

Microprocessors & Microcontrollers

SUBJECT CODE: 17CS44

LECTURE BY: PRAAHAS AMIN

REFERENCE MATERIAL:

THE X86 PC ASSEMBLY LANGUAGE DESIGN AND INTERFACING, 5TH EDITION, PEARSON, 2013. - MUHAMMAD ALI MAZIDI, JANICE GILLISPIE MAZIDI, DANNY CAUSEY

ARM SYSTEM DEVELOPERS GUIDE, ANDREW N SLOSS, DOMINIC SYMES AND CHRIS WRIGHT, ELSEVIER, MORGAN KAUFMAN PUBLISHERS, 2008.

Module 1 (10 Hours)

The x86 microprocessor: Brief history of the x86 family, Inside the 8088/86, Introduction to assembly programming, Introduction to Program Segments, The Stack, Flag register, x86 Addressing Modes. Assembly language programming: Directives & a Sample Program, Assemble, Link & Run a program, More Sample programs, Control Transfer Instructions, Data Types and Data Definition, Full Segment Definition, Flowcharts and Pseudo code.

Text book 1: Ch 1: 1.1 to 1.7, Ch 2: 2.1 to 2.7

Brief History of the x86 Family

Table 1-1: Evolution of Intel's Microprocessors (from the 8008 to the 8088)

Product	8008	8080	8085	8086	8088
Year introduced	1972	1974	1976	1978	1979
Technology	PMOS	NMOS	NMOS	NMOS	NMOS
Number of pins	18	40	40	40	40
Number of transistors	3000	4500	6500	29,000	29,000
Number of instructions	66	111	113	133	133
Physical memory	16K	64K	64K	1M	1M
Virtual memory	None	None	None	None	None
Internal data bus	8	8	8	16	16
External data bus	8	8	8	16	8
Address bus	8	16	16	20	20
Data types	8	8	8	8/16	8/16

Brief History of the x86 Family

Table 1-2: Evolution of Intel's Microprocessors (from the 8086 to the Pentium Pro)

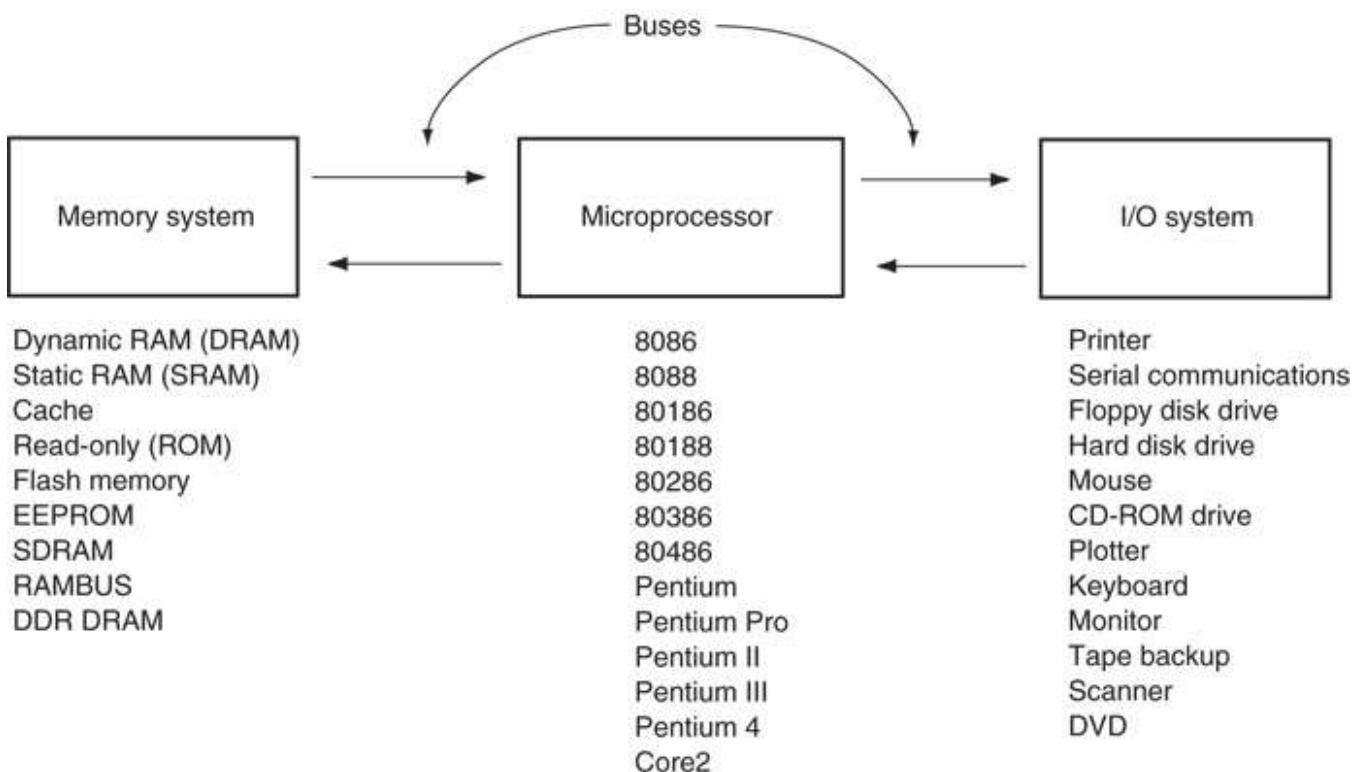
Product	8086	80286	80386	80486	Pentium	Pentium Pro
Year Introduced	1978	1982	1985	1989	1993	1995
Technology	NMOS	NMOS	CMOS	CMOS	BICMOS	BICMOS
Clock rate (MHz)	3–10	10–16	16–33	25–33	60, 66	150
Number of pins	40	68	132	168	273	387
Number of transistors	29,000	134,000	275,000	1.2 mill.	3.1 mill.	5.5 mill.
Physical memory	1M	16M	4G	4G	4G	64G
Virtual memory	None	1G	64T	64T	64T	64T
Internal data bus	16	16	32	32	32	32
External data bus	16	16	32	32	64	64
Address bus	20	24	32	32	32	36
Data types	8/16	8/16	8/16/32	8/16/32	8/16/32	8/16/32

Brief History of the x86 Family

Table 1-3: Evolution of Intel x86 Microprocessors: From Pentium II to Itanium

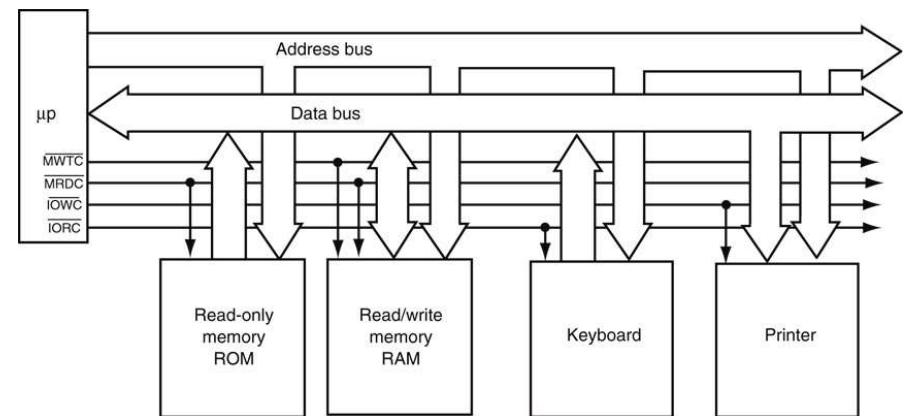
Product	Pentium II	Pentium III	Pentium 4	Itanium II
Year introduced	1997	1999	2000	2002
Technology	BICMOS	BICMOS	BICMOS	BICMOS
Number of transistors	7.5 mill.	9.5 mill.	42 mill.	220 mill.
Cache size	512K	512K	512K	3MB
Physical memory	64G	64G	64G	64G
Virtual memory	64T	64T	64T	64T
Internal data bus	32	32	32	64
External data bus	64	64	64	64
Address bus	36	36	36	64
Data types	8/16/32	8/16/32	8/16/32	8/16/32/64

Microprocessor Based Personal Computer System



The Microprocessor

- The CPU (**central processing unit**) :
 - The controlling element in a computer system.
 - Controls memory and I/O through connections called buses.
 - buses select an I/O or memory device, transfer data between I/O devices or memory and the microprocessor, control I/O and memory systems
 - Memory and I/O are controlled via instructions stored in memory, executed by the microprocessor.
 - A common group of wires that interconnect components in a computer system. They are used to transfer address, data, & control information between microprocessor, memory and I/O. Three buses exist for this transfer of information: address, data, and control. Figure shows how these buses interconnect various system components.
- Microprocessor performs three main tasks:
 - data transfer → between itself and the memory or I/O systems
 - processing → simple arithmetic and logic operations
 - program flow via simple decisions



The block diagram of a computer system showing the address, data, and control bus structure.

Intel 8086 Internal Architecture

Address bus:

- The address bus consists of 20 parallel lines.
- On these lines the CPU sends out the address of the memory locations that are to be written to or read from. The number of memory locations that the CPU can address is determined by the number of address lines, i.e if there are "n" address lines then it can directly address 2^n memory location.
- When the CPU reads data from or writes data to a port, it sends the port address on the address bus. Ex: CPU has 20 address lines can address 2^{20} or 1M(1048576) memory locations.

Data bus:

- It consists of 16 parallel lines. The data bus lines are bidirectional. This means that the CPU can read data from memory or from a port on these lines, or it can send data out to memory or to port on these lines.

Control bus:

- The control bus consists of 4 to 10 parallel signals lines. The CPU sends out signals on the control bus enable the outputs of addressed memory devices or port devices. Typical control bus signal are memory read, memory write, I/O read and I/O write.

Pin Diagram of 8086

Pin Diagram of 8086		MAX MODE	MIN MODE
GND	1	40	VCC
AD14	2	39	AD15
AD13	3	38	A16/S3
AD12	4	37	A17/S4
AD11	5	36	A18/S5
AD10	6	35	A19/S6
AD9	7	34	BHE/S7
AD8	8	33	MN/MX
CPU	9	32	RD
	10	31	RD/GT0 (HOLD)
	11	30	RD/GT1 (HLDA)
	12	29	LOCK (WR)
	13	28	S2 (M/I)
	14	27	S1 (DT/R)
	15	26	S0 (DEN)
	16	25	QS0 (ALE)
NMI	17	24	QS1 (INTA)
INTR	18	23	TEST
CLK	19	22	READY
GND	20	21	RESET

Power supply and frequency signals : It uses 5V DC supply at V_{CC} pin 40, and uses ground at V_{SS} pin 1 and 20 for its operation.

Clock signal: Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus: AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus: A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

S7/BHE: BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

RD : It is available at pin 32 and is used to read signal for Read operation.

READY: It is available at pin 32. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

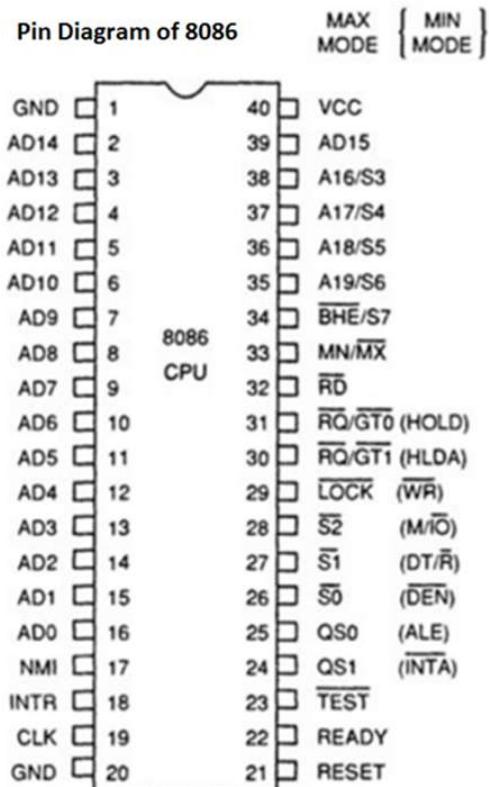
RESET: It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR : It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

INTA : It is an interrupt acknowledgement signal and is available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

NMI: It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

Pin Diagram of 8086



TEST: This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

MN/MX: It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

ALE: It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN: It stands for Data Enable and is available at pin 26. It is used to enable Transceiver 8286. The transceiver is a device used to separate data from the address/data bus.

DT/R : It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transceiver. When it is high, data is transmitted out and vice-a-versa.

M/I/O : This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR: It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/I/O signal.

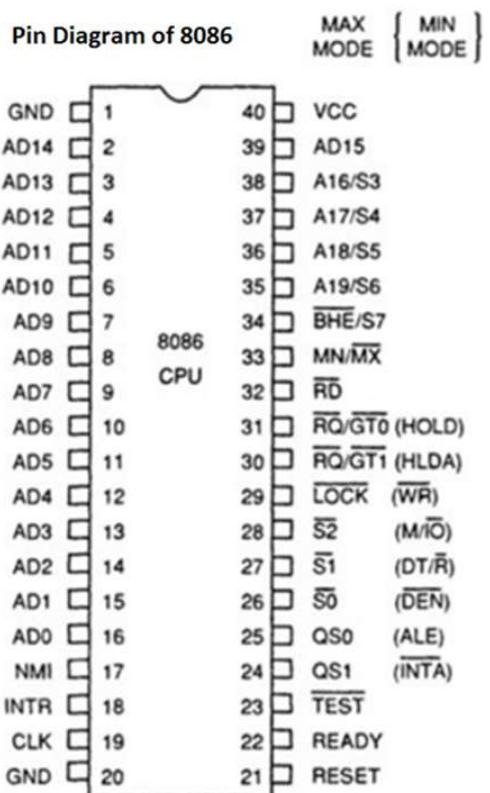
HLDA: It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD: This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

LOCK: When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

RQ/GT1 and **RQ/GT0**: These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. **RQ/GT0** has a higher priority than **RQ/GT1**.

Pin Diagram of 8086

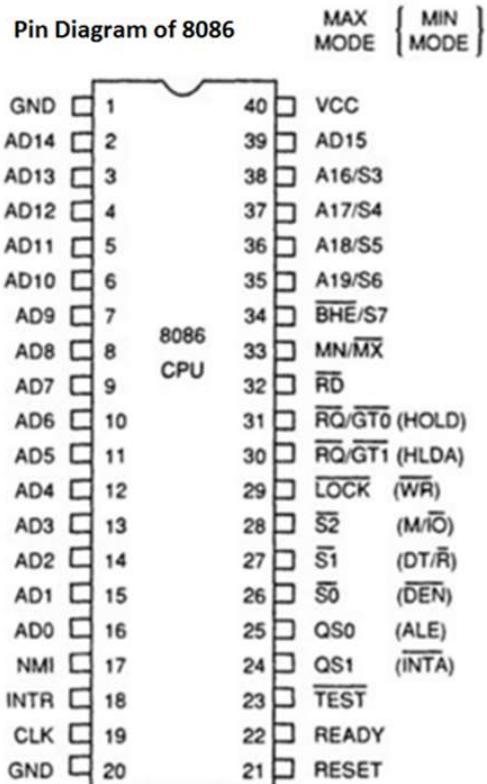


QS₁ and QS₀

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table –

QS0	QS1	Status
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

Pin Diagram of 8086



S0, S1, S2: These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status –

	S2	S1	S0	Status
	0	0	0	Interrupt acknowledgement
	0	0	1	I/O Read
	0	1	0	I/O Write
	0	1	1	Halt
	1	0	0	Opcode fetch
	1	0	1	Memory read
	1	1	0	Memory write
	1	1	1	Passive

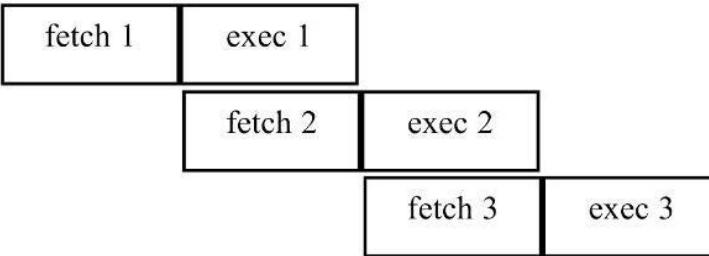
Pipelining

Consider what happens when the 8086 is first started.

Nonpipelined
(e.g., 8085)

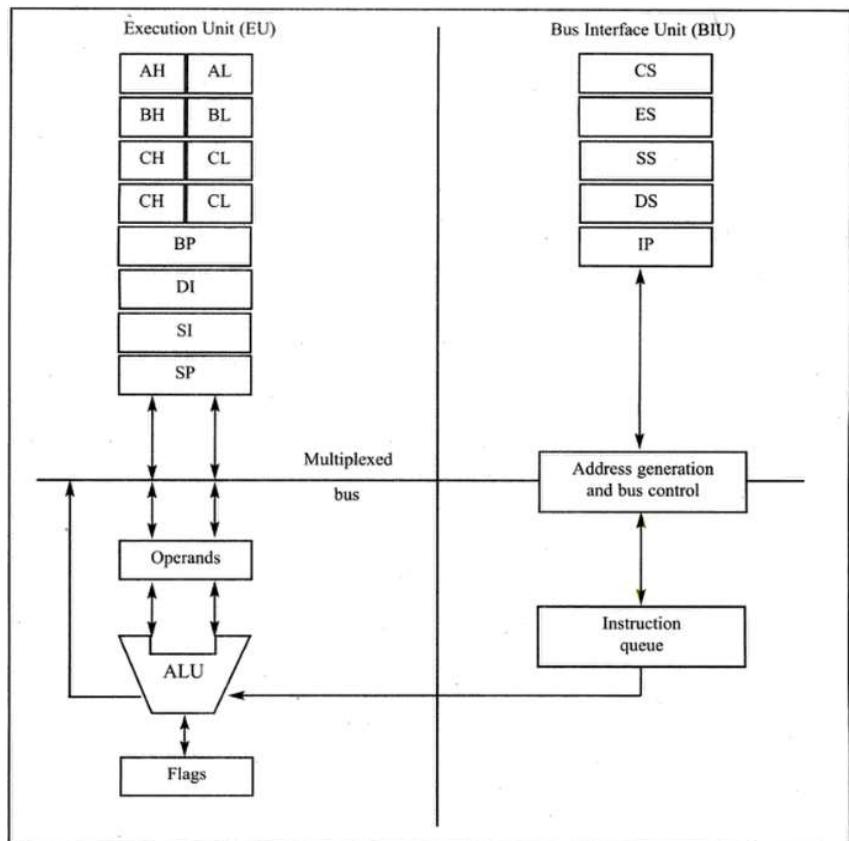


Pipelined
(e.g., 8086)



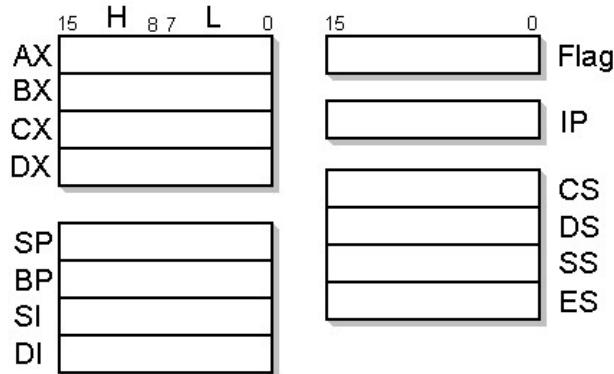
1. The BIU outputs the contents of the instruction pointer register (IP) onto the address bus, causing the selected byte or word to be read into the BIU.
2. Register IP is incremented by 1 to prepare for the next instruction fetch.
3. Once inside the BIU, the instruction is passed to the queue. This is a first-in, first-out storage register sometimes likened to a "pipeline".
4. Assuming that the queue is initially empty, the EU immediately draws this instruction from the queue and begins execution.
5. While the EU is executing this instruction, the BIU proceeds to fetch a new instruction. Depending on the execution time of the first instruction, the BIU may fill the queue with several new instructions before the EU is ready to draw its next instruction.

8086 Architecture

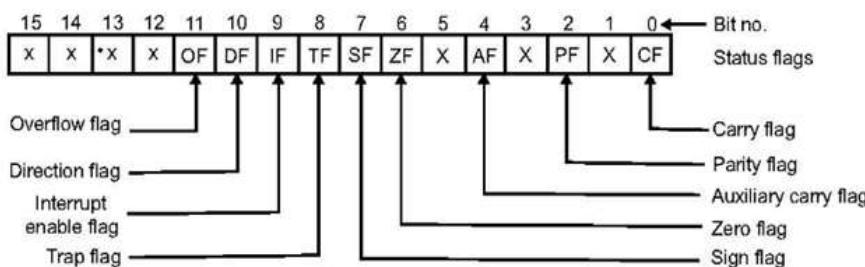


Registers

Registers of 8086



Status Flags of 8086



Used to store information temporarily.

General Purpose Registers

1. Accumulator (AX)
2. Base Index (BX)
3. Count (CX)
4. Data (DX)
5. Base Pointer (BP)
6. Source Index (SI)
7. Destination Index (DI)

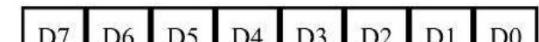
Special Purpose Registers

1. Instruction Pointer (RIP/EIP/IP)
2. Stack Pointer (RSP/ESP/SP)
3. Flags (RFLAGS/EFLAGS/FLAGS)
 - Carry (C)
 - Parity (P)
 - Auxiliary Carry (A)
 - Zero (Z)
 - Sign (S)
 - Trap (T)
 - Interrupt (I)
 - Direction (D)
 - Overflow (O)

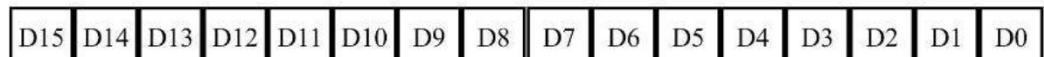
Segment Registers

1. Code Segment(CS)
2. Data Segment (DS)
3. Extra Segment(ES)
4. Stack Segment(SS)

8-bit register:



16-bit register:



Registers of x86

Table 1-4: Registers of the 8088/86/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note: The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

General Purpose Registers

- **AX** - 16-bit register (AX), or as either of two 8-bit registers (AH and AL).

The Accumulator is a general purpose register that is also used for instructions such as multiplication, division, and some of the adjustment instructions.

- **BX** - 16-bit register (BX), or as either of two 8-bit registers (BH and BL).

The Base Index Register is a general purpose register that is also used to hold the offset address of a location in the memory system in all versions of the microprocessor

- **CX** - 16-bit register (CX), or as either of two 8-bit registers (CH and CL).

The Count Register is a general purpose register that is also used to hold the count for various instructions

- **DX** - 16-bit register (DX), or as either of two 8-bit registers (DH and DL).

The Data Register is a general-purpose register that is also used to hold a part of the result from a multiplication or part of dividend before a division

- **BP** - 16-bit register (BP)

Base Pointer is a General purpose Register that is used to point to a memory location for memory data transfers

- **DI** - 16-bit register (DI) .

Destination Index is a general purpose register also used to addresses string destination data for the string instructions

- **SI** - 16-bit register (SI)

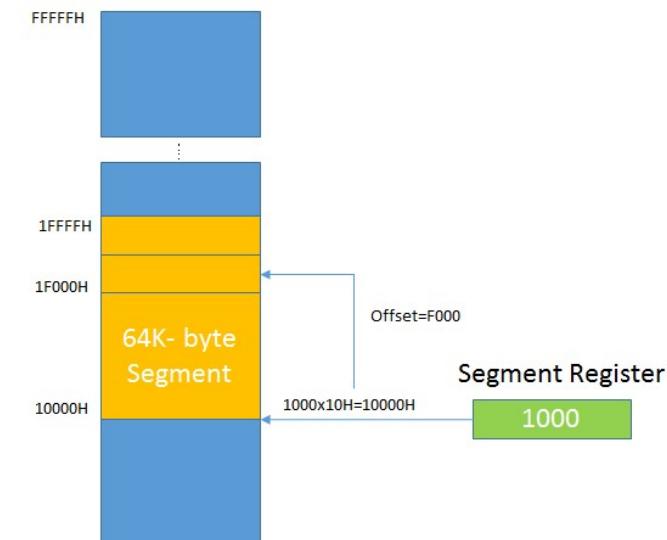
Source Index is a general purpose register also used to addresses source string data for the string instructions

Special Purpose Registers

- **IP** – Instruction Pointer addresses the next instruction in a section of memory (code segment)
- **SP** – Stack Pointer addresses an area of memory called the stack. Data is stored onto the stack through this pointer
- **FLAGS** indicate the condition of the microprocessor and control its operation. Flags never change for any data transfer or program control operation. Some of the flags are also used to control features found in the microprocessor.
 - **C (carry)** - Holds the carry after addition or borrow after subtraction. It also indicates error conditions. Set when theres a Carry out from D7(8-Bit ops) D15(16-Bit ops)
 - **P (parity)** – Parity is the count of ones in a number expressed as even or odd. Logic 0 for odd parity; logic 1 for even parity.
 - **A (auxiliary carry)** - Holds the carry (half-carry) after addition or the borrow after subtraction between bit positions 3 and 4 of the result(carry from D3 to D4).
 - **Z (zero)** - Shows that the result of an arithmetic or logic operation is zero.
 - **S (sign)** - Flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes.
 - **T (trap)** - The trap flag enables trapping through an on-chip debugging feature. Allows the program to single step for debugging.
 - **I (interrupt)** - Controls operation of the INTR (interrupt request) input pin. Set to Enable ,Clear to Disable.
 - **D (direction)** -Selects increment or decrement mode for the DI and/or SI registers.
 - **O (overflow)** - Occurs when signed number operatins are performed & the result has exceeded the capacity of the machine. Overflow is also used to detect errors in signed arithmetic operations.

Real Mode Memory Addressing

- Segment is an area of memory that includes 64K bytes and begins on an address evenly divisible by 16
- Real Mode Memory Addressing allows the microprocessor to address only the first 1 M byte.
- (Real Memory/DOS Memory/Conventional Memory)
- Addresses must consist of a Segment Address plus an Offset Address.
- Segment Address located within one of the Segment Registers defines the Beginning Address of any 64K byte memory segment.
- Offset Address(Displacement) selects any location within the selected Memory Segment.
- Segments in Real Mode always have a size of 64K Byte.
- Each Segment Register is internally Appended with a 0H on its rightmost end.
This forms the 20-Bit memory Address, allowing it to access the start of the Segment.
- The MP must generate a 20-Bit memory address to access a location within the first 1M byte of Memory
- Ending Address of a segment is found by adding FFFFH to the Segment Start Address
- Segment and offset address is sometimes written as 1000:2000.
 - a segment address of 1000H; an offset of 2000



—this shows a memory segment beginning at 1000H, ending at location FFFFH
•64K bytes in length

—also shows how an offset address, called a **displacement**, of F000H selects location 1F000H in the memory

Segment Registers

Used to generate memory addresses when combined with other registers in the microprocessor.

4 segment registers in 8086 microprocessor.

A segment register functions differently in real mode than in protected mode.

Used to "point" at location 0 (the base address) of each segment

CS (code) segment holds code (programs and procedures) used by the microprocessor.

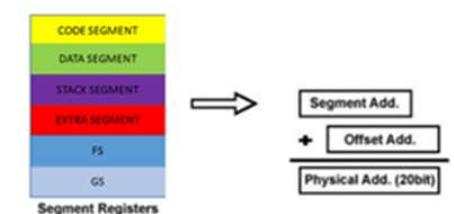
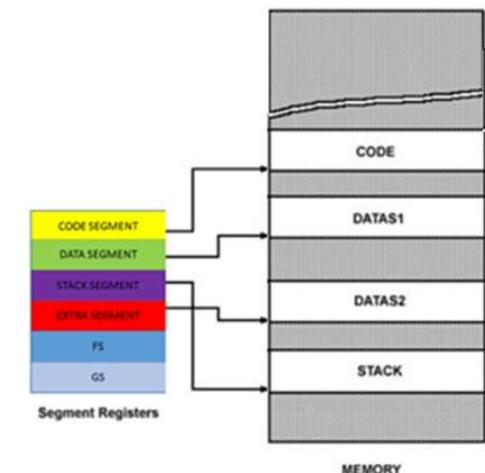
DS (data) contains most data used by a program.

- Data are accessed by an offset address or contents of other registers that hold the offset address

ES (extra) an additional data segment used by some instructions to hold destination data.

SS (stack) defines the area of memory used for the stack.

- stack entry point is determined by the stack segment and stack pointer registers
- the BP register also addresses data within the stack segment



Logical Address & Physical Address

8086 -16 Bit Microprocessor

Available Memory is 1 Mb (1048576 Locations)

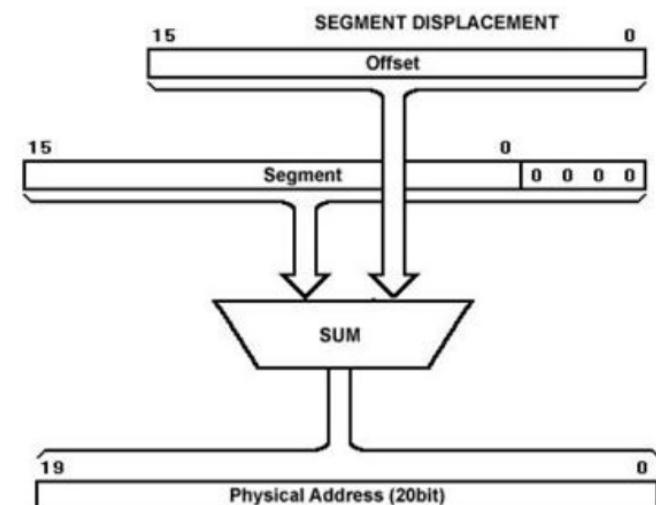
Accessible by 8086 64Kb (65536 Locations) corresponds to Segment Length

Multiply Segment Base Address by 10H. i.e Pad 4 Zeros

Add the Offset Address

This will generate the 20 Bit Physical Address.

Thus 1048576 Locations will now be accessible.



Physical Address Generation

Generally Physical Address (20 Bit) = Segment Base Address (SBA) + Effective Address (EA)

Code Segment: CS:IP

Physical Address (PA) = CS Base Address + Instruction Pointer (IP)

Data Segment (DS) : DS:BX / DS: SI/ DS:DI

PA = DS Base Address + EA can be in BX or SI or DI

Stack Segment (SS): SS:SP/ SS:BP

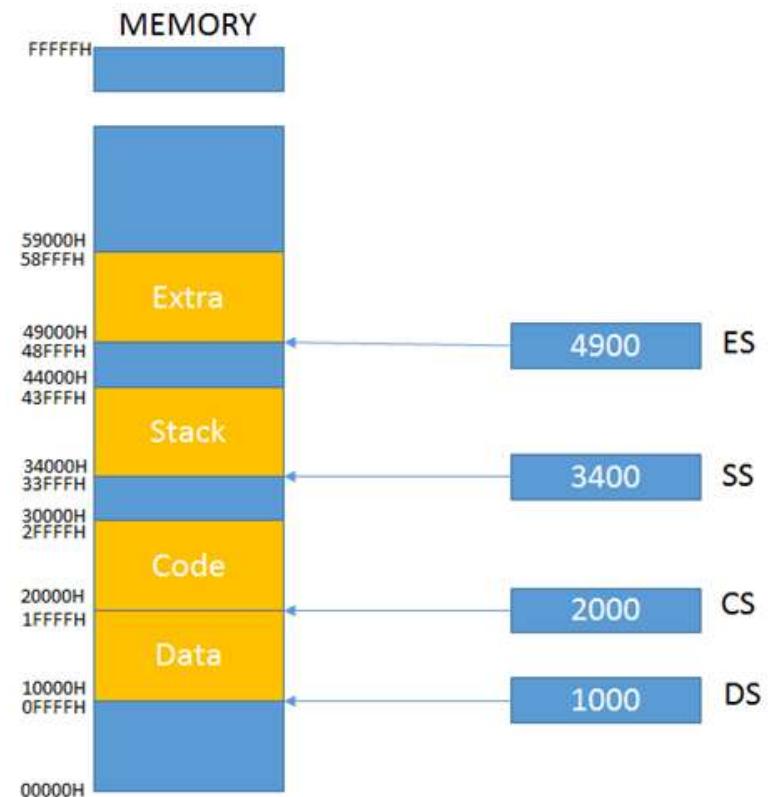
PA + SS Base Address + EA can be SP or BP

Extra Segment (ES): ES:DI

PA = ES Base Address + EA in DI

Default Segment & Offset Registers

SEGMENT	OFFSET REGISTER
Code Segment	IP
Data Segment	SI, DI,BX
Extra Segment	DI
Stack Segment	SP,BP



Advantages of Segmentation

Makes having a bigger memory model possible with a 16 bit processor(64Kb).

- 20 Bit Address Space on a 16 Bit Processor.

Relocation

Relocatable Program- Placed into any area of Memory & Executed without change.

Relocatable Data- Placed in any area of Memory and used without any change to the program.

Memory Segment can be relocated to any place in the memory system without changing any of the offset addresses.

Only the contents of the segment register must be changed to address the program in the new area of memory.

Ideal for use with General Purpose Computers where Memory Areas can vary from machine to machine

Introduction to Assembly Language Programming

Consists of a Mnemonic followed by 1 or 2 operands.

Eg: MOV destination,source ; copies source operand to destination

MOV AX,BX ; copies the data from BX to AX

MOV instruction does not affect the source operand.

Examples:

MOV CL,55H

Examples:

MOV CX,1234H

Examples:

MOV DS,1234H; Illegal. Values cannot be loaded directly into segment registers

MOV DL,CL

MOV AX,CX

MOV CS,5678H;Illegal

MOV AH,DL

MOV DX,AX

Values cannot be loaded directly into any Segment Register. First load the value into a non-segment register, then move it to a segment register.

MOV AL,AH

MOV DI,BX

MOV AX,1234H

MOV BH,CL

MOV SI,DI

MOV DS,AX

MOV CH,BH

MOV DS,SI

Moving a value less than FFH into a 16 bit register will cause the rest of the bits to assume 0s

MOV BP,DI

Moving a value greater than FFFFH into a 16 Bit register will cause an error

Examples

CS=24F6H

IP=634AH

Logical Address: 24F6:634A

Offset Address:634A

Physical Address: $[(24F6H \times 10H) + 634A] = 2B2AAH$

Lower Range: $24F60H + 0000H = 24F60H$

Upper Range: $24F60H + FFFFH = 34F5FH$

Logical vs Physical Address in CS

Logical Address CS:IP	Machine Language Opcode& Operand	Assembly Language Mnemonics & Operands
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL,72
1132:0106	89D1	MOV CX,DX
1132:0108	88C7	MOV BH,AL
1132:010A	B39F	MOV BL,9F
1132:010C	B420	MOV AH,20
1132:010E	01D0	ADD AX,DX
1132:0110	01D9	ADD CX,BX
1132:0112	05351F	ADD AX,1F35

Logical vs Physical Address

Logical Address	Physical Address	Machine Code Contents	Assembly Language
1132:0100	11420	B0	MOV AL,57
1132:0101	11421	57	
1132:0102	11422	B6	MOV DH,86
1132:0103	11423	86	
1132:0104	11424	B2	MOV DL,72
1132:0105	11425	72	
1132:0106	11426	89	MOV CX,DX
1132:0107	11427	D1	
1132:0108	11428	88	MOV BH,AL
1132:0109	11429	C7	

Logical Address	Physical Address	Machine Code Contents	Assembly Language
1132:010A	1142A	B3	MOV BL,9F
1132:010B	1142B	9F	
1132:010C	1142C	B4	MOV AH,20
1132:010D	1142D	20	
1132:010E	1142E	01	ADD AX,DX
1132:010F	1142F	D0	
1132:0110	11430	01	ADD CX,BX
1132:0101	11431	D9	
1132:0112	11432	05	ADD AX,1F35
1132:0113	11433	35	
1132:0114	11434	1F	

Data Segment

To add the 5 bytes of data Directly using Immediate values.

MOV AL,00H

ADD AL,25H

ADD AL,12H

ADD AL,15H

ADD AL,1FH

ADD AL,2BH

Data Segment

Assume DS offset begins at 200H, and Data is placed in the following locations	MOV AL,00H ADD AL,[0200]	MOV AL,00H MOV BX,0200H
DS:0200 = 25	ADD AL,[0201]	ADD AL,[BX]
DS:0201 = 12	ADD AL,[0202]	INC BX
DS:0202 = 15	ADD AL,[0203]	ADD AL,[BX]
DS:0203 = 1F	ADD AL,[0204]	INC BX
DS:0204 = 2B		ADD AL,[BX]

Note: Little Endian Convention- Low Byte goes to Lower Memory Address Location,
High Byte goes to Higher Memory Location.

MOV AX,35F3H

INC BX

MOV[1500H],AX

ADD AL,[BX]

DS:1500 = F3H ; DS:1501 = 35H

Example

Assume DS=5000H ; Offset is 1950. Calculate the Physical Address

DS= 5000H

LSHIFT DS => DS = 50000H

Add the Offset 50000H+1950H

Physical Address = 51950H

Example

If DS=7FA2H, & the offset is 438EH, calculate

Physical Address = 83DAEH

The Lower Range=7FA20H

Upper Range=8FA1FH

Logical Address = 7FA2:438AE

Assume that DS=578CH. To access a given byte of data at physical memory location 67F66H. Does the data segment cover the range where the data resides. If not what are the changes to be made.

Assume Memory location with the following contents DS:6826 = 48H & DS:6827 =22H. Show the contents of BX for the instruction

The Stack

It is a section of Read/Write Memory used by the CPU to store information temporarily.

Registers are limited in number.(Transistors are Expensive!)

Stack is however slower to access than Registers.

Registers used to access the Stack are SP Register & SS Register.

These Registers must be loaded with the appropriate values before the Stack is used.

Every Register except Segment Registers & SP can be stored & Retrieved from the Stack.

Storing into the Stack is called PUSH.

Loading from the Stack is called POP.

The Stack

SP points at the current memory location used as Top Of the Stack (TOS).

When data is PUSHed onto the stack SP is decremented by 2.

When data is POPped from the stack SP is incremented by 2.

Instruction used for Storing & Loading from the Stack are PUSH & POP.

PUSH & POP work with entire 16-Bit Register.

STACK Operation

When a word of data is PUSHed onto the Stack:

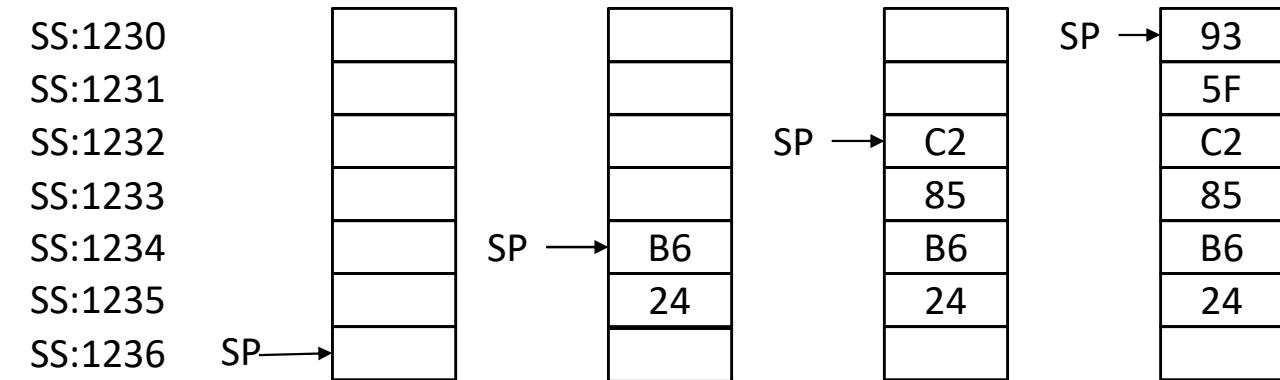
1. Higher order 8-Bits are placed in SP-1 & Lower Order 8-Bits are placed in SP-2
2. SP is then decremented by 2.
3. Next data is placed in the next available stack memory location.

When a word of data is POPped from the Stack:

1. Lower order 8-Bits are removed from the location addressed by SP & Higher Order 8-Bits are removed from the location addressed by SP+1
2. SP Register is then incremented by 2.
3. Data maybe POPped from the stack into any register or any segment register except CS Register.

- SP always points to an area of memory within Stack Segment
- SP Register adds to SSx10H to form the Stack Memory Address in Real Mode memory Addressing.

PUSH Operation Example



Register Values:

SP=1236H

AX=24B6H

DI=85C2H

DX=5F93H

Instruction Sequence:

PUSH AX

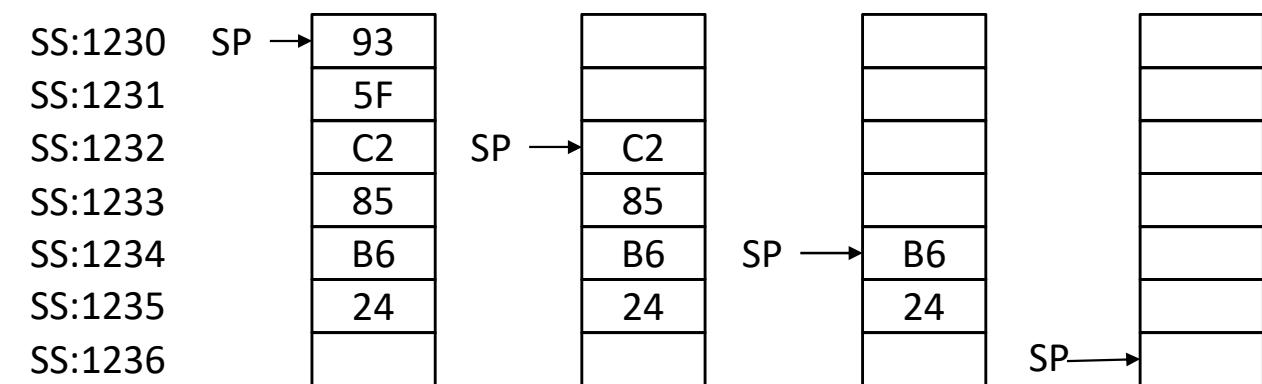
PUSH DI

PUSH DX

When a word of data is PUSHed onto the Stack:

1. Higher order 8-Bits are placed in SP-1 & Lower Order 8-Bits are placed in SP-2
2. SP is then decremented by 2.
3. Next data is placed in the next available stack memory location.

POP Operation Example



Register Values:

SP=1230H

AX=24B6H

DI=85C2H

DX=5F93H

Instruction Sequence:

POP AX

POP DI

POP DX

When a word of data is POPped from the Stack:

1. Lower order 8-Bits are removed from the location addressed by SP & Higher Order 8-Bits are removed from the location addressed by SP+1
2. SP Register is then incremented by 2.
3. Data maybe POPped from the stack into any register or any segment register except CS Register.

Example

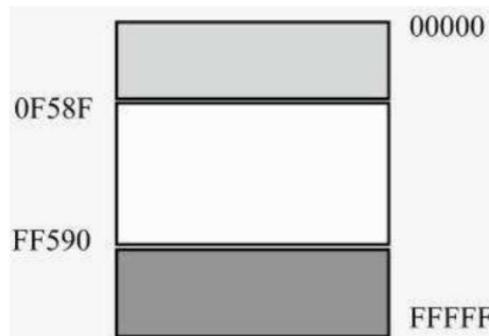
SS=3500H ; SP=FFFEH; Calculate:

- a) Physical Address b)Lower Range c)Upper Range d)Show the Stack's Logical Address
- A. 44FFEH (35000H+FFFEH)
- B. 35000H (35000H+0000H)
- C. 44FFFH (35000H+FFFFH)
- D. 3500H:FFFEH

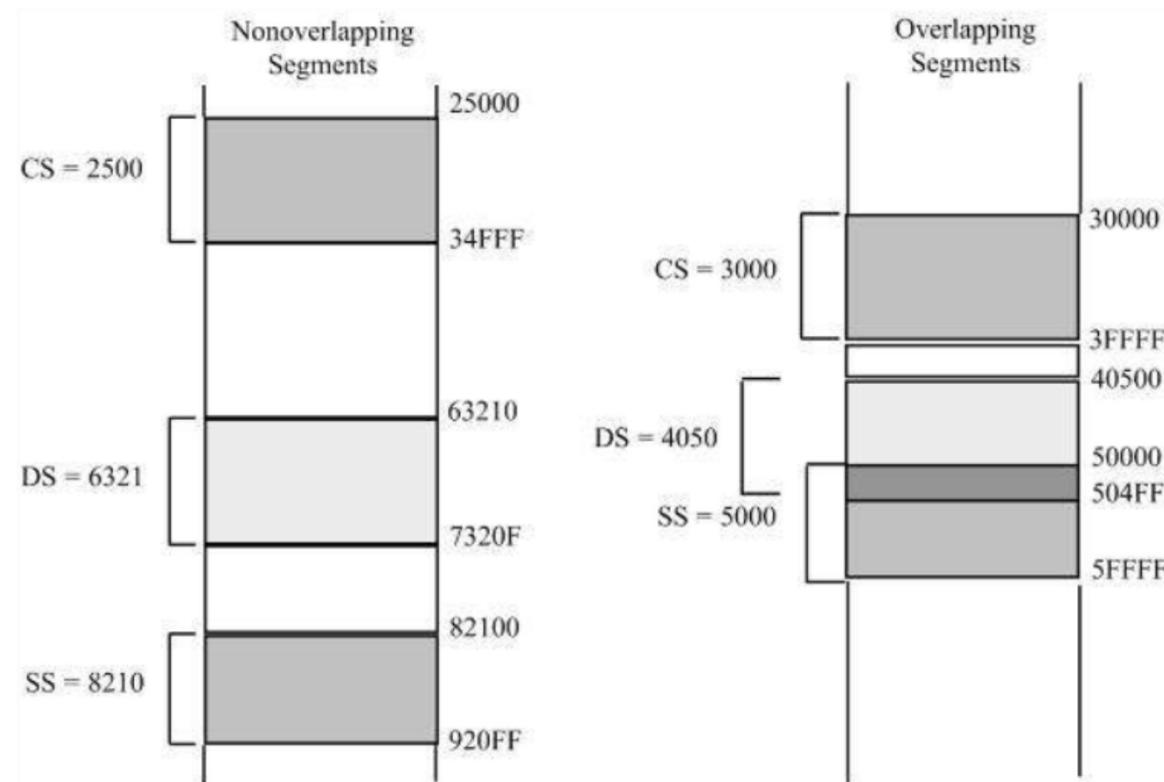
Range of Physical Addresses for CS=FF59H

Low Range is FF590H+0000H = FF590H

Upper Range goes to FFFFFH and wraps around to 00000 to 0F58FH as shown in the figure.

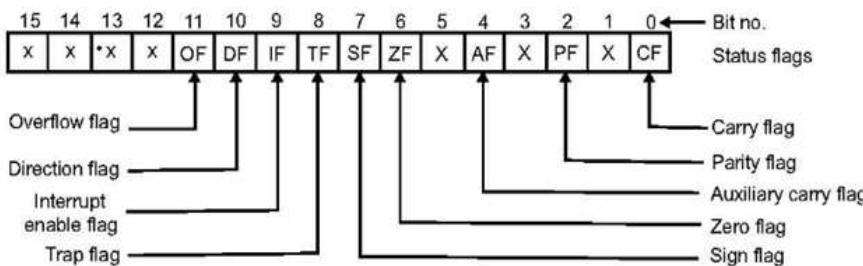


Overlapping Segments



FLAG Register

Status Flags of 8086



16-Bit Register also known as Status Register
6 of the flags are Conditional Flags (CF,PF,AF,ZF,SF & OF)
3 are Control Flags (TF,IF & DF)

- **C (carry)** - Holds the carry after addition or borrow after subtraction. It also indicates error conditions. Set when there's a Carry out from D7(8-Bit ops) D15(16-Bit ops)
- **P (parity)** - Parity is the count of ones in a number expressed as even or odd. Logic 0 for odd parity; logic 1 for even parity.
- **A (auxiliary carry)** - Holds the carry (half-carry) after addition or the borrow after subtraction between bit positions 3 and 4 of the result (carry from D3 to D4).
- **Z (zero)** - Shows that the result of an arithmetic or logic operation is zero.
- **S (sign)** - Flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes.
- **T (trap)** - The trap flag enables trapping through an on-chip debugging feature. Allows the program to single step for debugging.
- **I (interrupt)** - Controls operation of the INTR (interrupt request) input pin. Set to Enable, Clear to Disable.
- **D (direction)** - Selects increment or decrement mode for the DI and/or SI registers.
- **O (overflow)** - Occurs when signed number operations are performed & the result has exceeded the capacity of the machine. Overflow is also used to detect errors in signed arithmetic operations.

Example

Show how the Flag Register is affected by the addition of 38H & 2FH.

MOV BH,38H ; BH=38H

ADD BH,2FH ;add 2FH to 38H now BH is 67H

$$\begin{array}{r} 38H+ \\ 2FH \\ \hline 67H \end{array} \quad \begin{array}{r} 0011\ 1000\ + \\ 0010\ 1111 \\ \hline 0110\ 0111 \end{array}$$

CF=0 since there is no carry beyond D7

PF=0 since there is an odd number of 1s in the result

AF=1 since there is a carry from D3 to D4

ZF=0 since the result is not zero

SF=0 since D7 of the result is zero

Example

- Show how FLAG register is affected by

MOV AL,9CH

MOV DH,64H

ADD AL,DH

- Show how FLAG register is affected by

MOV AX,34F5H

ADD AX,95EBH

Example

- Show how FLAG register is affected by

MOV BX,AAAAH

ADD BX,5556H

- Show how the Flag register is affected by

MOV AX,94C2H

MOV BX,323EH

ADD AX,BX

MOV DX,AX

MOV CX,DX

Use of ZF for Looping

Looping refers to a set of instruction that is repeated for a number of times.

Counter can be used to keep track of how many times a loop must be repeated.

Each time the loop is executed the counter is decremented & ZF is checked.

When Counter becomes 0, ZF gets set & loop is stopped.

```
MOV CX,05H  
MOV BX,0200H  
MOV AL,00H  
ADD_LOOP:ADD AL,[BX]  
    INC BX  
    DEC CX  
    JNZ ADD_LOOP
```

Addressing Modes

Sl.no.	Addressing Mode	Operand	Default Segment	Example
1	<i>Register</i>	Register	NONE	MOV AX,BX
2	<i>Immediate</i>	Data	NONE	MOV AX,1000H
3	<i>Direct</i>	[Offset]	Data Segment	MOV AX,[1234H]
4	<i>Register Indirect</i>	[BX]/[SI]/[DI]	Data Segment	MOV AX,[BX] / MOV AX,[SI] / MOV AX,[DI]
		[SP]/[BP]	Stack Segment	MOV AX,[SP] / MOV AX,[BP]
		[IP]	Code Segment	

Addressing Modes

5 (a)	<i>Register Relative</i>			
	<i>Based Relative</i>	[BX]+Displacement	Data Segment	MOV AX,[BX+10H]
		[BP]+Displacement	Stack Segment	MOV AX,[BP+10H]
	<i>Indexed Relative</i>	[DI]+Displacement	Data Segment	MOV AX,[DI+10H]
		[SI]+Displacement	Data Segment	MOV AX,[SI+10H]
6	<i>Based Indexed Relative</i>	[BX][SI]+Displacement	Data Segment	MOV AX,[BX+SI+10H]
		[BX][DI]+Displacement	Data Segment	MOV AX,[BX+DI+10H]
		[BP][SI]+Displacement	Stack Segment	MOV AX,[BP+SI+10H]
		[BP][DI]+Displacement	Stack Segment	MOV AX,[BP+DI+10H]

Register Addressing Mode

- Data is moved between 2 Registers.
- Involves the use of Registers to hold the data to be manipulated.
- Memory is not accessed.
- Relatively Fast.

Operand: Register

Default Segment: None

Example: MOV AX,BX

Immediate Addressing Mode

- A Constant Value is moved into Destination Register
- Source Operand is a constant.
- During Assembly the operand comes immediately after the opcode.
- Usually used to load data into any registers (except segment registers & flag register)

Operand: Immediate Data / Constant

Default Segment: None

Example: MOV AX,1000H

Direct Addressing Mode

- Copies data between a memory location & Register
- Data is in some memory location.
- Address of the data in memory comes immediately after the instruction.
- This address is the offset address. Physical Address is formed by Adding Displacement to the Left Shifted Default Segment Address.
- Memory to Memory Transfer is not possible except with MOVS.

Operand: [Offset]

Default Segment: DS

Example: MOV AX,[1234H] ; Moves contents of DS:1234H & DS:1235H into AX

Example: MOV AL,[1234H] ; Moves contents of DS:1234H into AL

Register Indirect Addressing Mode

- Transfers data between a Register & Memory Location.
- The address of the memory location where the operand resides is held by a register.
- Registers used for this addressing mode are SI,DI & BX

Operand: [BX]/[SI]/[DI]

Default Segment: DS

Example: MOV AX,[BX] / MOV AX,[SI] / MOV AX,[DI]

Operand: [SP]/[BP]

Default Segment: SS

Example: MOV AX,[SP] / MOV AX,[BP]

Operand: [IP]

Default Segment : CS

Based Relative Addressing Mode

- Transfers Data between a Register & Memory Location addressed by a Base Register plus a Displacement.
- Base Registers BX or BP as well as a Displacement is used to calculate the Effective Address

Operand: [BX]+Displacement

Default Segment: DS

Example: MOV AX,[BX+10H]

Operand:[BP]+Displacement

Default Segment: SS

Example:MOV AX,[BP+10H]

Indexed Relative Addressing Mode

- Transfers Data between a Register & Memory Location addressed by an IndexRegister plus a Displacement.
- Index Registers SI or DI as well as a Displacement is used to calculate the Effective Address

Operand: [SI]+Displacement

Default Segment: DS

Example: MOV AX,[SI+10H]

Operand:[DI]+Displacement

Default Segment: DS

Example:MOV AX,[DI+10H]

Based Indexed Relative Addressing Mode

- Transfers data between a Register & a Memory Location addressed by a Base Register, an Index Register plus a Displacement.

Operand: [BX][SI]+Displacement

Default Segment: DS

Example: MOV AX,[BX+SI+10H]

Operand: [BP][SI]+Displacement

Default Segment: SS

Example: MOV AX,[BP+SI+10H]

Operand: [BX][DI]+Displacement

Default Segment: DS

Example: MOV AX,[BX+DI+10H]

Operand:[BP][DI]+Displacement

Default Segment: SS

Example: MOV AX,[BP+DI+10H]

Default Segment & Offset Registers

SEGMENT	OFFSET REGISTER
Code Segment	IP
Data Segment	SI, DI,BX
Extra Segment	DI
Stack Segment	SP,BP

Segment Override Examples

MOV AX,CS:[BP] ; segment used: CS:BP / Default: SS:BP

MOV DX,SS:[SI] ; segment used: SS:SI / Default :DS:SI

MOV AX,DS:[BP] ; segment used: DS:BP / Default : SS:BP

MOV CX,ES:[BX]+12 ; segment used ES:BX+12 / Default : DS:BX+12

MOV SS:[BX][DI]+32,AX ; segment used SS:BX+DI+32 / Default DS:BX+DI+32

Assembly Language Instruction

- [label:]mnemonic [operands] [;comment]
- Brackets indicate optional field. Do not type the brackets.
- Label field allows the program to refer to a line of code by name. It cannot exceed 31 characters. Labels or directives need not end with a colon. Label for an opcode generating instruction must end with a colon.
- Mnemonic along with operands perform some task.
- Comment field begins with a ";" . Comments maybe at the end of the line or they maybe on a line by themselves. They are optional but recommended as they make it easier to read & understand the program.

Model Definition

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
        .MODEL SMALL
        .STACK 64
        .DATA
DATA1      DB      52H
DATA2      DB      29H
SUM        DB      ?
        .CODE
MAIN       PROC   FAR           ;this is the program entry point
          MOV    AX, @DATA        ;load the data segment address
          MOV    DS, AX           ;assign value to DS
          MOV    AL, DATA1         ;get the first operand
          MOV    BL, DATA2         ;get the second operand
          ADD    AL, BL            ;add the operands
          MOV    SUM, AL           ;store the result in location SUM
          MOV    AH, 4CH            ;set up to return to OS
          INT    21H              ;
MAIN       ENDP
END     MAIN           ;this is the program exit point
```

Model Definition

- .MODEL SMALL – Maximum of 64K Bytes of memory for code & 64K Bytes for data
- .MODEL MEDIUM – Data must fit into 64K Bytes, but Code can exceed 64K Bytes
- .MODEL COMPACT – Data can exceed 64K Bytes, but Code cannot exceed 64K Bytes
- .MODEL LARGE – Both Data & Code can exceed 64K Bytes, but no single set of data should exceed 64K Bytes
- .MODEL HUGE – Both Data & Code can exceed 64K Bytes. Data Items can exceed 64K Bytes
- .MODEL TINY – Data & Code must fit into 64K Bytes. Used with COM files.

Segment Definition

.CODE – Contains the assembly language instructions

.DATA – Defines the data that the program will use

.STACK – Defines storage for the stack

Sample Shell of an Assembly Language Program

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
; USING SIMPLIFIED SEGMENT DEFINITION
    .MODEL SMALL
    .STACK 64
    .DATA
    ;
    ;place data definitions here
    ;
    .CODE
MAIN    PROC FAR           ;this is the program entry point
        MOV AX,@DATA      ;load the data segment address
        MOV DS,AX         ;assign value to DS
        ;
        ;place code here
        ;
        MOV AH,4CH        ;set up to
        INT 21H          ;return to OS
MAIN    ENDP
        END MAIN         ;this is the program exit point
```

Assemble, Link & Run

■ Edit The Program –

Input: Keyboard

Program: Editor

Output: myfile.asm

■ Assemble the Program –

Input: myfile.asm

Program: MASM(Assembler)

Output: myfile.obj

myfile.lst

myfile.crf

■ Link the Program –

Input: myfile.obj

Program: LINK (Linker)

Output: myfile.exe

myfile.map

■ .asm file : Source File. Created with editor/word processor.

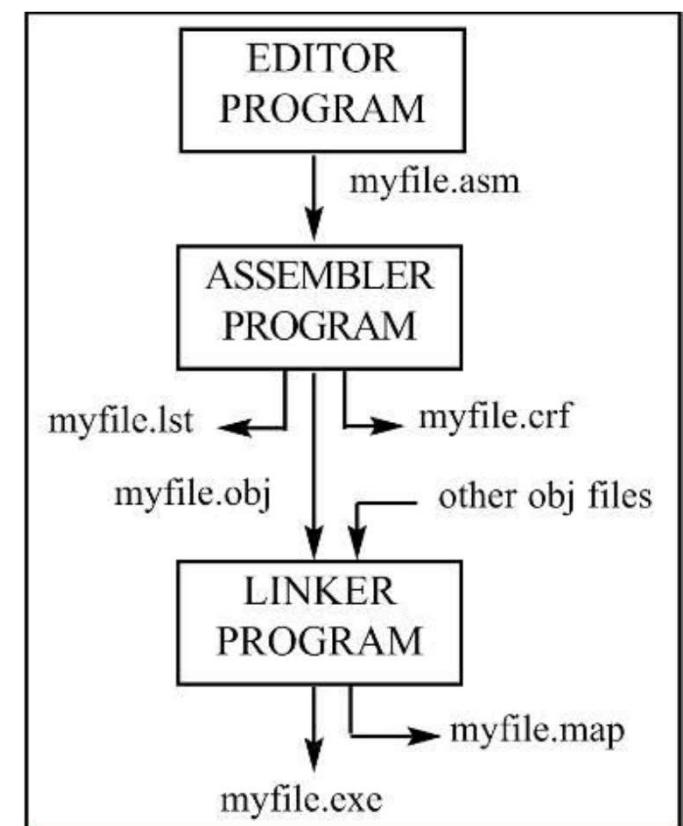
■ .obj file: Assembler converts source file(assembly language instructions) into Machine Language.

■ .lst file: Lists all opcodes & offset addresses as well as errors that the assembler detected. To make it readable use PAGE & TITLE Directive. Syntax: PAGE [lines] [columns]

■ .crf file: Cross Referencing File. Provides an alphabetical list of all symbols & labels used in the program as well as program line numbers in which they are referenced.

■ .exe file: the executable file produced by the Linker.

■ .map file: Provides the name of each segment, where it starts & stops & its size in bytes.



Control Transfer Instructions- Conditional Jumps

Table 2-1: 8086 Conditional Jump Instructions

Note: “Above” and “below” refer to the relationship of two unsigned values; “greater” and “less” refer to the relationship of two signed values.

Mnemonic	Condition Tested	“Jump IF ...”
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

***Note:**

All conditional Jumps are SHORT Jumps.
i.e address of the target must be within -128(Back) to +127(Forward) bytes of the current IP. If violated will give error “relative jump out of range”

Conditional Jumps are 2 Byte Instructions.

1 Byte is the opcode of the J Condition

1 Byte is a value between 00 & FF

The 2nd Byte is added to IP to perform the Jump. If Jump is backward then the 2nd byte is the 2's complement of the displacement

Control Transfer Instructions- Conditional Jumps- Backward Jump

1067:0000	B86610		MOV	AX, 1066	"JNZ AGAIN" was assembled as "JNZ 000D", and 000D is the address of the instruction with the label AGAIN.
1067:0003	8ED8	MOV	DS, AX		"JNZ 000D" has the opcode 75 and the target address FA.
1067:0005	B90500		MOV	CX, 0005	
1067:0008	BB0000		MOV	BX, 0000	
1067:000D	0207	ADD	AL,[BX]		The IP value of MOV,0013, is added to FA to calculate the address of label AGAIN, and the carry is dropped.
1067:000F	43		INC	BX	
1067:0010	49		DEC	CX	
1067:0011	75FA	JNZ	000D		FA is the 2's complement of -6.
1067:0013	A20500		MOV	[0005] , AL	
1067:0016	B44C	MOV	AH, 4C		
1067:0018	CD21	INT	21		

Control Transfer Instructions- Conditional Jumps – Forward Jump

0005	8A	47	02	AGAIN:	MOV	AL,[BX]	+2	"JB NEXT" has the opcode 72, the target address 06 and is located at IP = 000A and 000B.
0008	3C	61			CMP	AL, 61H		
000A	72	06			JB	NEXT		
000C	3C	7A			CMP	AL, 7AH		The jump is 6 bytes from the next instruction, is IP = 000C.
000E	77	02			JA	NEXT		
0010	24	DF			AND	AL, ODFH		
0012	88	04		NEXT:	MOV	[SI] , AL		Adding gives us 000CH + 0006H = 0012H, which is the exact address of the NEXT label.

Control Transfer Instructions – Unconditional Jumps

- SHORT JUMP(EB)-2 Byte Instruction – 1 Byte Displacement (Range is -128 to +127 bytes of the IP) 00-FF i.e 0 to 255. Both Forward & Backward Jumps are possible, therefore -128 to +127 from IP. (**CS remains Same. Only IP updated**)

EB	Displacement (+127 to -128)	Format: JMP SHORT label
----	--------------------------------	--------------------------------

- NEAR JUMP(E9)-3 Byte Instruction – 2 Byte Displacement (Range is -32767 to +32767 of IP) can cover entire CS. Intrasegment Control Transfer (**CS remains Same. Only IP updated**)

E9	Displacement (Low Byte)	Displacement (High Byte)	Format: JMP label
----	----------------------------	-----------------------------	--------------------------

- FAR JUMP(EA)- 5 Byte Instruction- Intersegment Control Transfer(**CS & IP Both Updated**) – 4 Byte Displacement.

EA	IP (Low Byte)	IP (High Byte)	CS (Low Byte)	CS (High Byte)	Format: JMP FAR PTR label
----	---------------	----------------	---------------	----------------	----------------------------------

Control Transfer Instructions – Unconditional Jumps

NEAR JUMP

Direct Jump- exactly like the short jump except that the target address can be anywhere in the segment in the range +32767 to -32768 of the current IP

Register Indirect Jump- target address is in a register. In "JMP BX", IP takes the value BX.

Memory Indirect Jump - target address is the contents of two memory locations, pointed at by the register.

JMP [DI] will replace the **IP** with the contents of memory locations pointed at by **DI** and **DI+1**.

Control Transfer Instructions – CALL

The CALL instruction is used to call a procedure, to perform tasks that need to be performed frequently. Makes program more structured

NEAR CALL - target address is be in the current segment.

FAR CALL - target address is outside the current CS segment.

The Microprocessor must know where to return after executing the subroutine.

The microprocessor saves the address of the instruction following the CALL on the stack.

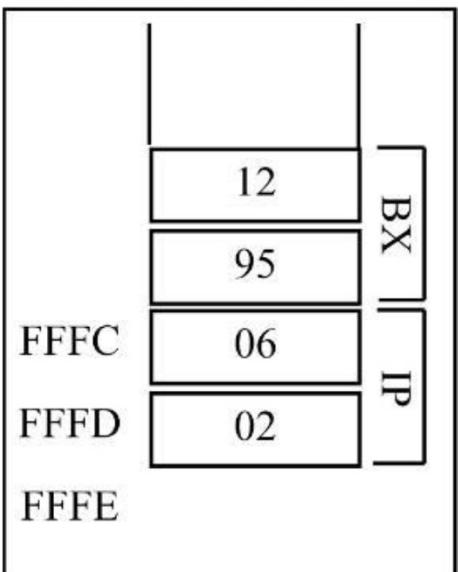
In NEAR CALL only IP is saved on the stack. In FAR CALL both CS & IP are saved on the stack.

For control to be transferred back to the caller, the last subroutine instruction must be RET (return).

Control Transfer Instructions -CALL

12B0:0200 BB1295 MOV BX, 9512 ASSUME: SP=FFFEH:
12B0:0203 E8FA00 CALL 0300 Since this call is NEAR CALL only IP is stored on the
12B0:0206 B82F14 MOV AX, 142F STACK.

The IP value corresponding to MOV AX,142FH is saved on the stack.



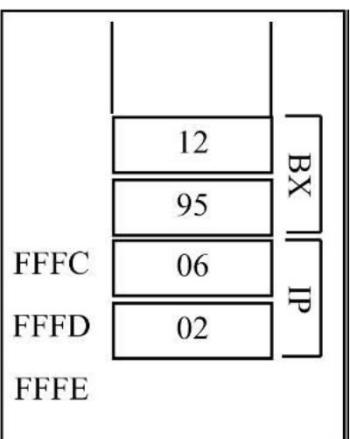
Control Transfer Instructions - RET

12B0:0300	53	PUSH BX
12B0:0301
.....
12B0:0309	5B	POP BX
12B0:030A	C3	RET

The last instruction of the called subroutine must be a RET instruction that directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address 0206.

The number of PUSH and POP instructions (which alter the SP) **must match**.

- For every PUSH there must be a POP.



Data Types & Data Definition

Assembler Directives	Description
DB	Used to declare a BYTE. Allows allocation of memory in byte sized chunks.
DW	Used to declare a WORD. Allows allocation of memory in 2-byte(Word) sized chunks.
DD	Used to declare a DOUBLEWORD. Allows allocation of memory in 4-byte sized chunks.
DQ	Used to declare a QUADWORD. Allows allocation of memory in 8-byte sized chunks.
DT	Used to declare Ten Bytes for memory allocation of Packed BCD numbers.
DUP	Used to duplicate a given number of characters
EQU	Used to give name to some value or symbol. Each time the assembler finds the given names in the program, it will replace the name with the value or a symbol.
ORG	Used to indicate the beginning of Offset Address of the segment.

.MODEL Definition vs Full Segment Definition

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
; USING SIMPLIFIED SEGMENT DEFINITION
    .MODEL SMALL
    .STACK 64
    .DATA
    ;
    ;place data definitions here
    ;
    .CODE
MAIN    PROC FAR      ;this is the program entry point
        MOV AX,@DATA   ;load the data segment address
        MOV DS,AX      ;assign value to DS
        ;
        ;place code here
        ;
        MOV AH,4CH     ;set up to
        INT 21H        ;return to OS
        ENDP
MAIN    END MAIN      ;this is the program exit point
```

```
;FULL SEGMENT DEFINITION          ;SIMPLIFIED FORMAT
;--- stack segment ---           .MODEL  SMALL
name1 SEGMENT                   .STACK   64
    DB 64 DUP (?)               ;
name1 ENDS                      ;
;--- data segment ---           ;-----.
name2 SEGMENT                   . DATA
;place data definitions here   ;place data definitions here
name2 ENDS                      ;
;--- code segment ---           ;-----.
name3 SEGMENT                   .CODE
MAIN    PROC FAR
        ASSUME ...
        MOV AX,name2
        MOV DS,AX
        ...
MAIN    ENDP
name3 ENDS
        END   MAIN
```

EXE vs. COM Files

EXE File	COM File
Unlimited Size	Maximum size 64K Bytes
Stack Segment is defined	No Stack Segment Definition
Data Segment is defined	Data segment defined in Code Segment
Code, Data defined at any offset address	Code & Data begin at offset 0100H
Larger file (Takes more Memory)	Smaller File (Takes Less memory)

Limited amount of Memory necessitates Compact Code

EXE file can be of any size.

COM Files are Compact(cannot be greater than 64KB) & must fit into a Single Segment

Flowcharts & Pseudocode

Structured Programming – Term used to denote Programming techniques that can make a program easier to code, debug & maintain over time. The following are the principles:

- Program should be designed before it is coded. – Flowcharts/Pseudocode.
- Using comments within the program & documentation accompanying the program.
- Main routine should primarily contain calls to sub-routines- Top Down Programming
- Data Control

Commonly used Flowchart Symbols

Terminal

Process

Decision
?

Subroutine

Input/
Output



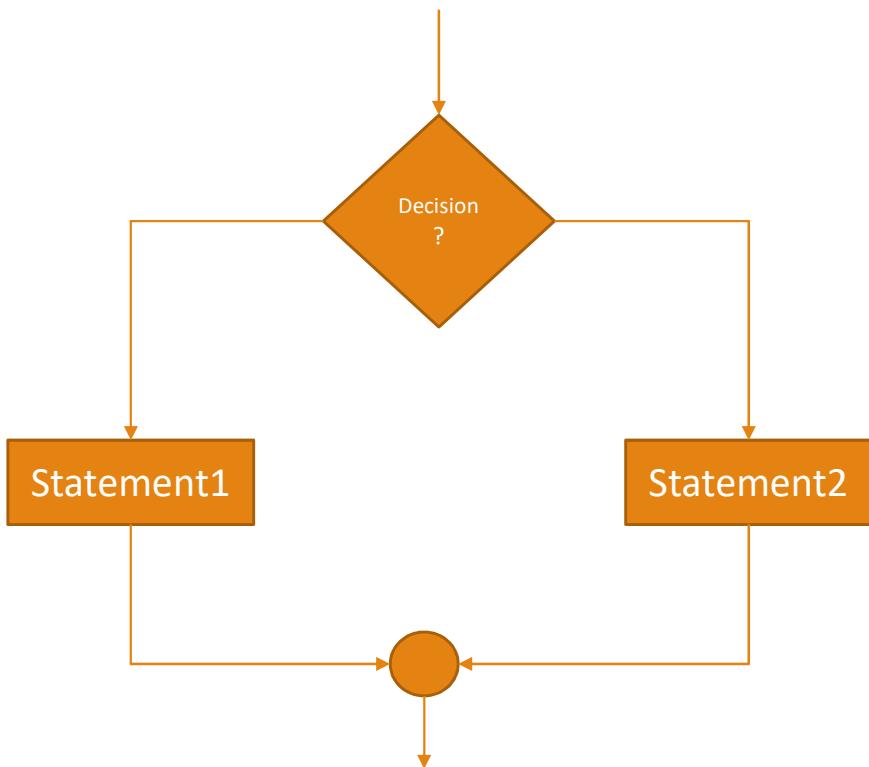
Sequence



Statement 1

Statement 2

IF-THEN-ELSE



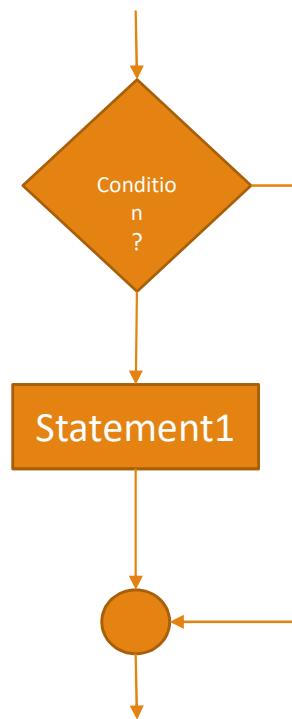
IF (condition) THEN

Statement 1

ELSE

Statement 2

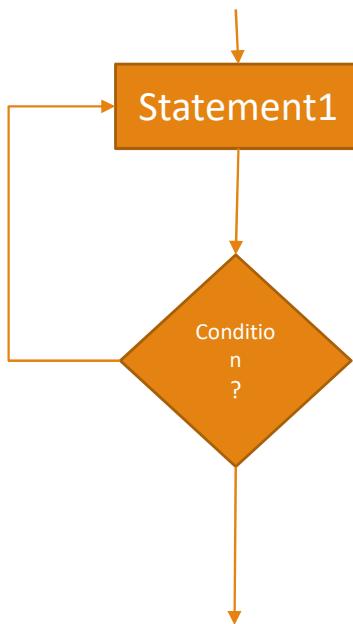
IF-THEN



IF (condition) THEN

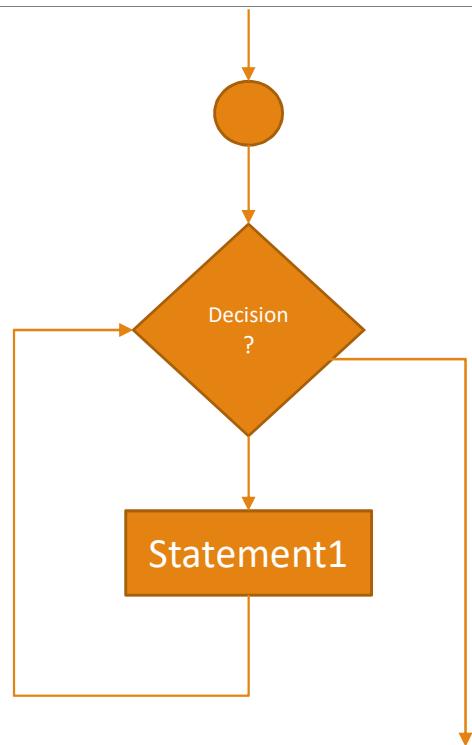
Statement 1

REPEAT-UNTIL



REPEAT
Statement 1
UNTIL (Condition)

WHILE-DO



WHILE (condition) DO

Statement 1

Pseudo-Code & FC (Add 5 Bytes)

Count =5

REPEAT

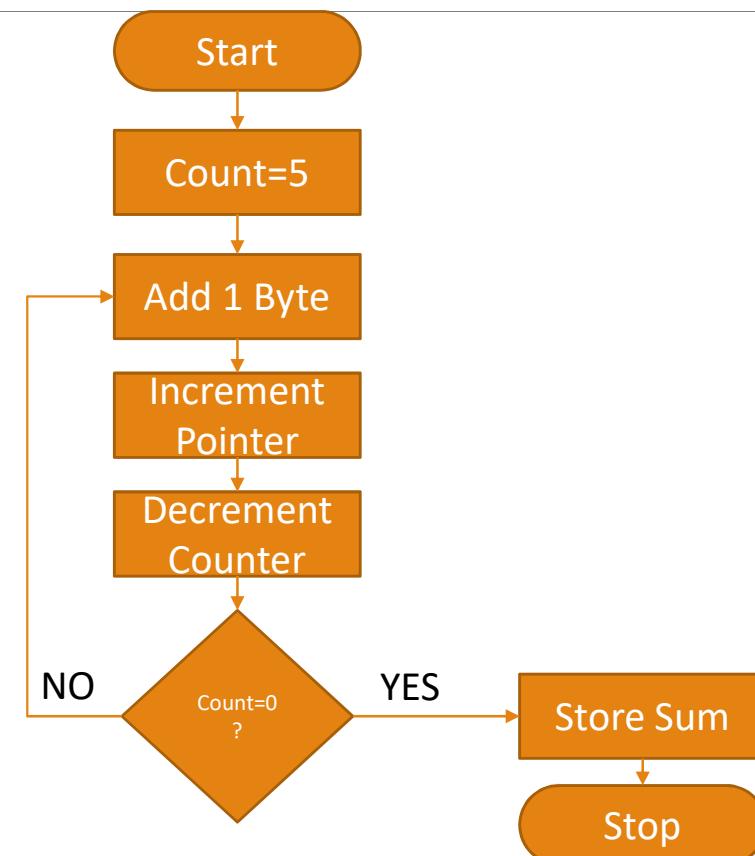
Add next byte

Increment Pointer

Decrement Count

UNTIL Count =0

STORE SUM



Module 2 (10 Hours)

x86: Instructions sets description, Arithmetic and logic instructions and programs: Unsigned Addition and Subtraction, Unsigned Multiplication and Division, Logic Instructions, BCD and ASCII conversion, Rotate Instructions.

INT 21H and INT 10H Programming : Bios INT 10H Programming , DOS Interrupt 21H.

Interrupts in x86 PC: 8088/86 Interrupts, x86 PC and Interrupt Assignment.

Text book 1: Ch 3: 3.1 to 3.5, Ch 4: 4.1 & 4.2 Ch 14: 14.1 & 14.2

Instruction Set

Arithmetic Instructions	Logical Instructions	Shift & Rotate Instructions	BCD & ASCII Instructions	STRING Instructions	STACK Instructions
■ ADD	■ AND	■ SHR	■ DAA	■ MOVS(B/W)	■ PUSH
■ INC	■ OR	■ SHL	■ DAS	■ SCAS(B/W)	■ POP
■ ADC	■ XOR	■ SAR	■ AAA	■ CMPS(B/W)	■ PUSHA
■ SUB	■ NOT	■ SAL	■ AAS	■ STOS	■ POPA
■ DEC	■ NEG	■ ROR	■ AAM	■ LOADS	■ PUSHF
■ SBB	■ TEST	■ ROL	■ AAD	■ REP(Z/NZ/E/NE)	■ POPF
■ MUL		■ RCL		■ CLD	Data Transfer Instructions
■ IMUL		■ RCR			■ MOV
■ DIV					■ CMOV
■ IDIV					■ MOVSX
					■ MOVZX

Instruction Set

Transfer Control Instructions	Transfer Control Instructions	Other Instructions	Other Instructions	I/O Instructions
■ JMP	■ JG/JNLE ■ JGE/JNL ■ JL/JNGE ■ JLE/JNG ■ JNC	■ XLAT ■ BSF ■ BSR ■ CMP ■ CMPXCHG	■ CMC ■ HLT ■ INT ■ LAHF ■ SAHF	■ IN
■ CALL	■ JNE/JNZ ■ JNO	■ BT ■ BTC	■ LDS	■ OUT
■ LOOP	■ JNP/JPO ■ JP/JPE	■ BTR ■ BTS	■ LEA ■ LES	■ INS(B/W)
■ LOOPE/LOOPZ	■ JNS	■ CBW/CWD	■ LOCK	■ OUTS(B/W)
■ LOOPNE/LOOPNZ	■ JO	■ CLC/CLD/CLI		
■ JA/JNBE	■ JS	■ STC/STD/STI		
■ JAE/JNB	■ RET	■ NOP		
■ JB/JNAE	■ RETF	■ XADD		
■ JBE/JNA	■ IRET			
■ JC				
■ JCXZ				
■ JE/JZ				

Arithmetic Instructions

- ADD
- INC
- ADC
- SUB
- DEC
- SBB
- MUL
- IMUL
- DIV
- IDIV

ADD/ADC – Unsigned Addition

Data operand can be between 00H-FFH for 8-Bit (0-255)

Data operand can be between 0000H-FFFFH for 16-Bit (0-65535)

Syntax: ADD destination,source; dst=dst+src

ADC is used for multiword Addition & will take care of carry from the lower operand

Source operand can be- Register/in Memory/Immediate.

Destination can be Register/in Memory

The only types of addition *not* allowed are memory-to-memory and segment register.

- segment registers can only be moved, pushed or popped

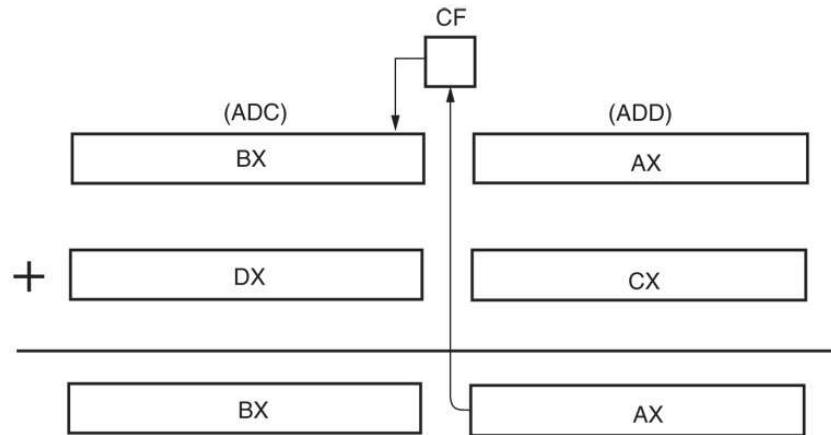
Increment instruction (INC) is a special type of addition that adds 1 to a number.

The assembler program cannot determine if the INC [DI] instruction is a byte-, word-, or doubleword-sized increment.

The size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives.

The instruction could affect ZF/SF/AF/CF/PF

ADD/ADC



Example- ADD

Example 3-1

Show how the flag register is affected by

```
MOV AL, 0F5H  
ADD AL, 0BH
```

Solution:

$$\begin{array}{r} \text{F5H} & 1111\ 0101 \\ + \underline{\text{OBH}} & + \underline{0000\ 1011} \\ \text{100H} & 0000\ 0000 \end{array}$$

After the addition, the AL register (destination) contains 00 and the flags are as follows:

CF = 1, since there is a carry out from D7

SF = 0, the status of D7 of the result

PF = 1, the number of 1s is zero (zero is an even number)

AF = 1, there is a carry from D3 to D4

ZF = 1, the result of the action is zero (for the 8 bits)

XADD

Exchange and add (XADD) appears in 80486 and continues through the Core2.

XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition.

- after the addition takes place, the original value of the destination is copied into the source operand

One of the few instructions that change the source.

Case1: Add Individual Byte Data/Word Data

Case 2: Add Multiword Numbers

SUB/SBB – Unsigned Subtraction

Data operand can be between 00H-FFH for 8-Bit (0-255)

Data operand can be between 0000H-FFFFH for 16-Bit (0-65535)

Syntax: SUB destination,source ; dst-dst-src

A second form of subtraction, is called **subtract with borrow i.e** the SBB instruction.

SBB is used for multiword subtraction & will take care of borrow of the lower operand

Source operand can be- Register/in Memory/Immediate.

Destination can be Register/in Memory

The only types of subtraction *not* allowed are memory-to-memory and segment register.

- segment registers can only be moved, pushed or popped

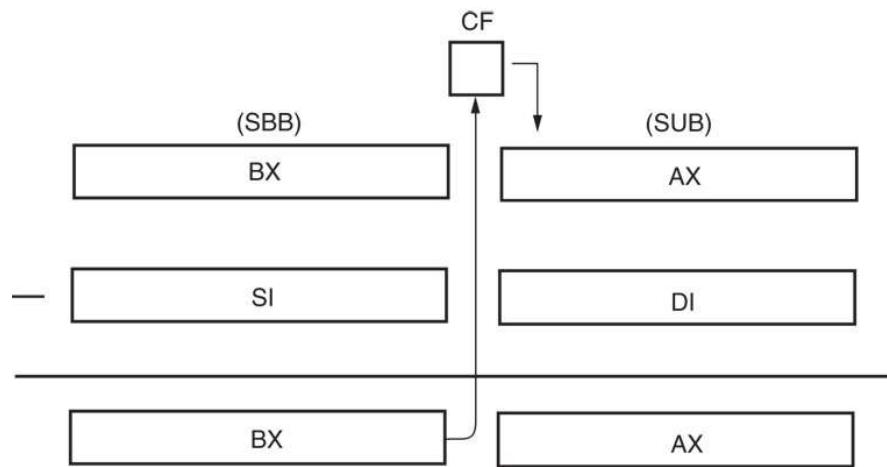
Decrement instruction (DEC) is a special type of subtraction that subtracts 1 from a number.

The assembler program cannot determine if the DEC[DI] instruction is a byte-, word-, or doubleword-sized decrement.

The size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives.

The instruction could affect ZF/SF/AF/CF/PF

SUB/SBB



CMP- Compare

The comparison instruction (CMP) is a subtraction that changes only the flag bits.

- destination operand never changes

Useful for checking the contents of a register or a memory location against another value.

A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

CMPXCHG- Compare & Exchange

Compare and exchange instruction (CMPXCHG) compares the destination operand with the accumulator.

- found only in 80486 - Core2 instruction sets

If they are equal, the source operand is copied to the destination; if not equal, the destination operand is copied into the accumulator.

- instruction functions with 8-, 16-, or 32-bit data

MUL- Unsigned Multiplication

In multiplying two numbers in the x86 processor, use of registers AX, AL, AH, and DX is necessary.

–The function assumes the use of those registers.

Three multiplication cases:

–byte times byte; word times word; byte times word.

Syntax: MUL src; AL=AL x src / AX=AL x src / AX= AX x src / DX-AX= AX x src

8 Bit Unsigned Multiplicand is stored in AL. 8 Bit Multiplier can be any 8 Bit Register or Memory Location. (Immediate data not allowed) 16 Bit Product is stored in AX.

16 Bit Unsigned Multiplicand is stored in AX. 16 Bit Multiplier can be any 16 Bit Register or Memory Location. (Immediate data not allowed) 32 Bit Product is stored in DX-AX.

Multiplication- Unsigned

Table 3-1: Unsigned Multiplication Summary

Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX
word × word	AX	register or memory	DX AX
word × byte	AL = byte, AH = 0	register or memory	DX AX

DIV – Unsigned Division

In dividing two numbers in the x86 processor, use of registers AX, AL, AH, and DX is necessary.

–The function assumes the use of those registers.

Three division cases:

–byte over byte; word over word; word over byte; doubleword over word.

Syntax: DIV src; AH=00, AL=Dividend, AL=AL/src and AH=AL%src (8-Bit Division)

Syntax: DIV src; DX=00, AX=Dividend, AX=AX/src and DX=AX%src (16-Bit Division)

Dividend is stored in AX. Divisor can be any 8-Bit Register or Memory Location. Quotient is stored in AL, Remainder is stored in AH. Quotient can be + or -, Remainder assumes Dividends sign. IN 8 Bit Division, the 8 Bit Dividend in AL must be converted to 16 Bit Number in AX using CBW (Sign Extend therefore it is for unsigned division).

Dividend is stored in DX-AX. Divisor can be any 16-Bit Register or Memory Location. Quotient is stored in AX, Remainder is stored in DX. Quotient can be + or -, Remainder assumes Dividends sign. IN 16 Bit Division, the 16 Bit

Division - Unsigned

Table 3-2: Unsigned Division Summary

Division	Numerator	Denominator	Quotient	Rem.
byte/byte	AL = byte, AH = 0	register or memory	AL ¹	AH
word/word	AX = word, DX = 0	register or memory	AX ²	DX
word/byte	AX = word	register or memory	AL ¹	AH
doubleword/word	DXAX = doubleword	register or memory	AX ²	DX

Notes: 1. Divide error interrupt if AL > FFH. 2. Divide error interrupt if AX > FFFFH.

A division can result in two types of errors:

- attempt to divide by zero
- other is a divide overflow, which occurs when a small number divides into a large number

In either case, the microprocessor generates an interrupt if a divide error occurs.

In most systems, a divide error interrupt displays an error message on the video screen.

Logic Instructions

- AND
- OR
- XOR
- NOT
- NEG
- TEST

AND

Performs Logical AND on the operands & places the result in the destination.

Source Operand can be a Register, in Memory or Immediate

Destination Operand can be a Register or In Memory

Syntax: AND destination,source

AND clears the CF & OF and PF,ZF and SF are set according to result. Rest are undecided/unaffected

AND Can be used to Test a zero operand

x x x x	x x x x	Unknown number
• 0 0 0 0	1 1 1 1	Mask
<hr/>		Result

A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

(a)



(b)

OR

Performs Logical OR on the operands & places the result in the destination.

Source Operand can be a Register, in Memory or Immediate

Destination Operand can be a Register or In Memory

Syntax: OR destination,source

Clears the CF & OF and PF,ZF and SF are set according to result. Rest are undecided/unaffected

Can be used to Test a zero operand

x x x x	x x x x	Unknown number
+	0 0 0 0	1 1 1 1
Result		
x x x x	1 1 1 1	Result

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

XOR

Performs exclusive-OR operation on the operands & places the result in the destination.

Source Operand can be a Register, in Memory or Immediate

Destination Operand can be a Register or In Memory

Syntax: XOR destination,source

Clears the CF & OF and PF,ZF and SF are set according to result. Rest are undecided/unaffected

Can be used to check if two register have the same value.

Also used to toggle the bits of an operand

x x x x x x x x Unknown number

$\oplus 0000\ 1111$ Mask

$x x x x \bar{x} \bar{x} \bar{x} \bar{x}$ Result

A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

NOT

Performs 1s Complement of the Destination Operand which can be Register or in Memory

Syntax: NOT destination

NEG

Performs 2s Complement of the Destination Operand which can be Register or in Memory

Syntax: *NEG destination*

TEST

Performs Logical AND operation on the contents of the destination Register and only updates the flag.

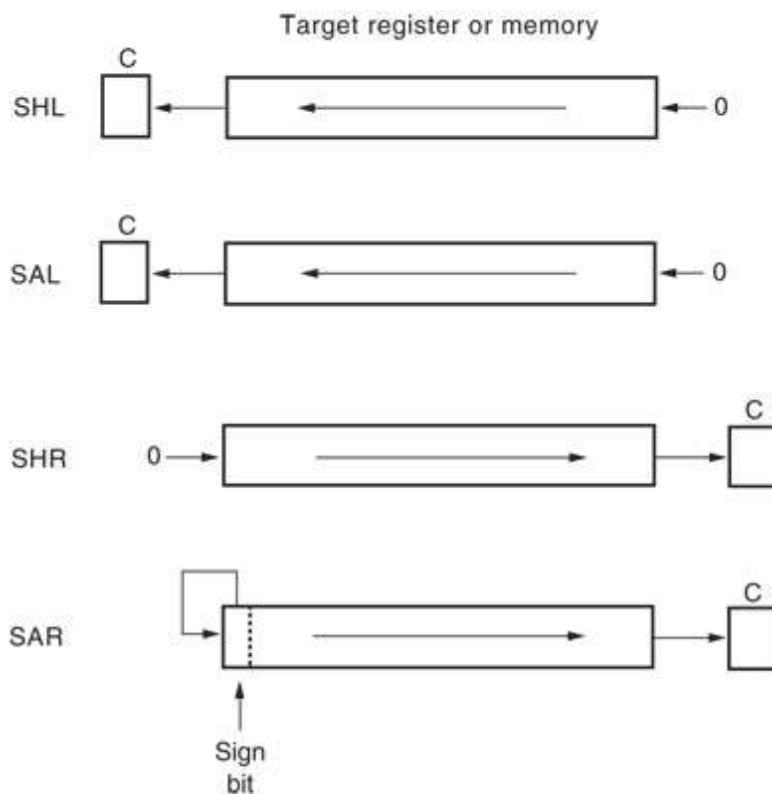
DOES NOT UPDATE DESTINATION REGISTER.

Syntax: TEST source,destination

Shift & Rotate

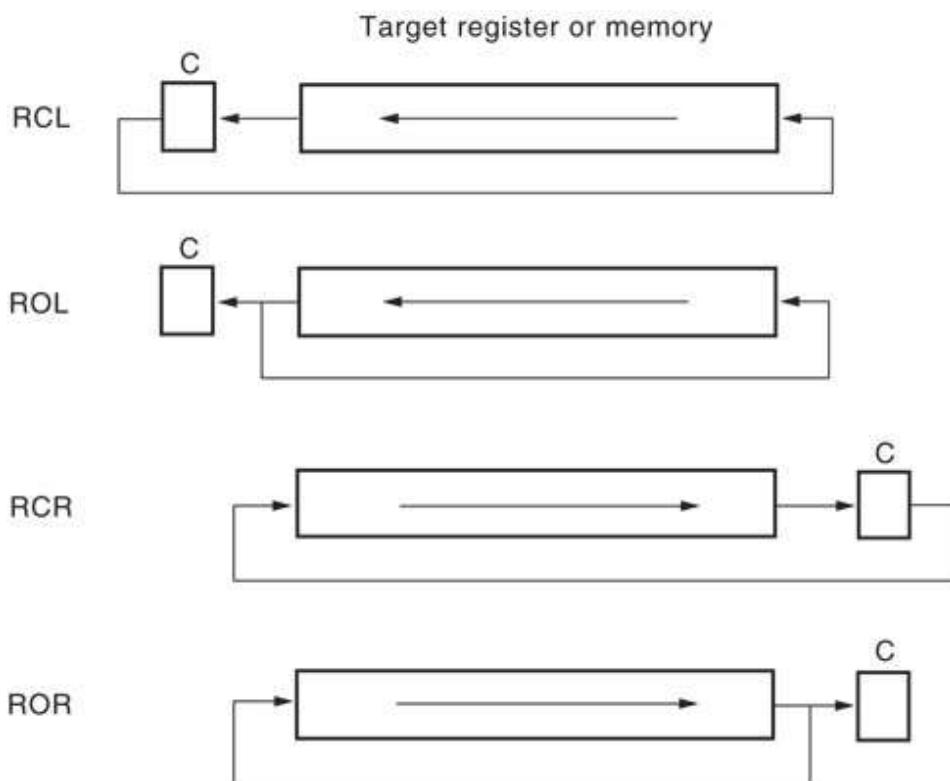
- SHR
- SHL
- SAR
- SAL
- ROR
- ROL
- RCL
- RCR

Shift



- logical shifts move 0 in the rightmost bit for a logical left shift;
- 0 to the leftmost bit position for a logical right shift
- arithmetic right shift copies the sign-bit through the number
- logical right shift copies a 0 through the number.

Rotate



A rotate count can be immediate or located in register CL.

if CL is used for a rotate count, it does not change

Rotate instructions are often used to shift wide numbers to the left or right.

BCD & ASCII Conversion

- DAA
- DAS
- AAA
- AAS
- AAM
- AAD

BCD

BCD- Binary Coded Decimal (0-9)	Binary
0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	0101
0110	0110
0111	0111
1000	1000
1001	1001
	1010
	1011
	:

Unpacked BCD

0000 0001

Lower 4 Bits Represent the BCD Number.

0000 0010

Rest of the bits are 0

0000 0011

1 Byte required to store BCD Number

0000 0100

0000 0101

0000 0110

0000 0111

0000 1000

0000 1001

Packed BCD

0000 0001

Single Byte has 2 BCD numbers in it.

0000 0010

One in Lower 4 Bits One in Upper 4 Bits

0000 0011

0000 0100

0000 0101

0000 0110

0000 0111

0000 1000

0000 1001

ASCII

Key	ASCII (Hex)	Binary	BCD	Key	ASCII (HEX)	Binary
0	30H	011 0000	0000 0000	A	41H	100 0001
1	31H	011 0001	0000 0001	B	42H	100 0010
2	32H	011 0010	0000 0010	C	43H	100 0011
3	33H	011 0011	0000 0011	D	44H	100 0100
4	34H	011 0100	0000 0100	E	45H	100 0101
5	35H	011 0101	0000 0101	F	46H	100 0110
6	36H	011 0110	0000 0110	:	:	:
7	37H	011 0111	0000 0111	:	:	:
8	38H	011 1000	0000 1000	Z	5AH	101 1010
9	39H	011 1001	0000 1001	a-z	61H-7AH	

ASCII to Unpacked BCD Conversion

Get Rid of the tagged 011 in the higher 4 bits of the ASCII

ASCII Number is ANDed with 0FH(i.e 0000 1111)

Eg: Store Numbers as String (i.e their ASCII value) in Memory. Unpack the numbers and store the Unpacked values in another location in Memory

ASCII to Packed BCD Conversion

First Convert ASCII to Unpacked BCD by getting rid of the 3 (011)in the Higher Byte.

Then Combine to make packed BCD.

Eg: When 95H is entered through the keyboard, 39H & 35H. Goal is to Produce 95H.

Remove the 3 from the Higher byte from 39H & 35H by ANDing with 0FH. 39H will become 09H and 35H will become 05H. Left Shift 09H by 4 Bits. It will now become 90H. This is ANDed with 05H to obtain 95H.

Packed BCD to ASCII Conversion

First Convert Packed BCD to Unpacked BCD.

The Unpacked BCD is then Tagged with 30H.

Eg: Store a Packed BCD number (29H) in Memory.

Load it into AL. Store a Copy of it in AH. Now AH & AL both have 29H.

AND AX with F00FH. This will remove the 9 from 20H in AH and will remove the 2 from 29H in AL. AH will hold 20H and AL will hold 09H

SHR AH by 4 bits, AH will now hold 02H and AL will hold 09H.

OR AX with 3030H. AX will now hold the ASCII values of 2 & 9 i.e 32H & 39H. Store result in Memory.

BCD addition & Subtraction

After Adding Packed BCD numbers the result is no longer BCD.

Eg: $17H + 28H = 3FH$

But for BCD $17 + 28 = 45$.

$3F+06H =45H$.To correct the problem $06H$ must be added to the Result.

The same can happen in the upper digit as well. Eg: $52H+87H =D9H$

To solve this 6 must be added to the upper digit, i.e add $60H$ $D9H+60H =139H$

DAA- Decimal Adjust for Addition

Decimal Adjust AL After Addition.

After an Addition operation

If Lower Nibble of AL >9 or AF=1, then AL=AL+6; AF=1.

If AL>9Fh or CF=1 then AL=AL+60H; CF=1.

DAA will Add 6 to the lower nibble or higher nibble if needed. Otherwise it will leave the result alone.

DAA works only with AL. So the Destination of the ADD operation before DAA must be AL.

DAA must be used after addition of BCD numbers. (numbers being added cannot have digits greater than 9)

(i.e A-F cannot be used)

**DAA will not work after INC instruction.*

Use of AF will be seen only for BCD addition-Eg: 29+18

DAA

$29H + 18H = 41H$

But for BCD Addition $29+18 = 47$

i.e $29H + 18H = 41H + 06H$; $06H$ must be added though the result is not having digits greater than 9

Therefore AF flag will be checked and in this case AF will be set to 1.

Therefore DAA will add 6 to the result considering AF.

DAS- Decimal Adjust for Subtraction

Decimal Adjust AL After Subtraction.

After a Subtraction operation

If Lower Nibble of AL >9 or AF=1, then AL=AL-6; AF=1.

If AL>9Fh or CF=1 then AL=AL-60H; CF=1.

DAS will Subtract 6 from the lower nibble or higher nibble if needed. Otherwise it will leave the result alone.

DAS works only with AL. So the Destination of the SUB operation before DAS must be AL.

DAS must be used after subtraction of BCD numbers. (numbers being subtracted cannot have digits greater than 9) (i.e A-F cannot be used)

AAA

*ASCII Adjust AL After Addition. Follows ADD or ADC. Uses AL as Source & AX as Destination.
Unpacks the 8 Bit number stored in AL .*

*Adjusts the Binary Result of ADD or ADC assuming AL contained Binary result of adding 2 ASCII
Digits*

If lower nibble of AL>9 or AF=1 then: AL=AL+6;AH=AH+1;AF=1;CF=1

else AF=0; CF=0

Higher Nibble of AL is cleared in both cases.

Note: CLEAR AH BEFORE USE OF AAA

MOV AH,00

MOV AL,'8' ; Store ASCII 8 i.e 38H in AL

ADD AL,'2' ; ADD ASCII 2 i.e 32H to 38H giving 6AH

AAA ; Gives AX=0100H AH=01H AL=00H

OR AX,3030H ; to convert result to ASCII values

AAS

ASCII Adjust AL After Subtraction.

Converts the result of Subtraction of 2 unpacked BCD into a single BCD.

AL is the implicit operand.

The 2 operands of subtraction must have its lower 4 bits in the range from 0-9.

If lower nibble of AL>9 or AF=1 then: AL=AL-6;AH=AH-1;AF=1;CF=1 else AF=0; CF=0

Higher Nibble of AL is cleared in both cases.

Eg:

MOV AH,00

MOV AL,'8' ; Store ASCII 8 i.e 38H in AL

SUB AL,'2' ; ADD ASCII 2 i.e 32H to 38H giving 6AH

AAS ; Gives AX=0100H AH=01H AL=00H

OR AX,3030H ; to convert result to ASCII values

AAM – BINARY to Unpacked BCD Conversion

ASCII Adjust AX After Multiplication.

Follows Multiply.

*Converts **Binary** to **Unpacked BCD**.*

Converts the product of 2 valid unpacked BCD into a valid unpacked BCD. AX is the implicit operand.

AH=AL/10 AL= Remainder

If a Binary Number between 0000H & 0063H i.e 00 & 99 Decimal is present in AX, then AAM converts it into Unpacked BCD.

Eg: if AX=0019H, AAM converts it to 0205H

AAD- Unpacked BCD to Binary

ASCII Adjust AX Before Division.

*Converts **unpacked BCD** between 00 & 99 in AH & AL into **Binary Number** in AX to prepare it for DIV operation.*

AL=(AH*10)+AL or AL=(AH*0AH)+AL

Eg: If AX=0205H, after AAD, 02x0AH =14H+5H = 19H => 25D

Eg: 67/9 = 7

AX=0607H CH=09H; After AAD AX=43H = 67D. After DIV CH, AL=7, AH=4

BIOS INT 10H Programming

BIOS-ROM based Interrupt

INT 10H subroutines are burned into the ROM BIOS of the x86 based IBM PC

Used for communication with computers screen video

Used for manipulation of Screen Text or Graphics

Functions: Eg:

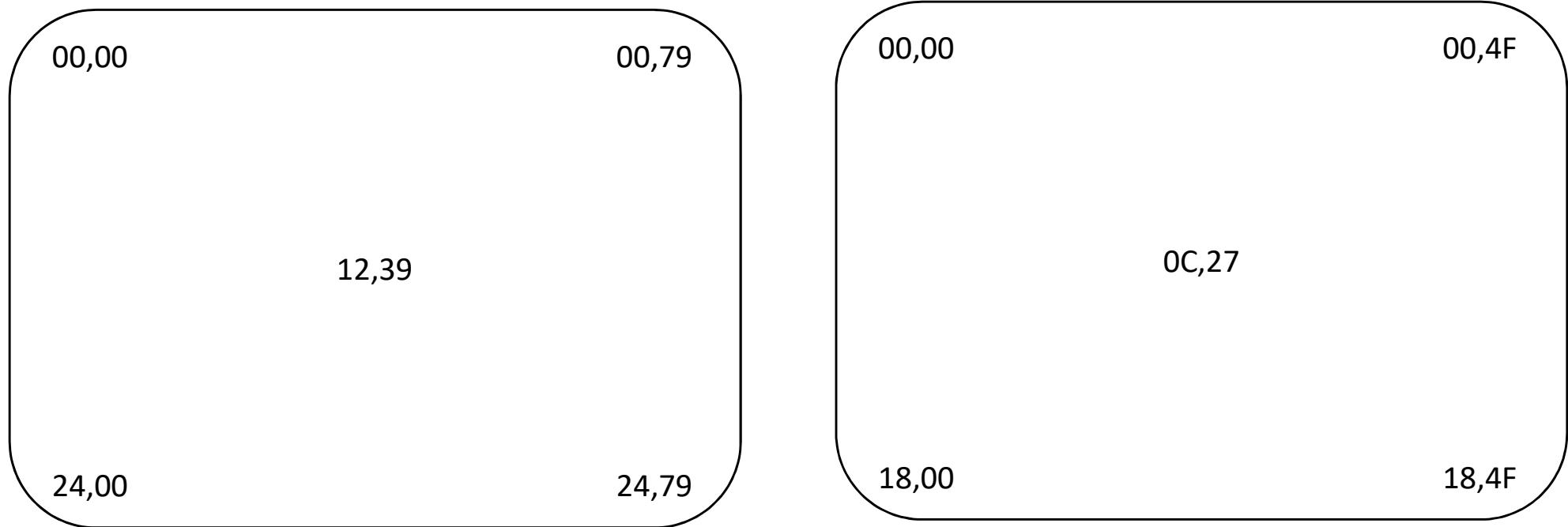
Change the color of characters, change background color, clear the screen, change location of cursor

Functions are chosen by loading specific values in Register AH

Monitor Screen in Text Mode

80 Columns (0-79) (00H-4FH)

25 Rows (0-24) (00H-18H)



Clear Screen using INT 10H function 06H

AH=06H ; SCROLL FUNCTION

AL=00H ;ENTIRE PAGE

BH=07H ;NORMAL ATTRIBUTE

CH=00H ; START ROW VALUE

CL=00H ; START COLUMN VALUE

DH=24 ;END ROW VALUE

DL=79 ;ENDF COLUMN VALUE

After setting these values call INT 10H

This Scrolls the entire Page Up.

Set cursor to specific location using INT 10H Function 02H

AH=02H ; SET CURSOR FUNCTION

BH=00H ;SET PAGE 0.VIDEO RAM CAN HAVE MULTIPLE PAGES OF TEXT,BUT ONLY 1 CAN BE VIEWED AT A TIME

DL=25 ; COLUMN VALUE

DH=15 ; ROW VALUE

Then call INT 10H

Get current cursor position using INT 10H Function 03H

AH=03H

BH=00; SET PAGE 0

Call INT 10H

After execution of INT,

DH= CURRENT ROW POSITION

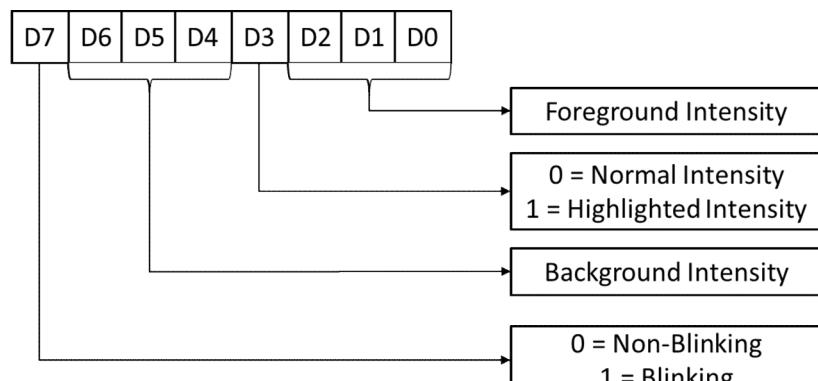
DL=CURRENT COLUMN POSITION

Changing Video Mode

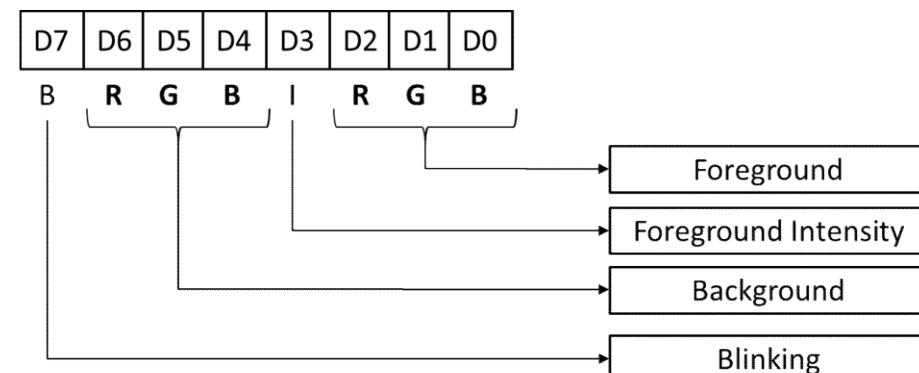
AH=00H

AL= VIDEO MODE

INT 10H is called



Attribute Byte for Monochrome Monitors



CGA Text Mode Attribute Byte

Monochrome Monitors

00 – White Text on White BG

07 – White Text on Black BG

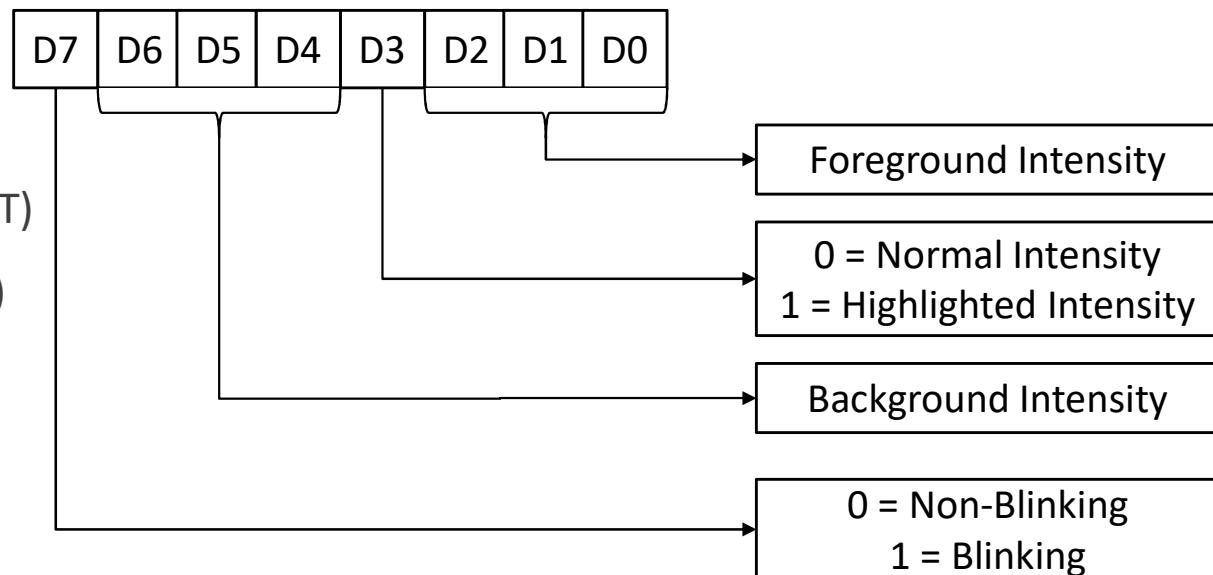
0F – White Text on Black BG (HIGHLIGHT)

87 – White Text on Black BG (BLINKING)

77 – Black Text on Black BG

70 – Black Text on White BG

F0 – Black Text on White BG (BLINK)



Attribute Byte for Monochrome Monitors

CGA-Text Mode

Color Graphics Adapter

BG can take 8 different colors

BINARY HEX Color Effect

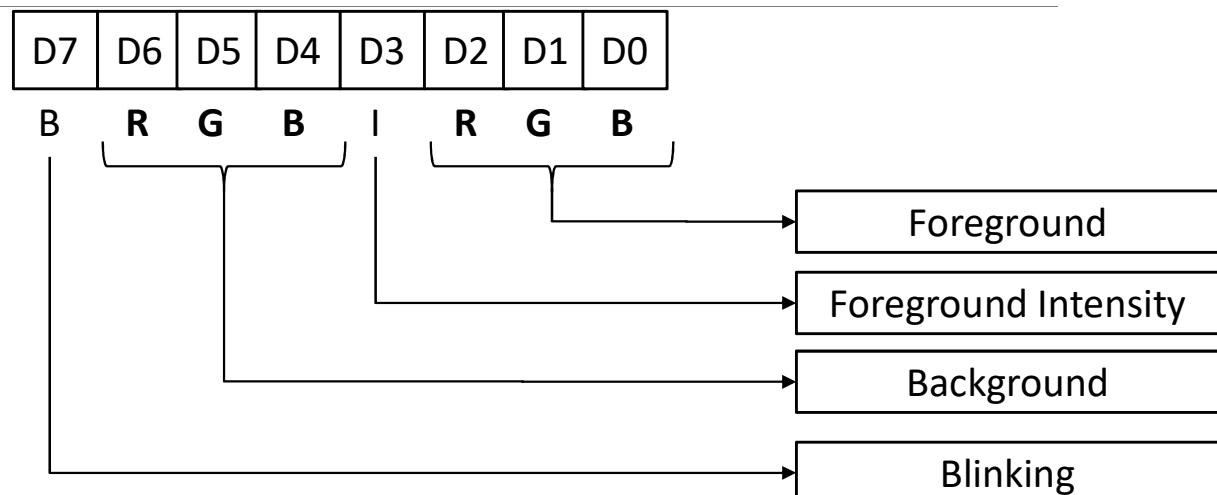
0000 0000 00 Black on Black

0000 0001 01 Blue on Black

0001 0100 14 Green on Blue

0001 1111 1F High Intensity White on Blue

IRGB	Color	IRGB	Color	IRGB	Color
0000	black	0110	brown	1100	light red
0001	blue	0111	white	1101	light magenta
0010	green	1000	gray	1110	yellow
0011	cyan	1001	light blue	1111	High intensity white
0100	red	1010	light green		
0101	magenta	1011	light cyan		



CGA Text Mode Attribute Byte

CGA – Text Mode

Screen is viewed as a matrix of rows & columns of characters.

1 Character requires 1 Byte + Each character has 1 attribute byte

IN 1 Frame there are 80x25 characters = 2000 i.e 2K Bytes of memory required for characters

Therefore 2K bytes required for each attribute.

Total of 4K bytes required for each screen frame.

CGA has 16K bytes of video memory

Thus CGA supports 4 pages of data where each page represents 1 full screen.

16 colors are supported.

AL=03H

AH=00H

INT 10H

CGA Graphics Mode

Screen is viewed as matrix of horizontal & vertical pixels.

The number of pixels depends on monitor resolution & video board.

- Location Of the Pixel
 - Pixel Attributes
- } Stored in the video RAM

Higher the Number of Pixels & Colors, larger is the memory needed.

CGA Graphics Mode

CGA mode can have maximum of 16K bytes of Video Memory. (16K x 8 Bits)

Graphics Mode of 320x200 (medium res)

320 Columns x 200 Rows = 64000 Pixels

Total Video Memory is 16K Bytes or 128 K Bits

128K Bits/64000 pixels gives 2 bits for the color of each pixel. (thus 4 colors)

Thus 320x200 resolution CGA cannot support more than 4 Colors

Select AL=04 to Choose this Mode

Graphics Mode of 640x200 (High res)

640 Columns x 200 Rows = 128000 Pixels

Total Video Memory is 16K Bytes or 128 K Bits

128K Bits/128000 pixels gives 1 bit for the color of each pixel. (thus 2 colors)

Thus White(Bit=1) or Black (Bit=0)

Thus 320x200 resolution CGA cannot support more than 4 Colors

Select AL=06 to Choose this Mode

INT 10H Pixel Programming

AH=0CH

CX= Column Point(X Co-ordinate)

DX= Row Point(Y Co-ordinate)

If display supports more than 1 Page BH= Page Number otherwise it is ignored.

AL=1 (Pixel ON) WHITE

AL=0 (Pixel OFF) BLACK

Examples

1. Write an ALP to clear the screen.
2. Write the ALP to set the cursor position to row=15(0FH) & Column = 25(19H)
3. Write an ALP to clear the screen & Set Cursor to center of the screen.
4. Write an ALP using INT 10H to (a) Change the Video Mode (b) Display the letter D in 200H locations with attributes Black on White Blinking.
5. Write an ALP to (a) Clear the Screen, (b)Set the mode to CGA of 640x200 Res, (c)Draw a horizontal line starting at column=100, row=50 & end at column=200 ,row=50

DOS INT 21H

DOS Based Interrupt.

Invoked to perform some useful functions such as:

Input a character from keyboard(with/without echo)

Output a character

Input a String of data from keyboard

Output a String

Get system Time

Get System Date

Exit a Program

INT 21H Option 01H

Input a single character with echo.

Function waits until a character is input from the keyboard, then echoes it to the monitor.

After the interrupt, the input character's ASCII value will be in AL.

Usage:

```
MOV AH,01H
```

```
INT 21H
```

AL now holds the ASCII value of Key pressed.

INT 21H Option 02H

Output a single character.

Function outputs a single character to the monitor.

The ASCII value of the character to be displayed is loaded into DL Register.

AH is then loaded with 02H and the Interrupt INT 21H is invoked to display the character

Usage:

```
MOV DL,'A'
```

```
MOV AH,02H
```

```
INT 21H
```

A is now displayed on the Screen. This can also be used to Display "\$"(more on this later)

INT 21H Option 07H

Input a single character without echo.

Function waits until a character is input from the keyboard, but does not echo it to the monitor as with function 01H

After the interrupt, the input character's ASCII value will be in AL.

Usage:

```
MOV AH,07H
```

```
INT 21H
```

AL now holds the ASCII value of Key pressed.

INT 21H Option 09H

Output a String of data to the monitor.

AH must be set with 09H and DX must be loaded with the offset address of the ASCII DATA to be displayed. INT 21H is then invoked.

The ASCII Data pointed at by DX will be displayed until it encounters a “\$” sign.

Usage:

```
MSG DB 'HELLO$'
```

```
:
```

```
LEA DX,MSG
```

```
MOV AH,09H
```

```
INT 21H
```

INT 21H Option 0AH

Input a String of data from the Keyboard & store it in a pre-defined area in memory in DATA SEGMENT. (WITH ECHO)

AH must be set with 0AH and DX must be loaded with the offset address from which the string of data is stored(BUFFER AREA) in the DATA SEGMENT. INT 21H is then invoked.

First byte- BUFFER SIZE

Second Byte-Number of Characters Entered through the keyboard

Keyed in Data is placed from the Third Byte onwards

Inputting data more than the Buffer Size will cause the computer to sound the speaker.

The count of keyed-in data does not include Carriage Return(0DH). 0DH will be in the buffer, but will not be counted.

Usage:

INT 21H Option 0AH

ORG 0010H

DATA DB 6,?,6 DUP(FF); 0010H =06H(BUFFER SIZE) 0011H=TO HOLD COUNT 0012H to 0017H=FFH

MOV AH,0AH

MOV DX,OFFSET DATA

INT 21H

0010 0011 0012 0013 0014 0015 0016 0017

06 00 FF FF FF FF FF FF

"USA"<RETURN>

0010 0011 0012 0013 0014 0015 0016 0017

06 03 55 53 41 0D FF FF

Note:

Carriage Return (10 or 0AH)

Line Feed(13 or 0DH)

Only CR brings cursor to beginning of the Line. May cause overwriting of the line if new data is printed.

Line Feed after CR will bring the cursor to the Next Line.

LABEL Directive to define String Buffer

JOE LABEL BYTE

TOM DB 20 DUP (0)

Example

Write an ALP to

1. Clear the screen
2. Set the cursor at the beginning of the third line from the top of the screen
3. Accept the message “hello WORLD” from the keyboard.
4. Convert the lowercase letters of the message into uppercase
5. Display Converted Results on the next line.

Example

Write an ALP to

1. Clear the screen
2. Set the cursor at row5 , Column 1 of the screen
3. Prompt “THERE IS A MESSAGE FOR YOU. TO READ IT ENTER Y”.
4. If the user Enters Y or y the n the message “HI! HAVE A GREAT DAY” will appear on the screen
5. If the user enters any other key, then prompt “NO MORE MESSAGES FOR YOU” should appear on the next line.

8086/88 INterrupts

Interrupt- An external event that informs the CPU that a device needs it's service

In 8086/88 there total of 256 Interrupts. (INT 00, INT 01,.....INT FFH)

When an Interrupt is executed, the microprocessor automatically saves the FLAG Register (FR), The Instruction Pointer (IP) and the Code Segment Register (CS) on the STACK & goes to a fixed memory location.

In x86 PC the memory location to which an interrupt goes to is always 4 times the value of the Interrupt Number.

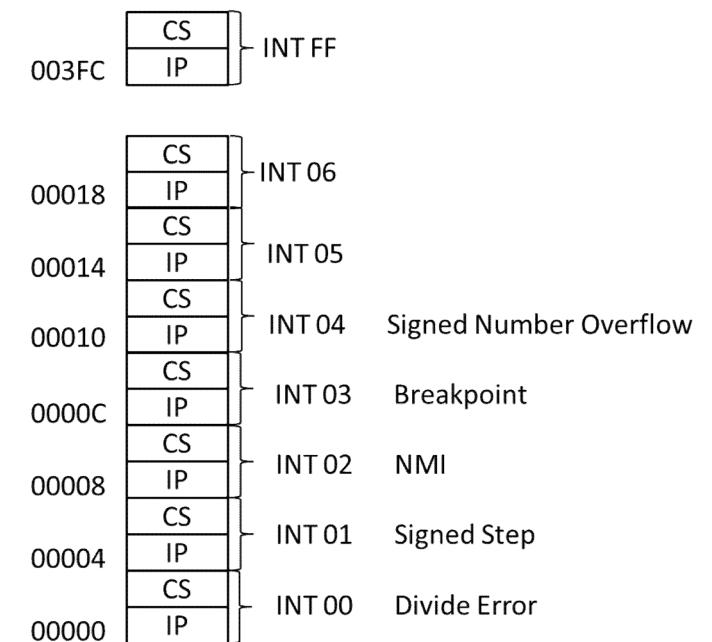
Eg: INT 03H will go to the address 0000CH ($4 \times 3 = 12 = 0CH$)

INT nn – 2 Byte instruction- 1st Byte is Opcode, 2nd Byte is the Interrupt Number.

256 Interrupts.

8086/88 Interrupts - IVT

INT Number	Physical Address	Logical Address
INT 00	00000	0000-0000
INT 01	00004	0000-0004
INT 02	00008	0000-0008
INT 03	0000C	0000-000C
INT 04	00010	0000-0010
INT 05	00014	0000-0014
:	:	:
INTFF	003FC	0000-03FC



ISR-Interrupt Service Routine

For Every Interrupt there is a program associated with it.

When an Interrupt is invoked, it is asked to run a program to perform certain service

This is called Interrupt Service Routine.

Also called Interrupt Handler.

For every interrupt there are 4 Bytes of Memory Allocated in the Interrupt Vector Table(IVT)

Two Bytes for the Lower 2 Bytes for IP and the Higher 2 Bytes for CS.

These 4 Bytes provide the Address of the ISR

There are 256 Interrupt and 4 Bytes for Each ISR Address. Therefore the lowest 1024 ($256 \times 4 = 1024$) bytes of memory space are set aside for IVT and must not be used for any other function as defined by Intel.

Example

Find the Physical & Logical Addresses in the IVT associated with

1. INT 12H
2. INT 08H

Physical Address for INT 12H are 00048H-0004BH since $4 \times 12H = 48H$

i.e 48H,49H,4AH and 4BH are set aside for CS & IP of the ISR belonging to INT 12H.

Logical Address is 0000:0048H – 0000:004BH

Physical Address for INT 08H are 00020H-00023H since $4 \times 8 = 32 \Rightarrow 20H$

i.e 20H,21H,22H and 23H are set aside for CS & IP of the ISR belonging to INT 08H.

Logical Address is 0000:0020H – 0000:0023H

INT vs CALL

CALL FAR PROCEDURE

1. Can jump to any location within the 1MB address range of 8086/88 CPU.
2. Used by the programmer in the sequence of instructions in the program.
3. Cannot be masked(disabled)
4. Automatically saves CS:IP of the next instruction on the STACK.
5. RETF is the last instruction(POPs CS & IP from the STACK)

INT

1. Jumps to a fixed memory location in the IVT to get the address of the ISR.
2. Externally activated hardware interrupt can come in at any time requesting the attention of the CPU.
3. Externally Activated Hardware interrupts can be disabled (Masked)
4. Automatically saves CS:IP of the next instruction on the STACK along with FLAG Register.
5. IRET is the last instruction(POPs FLAG Register Contents along with CS & IP from the STACK)

Categories of Interrupts – Hardware Interrupts

INTR(Interrupt Request) – It is an Input Signal to the CPU to request the service of the CPU. It may be ignored(masked). Masking and Unmasking is done through the use of CLI & STI.

INTA(Interrupt Acknowledge) – Used to send an acknowledgement to the Device Interrupting the CPU.

NMI(Non-Maskable Interrupt) - It is an Input Signal to the CPU to request the service of the CPU. But It cannot be Masked(ignore). INT 02H

INTR & NMI are activated by applying 5V on the corresponding pins.

When either of these is activated, x86 finishes the currently executing instruction and then pushes FR, CS & IP onto the STACK, Then Jumps to a fixed location in the IVT & Fetches the CS:IP value for the ISR from the IVT. At the end of the ISR execution, the IRET instruction will cause the CPU to POP the CS:IP from the stack causing the CPU to continue executing the instruction where it left off when the Interrupt occurred.

Categories of Interrupts – Software Interrupts

If the ISR is invoked as a result of the execution of an instruction such as “INT nn” then it is referred to as SOFTWARE INTERRUPT as it was invoked from software and not from external hardware.

Eg: INT 21H, INT 10H, INT 00, INT 01, INT 04.

INT 00 to INT 04 have pre defined functions.

INT 05 to INT FF can be used to implement either software or Hardware Interrupts.

Interrupts & FLAG Register

IF,TF,OF

IF =0 (by using CLI)All H/W interrupts through INTR are Ignored. (no effect on NMI).

IF =1 (by using STI)Allow H/W interrupt request through INTR.

TF =1 for Single Stepping /Trace.

Processing Interrupts

FLAG Register is PUSHed onto the STACK. SP is decremented by 2(FR is a 2 Byte Register)

IF & TF are cleared (Ignores other interrupts when current interrupt is being serviced & avoid single stepping during ISR execution) Depending on the nature of ISR, the programmer can unmask the INTR using STI.

Current CS is PUSHed onto the STACK. SP is decremented by 2.

Current IP is PUSHed onto the Stack. SP is decremented by 2.

The INT number is multiplied by 4 to get Physical Address of the location within the IVT to fetch the CS & IP of the ISR.

From the new CS:IP, the CPU starts to fetch & execute the instructions of the ISR.

Last instruction of the ISR is IRET. On execution of IRET, IP,CS and FR are POPped from the STACK to resume execution where the CPU left off before the Interrupt occurred.

INT 00(Divide Error)

Conditional or Exception Interrupt

Invoked by the Microprocessor when there are exceptions that the CPU cannot handle.

Attempt to Divide a number by 0.

ISR is responsible for displaying the message “DIVIDE ERROR” on the screen.

Also invoked if the quotient is too large to fit the assigned register when executing DIV.

INT 01(Single Step)

When executing a sequence of instructions, there maybe a need to examine the contents of the CPU Registers & System Memory.

Execute the program one instruction at a time & examine the Registers/Memory.

Single Stepping/ Trace

Set TF=1.

After every instruction, CPU automatically jumps to physical location 00004 to fetch 4 Bytes of CS:IP of the ISR whose one of the jobs is to dump the register contents onto the screen.

How to CLEAR TF

PUSHF

POP AX

AND AX,1111111011111111B

PUSH AX

POPF

INT 02(NMI)

Active High input pin.

Activated by a +5V (HIGH) Signal.

CPU Jumps to physical Memory 00008 to fetch CS:IP of the ISR associated with NMI

INT 03(Breakpoint)

Allow Implementation of Breakpoints

Allows to check the contents of Registers & Memory after executing a group of Instructions.

INT 03H is a 1 Byte Instruction. Unlike other “INT nn” instructions which are 2 Byte instructions

INT 04(Signed Number Overflow)

Invoked by a Signed Number Overflow condition.

There is an INTO instruction associated with this condition.

If INTO is placed after a signed number arithmetic/logical operation such as IMUL, or ADD, the CPU will activate INT 04 if OF=1. IF OF=0, then INTO is bypassed & acts as NOP.

If executed then CPU jumps to physical location 00010H of the IVT to get the CS:IP of the ISR

Module 3 (10 Hours)

Signed Numbers and Strings: Signed number Arithmetic Operations, String operations.

Memory and Memory interfacing: Memory address decoding, data integrity in RAM and ROM, 16-bit memory interfacing.

8255 I/O programming: I/O addresses Map of x86 PC's, programming and interfacing the 8255.

Text book 1: Ch 6: 6.1, 6.2. Ch 10: 10.2, 10.4, 10.5. Ch 11: 11.1 to 11.4

8-Bit Signed Numbers (Byte Sized)

Unsigned Numbers – Entire 8 Bit operand was used to represent magnitude

Signed Numbers – MSB is set aside for Sign(+ or -).

Sign Bit 0 Represents a Positive Number.

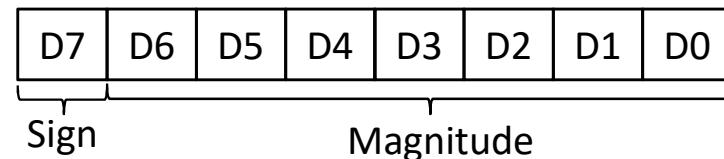
Sign Bit 1 Represents a Negative Number.

D7 (MSB) used to represent Sign.

D6-D0 used to represent Magnitude

If D7=0, The number is positive whose magnitude is represented by D6-D0

If D7=1, The number is negative whose magnitude(2's Complement) is represented by D6-D0



8-Bit Signed Numbers (Byte Sized)

Positive Range of 8 Bit Numbers is therefore (0 to +127) (i.e 00H to 7FH) (i.e 0000 0000 to 0111 1111)

Negative Range of 8 Bit Numbers is therefore (-128 to -1) (i.e 80H to FFH) (i.e 1000 0000 to 1111 1111)

Negative Number is in 2's Complement Representation.

- Write the magnitude of a number in 8-Bit Binary(no Sign).
- Invert Each Bit
- Add 1 to it

16-Bit Signed Numbers (Word Sized)

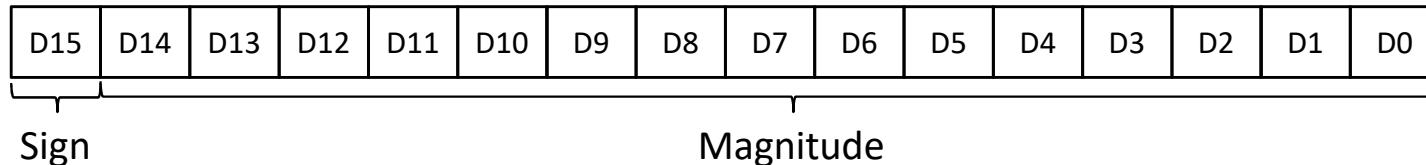
Unsigned Numbers – Entire 16Bit operand was used to represent magnitude

Signed Numbers – MSB is set aside for Sign(+ or -).

Sign Bit 0 Represents a Positive Number.

Sign Bit 1 Represents a Negative Number.

D15 (MSB) used to represent Sign.



D14-D0 used to represent Magnitude

If D15=0, The number is positive whose magnitude is represented by D14-D0

If D15=1, The number is negative whose magnitude (2's Complement) is represented by D14-D0

16-Bit Signed Numbers (Word Sized)

Positive Range of 16 Bit Numbers is therefore (0 to +32767) (i.e 0000H to 7FFFH)

(i.e 0000 0000 0000 0000 to 0111 1111 1111 1111)

Negative Range of 16 Bit Numbers is therefore (-32768 to -1) (i.e 8000H to FFFFH)

(i.e 1000 0000 0000 0000 to 1111 1111 1111 1111)

Negative Number is in 2's Complement Representation.

- Write the magnitude of a number in 8-Bit Binary(no Sign).
- Invert Each Bit
- Add 1 to it

Overflow Problem in Signed Number Operation

CPU understands only 0s & 1s.

CPU ignores the human convention of Positive & Negative Numbers.

CPU indicates the existence of this problem using OF

IT is upto the programmer to handle it.

If the result of an operation on signed numbers is too large for the register an overflow occurs & the programmer must be notified

DATA1 DB +96	AL=60H (0110 0000)	1010 0110	AL can handle results only till +127
DATA2 DB +70	BL=46H (0100 0110)	Ignore Sign and take 2's Complement of magnitude .	OF exists for this purpose, to inform the programmer that the result of a signed operation is erroneous because the result is going beyond the range of the Register.
.....	After Add		
MOV AL,DATA1	AL=A6H (1010 0110)	101 1010 = 90	
MOV BL,DATA2	i.e 166 which is greater than signed number range	Therefore signed result would be considered -90	
ADD AL,BL			

When is OF set in 8-bit Operations

OF is set in 8-bit Operations if either of the following 2 conditions occur

1. There is a Carry from D6 to D7 but no carry out of D7(i.e CF=0)
2. There is a Carry Out from D7(i.e CF=1) but no carry from D6 to D7

In other words OF is set if there ia Carry from D6 to D7 or there is a Carry Out from D7 but not both.

When is OF set in 16-bit Operations

OF is set in 8-bit Operations if either of the following 2 conditions occur

1. There is a Carry from D14 to D15 but no carry out of D15(i.e CF=0)
2. There is a Carry Out from D15(i.e CF=1) but no carry from D14 to D15

In other words OF is set if there ia Carry from D14 to D15 or there is a Carry Out from D15 but not both.

Avoiding Erroneous Results in Signed Number Operations

Use of OF flag, CBW,CWD

```
DATA1 DB +96  
DATA2 DB +70  
.....
```

```
SUB AH,AH  
MOV AL,DATA1  
MOV BL,DATA2  
ADD AL,BL  
JNO COMPLETE  
MOV AL,DATA2  
CBW  
MOV BX,AX  
MOV AL,DATA1  
CBW  
ADD AX,BX  
COMPLETE: MOV RESULT,AX
```

CBW- Copies D7 of AL to all bits of AH

CWD- Copies D15 of AX to all bits of DX

CBW & CWD perform Sign Extension of AL and AX into AH and DX Respectively

As a rule, if the possibility of overflow exists, all byte sized signed operands must be sign extended into a word and all word sized signed operands must be sign extended into a double word.

IDIV (Signed Number Division) – Integer Division

Functions same as Unsigned Division(DIV).

The Byte sized Signed Dividend in AL must first be sign extended into AH. Using CBW.

Then IDIV is performed.

Divide Error occurs if $-128 > AL > +127$ for Word/Byte IDIV

Quotient in AL, Remainder in AH

The Word sized Signed Dividend in AX must first be sign extended into DX. Using CWD.

Then IDIV is performed.

Divide Error occurs if $-32768 > AX > +32767$ for DoubleWord/Word IDIV

Quotient in AX, Remainder in DX

IMUL (Signed Number Multiplication) – Integer Multiplication

Signed number Multiplication(IMUL) is similar in operation to the Unsigned Multiplication (MUL)

Operands can be positive or negative

Result must therefore indicate the sign.

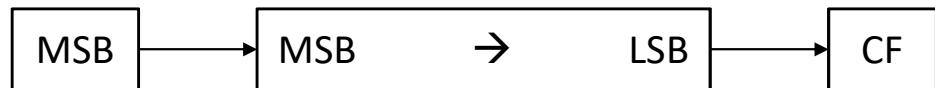
Arithmetic Shift

SAR-Shift Arithmetic Right:

Same as Logical Shift Right, But instead of a 0 being copied into MSB on shifting, Sign Bit, i.e MSB is copied into MSB on Shifting

SAL-Shift Arithmetic Left:

Same as Logical Shift Left.



Signed Number Comparision

CMP Dest,Src

Same as Comparison of Unsigned Numbers

For Unsigned CF & ZF are checked for determining, Greater than, Less than or Equality

For Signed Numbers OF,ZF and SF are checked

String & Table Operations

STRING Instructions

- CMPSB
- CMPSB
- SCASB
- SCASW
- STOSB
- STOSW
- LODSB
- LODSW
- MOVSB
- MOVSW
- CLD
- STD

STRING Instructions REP Prefix

- REPZ
- REPNZ
- REPE

Table Instructions

- XLAT

Use of DS:SI and ES:DI, DF and REP Prefix

For String Operations to work certain register must be set aside for specific functions

These Registers must permanently provide the Source & Destination of the Operands

SI Points to the Source Operand. DI points to the Destination Operand

To generate Physical Address: SI is always an offset to DS. DI is always an offset to ES

ES must be initialized for String Operations to work.

In String Operations, the operand can be a Byte or Word, specified by letters B or W in the mnemonic.

(CMPSB/CMPSW, SCASB/SCASW, LODSB/LODSW, STOSB/STOSW, MOVSB/MOVSW) .

Use of DS:SI and ES:DI, DF and REP Prefix

Direction Flag (DF): The Pointers SI & DI have to be incremented or decremented to process operands located in consecutive memory locations.

String instruction Increment/Decrement SI & DI depending on DF.

If DF=0 (using CLD), SI & DI will Increment Automatically (Auto-Increment)

If DF=1 (using STD), SI & DI will Decrement Automatically (Auto-Decrement)

The REP(repeat) Prefix allows a string instruction to perform repeatedly. REP assumes that CX holds the Number of Times the instruction must be repeated.

REP tells the CPU to perform the string operation and then Decrement the CX Register.

REPE,REPZ,REPNE and REPNZ along with assuming that CX holds count also checks the status of ZF and allows a String instruction to be performed repeatedly as long as the specified Condition (REPZ/REPE => ZF=1) or (REPNZ/REPNE => ZF=0) is satisfied and as long as CX is not 0.

MOVS- MOVS_B/MOVS_W

MOVSB- Move String Byte.

- Transfers a byte from DS location addressed by SI to ES location Addressed by DI.
- SI & DI are incremented (DF=0) or decremented (DF=1) by 1.
- Source Segment maybe overridden, Destination Segment is always ES.
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

MOVSW- Move String Word.

- Transfers a word from DS location addressed by SI & (SI+1) to ES location Addressed by DI & (DI+1).
- SI & DI are incremented (DF=0) or decremented (DF=1) by 2
- Source Segment maybe overridden, Destination Segment is always ES.
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

STOS- STOSB/STOSW

STOSB- Store String Byte.

- Stores data in AL at the ES memory location addressed by DI Register.
- DI is incremented by 1 if DF=0 or decremented by 1 if DF=1
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

STOSW- Store String Word.

- Stores data in AX at the ES memory location addressed by DI Register. (AL in ES:DI, AH in ES:(DI+1))
- DI is incremented by 2 if DF=0 or decremented by 2 if DF=1
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

LODS-LODSB/LODSW

LODSB- Load String Byte.

- Loads AL with data at segment offset address indexed by SI register.
- SI is incremented by 1 if DF=0 or decremented by 1 if DF=1.
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

LODSW- Load String Word.

- Loads AX with data at segment offset address indexed by SI register.(DS:SI in AL, DS(SI+1) in AH)
- SI is incremented by 2 if DF=0 or decremented by 2 if DF=1.
- If used with REP prefix the operation can be repeated for a specified number of times assuming CX holds the count

CMPS-CMPSB/CMPSW

CMPSB- Compare String Byte.

- *Contents of DS Memory location addressed by SI (DS:SI) are compared with ES memory Location addressed by DI (ES:DI).*
- *SI and DI are incremented by 1 if DF=0 or decremented by 1 if DF=1*
- *If used with REPE/REPZ/REPNE/REPNZ prefix then operation can be repeated for a specified number of times assuming CX holds the count and the Condition of ZF specified by the prefix is satisfied.*

CMPSW- Compare String Word.

- *Contents of DS Memory location addressed by SI (DS:SI) are compared with ES memory Location addressed by DI (ES:DI). Contents of DS Memory location addressed by SI+1 (DS:SI+1) are compared with ES memory Location addressed by DI+1 (ES:DI+1).*
- *Thus a word pointed at by DS:SI and ES:SI are compared.*
- *SI and DI are incremented by 2 if DF=0 or decremented by 2 if DF=1*
- *If used with REPE/REPZ/REPNE/REPNZ prefix then operation can be repeated for a specified number of times assuming CX holds the count and the Condition of ZF specified by the prefix is satisfied.*

SCAS-SCASB/SCASW

SCASB- Scan String Byte.

Compares Byte of data in AL with Byte from ES Memory location pointed at by DI(ES:DI).

DI is incremented if DF=0) or decremented if DF=1

If used with REPE/REPZ/REPNE/REPNZ prefix then operation can be repeated for a specified number of times assuming CX holds the count and the Condition of ZF specified by the prefix is satisfied.

SCASW- Scan String Word.

- A Word in AX Register is compared with ES memory Location addressed by DI (ES:DI). Byte in AL is compared with ES memory Location addressed by D (ES:DI). Byte in AH is compared with ES memory Location addressed by DI+1 (ES:DI+1).
- Thus a word pointed at by DS:SI and ES:SI are compared.
- SI and DI are incremented by 2 if DF=0 or decremented by 2 if DF=1
- If used with REPE/REPZ/REPNE/REPNZ prefix then operation can be repeated for a specified number of times assuming CX holds the count and the Condition of ZF specified by the prefix is satisfied.

REP Prefix -REPZ/REPNZ or REPE/REPNE

Repeat Prefix. Works with CMPS, SCAS, MOVS, LODS and STOS instructions. Executes the specified instruction repeatedly until CX=0.

Repeat If Equal/Zero. Works with CMPS, SCAS, LODS and STOS instructions. Executes the specified instruction repeatedly until CX=0 or as long as Equal condition is satisfied, i.e ZF=1. Exits the repeat cycle if CX becomes 0 or ZF=0

Repeat If Not Equal/Not Zero. Works with CMPS, SCAS, MOVS, LODS and STOS instructions. Executes the specified instruction repeatedly until CX=0 or as long as Not Equal/Not Zero condition is satisfied, i.e ZF=0. Exits the repeat cycle if CX becomes 0 or ZF=1

XLAT

Translates or converts the contents of AL Register into a number stored in a memory Table. Performs Direct Table Lookup Technique. It adds the contents of AL to BX to form a Memory Address within DS and copies the content of this address to AL.

Example:

```
SQR_TABLE DB 0,1,4,9,16,25,36,49,64,81; Defined in Data Segment
```

```
;In Code Segment  
:  
MOV BX,OFFSET SQR_TABLE ;This code will retrieve the 6th element from the table and put it in AL  
MOV AL,05  
XLAT                      ;AL will have 25 i.e 19H after execution of XLAT
```

Memory Address Decoding

CPLD (Complex Programmable Logic Devices)- Memory & Address Decoding Circuitry Integrated into one Programmable Chip.

NAND Gate

Decoder – IC 74LS138

SRAM and SROM considered for the sake of simplicity

Memory Address Decoding

Memory Capacity:

Number of Bits a Semiconductor memory chip can Store- Chip Capacity

Units- K Bits(kilo bits), M Bits(megabits) etc. unlike Memory capacity of a computer (in Bytes)

Memory Organization:

Memory Chips are organized into a number of locations within the IC

Each location can hold 1bit,4bits, 8bits or 16 bits.- Depends on Internal Design of chip.

Number of Bits in Each Location depends on Number of lines in the Chip's Data Bus

Number of Locations in a chip depends on Number of lines in the Chip's Address Bus

Memory Address Decoding

Each Memory chip contains 2^x locations, where x = number of pins on address bus

Each Location contains y bits, where y= number of pins on the data bus

Entire Chip contains $2^x \times y$ bits

$2^x \times y$ is referred to as the ORGANIZATION of the memory chip.

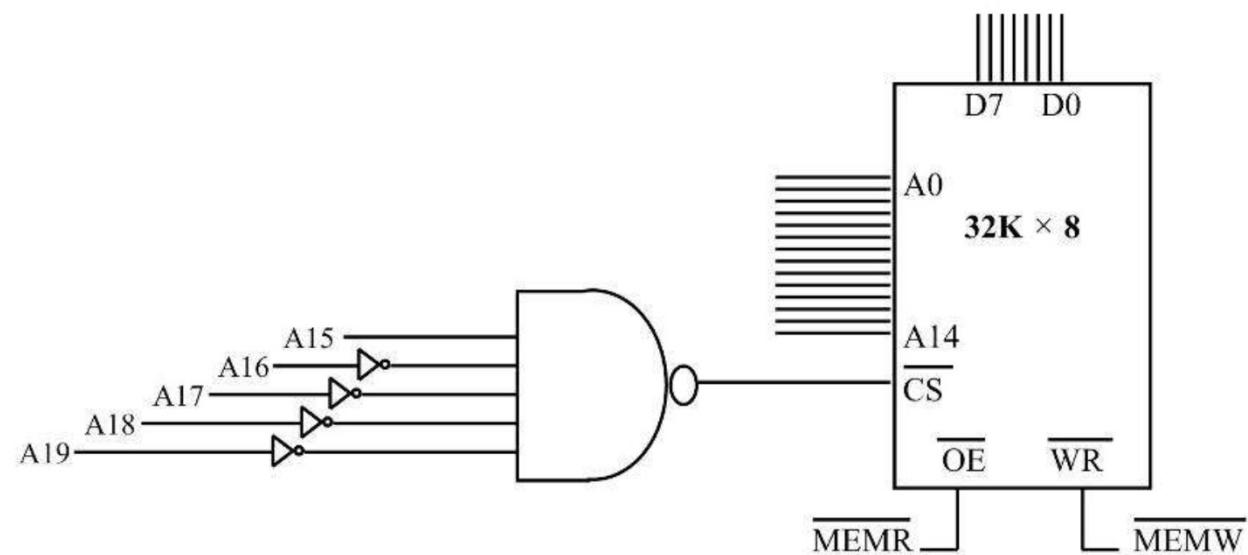
Eg: Memory Chip has 12 Address Pins and 8 Data pins. Find Organization & Capacity

Organization $2^{12} \times 8 = 4K \times 8$

Capacity = $4K \times 8$ Bits = 32K bits

x	2^x
10	1K
11	2K
12	4K
13	8K
14	16K
15	32K
16	64K
17	128K
18	256K
19	512K
20	1M
21	2M
22	4M
23	8M
24	16M

Simple NAND Gate as Decoder



NAND output is LOW when all inputs are HIGH
Otherwise output is 1
HIGH output of NAND deactivates Decoder
LOW Output of NAND activates Decoder
CS is Active LOW

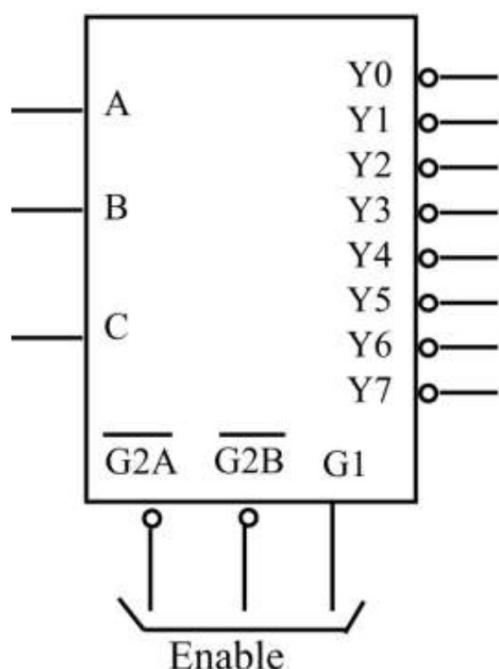
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Start Address: 08000H
End Address: 0FFFFH] Address Range

After 0FFFFH next address is 10000H.
For that address A16 becomes 1 thereby giving a

74LS138 Decoder

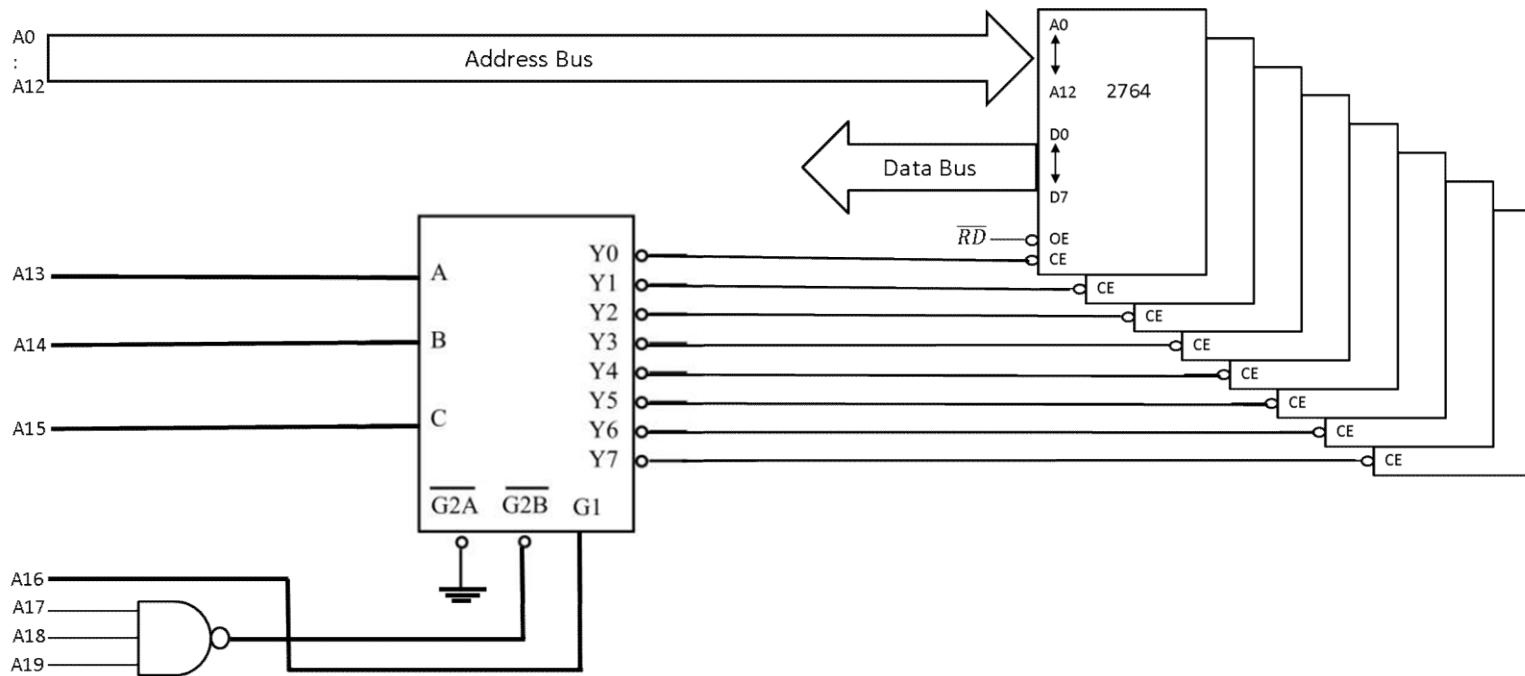
Block Diagram



Function Table

Inputs		Outputs
Enable	Select	Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7
G1G2	C B A	Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7
X H	XXX	H H H H H H H H
L X	XXX	H H H H H H H H
H L	LLL	L H H H H H H H
H L	LLH	H L H H H H H H
H L	LHL	H H L H H H H H
H L	LHH	H H H L H H H H
H L	HLL	H H H H L H H H
H L	HLH	H H H H H L H H
H L	HHL	H H H H H H L H
H L	HHH	H H H H H H H L

74LS138 as Memory Address Decoder



Address Space Selected
by This Circuit Design

F0000 – F1FFF

F2000 – F3FFF

F4000 – F5FFF

F6000 – F7FFF

F8000 – F9FFF

FA000 – FBFFF

FC000 – FDFFF

FE000 – FFFFF

DECODING LOGIC FOR G2A, G2B AND G1					C	B	A	TO THE 13 ADDRESS LINES OF 8K RAMS												
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

F0000H – F1FFFH

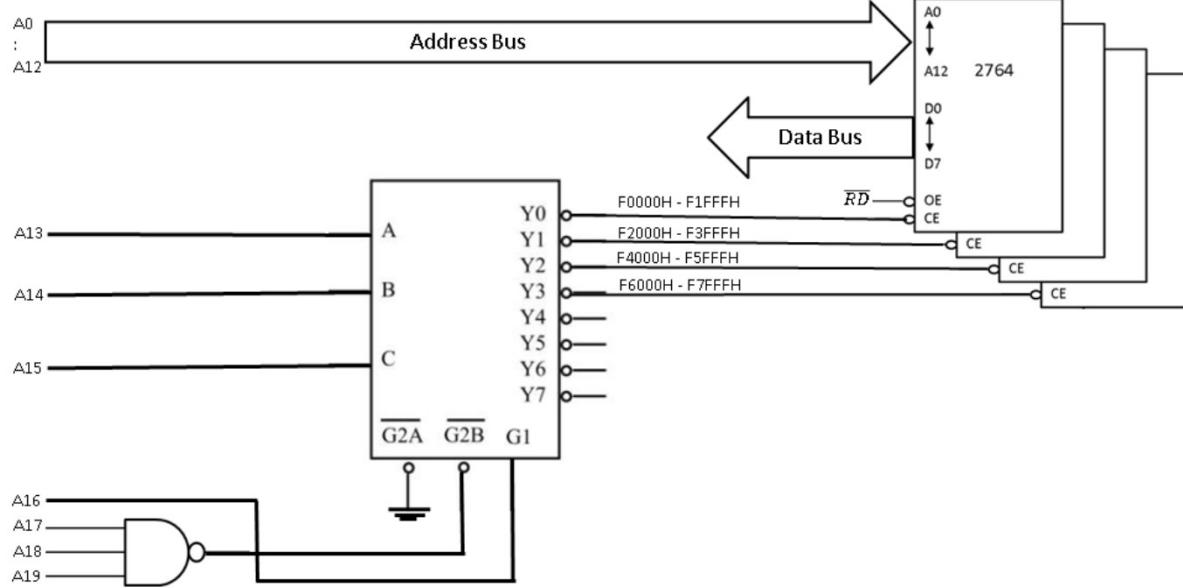
F2000H – F3FFFH

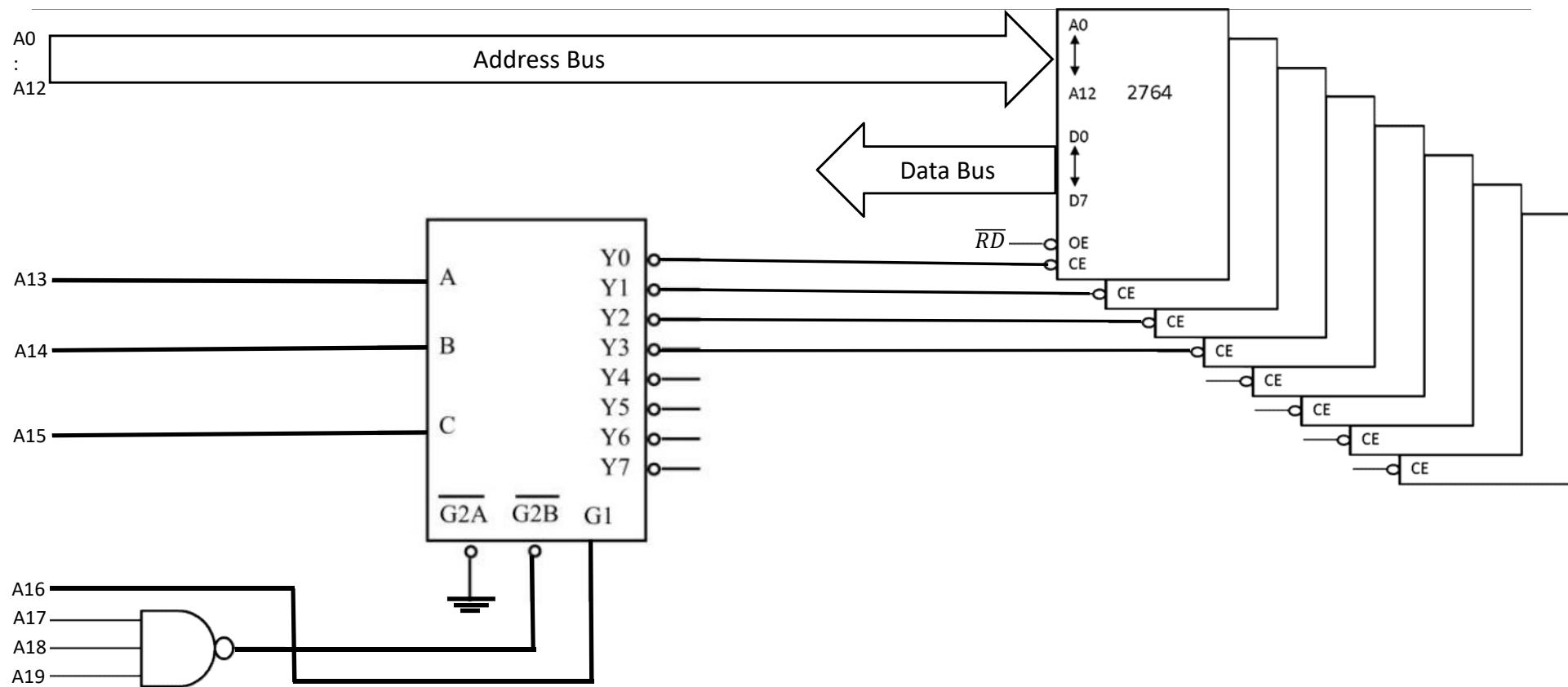
F4000H – F5FFFH

F6000H – F7FFFH

DECODING LOGIC FOR G2A,G2B AND G1		C	B	A	TO THE 13 ADDRESS LINES OF 8K RAMS														
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

F0000H - F1FFFH
 F2000H - F3FFFH
 F4000H - F5FFFH
 F6000H - F7FFFH





Examples

- Design a Memory System for 8086 for the following Specifications:
 - a. 32Kbytes EPROM using 16KByte Devices
 - b. 64KBytes SRAM using 16Kbyte Devices

Using 74LS138 Decoder

- Explain how 74LS138 Decodes 2732 EPROMS for 32K x 8 section of Memory. Assume Starting Address is 40000H. Give Detailed Memory Map

- Design an 8086 based system to interface with
 - a. 64K Byte EPROM
 - b. 64K Byte SRAM

Assume RAM is connected at 30000H and EPROM is connected at F0000H

- Explain how a 3-8 Decoder could be used to interface eight 8K memory chips

Data Integrity in RAM & ROM

Ensure that data retrieved is same as data

Same applies when transferring data from one place to another

1. Checksum Method
2. Parity Method

Checksum Byte:

Used to detect corruption of data(maybe due to power surge)

Uses Checksum Byte.- Extra Byte that is tagged at the end of a series of bytes.

Add the Bytes together & Drop the carries

Take the 2's Complement of the total sum. That is the checksum byte, the last byte of the stored information

To perform checksum operation, add all bytes including checksum byte. The result must be 0. If result is non zero, then one or more bytes have been corrupted.

Example

Assume that there are 4 Bytes of Hexadecimal data 25H,62H,3FH and 52H

Find Checksum Byte

Perform Checksum Operation to ensure data integrity

IF the 2nd byte 62H has changed to 22H, show how checksum detects error.

25H	Dropping the Carry we get 18H	25H	25H	
+62H	Taking 2's Complement we get E8H	+ 62H	+ 22H	Data Corrupted 62H changed to 22H
+3FH	Add it along with the other bytes.	+ 3FH	+ 3FH	
+52H	Drop the carry. We get 00H.	+ 52H	+ 52H	
	Therefore No Corruption of data	+ E8H	+ E8H	
1 18H		2 00H	1 C0H	Not 0

Checksum

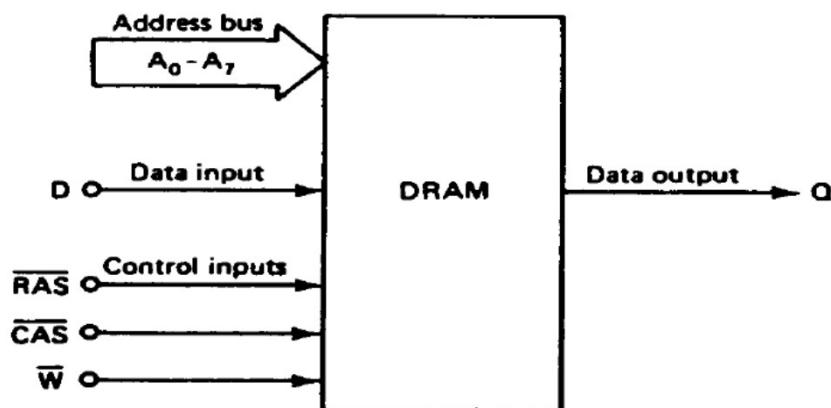
```
2411 ;-----  
2412 ;      ROS CHECKSUM SUBROUTINE      :  
2413 ;-----  
EC4C     2414 ROS_CHECKSUM PROC NEAR ;NEXT_ROS_MODULE  
EC4C B90020 2415           MOV CX,8192 ;NUMBER OF BYTES TO ADD  
EC4F     2416 ROS_CHECKSUM_CNT: ;ENTRY PT. FOR OPTIONAL ROS TEST  
EC4F 32C0   2417           XOR AL,AL  
EC51     2418 C26:  
EC51 0207   2419           ADD AL,DS:[ BX]  
EC53 43    2420           INC BX ;POINT TO NEXT BYTE  
EC54 E2FB   2421           LOOP C26 ;ADD ALL BYTES IN ROS MODULE  
EC56 0AC0   2422           OR AL,AL ; SUM = 0?  
EC58 C3    2423           RET  
2424 ROS_CHECKSUM ENDP
```

Use of Parity Bit in DRAM Error Detection

DRAM Memory Banks-Arrangement of DRAM chips on the system or memory Modules

Eg: 64K Bytes of DRAM can be arranged as :

1 bank of 8 IC chips of 64K x 1 organization or 4 banks of 16K x 1 organization



DRAM Memory Banks

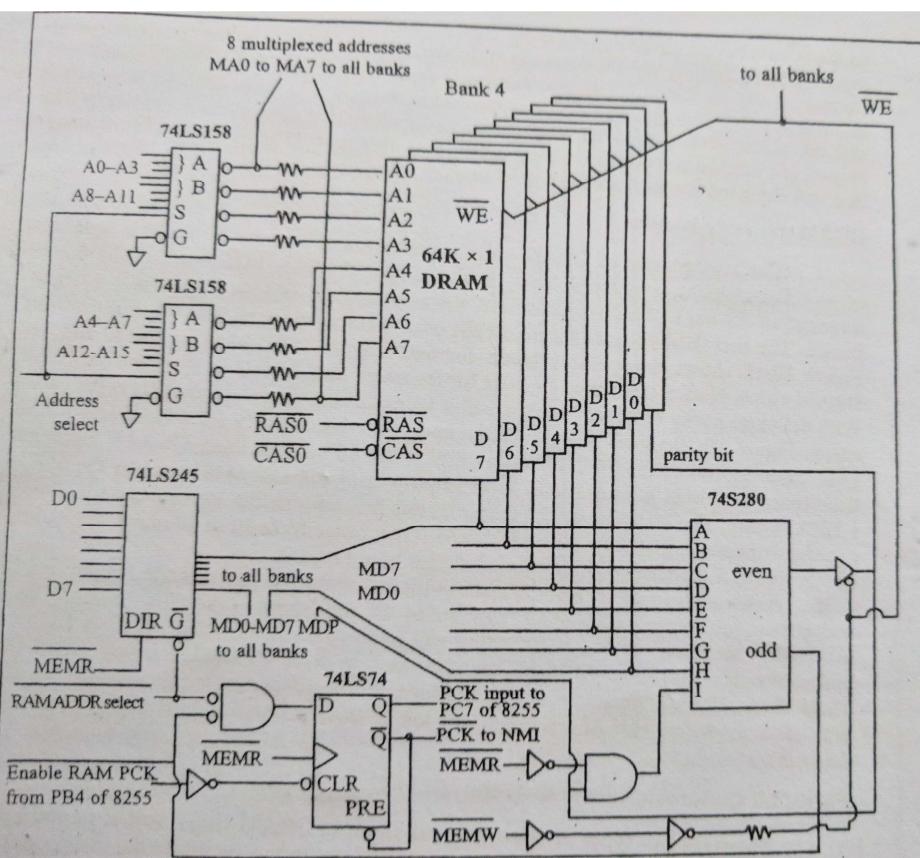
	d7 ...	d4	d3 ...	d0	Parity	640K DRAM	
Bank 3: $64K \times 9$						Single 256K-Bit Chip	$64K \times 4$
	$64K \times 4$		$64K \times 4$		$64K \times 1$		
Bank 2: $64K \times 9$						Single 1M-Bit Chip	$256K \times 4$
	$64K \times 4$		$64K \times 4$		$64K \times 1$		
Bank 1: $256K \times 9$						Figure shows the Memory banks of 640 K Bytes of RAM using 256K & 1M DRAM chips.	
	$256K \times 4$		$256K \times 4$		$256K \times 1$		
Bank 0: $256K \times 9$						1 Extra bit called Parity is used for every byte stored.	
	$256K \times 4$		$256K \times 4$		$256K \times 1$		

640K Byte RAM

	d7 ...	d4	d3 ...	d0	Parity
Bank 3: 64K × 9	64K × 4		64K × 4		64K × 1
Bank 2: 64K × 9	64K × 4		64K × 4		64K × 1
Bank 1: 256K × 9	256K × 4		256K × 4		256K × 1
Bank 0: 256K × 9	256K × 4		256K × 4		256K × 1



Parity Bit Generator/Checker



The o/p of MUX MA0-MA7 will go to all banks

Memory Data MD0-MD7 and Memory Data Parity MDP will go to all banks.

IC 74LS245 Buffers and Boosts MD0-MD7 to drive all DRAM I/P

There are 2 types of errors:

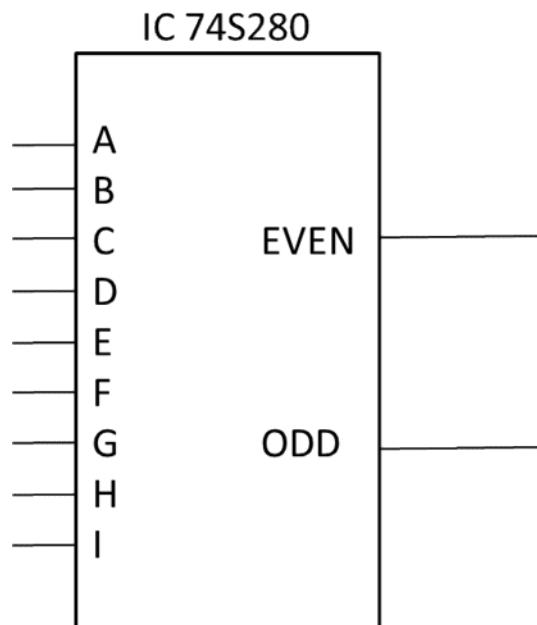
Hard Error- Some Bits or an entire row of memory cells inside the memory chip gets stuck to HIGH or LOW permanently, thereby producing 1 or 0 regardless of what is written into the cells.

Soft Error- A single Bit is changed from 1 to 0 or from 0 to 1 due to current surge or certain kinds of particle radiation in the air.

Parity is used to detect such errors.

This can only indicate if there is a difference between the data that was written to memory and the data that was read.

Parity Bit Generator/Checker



Parity Generator/Checker Chip (IC 74S280)-

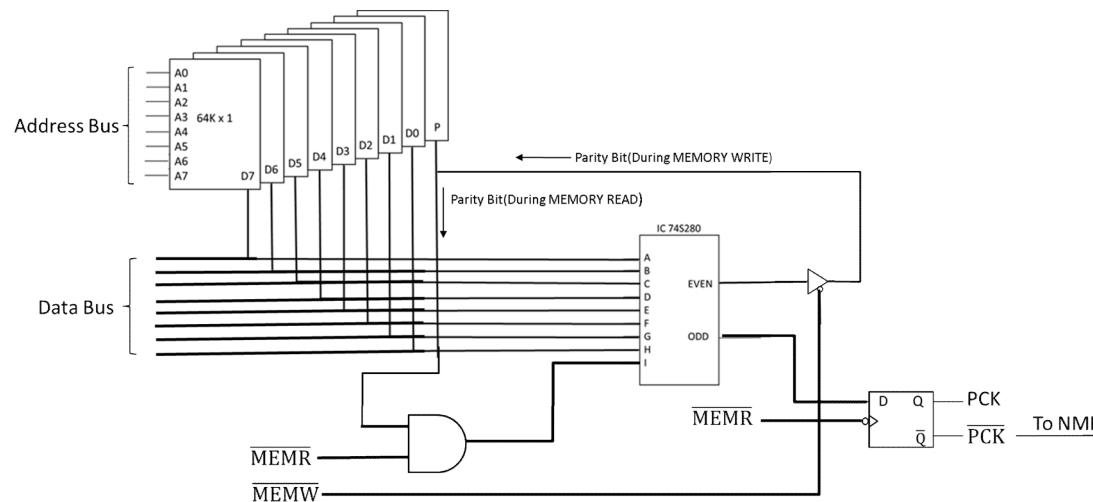
9 inputs / 2 outputs

If ODD number of 1s at the Input, ODD output is 1, EVEN output is 0

If EVEN number of 1s at the Input, ODD output is 0, EVEN output is 1

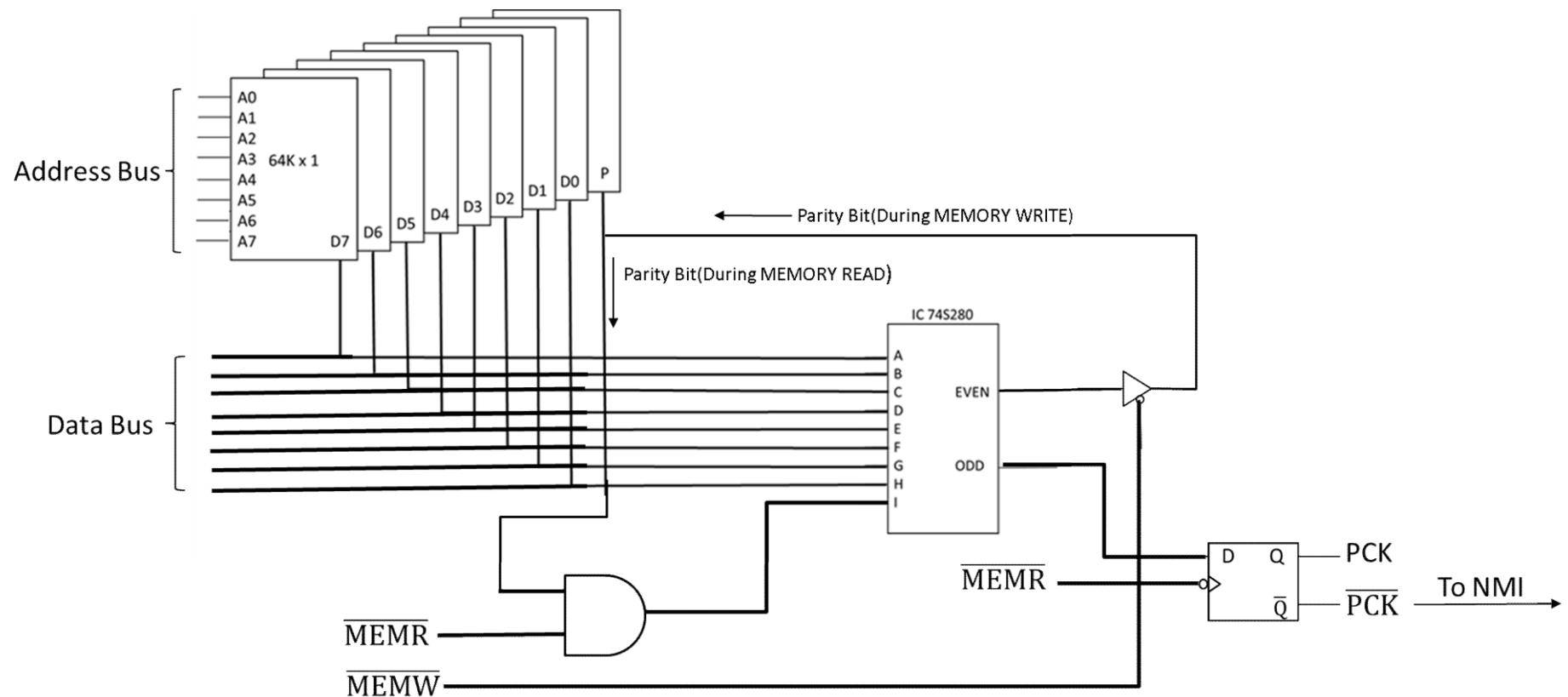
	INPUTS		OUTPUTS	
CASE	A-H	I	EVEN	ODD
1	EVEN	0	1	0
2	EVEN	1	0	1
3	ODD	0	0	1
4	ODD	1	1	1

Parity Bit Generator/Checker



- Storing EVEN data in D7-D0 generates a 1 in the EVEN output which stores a 1 in the Parity Bit thereby making the Parity ODD for the 9 bits including Parity Bit [D7-D0-P]
- Storing ODD data in D7-D0 generates a 0 in the EVEN output which stores a 0 1 in the Parity Bit thereby making the Parity ODD for the 9 bits including Parity Bit [D7-D0-P]
- A-H are connected to Data Bus. D7- D0 are connected to Data Bus
- I is used as Parity Bit to check correctness of the byte of data read from memory
- When a byte is written, Even parity Bit is generated & saved in the 9th bit position.($\overline{\text{MEMW}}$ is active (0))
- If D7-D0 is having Even Parity, Then a 1 is stored in Parity Bit Position Making the parity of the 9-Bit Data ODD.
- If D7-D0 is having ODD Parity, Then a 0 is stored in Parity Bit Position Making the parity of the 9-Bit Data ODD
- At this point I=0 since $\overline{\text{MEMR}}$ is 0
- When a Byte of Data is READ i.e $\overline{\text{MEMR}} = 0$, Parity bit is gated into I input through the AND gate. This gives the Parity of the 9-Bit Data including Parity Bit. If no Bit has changed then the Parity of the data is ODD(implies data is not corrupted) i.e ODD output will be HIGH. If a bit has changed its state and the data is corrupted then the ODD output would be LOW . The ODD output is given to a D Flip Flop which is passed to the output of the Flip flop when $\overline{\text{MEMR}} = 0$
- If ODD output is High, Q=1 and $\bar{Q}=0$ indicating NO ERROR.
- If ODD Output is Low, Q=0 and $\bar{Q}=1$ Indicating an ERROR. This signal is given to NMI pin to take the required action in case of ERROR

Parity Bit Generator/Checker



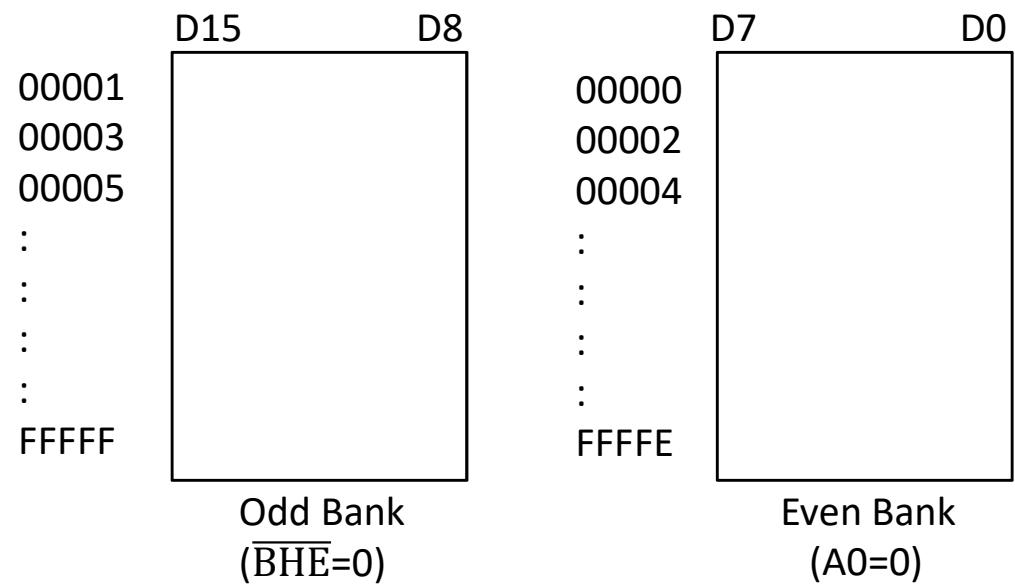
16-Bit Memory Interfacing

Consider 80286.

Memory Locations from 00000H to FFFFFH are designated as ODD and EVEN Bytes.

To Distinguish between ODD & EVEN Bytes CPU has \overline{BHE} (Bus High Enable) in association with A0

\overline{BHE}	A0		
0	0	Even Word	D0-D15
0	1	Odd Byte	D8-D15
1	0	Even Byte	D0-D7
1	1	None	

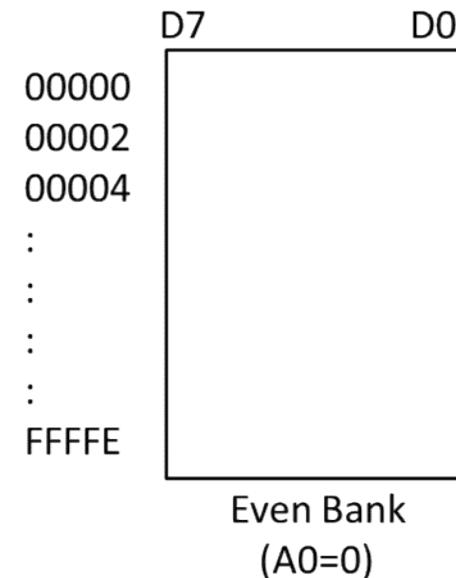
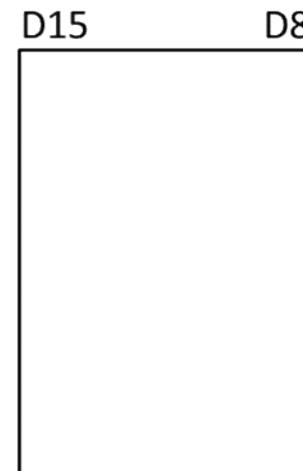


16-Bit Memory Interfacing

00000
00001
00002
00003
00004
00005
:
:
:
:
:
:
:
:
FFFFE
FFFFF

A0 & \overline{BHE} are used ad Bank Selectors

00001
00003
00005
:
:
:
:
FFFFF

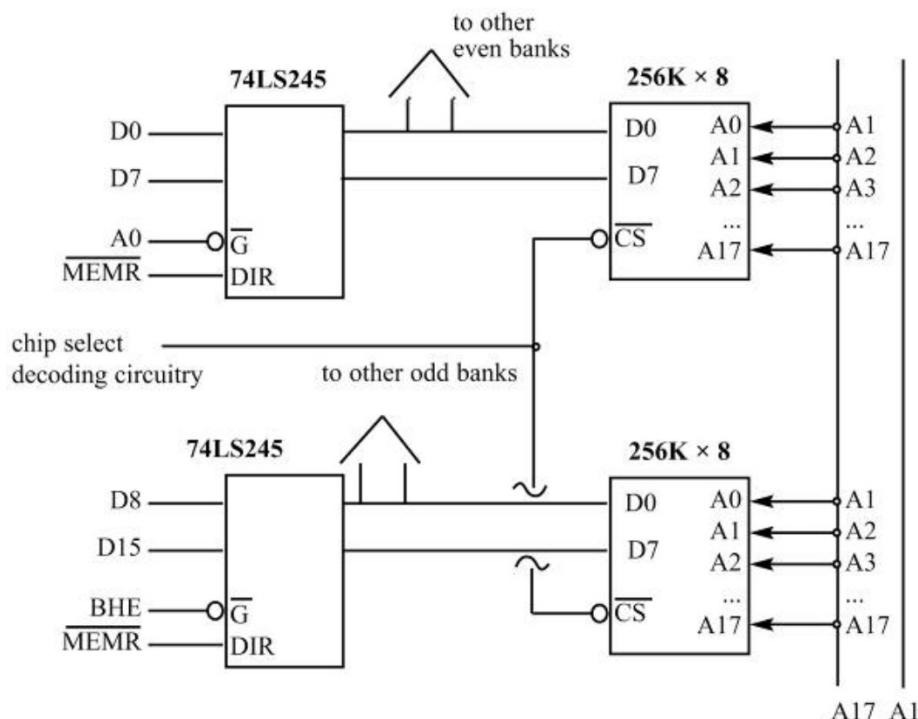


\overline{BHE}	A0		
0	0	Even Word	D0-D15
0	1	Odd Byte	D8-D15
1	0	Even Byte	D0-D7
1	1	None	

640K Byte DRAM with ODD & EVEN Banks

Parity	D15	D12	D11	D8	Parity	D7	D4	D3	D0
256K x 1	256K x 4	256K x 4	256K x 1	256K x 4					
64K x 1	64K x 4	64K x 4	64K x 1	64K x 4					

16-Bit Data Connection in systems with 16-Bit Bus



A0 is used to Enable Even Bank

\overline{BHE} is used to enable Odd Bank. Indicates transfer of Higher Byte of Data

When an Even Address is generated the A0 bit will be 0 and this will activate the Even Bank

For an Odd Address \overline{BHE} will be 0 and A0 will be 1 this will deactivate the Even Bank and Select the Odd Bank

For accessing a WORD, Both \overline{BHE} and A0 will be 0 selecting both Odd and Even Bank and a 16-Bit word will be used.

Memory Cycle Time & Wait States

To access an IO or Memory, The CPU provides fixed amount of time called a BUS CYCLE TIME.

During this time, the READ and WRITE operation of the Memory or IO must be completed.

The BUS CYCLE TIME used for accessing Memory is called MEMORY CYCLE TIME

The time from when the CPU provides the address at its ADDRESS PINS to when the Data is expected at its DATA PINS is called MEMORY READ CYCLE TIME. – 2 Clock Cycles 80286 onwards

If memory is slow and its access time does not match the CPU's Memory Cycle Time, extra time can be requested by from the CPU to extend the read cycle time. This is called a WAIT STATE.

TO avoid too many WAIT STATES – CACHE/High Speed DRAM

Memory Cycle Time & Wait States

Memory Access Time is the largest factor for slow down, but its not the only one. There is also the delay associated with signals going through data & address paths.

Delay Associated with reading data stored in memory has 2 components :

1. The time taken for address signals to go from CPU pins to memory pins, going through decoders & buffers plus the time it takes for the data to travel from memory to CPU is referred to as PATH DELAY
2. The Memory Access Time to get data out of the memory chip. This is the largest delay. (about 80% of the READ CYCLE TIME)

The total sum of these 2 must be equal to the memory read cycle time provided by the CPU.

Examples

Calculate the Memory Cycle Time of a 20 MHz 80386 system with

1. 0 WS
2. 1 WS
3. 2 WS

Assume that the bus speed is same as the processor speed.

$$f = 20 \text{ M Hz}$$

$$T = 1/f = 1/20 \text{ MHz} \Rightarrow 50 \text{ ns}$$

i.e the Processor clock period.

80386 has bus cycle time of 0 WS as 2 clocks.

$$\text{Memory Cycle Time with 0 WS} = 2 \times 50 \text{ ns} = 100 \text{ ns}$$

$$\text{Memory Cycle Time with 1 WS} = 100 \text{ ns} + 50 \text{ ns} = 150 \text{ ns}$$

$$\text{Memory Cycle Time with 2 WS} = 100 \text{ ns} + 50 \text{ ns} + 50 \text{ ns} = 200 \text{ ns}$$

WS have to be Integer values & cannot take values such as 1.25, 1.5 etc.

Examples

A 20M Hz 80386 based system is using a ROM of 150ns speed. Calculate the number of wait states needed if the path delay is 25ns.

If ROM access time is 150ns and the path delay is 25ns, every time the 80386 accesses the ROM it must spend a total of 175ns to get data into the CPU

A 20MHz CPU with 0 WS provides only 100ns (MEMORY READ CYCLE TIME. – 2 Clock Cycles 80286 onwards.)

TO match the CPU bus speed with this ROM 2 WS must be inserted which makes the Memory Read Cycle time 200ns

Accessing Even & Odd Words.

Intel Defines 16-Bit data as WORD.

The address of a location in which a word is stored can start at an ODD Address or an EVEN address.

In Systems with 16-Bit Data bus accessing a word from an Odd Addressed location can be slower than accessing the same word if it was stored in an even addressed location.

In 8-Bit systems Accessing a word is treated as accessing 2 Bytes regardless of odd or even address.
Accessing a Byte takes 1 Memory Cycle, therefore Accessing a Word takes 2 Memory Cycle.

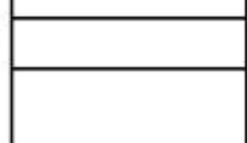
In 16-bit systems Accessing a word at an even address takes 1 Memory Cycle because 1 Byte is carried on D7-D0 and one byte is carried on D15-D8 in the same memory cycle.

In 16-bit systems Accessing a word at an odd address takes 2 Memory Cycle because 1 Byte (odd address) is carried on the 8 data lines out of the 16 in 1 cycle and the Other 1 Byte (Even Address) is carried on the 8 data lines out of the 16 in the 2nd cycle.

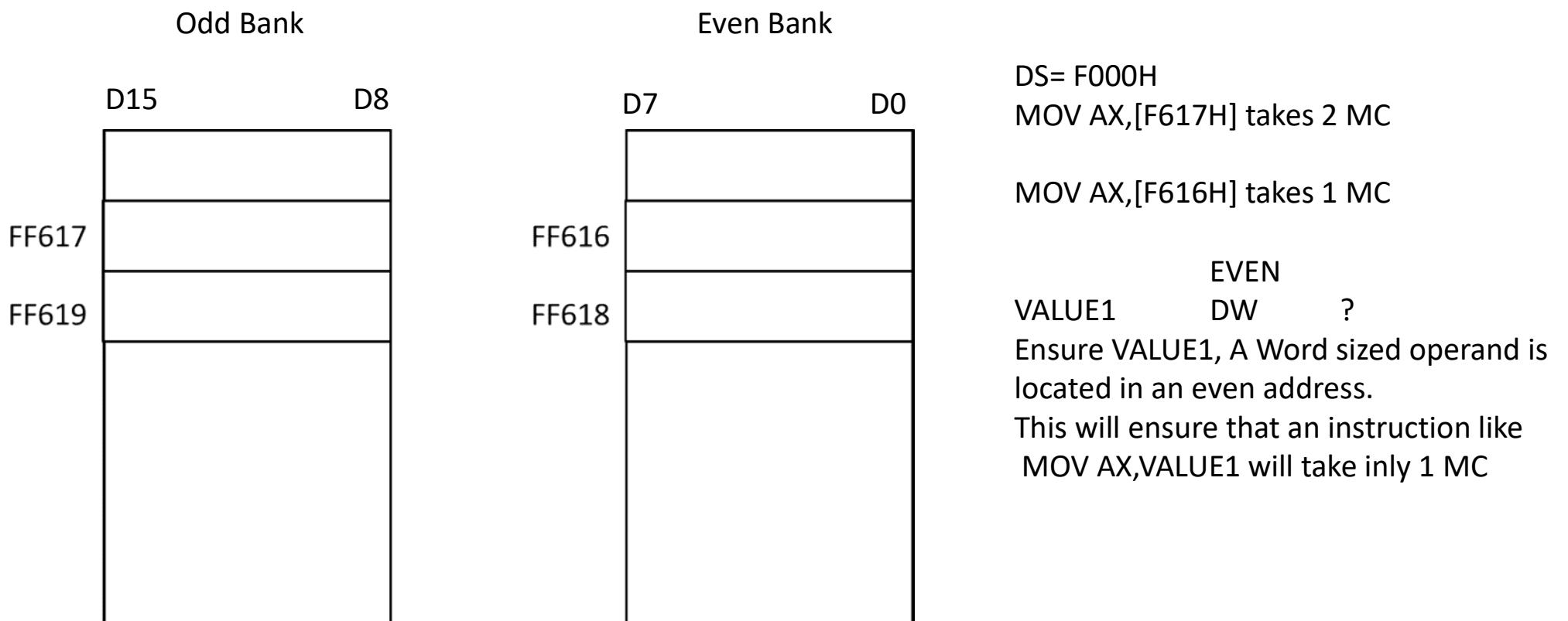
Therefore avoid storing words from odd addressed locations if the program is going to run on a 16-Bit system.

EVEN Directive.

Accessing Even & Odd Words in 8-Bit CPU

	D7	D0	MC (Memory Cycle)
FFF51			Assume that DS = F000
			“MOV AL,[FF51]” Odd byte takes 1 MC “MOV AL,[FF52]” Even byte takes 1 MC
FFF70			“MOV AX,[FF70]” Even word takes 2 MC
			“MOV AX,[FF91]” Odd word takes 2 MC
FFF91			

Accessing Even & Odd Words in 16-Bit CPU



Bus Bandwidth

Rate of Data Transfer between the CPU and the outside world is called BUS BANDWIDTH

It is a measure of how fast the buses transfer data between CPU and the Memory or Peripherals.

Wider the Data Bus, Higher the Bus Bandwidth.

Wider Bus Bandwidth comes at the cost of increased size of PCB.

Speed of the CPU must match with the Higher Bandwidth. CPU cannot process information that it does not have.

Bus Bandwidth is measured in MB/s

Bus Bandwidth=(1/Bus Cycle Time) x Bus Width in Bytes.

Example

Calculate the Memory Bus Bandwidth for the following microprocessors if the Bus speed is 20MHz

- a. 80286 with 0 WS and 1 WS (16-Bit Data Bus)
- b. 80386 with 0 WS and 1 WS(32-Bit Data Bus)

The memory cycle time for both 80286 & 80386 is 2 Clocks with 0 WS. With 20MHz bus speed, we have a bus clock of $1/20\text{MHz} = 50\text{ns}$.

With 0WS Bus Bandwidth = $(1/(2 \times 50\text{ns})) \times 2 \text{ Bytes} = 20\text{MB/s}$

With 1 WS Bus Bandwidth = $(1/(3 \times 50\text{ns})) \times 2 \text{ Bytes} = 13.3\text{MB/s}$

With 0WS Bus Bandwidth = $(1/(2 \times 50\text{ns})) \times 4 \text{ Bytes} = 40\text{MB/s}$

With 1 WS Bus Bandwidth = $(1/(3 \times 50\text{ns})) \times 4 \text{ Bytes} = 26.6\text{MB/s}$

Factors affecting Bus Bandwidth:
Read/write cycle time of CPU
Width of the Data Bus

Memory Mapped I/O

Memory Mapped I/O –

I/O Address space lies in the Memory Address Space. Portion of Address Space lost to I/O. –Memory Space Fragmentation

Memory location is assigned to be an input port or output port.

Makes use of instructions that access Memory location eg: MOV instruction for data transfer. MOV AL,[2000H] or MOV [2010H],AL

The Entire 20 Bit Address A19-A0 must be decoded. Therefore Segment Register DS must be loaded with Base Address. Since all 20 Bits are to be decoded The Decoding Circuitry is more expensive.

Control Signals used are MEMR and MEMW.

Can have 1M(1048576) Ports because addresses are 20 Bits wide.

Arithmetic & Logic operations can be performed directly on I/O Data without first moving it into the Accumulator.

Data can be transferred into any Register rather than only into Accumulator.

Isolated I/O

I/O Mapped I/O (Isolated I/O or Peripheral I/O) –

I/O space lies isolated from the Memory Address Space.

Entire Memory Address Space available for Memory.

Therefore MOV instruction cannot be used for Data Transfer. They Require different Instruction. 2 Instructions: IN & OUT

Control Signals used are \overline{IOR} and \overline{IOW} .

Limited to 65536 input ports & 65536 Output Ports. Because addresses are 16 Bits wide and there are separate \overline{IOR} and \overline{IOW} signals .

8-Bit Port Programming

8-Bit I/O operation is applicable to all x86 CPUs from 8088 to Pentium

The 8-Bit port uses D7-D0 data bus to communicate with the I/O Devices.

In 8-Bit Port Programming, AL is used as the SOURCE of data when using OUT & as the DESTINATION when using IN.

IN & OUT Instruction

IN instruction:

It can bring data from the ports into the Accumulator(AL)

IN destination,source

OUT destination,source

In format (1) port# is the address of the port and can range from 00H to FFH(0-255) allowing only 256 ports. i.e the address can only be 8-Bit wide. The 8-Bit address is carried on address bus A7-A0. No Segment Register is involved in computing the address in contrast to the way data is accessed from Memory.

In format (2) port# is the address of the port and can range from 0000H to FFFFH(0-65535) allowing 65536 ports. i.e the address can be 16-Bit wide. The 16-Bit address is carried on address bus A15-A0. No Segment Register is involved in computing the address in contrast to the way data is accessed from Memory. ONLY DX REGISTER IN INDIRECT ADDRESSING MODE CAN BE USED FOR HOLDING THE 16-BIT ADDRESSES OF PORTS

1. IN AL, port#
2. MOV DX, port#
IN AL, DX
1. OUT port#, AL
2. MOV DX, port#
OUT DX, AL

Example

ALP to send a byte of data to a fixed port address 43H (8-Bit)

MOV AL,36H

OUT 43H,AL

ALP to send a byte of data to a port address 300H

MOV DX,300H

MOV AL,55H

OUT DX,AL

Example

ALP to receive a byte of data from a fixed port address 43H (8-Bit)

IN AL,43H

ALP to receive a byte of data from a port address 300H

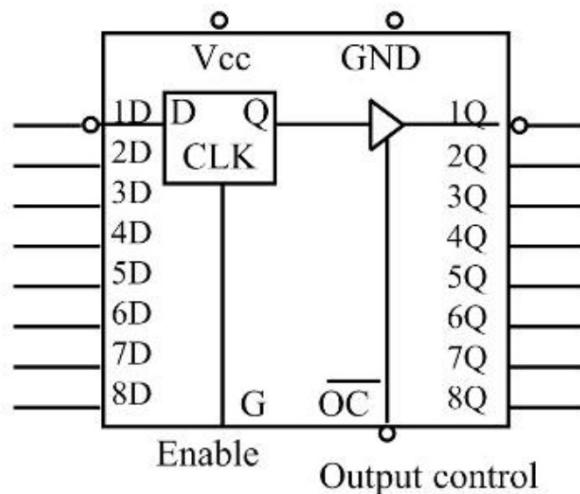
MOV DX,300H

IN AL,DX

Example

IN a given 8088 based system, port address 22H is an input port for monitoring temperature. Write an ALP to monitor that port continuously for the temperature of 100 degrees. If it reaches 100, then BH should contain 'Y'

IC 74LS373 D Latch



Function Table

Output Control	Enable		Output
	G	D	
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

I/O Address Decoding & Design

Using IC 74LS373 in an output Port Design:

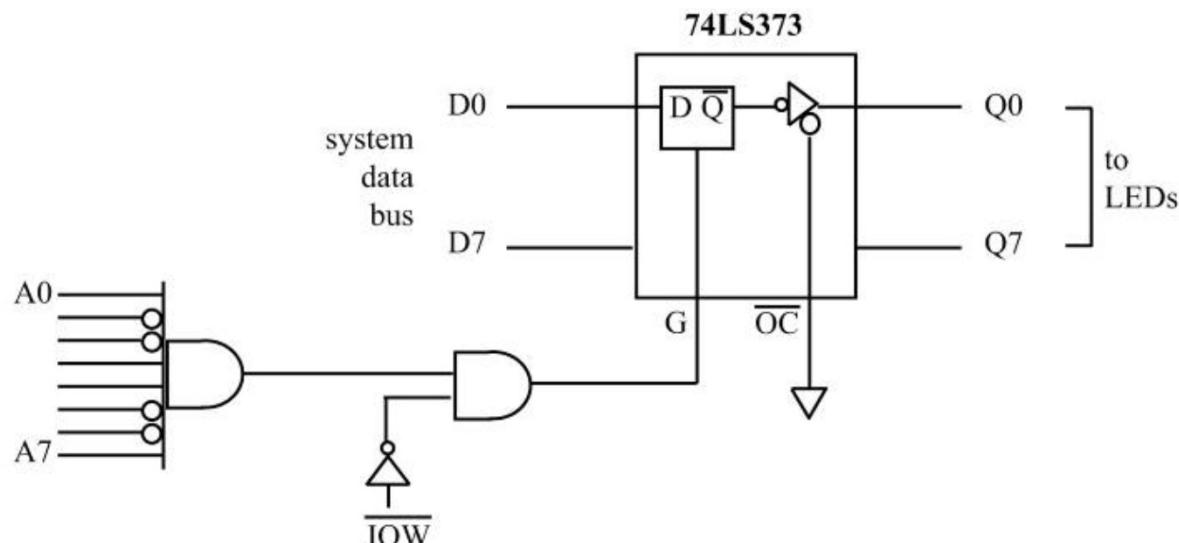
Whenever data is sent out by the CPU via the data bus, the data must be latched by the receiving device

Memories have an internal latch to grab the data, but a latching system must be designed for simple I/O ports. For this IC 74LS373 can be used.

The \overline{OC} must be grounded to make the 74LS373 to work as a latch.

It is common to AND the output of the Address decoder with the Control Signal \overline{IOW} to provide latching action

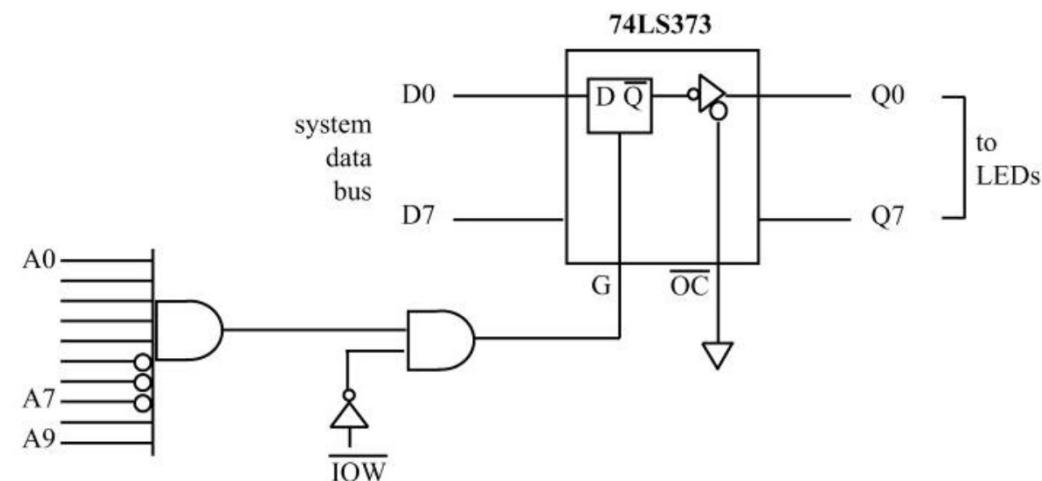
OUT 99H,AL will send the 8-Bit data in AL to port 99H



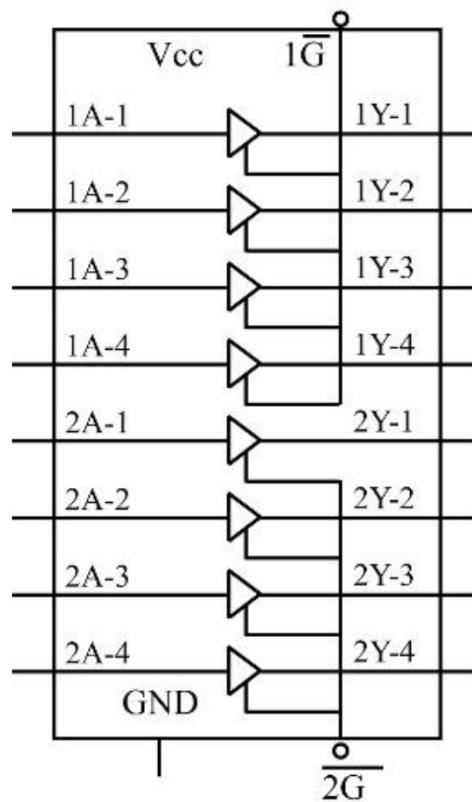
Design for Output Port Address 99H
OUT 99H,AL

I/O Address Decoding & Design

Design for Output Port of 31FH



IC 74LS244



I/O Address Decoding & Design

Using IC 74LS244 in an input Port Design:

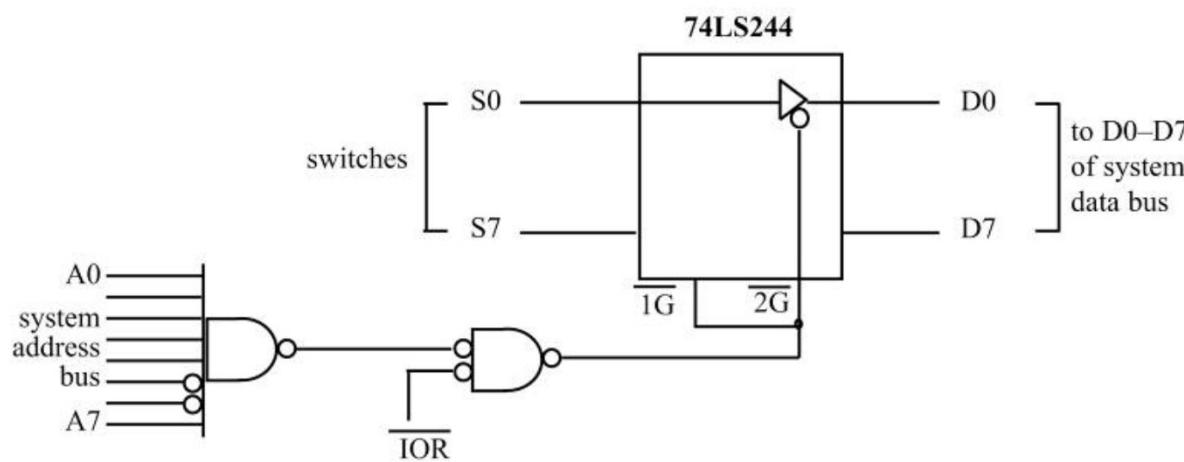
Whenever data is coming in via the data bus, the data must come in through a three-state buffer(Tristated).

Memories have an internal Tristate Buffer but for simple Input Ports IC 74LS244 can be used.

$\overline{1G}$ and $\overline{2G}$ each control only 4 Bits of the 74LS244. They both must be activated for 8-Bit input data.

In addition to Buffering it also provides sufficient driving capability to travel all the way to the CPU.

Design for IN AL,9FH

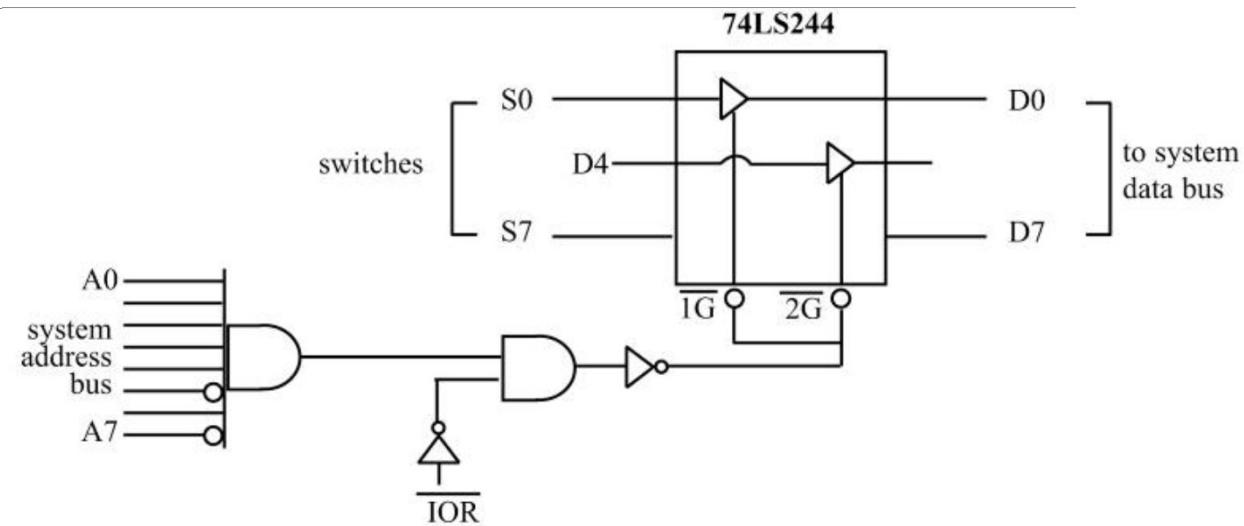


Design for Input Port Address 9FH
IN AL,9FH

I/O Address Decoding & Design

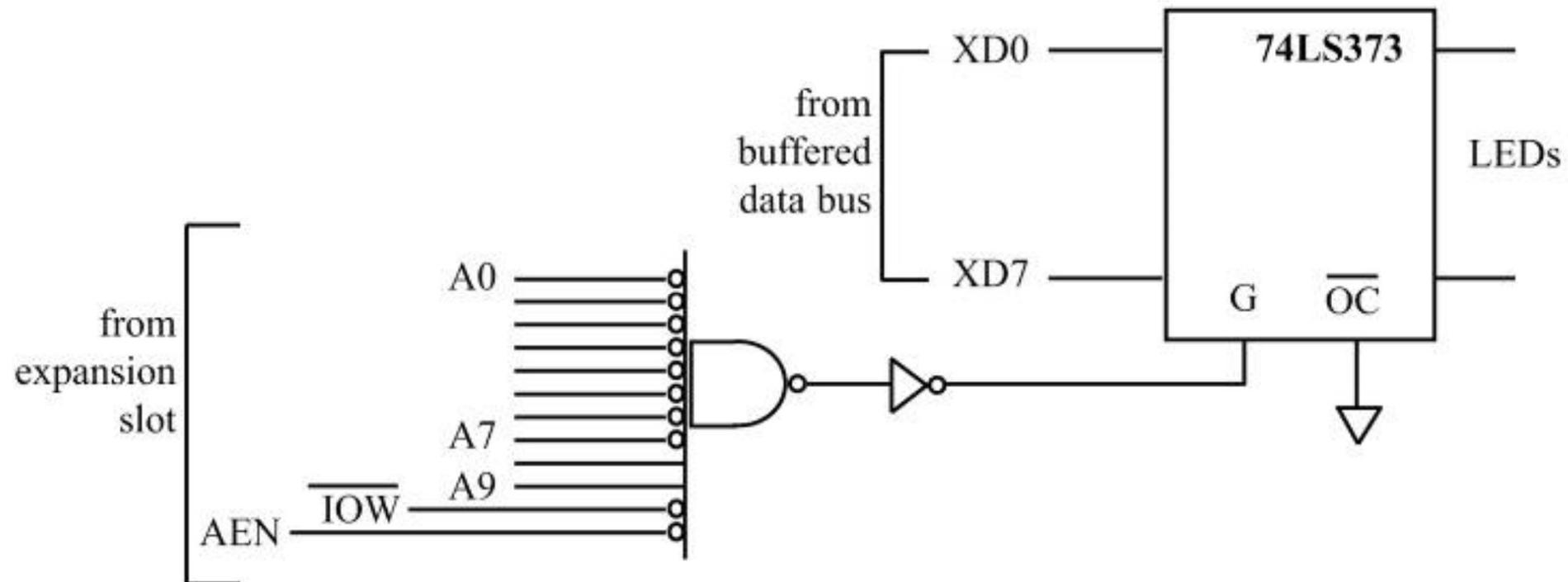
Design for Input Port 5FH

IN AL,5FH



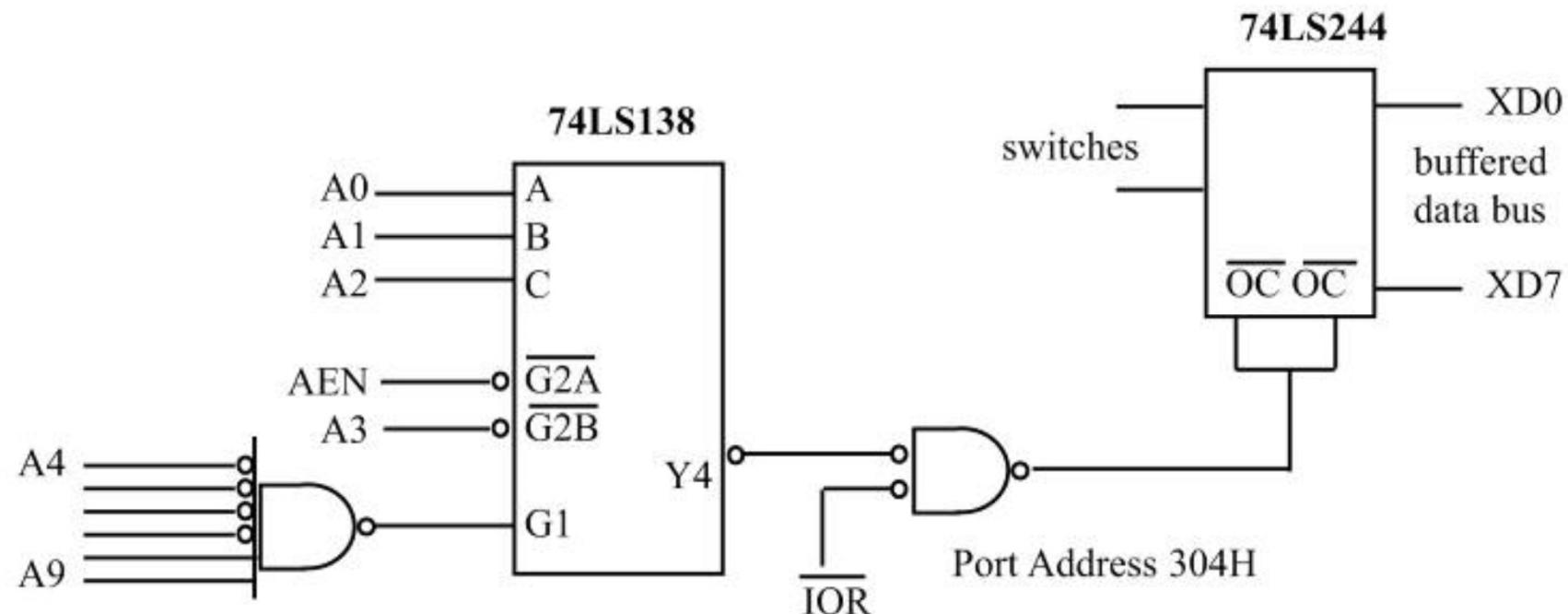
Use of Simple Logic Gates as IO Address Decoders

I/O Address 300H



Use of IC 74LS138 as IO Address Decoders

I/O Address – 304H



Port 61H & Time Delay Generation

Port 61H used to generate a time delay.

Will work with any type of processor from 80286 to Pentium.

I/O Port 61H has 8 –Bits (D7-D0)

Bit D4 of Port 61H changes its state every 15.085us.

This goes on as long as PC is on.

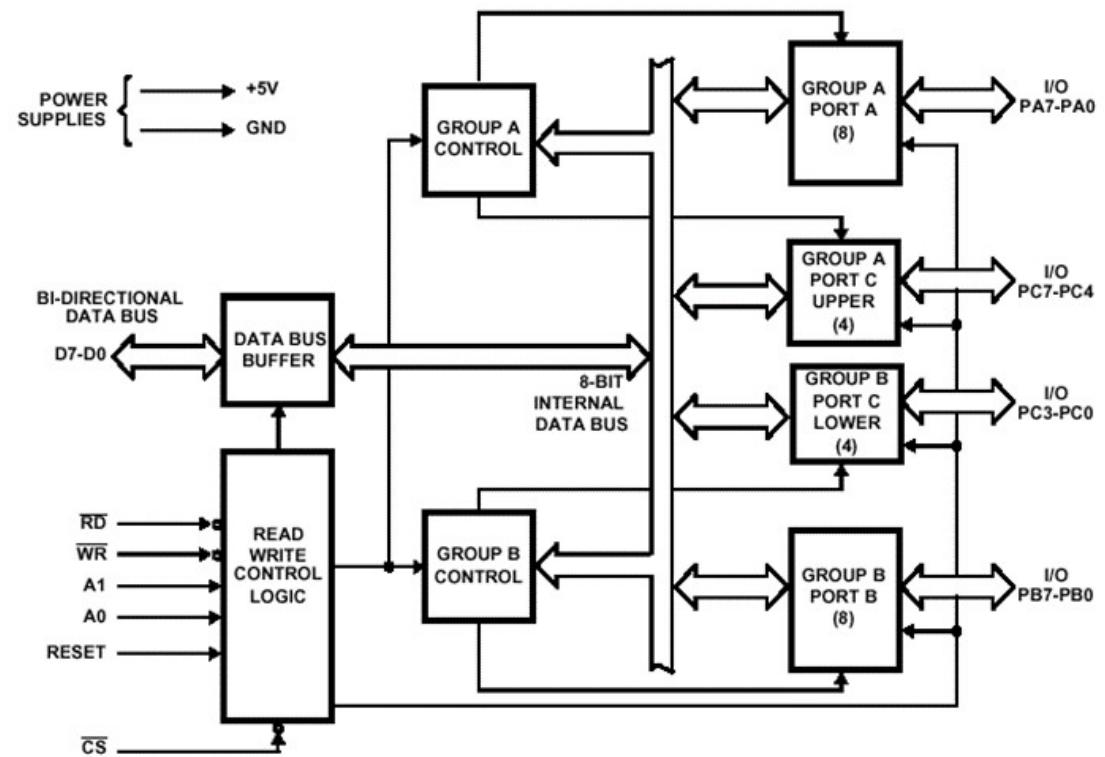
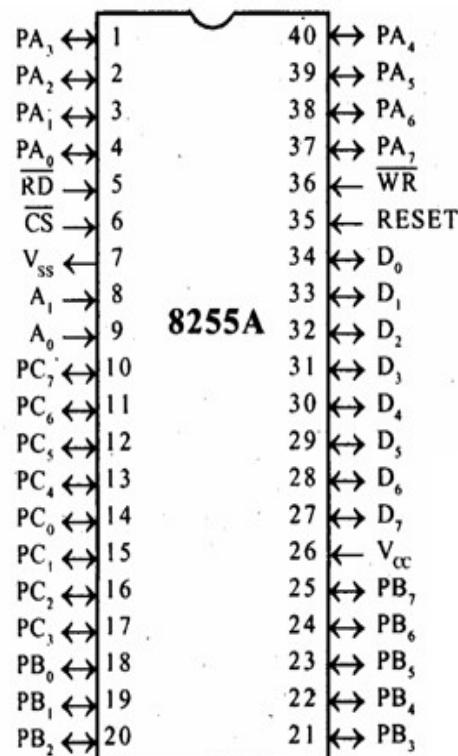
Example

```
;TOGGLING ALL BITS OF PORT 310H EVERY 0.5 SEC
        MOV     DX,310H
HERE:    MOV     AL,55H ;toggle all bits
        OUT     DX,AL
        MOV     CX,33144 ;delay=33144x15.085 us=0.5 sec
        CALL    TDELAY
        MOV     AL,0AAH
        OUT     DX,AL
        MOV     CX,33144
        CALL    TDELAY
        JMP    HERE
```

```
;CX=COUNT OF 15.085 MICROSEC
TDELAY      PROC NEAR
            PUSH AX          ;save AX
W1:         IN   AL,61H
            AND  AL,00010000B
            CMP   AL,AH
            JE    W1          ;wait for 15.085 usec
            MOV   AH,AL
            LOOP W1          ;another 15.085 usec
            POP   AX          ;restore AX
            RET
TDELAY      ENDP
```

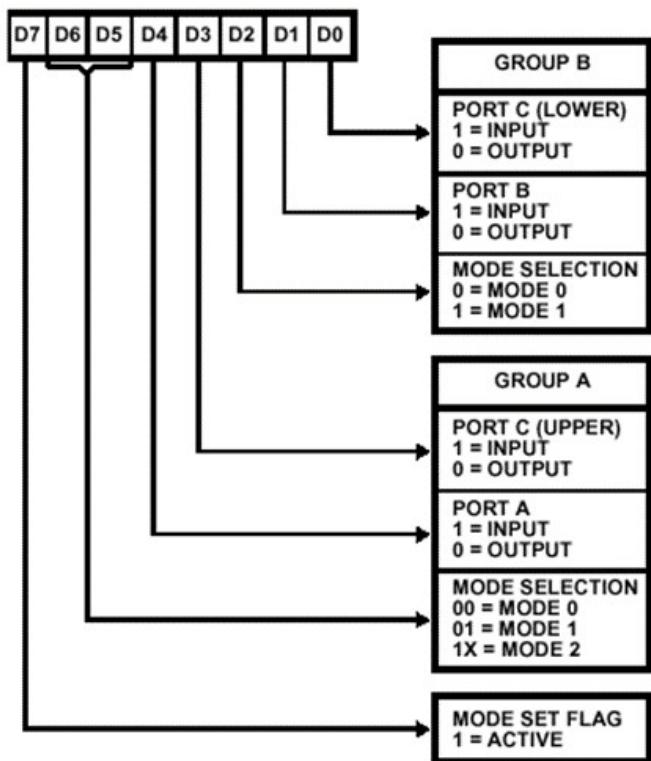
Absolute Vs. Linear Select Address Decoding

Programming & Interfacing the 8255 PPI

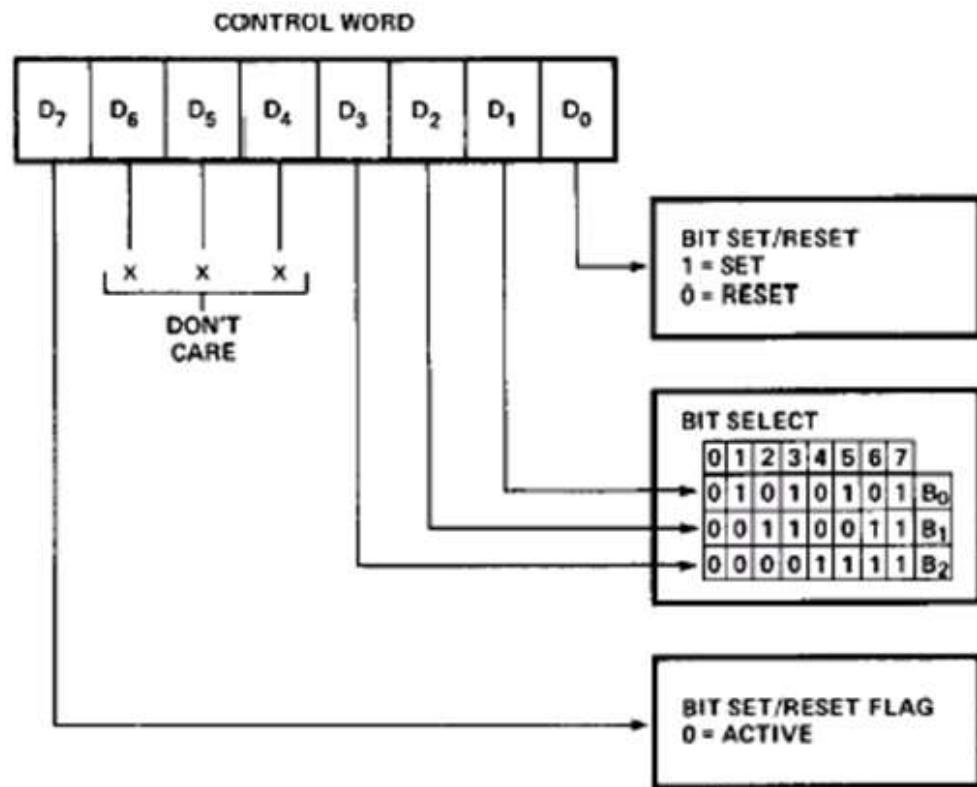


Input Output Mode

CONTROL WORD



BSR- Bit Set Reset Mode



Module 4 (10 Hours)

Microprocessors versus Microcontrollers,

ARM Embedded Systems: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software,

ARM Processor Fundamentals : Registers , Current Program Status Register , Pipeline, Exceptions, Interrupts, and the Vector Table , Core Extensions

Text book 2:Ch 1:1.1 to 1.4, Ch 2:2.1 to 2.5

ARM Embedded Systems

CISC Approach

- “MULTIPLY”
- Loads two values from the memory into separate registers .
- Multiplies the operands in the execution unit, and then stores the product in the appropriate register.
- The entire task of multiplying two numbers is completed with one instruction

RISC Approach

- RISC processors only use simple instructions that can be executed within one clock cycle.
- Thus MULTIPLY Instruction could be broken down into three parts
 - "LOAD," which moves data from the memory bank to a register
 - "PRODUCT," which finds the product of two operands located within the registers
 - "STORE," which moves data from a register to the memory banks.

CISC Advantages

Designed to decrease the memory cost –

1. Large programs need more storage, thus increasing the memory cost and large memory becomes more expensive.
2. To solve these problems, CISC attempts to minimize the number of instructions per program.
3. This can be reduced by embedding the number of operations in a single instruction, thereby making the instructions more complex.
4. The length of the code is relatively short, very little RAM is required to store instructions.
5. But this leads to increase in the number of cycles per instruction.
6. Compiler has to do very little work to translate a high-level language statement into assembly
7. The emphasis is put on building complex instructions directly into the hardware.

RISC Advantages

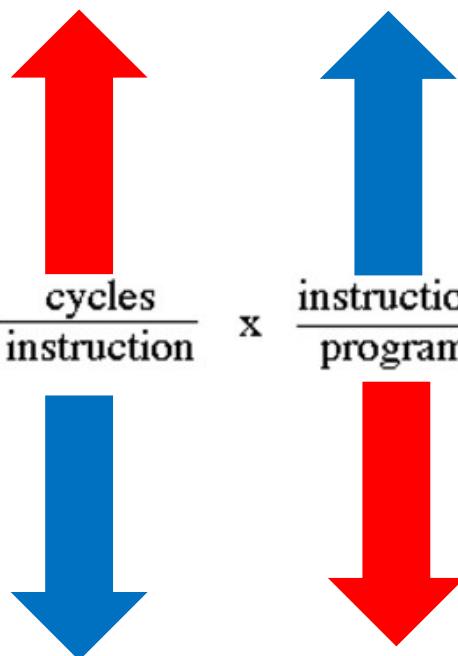
- Each instruction requires only one clock cycle to execute.
- These RISC Instructions require less hardware space than the complex instructions, leaving more room for general purpose registers.
- Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible. (In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.)
- Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform.
- After a CISC-style instruction is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.
- Emphasis on Compiler Complexity.
- Better Suited for RT Systems

Performance Equation

CISC

RISC

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$



The RISC Design Philosophy

The design philosophy aimed at delivering the following :

1. Simple but powerful instructions
2. Single cycle execution at a high clock speed
3. Intelligence in software rather than hardware (reduce complexity of instruction execution on H/W)
4. Provide greater flexibility on reducing the complexity of instructions.

RISC Design Philosophy

Instructions

Pipelines

Registers

Load-Store Architecture

RISC Philosophy – Instructions

Instructions –

1. RISC processors have a reduced number of instruction classes.
2. These instruction classes provide simple operations that can each execute in a single cycle.
3. The compiler or programmer synthesizes complicated operations (a divide operation) by combining several simple instructions.
4. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
5. (In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.)

RISC Philosophy – Pipelines

Pipelines –

- The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines.
- Ideally the pipeline advances by one step on each cycle for maximum throughput.
- There is no need for an instruction to be executed by a mini program called microcode as on CISC processors.

RISC Philosophy - Registers

Registers—

- RISC machines have a large general-purpose register set.
- Any register can contain either data or an address.
- (In contrast, CISC processors have dedicated registers for specific purposes.)

RISC Philosophy – Load Store Architecture

Load-store architecture—

- The processor operates on data held in registers.
- Separate load and store instructions transfer data between the register bank and external memory.
- Memory Accesses are costly, Thus separation of memory access from data processing is an advantage.
- Data held in register banks can be used multiple times without the need of memory access.

The ARM Design Philosophy

There are a number of physical features that have driven the ARM processor design.

1. Designed to be small to reduce power consumption and extend battery operation
2. High code density (Limited memory due to Cost/Size restrictions)
3. Price sensitive and use slow and low-cost memory devices.
4. Reduce the area of the die taken up by the embedded processor.
5. Hardware debug technology
6. ARM core is not a pure RISC architecture – Total Effective System Performance & Power Consumption

Instruction Set for Embedded Systems

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

Variable cycle execution for certain instructions—

Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred.

Inline barrel shifter leading to more complex instructions—

The inline barrel shifter is a hardware component that pre processes one of the input registers before it is used by an instruction.

Instruction Set for Embedded Systems

Thumb 16-bit instruction set—

ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions.

Conditional execution—

An instruction is only executed when a specific condition has been satisfied.

Enhanced instructions—

The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16 -bit multiplier operations and saturation.

Embedded Systems

An **embedded system** is a computer **system** with a dedicated function within a larger mechanical or electrical **system**, often with real-time computing constraints. It is **embedded** as part of a complete device often including hardware and mechanical parts.

Embedded systems are combinations of hardware and software that perform a specific function or perform specific functions within a larger system.

Embedded System Hardware

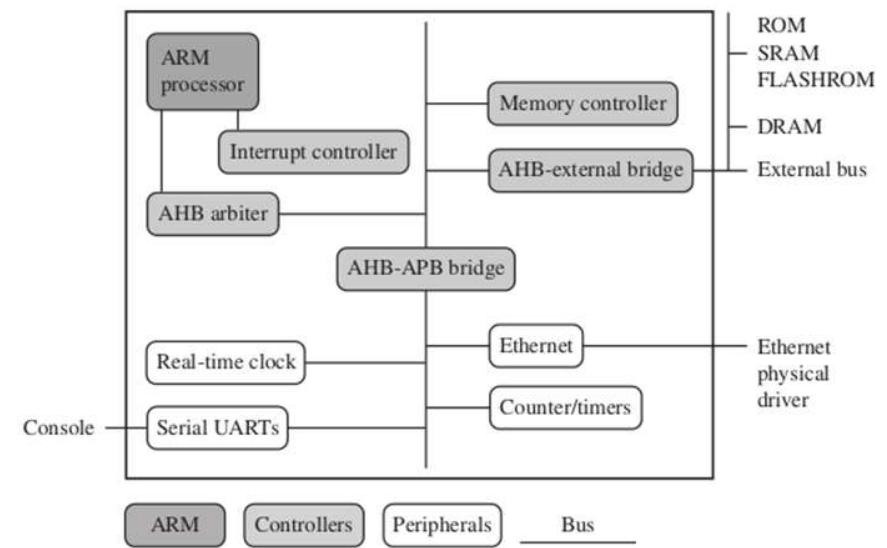
The ARM processor controls the embedded device.

An ARM processor comprises a core plus the surrounding components that interface it with a bus. These components can include memory management and caches.

Controllers co-ordinate important functional blocks of the system. Two commonly found controllers are interrupt controller and memory controller.

The peripherals provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.

A bus is used to communicate between different parts of the device.



An example of an ARM-based embedded device, a microcontroller.

Embedded System Hardware/Software

HARDWARE

AMBA Bus Technology
AMBA Bus Protocol
Memory
Peripherals

SOFTWARE

Initialization Code(Boot Code)
Operating System
Applications

ARM 7 TDMI

ARM Holdings neither manufactures nor sells CPU Devices

It Licenses the processor architecture to interested parties.

ARM7TDMI

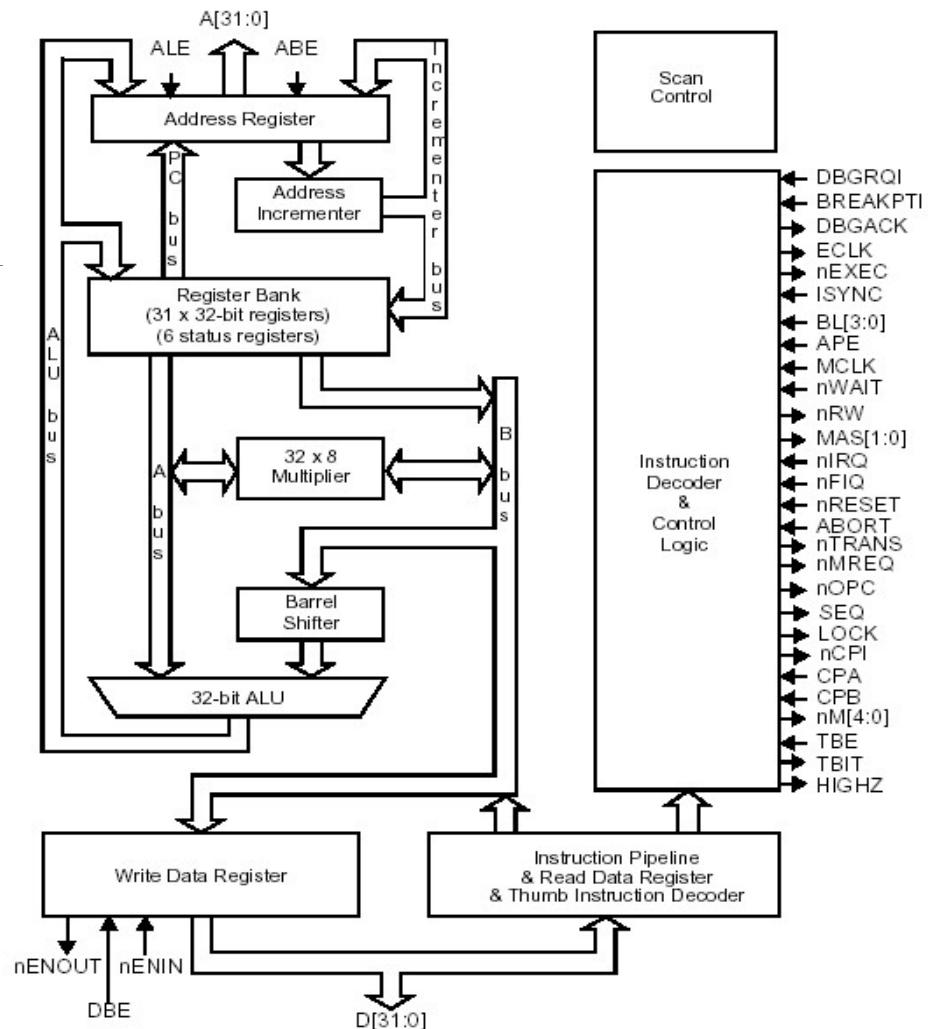
ARM7 - Core

T - 16 Bit Thumb

D – JTAG Debug

M – fast Multiplier

I – embedded ICE Macrocell (In-Circuit Emulator)



ARM Bus Technology

Embedded systems use different bus technologies.

The most common PC bus technology, the Peripheral Component Interconnect (PCI) bus, this type of technology is external or off-chip.

Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

There are two different classes of devices attached to the bus.

Bus master(ARM processor core)—a logical device capable of initiating a data transfer with another device across the same bus.

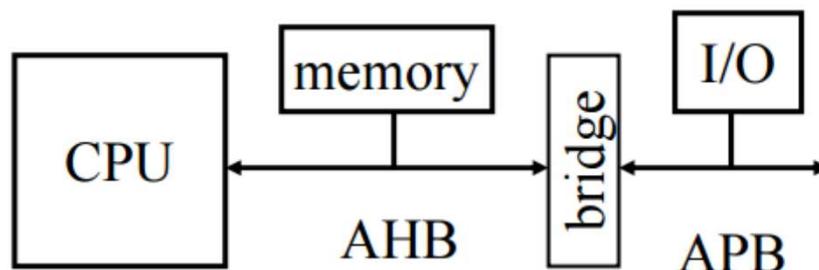
Bus slaves(Peripherals)—logical devices capable only of responding to a transfer request from a bus master device.

ARM Bus Technology

A bus has two architecture levels.

Physical level — that covers the electrical characteristics and bus width (16, 32, or 64 bits).

Second level (deals with protocol)—the logical rules that govern the communication between the processor and a peripheral.



A.M.B.A. Bus Protocol

The Advanced Microcontroller Bus Architecture (AMBA) has been widely adopted as the on-chip bus architecture used for ARM processors.

The first AMBA buses introduced were the

- 1. ARM System Bus(ASB)**
- 2. ARM Peripheral Bus(APB) - for slower peripherals**

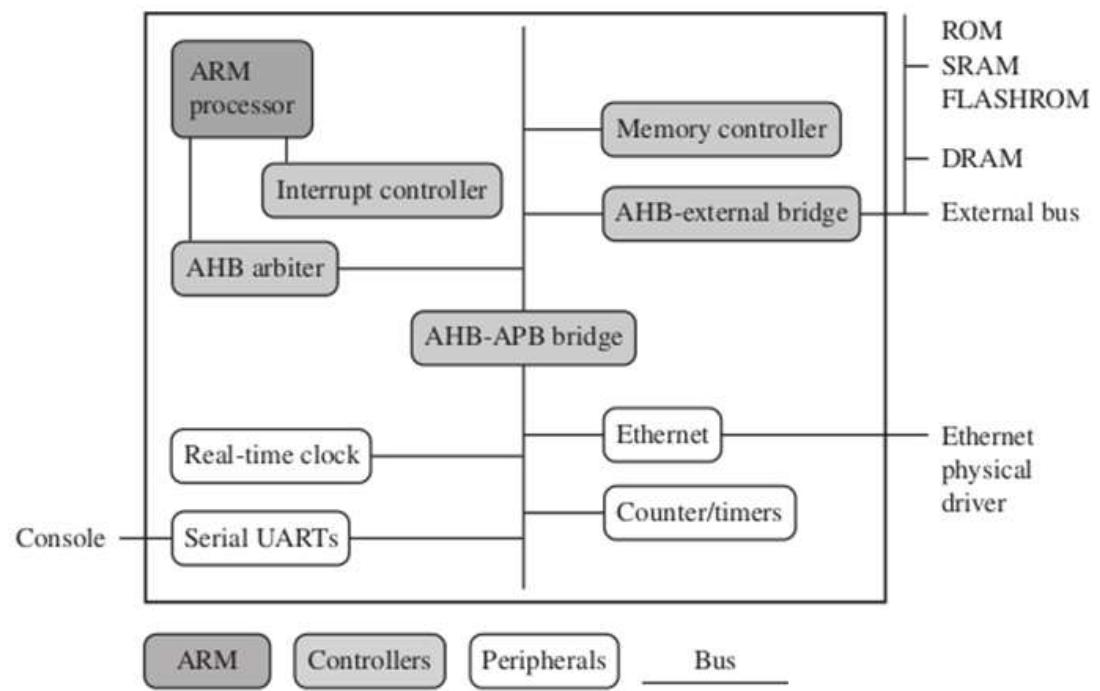
Later another bus design was introduced

- 3. ARM High Performance Bus(AHB)**

External Bus- Proprietary to this device.

AHB-External Bridge

AHB-APB Bridge



An example of an ARM-based embedded device, a microcontroller.

A.M.B.A. Bus Protocol

Plug-and-play interface for hardware developers —

Using AMBA, peripheral designers can reuse the same design on multiple projects.

A peripheral can simply be bolted onto the on-chip bus without having to redesign an interface for each different processor architecture.

A.M.B.A. Bus Protocol – AHB

Centralized multiplexed bus provides higher data throughput rather than the ASB bidirectional bus design.

- AHB runs at higher clock speed.
- It supports widths of 64 and 128 bits.
- Only 2 Bus Cycles- Address Phase & Data Phase
- Access to target device is controlled through MUX-
 - Multi-Layer AHB allows multiple active bus masters.
 - In such cases admit bus access to one bus master at a time

A.M.B.A. Bus Protocol – AHB

ARM has introduced two variations on the AHB bus:

1. Multi-layer AHB

Multi-layer AHB bus allows multiple active bus masters. (good for systems with multiple processors)

2. AHB-Lite.

AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.

Memory

An embedded system has to have some form of memory to store and execute code.

It has some specific memory characteristics, such as

Hierarchy

Width

Type

Deciding Factors : Price Performance & Power Consumption

Memory - Hierarchy

All computer systems have memory arranged in some form of hierarchy.

Memory trade-offs: The fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away.

Cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory.

Cache increases Performances but leads to loss of Predictable Execution Time. Does not help in RT System Response. Thus many small embedded systems do not need cache.

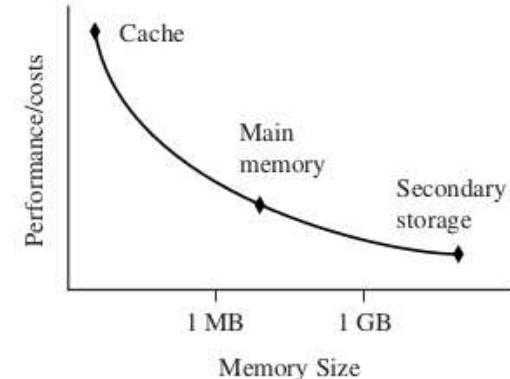
Cache is closer to processor core and more expensive

The **main memory** is around 256 KB to 256 MB or greater , depending on the application.

Generally it is stored in separate chips.

LOAD & STORE instructions access main memory if required values is not in cache.

Secondary Storage is the largest and slowest form of memory may vary from 600 MB to 60 GB.



Storage trade-offs.

Memory - Width

The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits.

Directly affects Overall Performance & Cost Ratio

Use of **thumb instructions** —

If you have an un-cached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction.

Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive.

In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory as a result of the core making only a single fetch to memory to load an instruction.

Memory – Width (Theoretical Cycle Times)

Instruction	size	8-bit memory	16-bit memory	32-bit memory
ARM	32-bit	4 cycles	2 cycles	1 cycle
Thumb	16-bit	2 cycles	1 cycle	1 cycle

Memory - Types

Read-only memory (ROM)

The least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed.

Used in high volume devices that require no Updates or Corrections.

Many devices use a ROM to hold boot code.

Flash ROM

Can be written to as well as read from.

It is slow to write. Its main use is for holding the device firmware and not to hold dynamic data.

The erasing and writing of flash ROM are completely software controlled & does not require additional hardware circuitry.

Memory - Types

Dynamic random access memory (DRAM)

The most commonly used RAM for devices.

It has the lowest cost per megabyte.

DRAM is dynamic. i.e it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.

Static random access memory (SRAM)

It is faster than traditional DRAM . But requires more silicon Area

The SRAM is Static i.e does not require refreshing.

The access time is shorter than equivalent DRAM because SRAM does not require a pause between data accesses.

Higher cost – therefore used for smaller high speed tasks i.e caching & fast memory

Memory - Types

Synchronous dynamic random access memory (SDRAM)

One of the many sub categories of DRAM

Run at much higher clock speeds than conventional memory

Synchronizes itself with processor bus as it is clocked.

Internally the data is fetched from the memory cells, pipelined & finally brought out on the bus in a burst

Peripherals

Outside world interaction of Embedded Systems.

A peripheral device performs input and output functions for the chip by connecting to other devices that are off-chip.

Each peripheral device usually performs a single function.

Peripherals range from a simple serial communication to complex 802.11 wireless device.

All ARM peripherals are memory mapped—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.

Peripherals - Controllers

Specialized peripherals called as Controllers that implement higher levels of functionality . Two important types

Memory controllers

Connect different types of memory to the processor bus.

On power-up a memory controller is configured in hardware to allow certain memory devices to be active.

Some memory devices must be set up by software.

Interrupt controllers

An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are two types of interrupt controller available for the ARM processor:

Standard interrupt controller (SIC)

Vector interrupt controller (VIC)

Standard Interrupt Controller

The SIC sends an interrupt signal to the processor core when an external device requests servicing.

It can be programmed to ignore or mask an individual device or set of devices.

The interrupt handler determines which device requires servicing by reading a device bitmap register in the interrupt controller.

Vector Interrupt Controller

The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts.

After associating a priority and a handler address with each interrupt, the VIC only asserts an interrupt signal to the core if the priority of a new interrupt is higher than the currently executing interrupt handler.

Depending on its type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler for the device from the VIC, or cause the core to jump to the handler for the device directly.

Embedded System Software

An embedded system needs software to drive it.

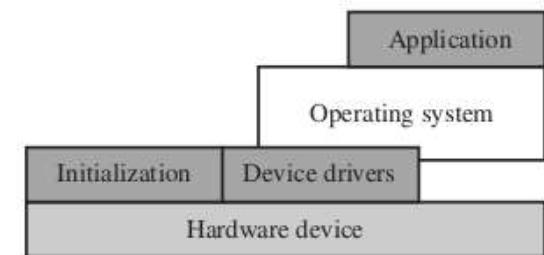
Initialization Code is the first code executed on the board and its sets up the minimum parts of the board before handing control over to the operating system.

Operating System provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system .

Device Drivers provide a consistent software interface to the peripherals on the hardware device.

Application performs one of the tasks required for a device.

Firmware is the software component that can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code).



Software abstraction layers executing on hardware.

Initialization (Boot) Code

Takes the processor from Reset state to a Run state where the operating system can run.

It configures : Memory controller | Processor caches | Initializes some devices

We can group tasks into three phases:

Initial hardware configuration

Setting up the target platform so it can boot an image.

Usually The target platform itself comes up in a standard configuration. May require Memory Remapping

Diagnostics

Tests the system by exercising the hardware target to check if the target is in working order.

It also tracks down standard system-related issues.

The primary purpose of diagnostic code is fault identification and isolation.

Booting

Booting involves loading an image and handing control over to that image.

The boot process itself can be complicated if the system must boot different OS or different versions of the same OS.

Operating System

The initialization process prepares the hardware for an operating system to take control.

An operating system organizes the system resources: the peripherals, memory, and processing time.

ARM processors support over 50 operating systems.

We can divide operating systems into two main categories:

RTOS – RTLinux , VxWorks, Windows CE

Platform OS – Windows, MAC OSX, Ubuntu etc.

RTOS

RTOS- provide guaranteed response times to events.

Different RTOS systems have different amounts of control over the system response time.

A hard real-time application requires a guaranteed response to work at all.

Performance Degrades immediately if deadline is missed. And System fails.

Good Response Time is the main criteria.

In contrast in Soft Real Time Systems, the system performance degrades gracefully when deadline is missed.

Firm Real Time Systems-The system does not fail if the deadline is missed a few times, but result may be of no use if deadline is missed.

Platform OS

Platform OS – User Convenience

Platform operating systems require a memory management unit to manage large, non-real time applications and tend to have secondary storage.

The Linux operating system is a typical example of a platform operating system.

Windows

Response Time is not the primary criteria. User Convenience is primary criteria.

Applications

An application implements a processing task

The operating system controls the environment. Schedules Tasks for execution

An embedded system can have one active application or several applications running simultaneously.

In contrast, ARM processors are not found in applications that require leading-edge high performance.

Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

Summary

ARM uses modified RISC design philosophy that targets good code density & low power consumption

An embedded system consists of a processor core surrounded by caches, memory & peripherals.

The system is controlled by an OS that manages application tasks.

Key points of RISC Design Philosophy:

Improve performance by reducing instruction complexity, speed up instruction processing by using pipeline, providing large register set to store data near the core & to use a Load-Store Architecture

ARM also uses certain NON-RISC ideas

Allows variable cycle execution on certain instructions to save power area & code size

Adds Barrel shifter to expand capability of certain instructions

Uses Thumb instruction set to improve code density

Improves code density & performance by conditionally executing instructions

It includes enhanced instructions to perform DSP type functions

Summary

ARM Processors are found embedded in chips

Peripherals are accessed through Memory Mapped Registers

Controller is a special type of peripheral used by embedded systems to configure higher level functions like memory & interrupts

AMBA on-chip bus is used to connect peripherals & processor together.

Embedded System also includes software components:

Initialization Code: Configures hardware to known state

Operating Systems: once configured OS is loaded & executed for the use of hardware resources & infrastructure

Device Drivers: provide standard interface to peripherals

Application: Perform task specific duties of an embedded system

Arm Processor Fundamentals

Introduction

ARM: Advanced RISC Machine

Leading Provider of 16/32 Bit Embedded RISC Systems

Licences its High Performance, Low Cost, Power Efficient RISC Processors , Peripherals & System-Chip Designs to leading international electronics companies.

Software development tools – RealView & Keil

ARM does not make Ics

ARM grants license of core to different silicon vendors like ATMEL,NXP, Cirrus Logic etc

Companies make the IC eg: LPC 2148 from NXP, AT91RM9200 from ATMEL

ARM Processors mainly used in Handheld devices, Robotics, automation, consumer electronics

ARMO processors are available for almost every domain

Products using ARM

iPhone

iPod

Nvidia graphic cards

Lego Mindstorm Robots

Etc...

ARM Features

RISC Processor

Large Register file R0 to R16

Load & Store Architecture – Data processing is only in Register contents

Uniform & Fixed length instruction (with a few multi cycle instructions)

32-Bit ARM Instructions

16-Bit Thumb Instructions

Good Speed & Power Consumption Ratio

High Code Density

Mostly Single Cycle execution

DSP enhanced instructions

Conditional Execution of all instructions

32-Bit Barrel Shifter

In built circuit for Debugging

Arm NOMENCLATURE

ARMxyzTDMIEJFS

x: Series

y: MMU

z: Cache

T: Thumb

D: Debugger

M: Multiplier – Hardware Multiplier Unit does multiplication

I: Embedded ICE Macrocell – H/W circuit used to generate Trace information for advanced debugging

E: Enhanced Instructions – Eg: DSP

J: Java Acceleration by Jazelle – H/W circuit used to run Java Byte Code

F: Vector Floating Point – Hardwired implementation of Floating Point Operations

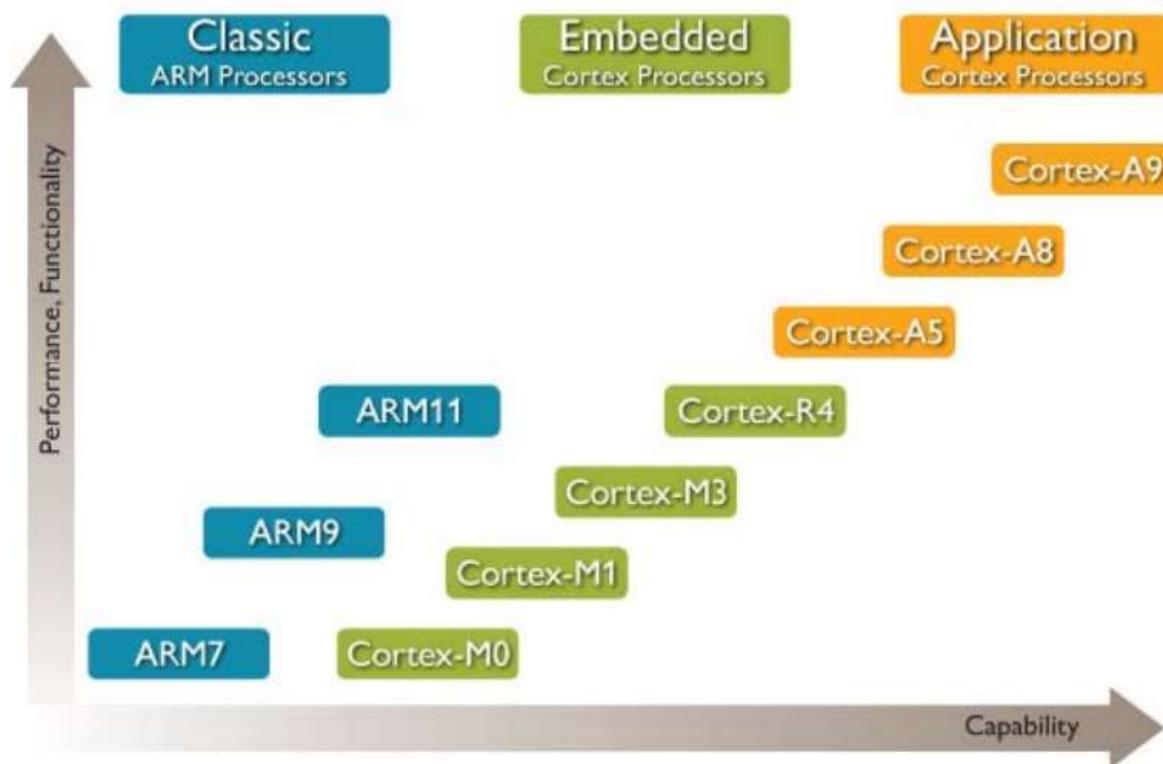
S: Synthesizable Version – ARM Architecture can be modified

ARM7TDMI

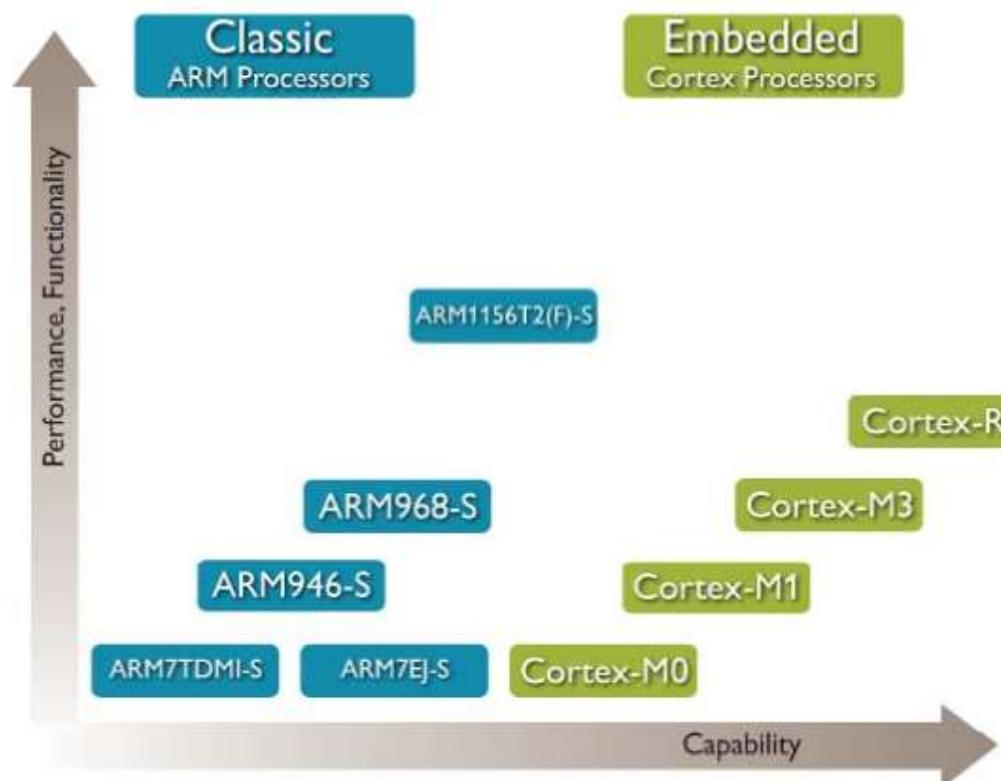
ARM7 family processor which has Thumb Instruction Set, Debug Unit, MMU, Trace Circuit is inside the core.

This is the basic core and all cores have TDMI

ARM FAMILY



ARM Family



ARM – Where to start

LPC214x

ARM7TDMI Family

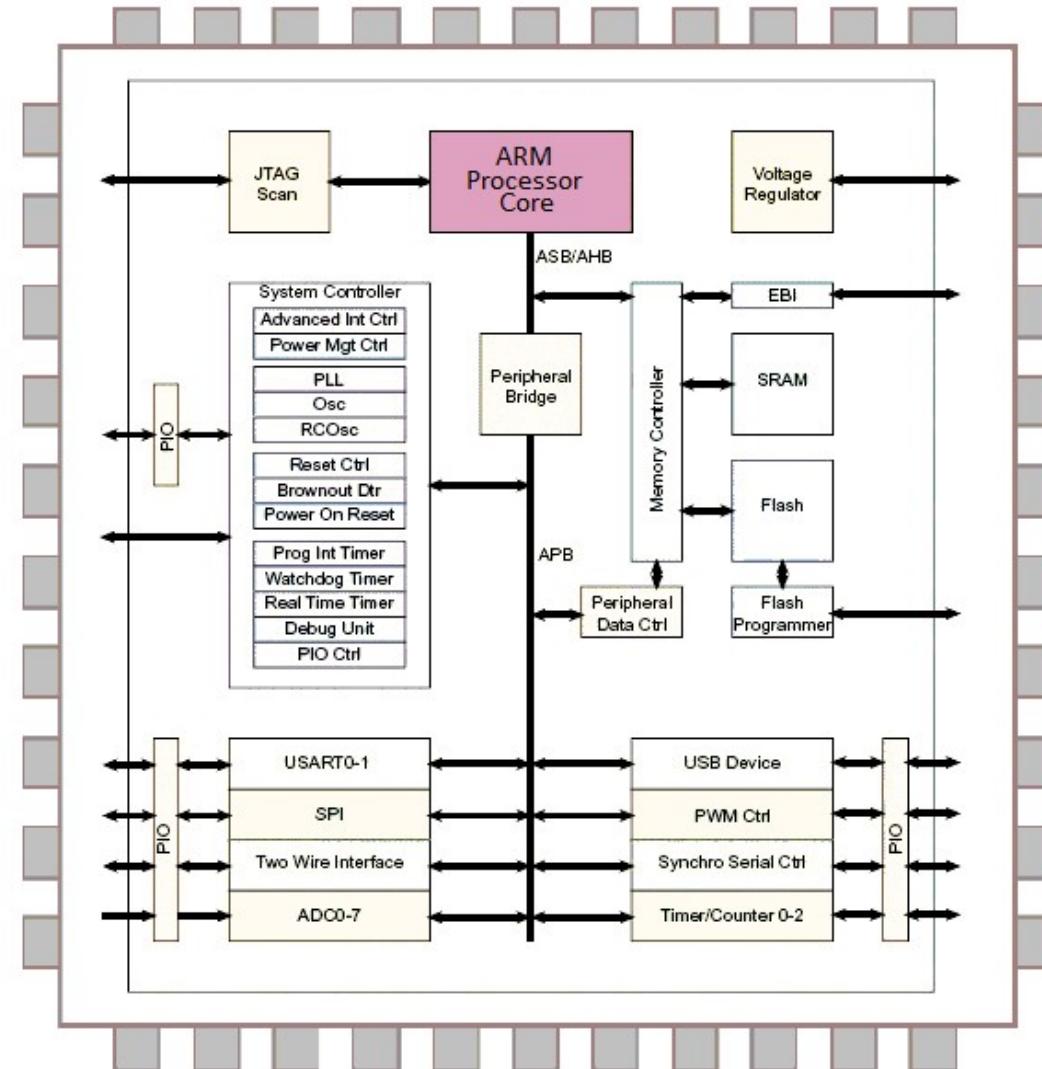
Best for entry point feature wise

Free development toolchain is available(Open source community & software vendors)

Development boards easily available in market

Support for RTOS

ARM CONTROLLER



ARM Core Dataflow

Programmer can think of ARM core as functional units connected by data buses

Arrows represent the flow of data

Lines represent the buses

Boxes represent either an operation unit or storage area

Figures shows data flow as well as abstract components that make up an ARM Core

Arm Core Data Flow Model

Data enters Processor Core through DATA BUS

Data may be an instruction to execute or a Data Item

Figure shows a Von-Neumann implementation of ARM – Data & Instructions share the same bus. Harvard implementation uses separate bus for Data & Instructions

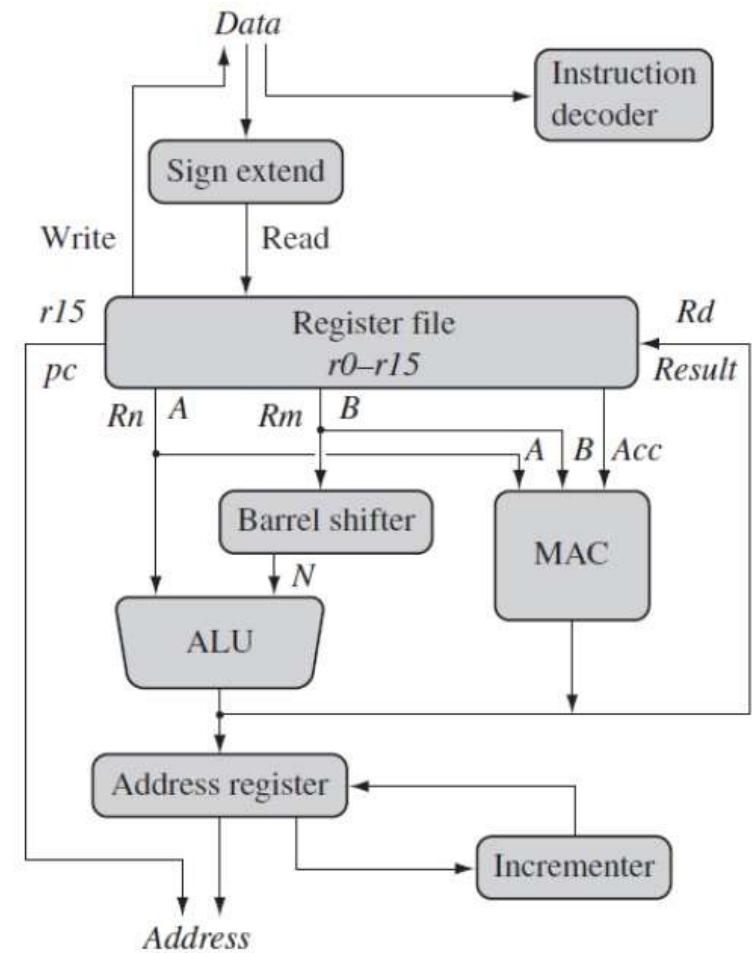
Instruction decoder- translates instructions before they are executed. Each instruction belongs to a particular instruction set

ARM Processor uses LOAD-STORE Architecture

There are no Data processing instructions that directly manipulate data in memory.

(happens only in Registers)

Data Items are placed in the Register File- A Storage Bank made up of 32-Bit registers



Arm Core Data Flow Mode

Since the ARM Core is a 32-Bit Processor most instructions treat the registers as holding signed or unsigned 32-Bit values

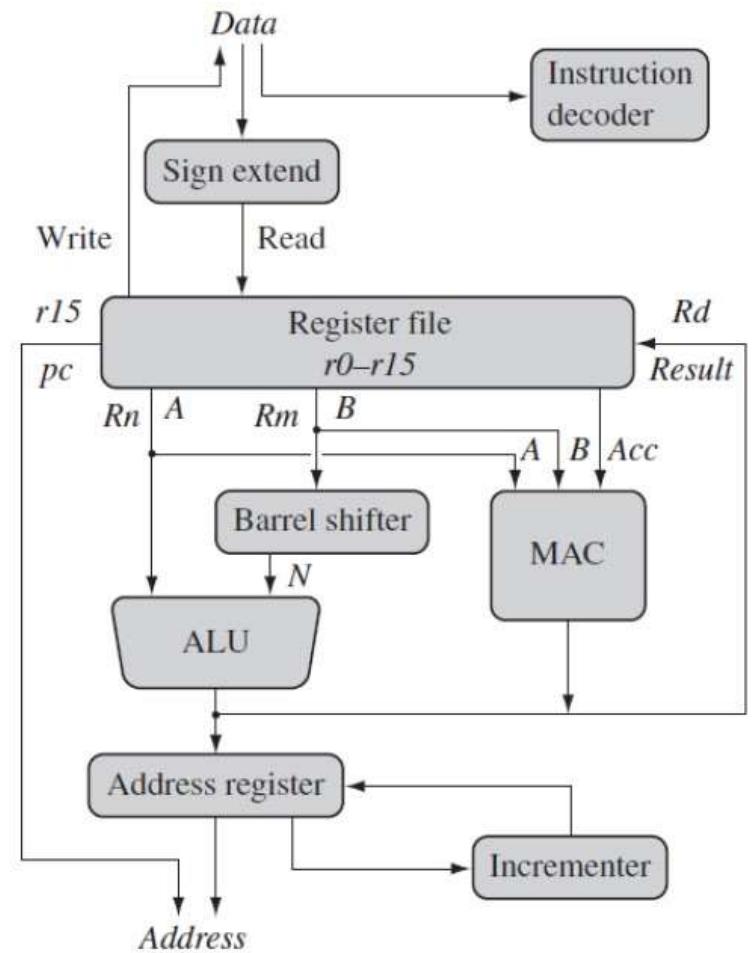
The Sign Extend Hardware converts signed 8-Bit & 16-Bit numbers to 32-Bit values as they are read from memory & placed in a Register

ARM instructions typically have 2 Source Registers Rn & Rm and a single Result or Destination Register Rd.

Source Operands are read from the Register File using the internal buses A & B

ALU or MAC(Multiply-Accumulate Unit) takes the Register values Rn & Rm from A & B buses & computes a result.

Data Processing instructions write the result into Rd directly to the Register File



Arm Core Data Flow Mode

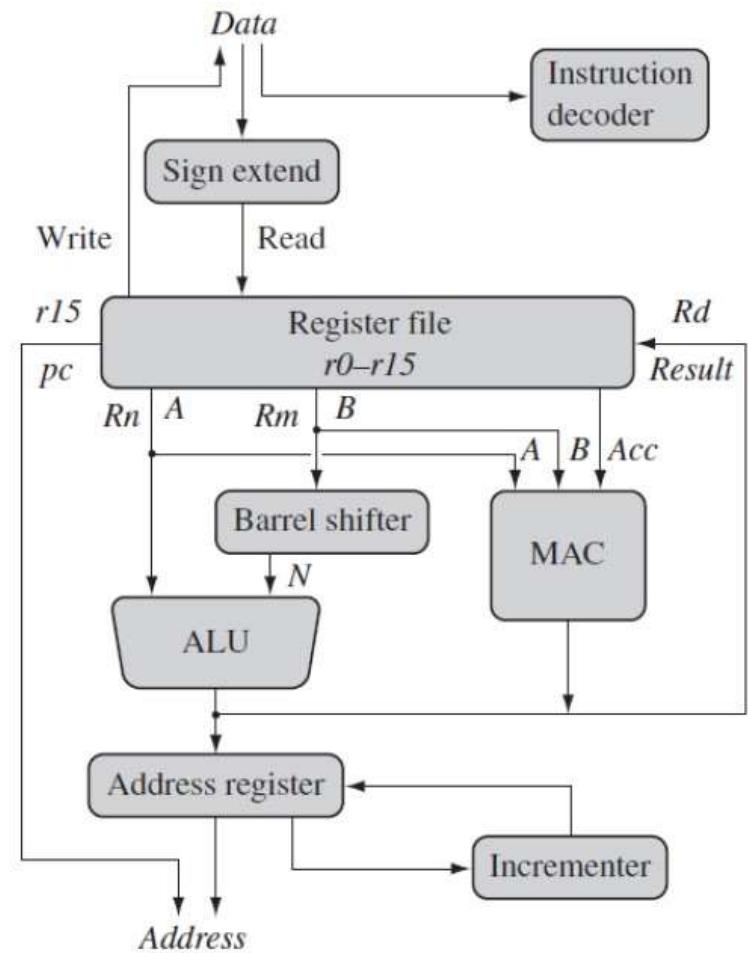
LOAD & STORE instructions use the ALU to generate an address to be held in the address Register & broadcast on the Address Bus

Register Rm alternatively can be pre-processed in the Barrel Shifter before it enters the ALU

Result in Rd is written back to the Register File using the Result Bus

For LOAD & STORE instructions incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

Processor continues execution until an exception or interrupt changes the normal execution flow.



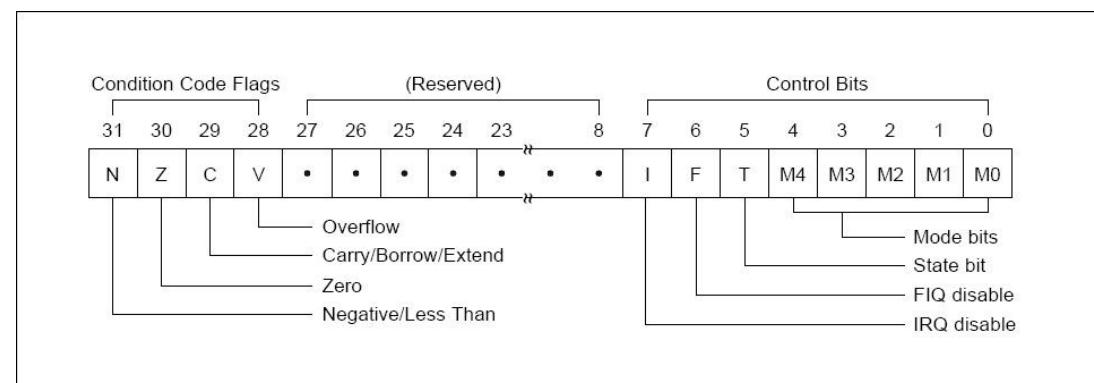
Registers

General registers and Program Counter

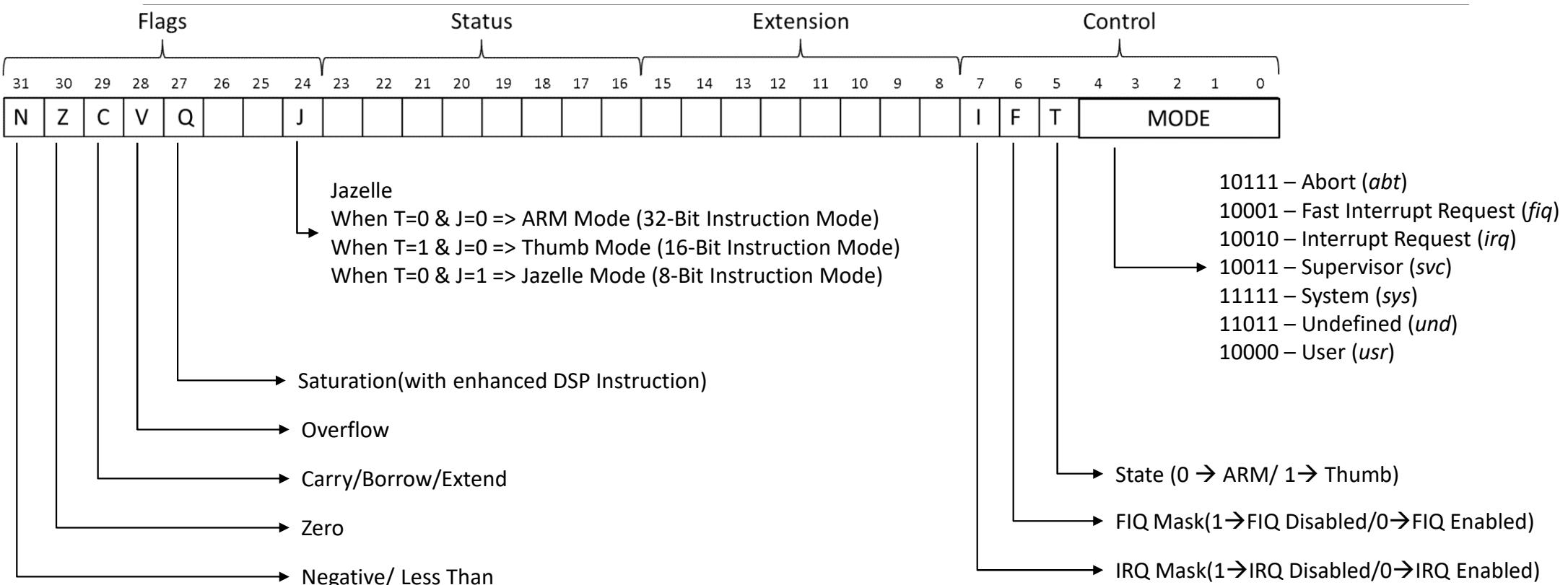
User32 / System	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8 fiq	r8	r8	r8	r8
r9	r9 fiq	r9	r9	r9	r9
r10	r10 fiq	r10	r10	r10	r10
r11	r11 fiq	r11	r11	r11	r11
r12	r12 fiq	r12	r12	r12	r12
r13 (sp)	r13 fiq	r13 svc	r13 abt	r13 irq	r13 undef
r14 (lr)	r14 fiq	r14 svc	r14 abt	r14 irq	r14 undef
r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)

Program Status Registers

cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
spsr_fiq	spsr_svc	spsr_abt	spsr_irq	spsr_undef	



A Generic Program Status Register(PSR)



Registers

Register File- Contains all the Register available to programmer(depends on current mode)

General Purpose Register- Hold Data or Address

Identified by label r prefixed to register number

Processor operates in 7 different modes

All Registers are 32 bits in size

18 Active Registers: 16 Data Registers and 2 Processor Status Registers

Data Registers visible to Programmer as r0 to r15

Registers

Special Function Register – r13,r14 and r15

r13 – Stack Pointer (sp) – Stores Head of the Stack in Current Processor Mode

r14 – Link Register (lr) – Core stores the Return Address whenever it calls a subroutine. IF the function is going to call another function then LR must be preserved on the stack i.e r13

r15 – Program Counter (pc) – Contains the address of the next instruction to be executed.

Depending on context r13 and r14 can also be used as GPR

Using r13 as GPR is dangerous when processor is running any form of OS because any OS often assumes r13 points to a valid stack frame.

CPSR – Current Program Status Register

SPSR – Saved Program Status Register

CPSR

Monitor & Control Internal Operations

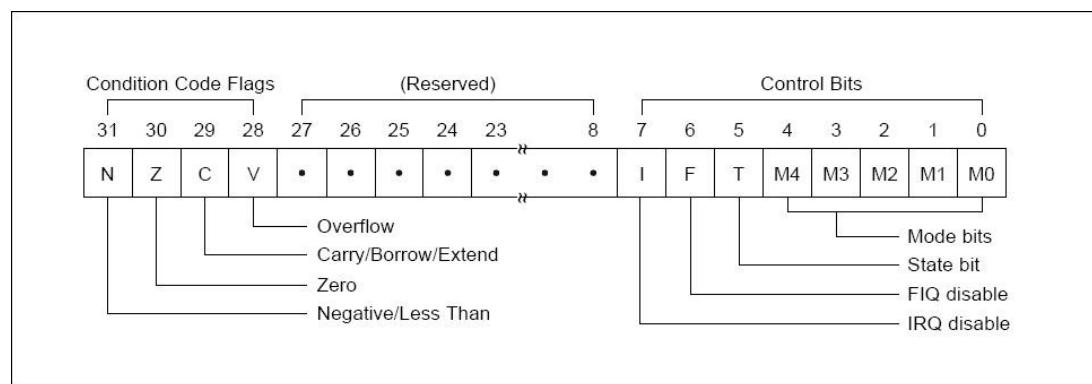
Dedicated 32-Bit Register

Resides in Register File

4 Fields, Each 8-Bit wide:

1. Flags – Conditional Flags
2. Status (reserved for future use)
3. Extension (reserved for future use)
4. Control – Processor Mode / State / Interrupt Mask Bits

Some cores have Additional Bits Eg: J Bit () in flag field in Jazelle enabled Processors which execute 8-Bit instructions.(Executes BYTE CODE)



Processor Modes

Determines which Registers are active for that Mode

Access Rights to CPSR

Each Mode is either

Privileged Mode - Full R/W access to CPSR

Non-Privileged Mode – READ Only Access to CPSR Control Field. But still allows R/W access to Conditional Flags

7 Modes-

6 Privileged – *abort, fast interrupt request, interrupt request, supervisor, system & undefined*

1 Non-Privileged - *user*

Processor Modes

Abort – entered when there is a failed attempt to access memory

Fast interrupt request

Interrupt request



Corresponds to two interrupt levels available on the ARM processor

Supervisor – Mode that the processor is in after Reset. Mode the OS operates in

System – Special User mode that allows full R/W access to CPSR

Undefined – Entered when Processor encounters a instruction that is not defined or not supported by the implementation

User – used for programs & Applications

Banked Registers

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7_fiq	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Denoted by _mode

Banked Registers available only in certain modes of Processor



Banked Registers

Abort mode: **r13_abt, r14_abt, spsr_abt , r0-r12,r15,cpsr**

Undefined mode: **r13_undef, r14_undef, spsr_undef, r0-r12,r15,cpsr**

Supervisor: **r13_svc, r14_svc, spsr_svc, r0-r12,r15,cpsr**

Interrupt Request: **r13_irq, r14_irq, spsr_irq, r0-r12,r15,cpsr**

Fast Interrupt Request: **r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, spsr_fiq , r0-r7,r15,cpsr**

State & instruction sets

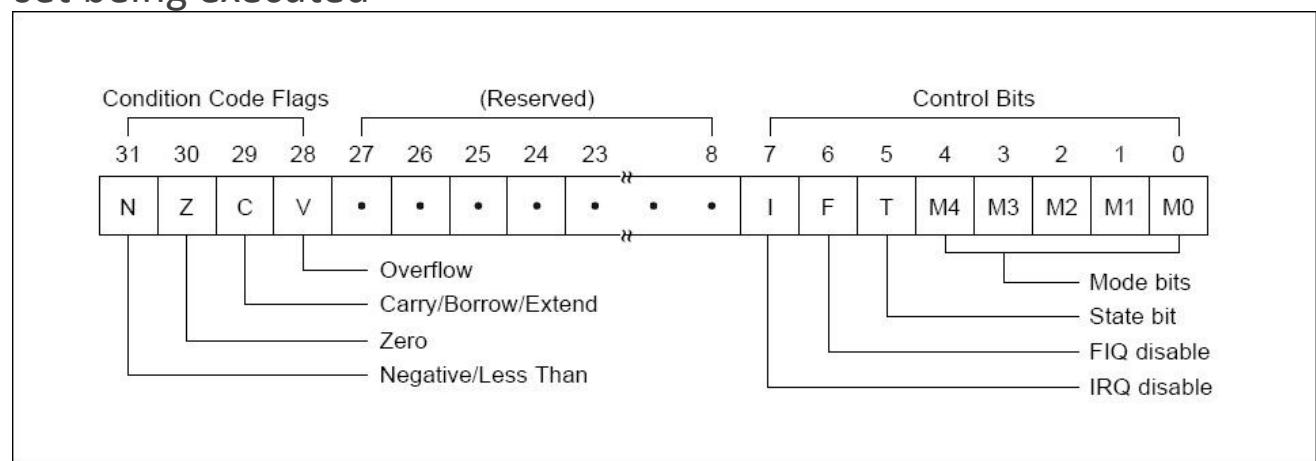
State determines the instruction set being executed

3 instruction sets:

32-Bit ARM instruction set

16-Bit Thumb instruction set

8-Bit Jazelle instruction set



Once set to a state the Processor executes the instruction set for that state.

Interrupt Masks

Used to stop specific interrupt requests from interrupting the processor

2 interrupt request levels on ARM

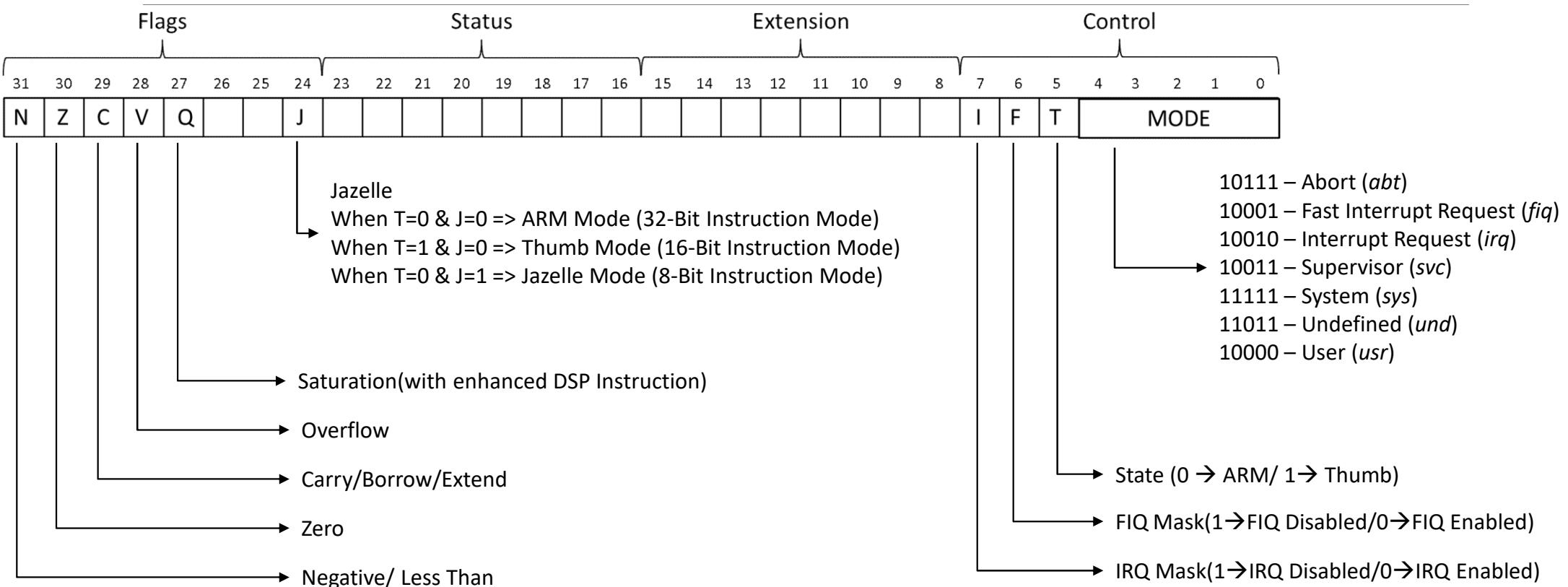
IRQ

FIQ

2 Interrupt Mask Bits I & F which masks IRQ or FIQ

MASKS when set to 1

A Generic Program Status Register(PSR)



Conditional Flags

Updated by Comparisons & the result of ALU operations that specify S instruction Suffix

Eg: SUBS if a result of subtraction is 0, then Z flag in the CPSR is set.

Q – Saturation /set when result causes overflow/saturation

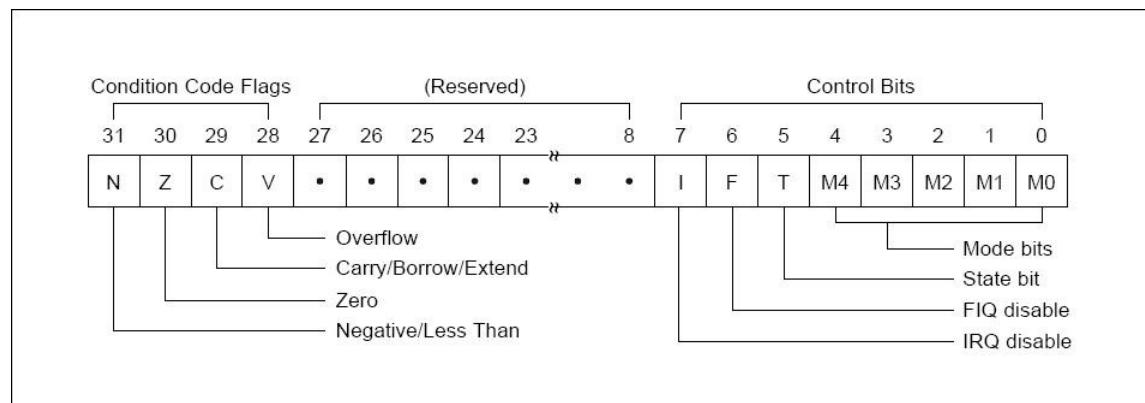
(sticky-set by h/w. to clear write to CPSR)

V – oVerflow /set when result causes signed overflow

C – Carry /set when result causes an unsigned carry

Z – Zero /set when result is 0

N – Negative /set when bit 31 of the result is a binary 1



Conditional Execution

Control whether core will execute an instruction or not

Most instructions have a condition attribute

Execution based on setting of Condition Flags

Prior to execution, the processor compares the condition attribute with Condition flag in CPSR

IF they match instruction is executed else it is ignored

Condition attribute is post fixed to the mnemonic, which is encoded into the instruction

When condition mnemonic is not present, the default behaviour is to set AL(ALways execute)

Conditional execution mnemonics

EQ - equal

NE – not equal

CS HS – carry set/
unsigned higher or same

CC LO – carry
clear/unsigned lower

MI – minus/negative

PL – plus/positive or zero

VS – overflow set

VC – overflow clear(no
overflow)

HI – unsigned higher

LS – unsigned lower or
same

GE – signed greater than
or equal

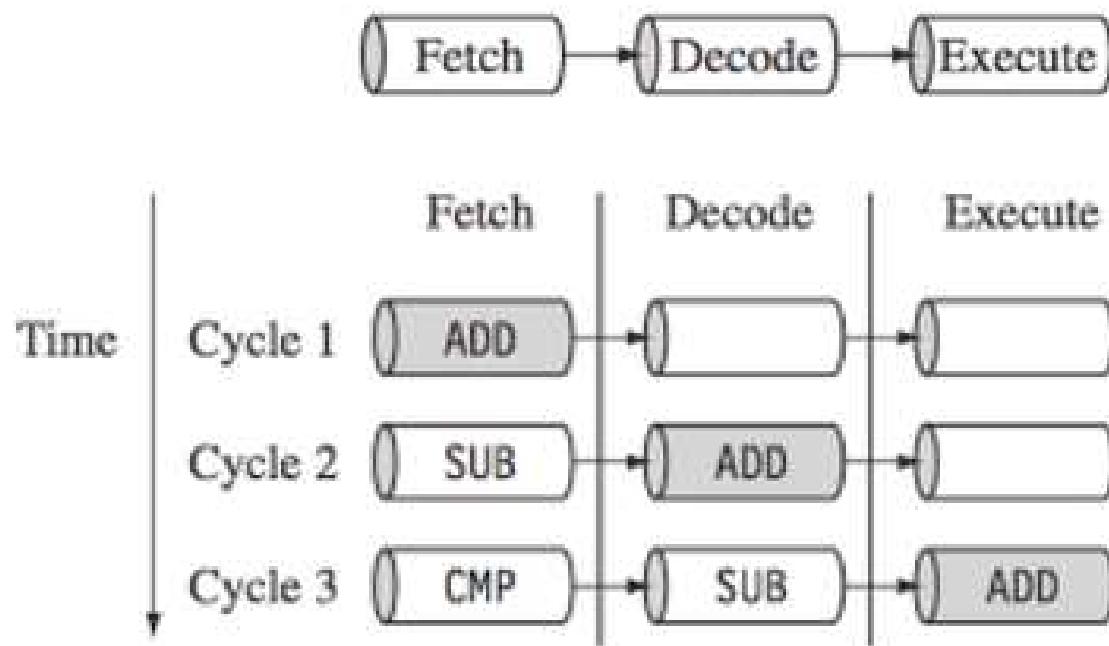
LT – signed less than

GT – signed greater than

LE – signed less than or
equal

AL - always

ARM Pipeline – 3 Stage



- Pipeline is flushed if branch instruction is encountered

Pipeline

As pipeline length increases, the amount of work done at each stage is reduced.

This enables processor to attain a higher operating frequency. This increases performance

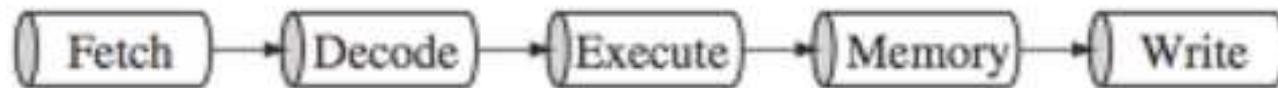
System Latency increases – more cycles taken to fill pipeline before core executes

Data dependency between certain stages

Write code to reduce the dependency using *instruction scheduling*.

Pipeline

ARM 9 – Fetch Decode Execute Memory Write – increase in instruction throughput by 13%



ARM 10 – Fetch Issue Decode Execute Memory Write – increase in instruction throughput by 34%



Code written for ARM7 will work on Arm 9 & ARM 11

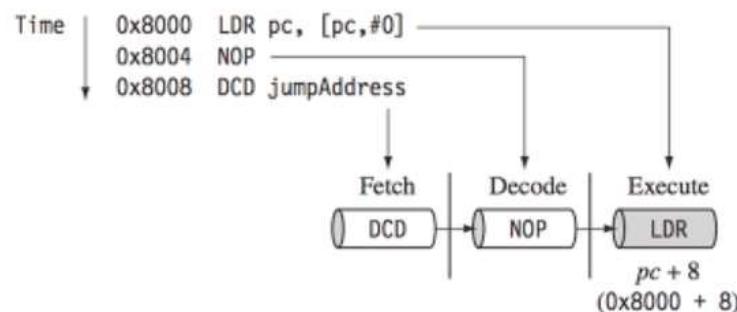
Pipeline Executing Characteristics

ARM pipeline has not processed an instruction until it has completely passed the execution stage

Eg: An ARM7 3 Stage Pipeline has executed an instruction only when the 4th instruction is fetched

In the Execute stage PC always points to the address of the instruction being executed plus 8 Bytes .i.e PC is running 8 bytes (2 instructions) ahead of the current instruction being executed.(care must be taken when calculating offsets used in PC relative addressing)

In Thumb Mode, PC is current instruction plus 4 Bytes.

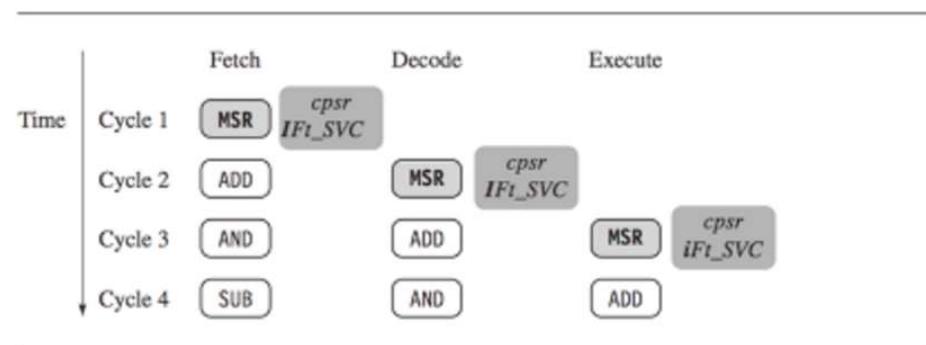


Pipeline Executing Characteristics

Execution of a Branch instruction or Branching By direct modification of PC causes the ARM core to flush its Pipeline

ARM10 uses Branch Prediction which reduces the effect of Pipeline Flush by predicting possible branches & loading the new branch address prior to the execution of the instruction

An instruction in the execute stage will complete even if an interrupt has been raised. Other instructions in the pipeline will be abandoned & processor will fill the pipeline with the appropriate entry in the vector table.



Exceptions, Interrupts & The Vector Table

Interrupt Occurs

Processor sets PC to a specific memory address

This address is within a special address range called vector table

Entries in vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt

When an exception or interrupt occurs, the processor suspends normal execution & starts loading instructions from VT. Each VT entry contains a form of Branch Instruction that points to the start of a particular Routine.

Memory Map Address 0x00000000 is reserved for Vector Table.

Optionally in some processors it can be located at higher addresses in memory starting at 0xFFFF0000(OS like Linux & MS Embedded Products take advantage of this)

Exceptions, Interrupts & The Vector Table

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0xFFFF0000
Undefined Instruction	UNDEF	0x00000004	0xFFFF0004
Software Interrupt	SWI	0x00000008	0xFFFF0008
Prefetch Abort	PABT	0x0000000C	0xFFFF000C
Data Abort	DABT	0x00000010	0xFFFF0010
Reserved	---	0x00000014	0xFFFF0014
Interrupt Request	IRQ	0x00000018	0xFFFF0018
Fast Interrupt Request	FIQ	0x0000001C	0xFFFF001C

Core Extensions

Standard components placed next to the ARM Core.

They Improve performance, manage resources, provide extra functionality

There are 3 H/W extensions ARM wraps around the core:

1. Cache & Tightly Coupled Memory
2. Memory Management
3. Co-Processors

Cache & Tightly Coupled Memory

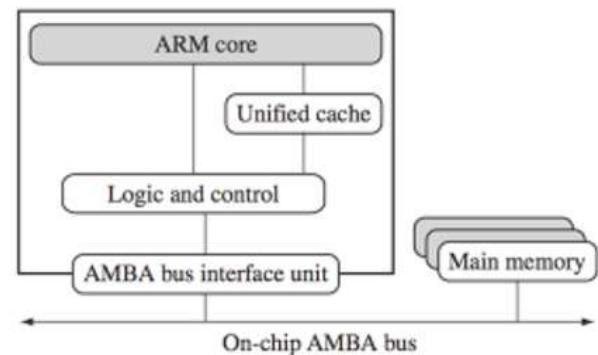
Cache is a block of fast memory placed between Main Memory & the Core

It allows efficient fetches from some memory types

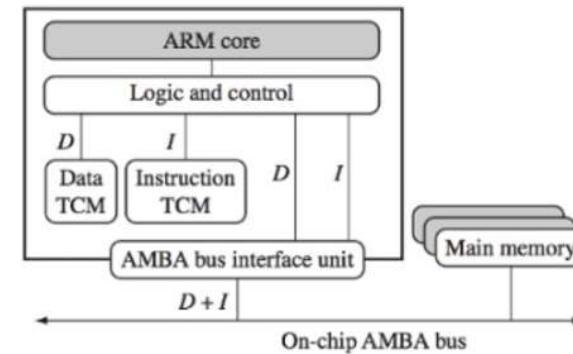
ARM has 2 forms of caches:

Simplified Von Neumann Architecture with cache- Combines both data & instruction into a single unified cache

Simplified Harvard Architecture with TCM-Separate caches for Data & Instructions



Simplified Von-Neumann Architecture with Cache



Simplified Harvard Architecture with TCM

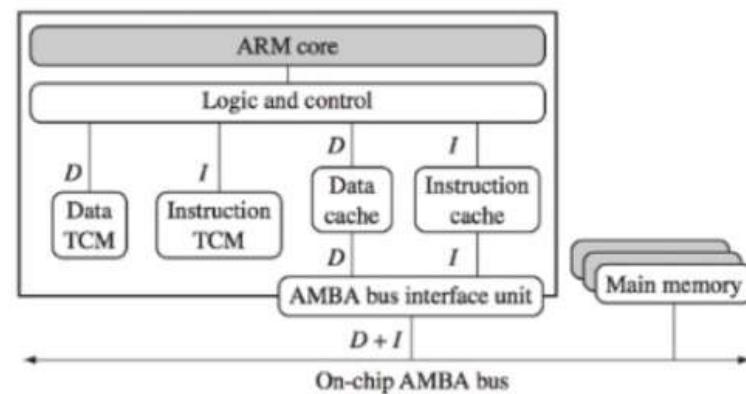
Cache & Tightly Coupled Memory

Cache provides an overall increase in performance at the cost of Predictable Execution

For RT systems, code execution has to be deterministic-time taken for loading & storing instructions or data must be predictable

This is achieved using TCM- fast SRAM located close to the core, guarantees clock cycles required to fetch instructions or data. Shows deterministic behaviour critical for RT systems

A combination of Cache & TCM will give an improved performance & Predictable RT response



Memory Management

Embedded Systems use multiple memory devices

These must be organized & protected

Achieved using Memory Management Hardware-

1. No extension-Provide no protection
2. MPU(Memory Protection Unit) – Provide limited Protection
3. MMU(Memory Management Unit) – Provide full protection

Memory management

No extension-Provide no protection

Fixed & Very little flexibility

Used for small Simple embedded systems that require no protection from rogue applications

MPU – Provide limited Protection

Used for systems that require memory protection but don't have a complex memory map.

MMU – Provide full protection.

Used for systems that run more sophisticated platform OS that support multitasking.

Has a set of translation tables to provide fine grained control over memory. These tables are stored in memory & provide virtual to physical address map as well as access permissions

Co-processors

Can be attached to ARM processor

Extends the processing features of a core by extending instruction set(eg: Vector Floating Point Operations) or by providing configuration registers

Multiple coprocessors can be added to the ARM Core via the coprocessor interface

Coprocessor can be accessed through a group of dedicated ARM instructions that provide a LOAD-STORE type interface.

Eg:coprocessor 15- ARM core uses coprocessor 15 registers to control cache,TCMs & memory management.

The new instructions are processed in the decode stage of the pipeline. If a coprocessor instruction is found then it is offered to the relevant coprocessor. If coprocessor is not present ARM decodes it as an **UNDEFINED INSTRUCTION EXCEPTION**

Module 5 (10 Hours)

Introduction to the ARM Instruction Set : Data Processing Instructions , Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants,

Simple programming exercises.

Text book 2: Ch 3:3.1 to 3.6 (Excluding 3.5.2)

Introduction to the ARM instruction set

PRAAHAS AMIN

ASSISTANT PROFESSOR

DEPT. OF ISE

Notable Features of ARM Instruction Set

The load-store architecture

3-address data processing instructions

Conditional execution of every instruction

The inclusion of every powerful load and store multiple register instructions

Single-cycle execution of all instruction

Open coprocessor instruction set extension

ARM Instruction Set

Data processing instructions

Branch instructions

LOAD-STORE Instructions

Software Interrupt Instruction

Program Status Register Instructions

Loading Constants

Conditional Execution

Every ARM instruction is CONDITIONALLY EXECUTED

The top 4 bits are ANDed with CPSR condition codes.

If they do not match then instruction is executed as a NOP

Therefore it is possible to perform a data processing operation and depending on the result, the following instructions may or may not be executed.

Basic Assembler instructions such as MOV or ADD can be prefixed with 16 conditional mnemonics which define the condition code to be tested for.

Each ARM instruction can be prefixed by one of the 16 condition codes. Therefore each instruction has 16 different variants

Conditional Execution

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Data processing instructions

Manipulation of data within registers

1. MOVE instructions
2. Arithmetic Instructions
3. Logical Instructions
4. Comparison Instructions
5. Multiply Instructions

Most data processing instructions can process one of their operands using Barrel Shifter
S suffix- updates flags in CPSR

MOVE & Logical Operations updates the C, N and Z flags

C flag is set from the result of the Barrel Shifter as the last bit shifted out

N flag is set to bit 31 of the result

Z flag is set if the result is 0

Data processing instructions

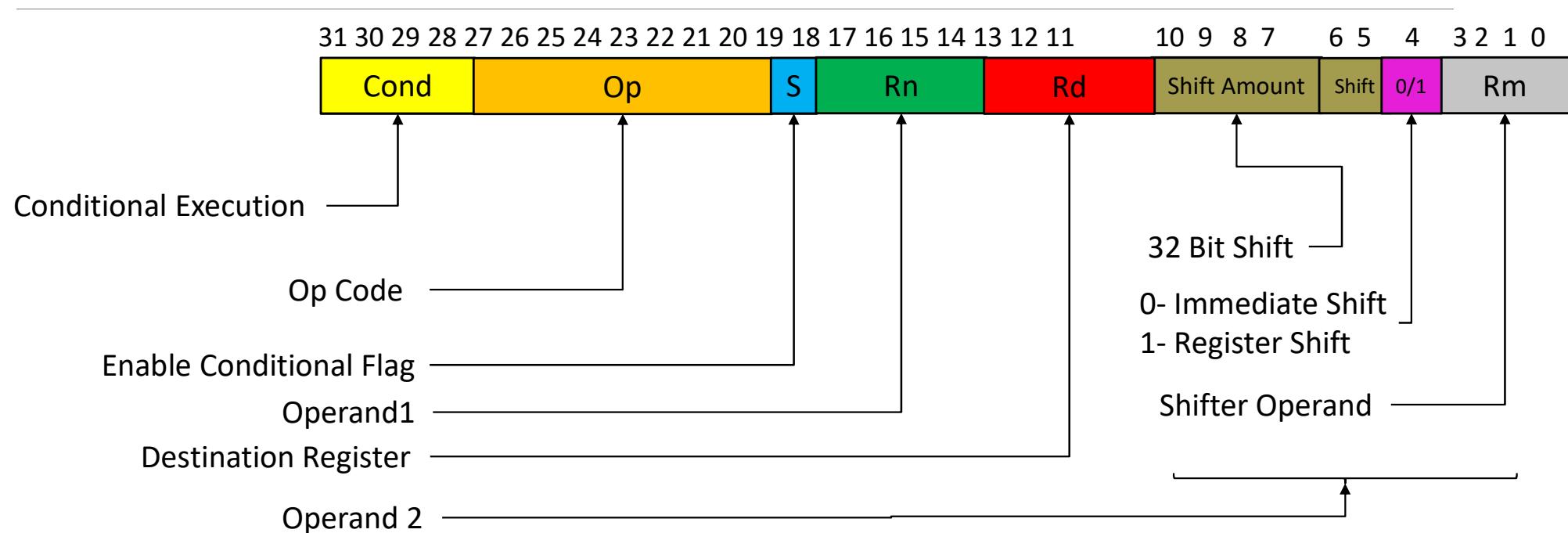
ARM Instruction Formats

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data processing immediate shift	cond	0	0	0	opcode	S	Rn		Rd		shift amount	shift	0		Rm																	
data processing register shift	cond	0	0	0	opcode	S	Rn		Rd		Rs	0	shift	1		Rm																
data processing immediate	cond	0	0	1	opcode	S	Rn		Rd		rotate		immediate																			
load/store immediate offset	cond	0	1	0	P	U	B	W	L		Rn		Rd			immediate																
load/store register offset	cond	0	1	1	P	U	B	W	L		Rn		Rd		shift amount	shift	0		Rm													
load/store multiple	cond	1	0	0	P	U	S	W	L		Rn				register list																	
branch/branch with link	cond	1	0	1	L							24-bit offset																				

- S = For data processing instructions, updates condition codes
- S = For load/store multiple instructions, execution restricted to supervisor mode
- P, U, W = distinguish between different types of addressing_mode
- B = Unsigned byte (B==1) or word (B==0) access
- L = For load/store instructions, Load (L==1) or Store (L==0)
- L = For branch instructions, is return address stored in link register

ARM Instruction Format

General Structure of Data processing Instructions



General Structure of Data Processing Instructions allows for conditional execution, a logical shift of up to 32-Bits & the data operation all in one cycle.

Shifter Operands

Shifter Operand is represented by the Least Significant 12 Bits of the instruction

It can take one of the 11 forms mentioned earlier

For instructions that use shifter operands, the C flag update is dependent on the form of the operand used.

MOVE instructions

Copies a 32-bit value into a destination Register Rd, where the 32-Bit value is a Register or Immediate value or a shifted register value.

Useful for setting initial values & transferring data between registers.

Syntax: <instruction>{<cond>} {S} Rd, N

- MOV {<cond>} {S} Rd,N
- MVN {<cond>} {S} Rd,N

Instruction	Description	
MOV	Move a 32-Bit value into a Register	Rd=N
MVN	Move the NOT of the 32-Bit value into a Register	Rd=^N

<cond> can be any of the 16 Conditional Mnemonics

{S} if S is suffixed to the instruction to set condition codes after execution of the <instruction>

Rd is the Destination Register

N is usually a constant immediate value preceded by # or a Register Rm

Possible Modes: Immediate, Register, Scaled Immediate, Scaled Register

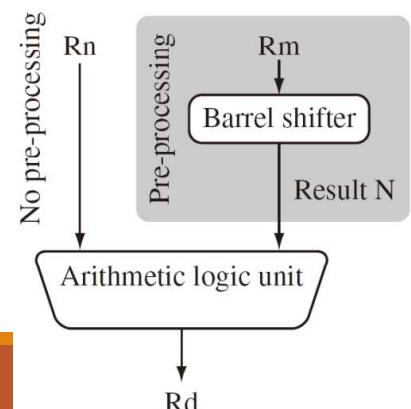
Barrel Shift operand Syntax for data processing instruction

N shift operations	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #shift_imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend	Rm, RRX

Barrel Shifter

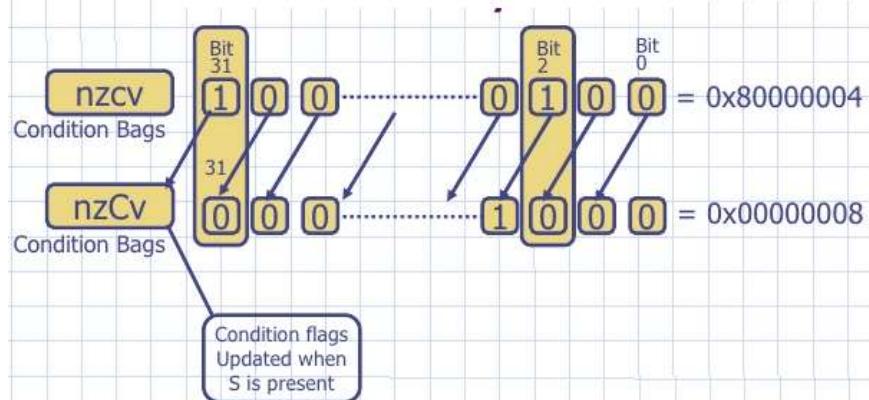
- First Operand must be a Register. Second Operand can be Register or an Immediate Value or a Register which has been pre-processed by the Barrel Shifter prior to being used by a data processing instruction.
- Data processing instructions are processed within the ALU
- A unique & powerful feature of ARM processor is the ability to shift the 32-Bit Binary Pattern in one of the source registers left or right by a specific number of positions before it enters the ALU
- This shift increases the power & flexibility of many data processing operations
- Particularly useful for loading constants into a register & achieving fast multiplies or divisions by power of 2.
- Eg: ADD R0,R2,R3,LSL #1
MOV R7,R5,LSL #2

PRE	r5=0x00000005
	r7=0x00000008
MOV r7,r5,LSL #2	
POST	r5=0x00000005
	r7=0x00000020



Barrel Shifter

Mnemonic	Description	Shift	Result	Shift amount y
LSL	Logical Shift Left	xLSLy	$x \ll y$	#0-31 or Rs
LSR	Logical Shift Right	xLSRy	(unsigned) $x \gg y$	#1-32 or Rs
ASR	Arithmetic Shift Right	xASRy	(signed) $x \gg y$	#1-32 or Rs
ROR	Rotate Right	xRORy	((unsigned) $x \gg y \mid (x \ll (32-y))$)	#1-31 or Rs
RRX	Rotate Right Extended	xRRX	(c flag $\ll 31$) ((unsigned) $x \gg 1$)	none



MOVE Examples

Syntax: MOV{<cond>}{S} Rd,Rm

MOV R7,R5

MOVNE R7,R5

MOVNES R7,R5 ;updates nzc flags

MVN R7,R5

MRS & MSR instructions

CPSR & SPSR are not part of the main Register Bank

Only 2 ARM instruction can operate on these Registers directly –

MRS- move PSR into GPR

MSR-move GPR value to PSR

These instructions will not work when CPU is in USER MODE. Will work in all other modes

Once in USER mode, the mode cannot be changed to Other modes except through an exception, reset FIQ,IRQ or SWI.

MRS{<cond>} Rd,CPSR

MRS{<cond>} Rd,SPSR

MSR{<cond>} CPSR_<fields>,#immediate

MSR{<cond>} CPSR_<fields>,Rm

MSR{<cond>} SPSR_<fields>,#immediate

MSR{<cond>} SPSR_<fields>,Rm

<fields>- limits any change just to the fields intended by the programmer
c-sets control field mask bit [7:0]
x – sets the extension field mask bit[15:8]
s – sets the status field mask bit[23:16]
f – sets the flags field mask bit[31:24]

MRS & MSR instruction

MSR allows an immediate value or Register contents to be transferred to the Condition Code Flag Bits nzcv of CPSR or SPSR_<mode> without affecting the control bits.

The top 4 bits of the specified register contents or 32 bit immediate value will be written to the top 4 bits of the relevant PSR

Operand Restrictions:

1. In USER Mode control bits of CPSR are protected from change, so only condition code flags of the CPSR can be changed. In other modes entire CPSR can be changed
2. Software must never change the T bit – leads to unpredictable state
3. SPSR Register that is accessed depends on the mode at the time of execution
4. R15(PC) must not be specified as Source or Destination Register
5. Do not attempt to Access SPSR in USER Mode as it does not exist in User Mode.

Arithmetic Instructions

ADD $Rd = Rn+N$

- ADD{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

ADC $Rd= Rn+N+Carry$

- ADC{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

SUB $Rd=Rn-N$

- SUB{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

SBC $Rd=Rn-N-!(carry\ flag)$

- SBC{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

RSB $Rd=N-Rn$

- RSB{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

RSC $Rd=N-Rn-!(Carry\ Flag)$

- RSC{<cond>} {S} Rd,Rn,N
- Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

Logical Instructions

AND $Rd = Rn \& N$

$AND\{cond\}\{S\} Rd, Rn, N$

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

ORR $Rd = Rn | N$

$OR\{cond\}\{S\} Rd, Rn, N$

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

EOR $Rd = Rn ^ N$

$EOR\{cond\}\{S\} Rd, Rn, N$

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

BIC $Rd = Rn \& \sim N$

$BIC\{cond\}\{S\} Rd, Rn, N$

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

BIT CLEAR OPERATION=> AND NOT OPERATION

Comparison Instructions

CMP flags set as a result of Rn-N

CMP{<cond>} Rn,N

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

CMN flags set as a result of Rn+N (compare negated)

CMN{<cond>} Rn,N

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

TST flags set as a result of Rn & N

TST{<cond>} Rn,N

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

TEQ flags set as a result of Rn ^ N

TEQ{<cond>} Rn,N

Where N – #imm or Rm or Shifted Rm (used with LSL,LSR)

Multiply Instructions

All operands & Destination must be Simple Registers

Immediate value or Shifted Register cannot be used for Operand 2

Destination & Operand 1 must be different

Registers R15 cannot be Destination

Multiply

MUL

MUL{<cond>} {S} Rd,Rm,Rs

Rd=Rm*Rs

MLA

MLA {<cond>} {S} Rd,Rm,Rs,Rn

Rd=(Rm*Rs)+Rn

SMULL – Signed Multiply Long

(32 bit x 32 bit) = 64 bit result

SMULL {<cond>} {S} RdLo,RdHi,Rm,Rs

[RdHi,RdLo] = Rm*Rs

UMULL – Signed Multiply Long

(32 bit x 32 bit) = 64 bit result

UMULL {<cond>} {S} RdLo,RdHi,Rm,Rs

[RdHi,RdLo] = Rm*Rs

SMLAL - Signed Multiply Accumulate Long

(32 bit x 32 bit)+64 bit = 64 Bit Result

SMLAL {<cond>} {S} RdLo,RdHi,Rm,Rs

[RdHi,RdLo] = [RdHi,RdLo]+(Rm*Rs)

UMLAL – Unsigned Multiply Accumulate Long

(32 bit x 32 bit)+64 bit = 64 Bit Result

UMLAL {<cond>} {S} RdLo,RdHi,Rm,Rs

[RdHi,RdLo] = [RdHi,RdLo]+(Rm*Rs)

Branch Instructions

B

pc = label

B{<cond>} label

BX

pc=Rm & 0xFFFFFFFF, T=Rm & 1

BX{<cond>} Rm

BL

pc = label

lr = address of next instruction after BL

BL{<cond>} label

BLX

Pc=label, T=1

Pc =Rm & 0xFFFFFFFF, T= Rm& 1

lr= address of next instruction after BLX

BLX{<cond>} Rm

BLX label

Examples for MOVE Instructions

Pre	r5=5	Pre	r5=5	Pre	cpsr=nzcvqiFt_USER
	r7=8		r7=8		r0=0x00000000
	MOV r7,r5		MOV r7,r5,LSL #2		r1=0x80000004
Post	r5=5	Post	r5=5	MOVS r0,r1,LSL #1	
	r7=5		r7=20	Post	cpsr=nzCvqiFt_USER
					r0=0x00000008
					r1=0x80000004

Examples for Arithmetic Instructions

Pre	r0=0x00000000	Pre	r0=0x00000000	Pre	r0=0x00000000	Pre	cpsr=nzcvqiFt_USER
	r1=0x00000002		r1=0x00000002		r1=0x00000002	o	r0=0x00000000 r1=0x00000001
	r2=0x00000001		r2=0x00000001		r2=0x00000001		r2=0x00000001
	ADD r0,r1,r2		SUB r0,r1,r2		RSB r0,r1,r2		SUBS r0,r1,r2
Post	r0=0x00000003	Post	r0=0x00000001	Post	r0=0xFFFFFFFF	Post	cpsr=nZCvqiFt_USER
	r1=0x00000002		r1=0x00000002		r1=0x00000002		r0=0x00000000 r1=0x00000001
	r2=0x00000001		r2=0x00000001		r2=0x00000001		r2=0x00000001

Examples for Arithmetic Instruction with Shifter Operand

Pre r0=0x00000000

 r1=0x00000005

 ADD r0,r1,r1,LSL #1

Post r0=0x0000000F

 r1=0x00000005

0101 → 1010 after LSL #1

That is added to 0101

Final Result is 1111

Examples For Logic Instructions

Pre	r0=0x00000000 r1=0x02040608 r2=0x10305070 ORR r0,r1,r2	Pre	r0=0x00000000 r1=0x0000000F r2=0xFFFF000F AND r0,r1,r2	Pre	r0=0x00000000 r1=0x5F5F5F5F r2=0xFFFFFFFF EOR r0,r1,r2	Pre	r0=0x00000000 r1=0xFFFFFFFF r2=0x000000FF BIC r0,r1,r2
Post	r0=0x12345678 r1=0x02040608 r2=0x10305070 OR Operation	Post	r0=0x0000000F r1=0x0000000F r2=0xFFFF000F AND Operation	Post	r0=0xA0A0A0A0 r1=0x5F5F5F5F r2=0xFFFFFFFF EOR Operation (Exclusive OR)	Post	r0=0xFFFFFFF0 r1=0xFFFFFFFF r2=0x000000FF BIC Operation (Bit Clear) (AND-NOT Operation)
			<th></th> <td><th></th><td></td></td>		<th></th> <td></td>		

Examples for Comparison Instructions

Pre	cpsr=nzcvqiFt_USER	Pre	cpsr=nzcvqiFt_USER
	r0=0x00000004		r0=0x00000004
	r1=0x00000004		r1=0x00000000
	CMP r0,r1		TST r0,r1
Post	cpsr=nZcvqiFt_USER	Post	cpsr=nZcvqiFt_USER
	r0=0x00000004		r0=0x00000004
	r1=0x00000004		r1=0x00000000

Examples for Multiply Instructions

Pre	r0=0x00000000 r1=0x00000002 r2=0x00000002 MUL r0,r1,r2	Pre	r0=0x00000000 r1=0x00000000 r2=0xF0000002 r3=0x00000002 UMULL r0,r1,r2,r3	Pre	r0=0x00000000 r1=0x00000002 r2=0x00000002 r3=0x00000001
Post	r0=0x00000004 r1=0x00000002 r2=0x00000002	Post	r0=0xE0000004 r1=0x00000001 r2=0xF0000002 r3=0x00000002	Post	MLA r0,r1,r2,r3 r0=0x00000005 r1=0x00000002 r2=0x00000002 r3=0x00000001

Examples for Branch Instructions

B fwd	bwd ADD r1,r2,#4
ADD r1,r2,#4	ADD r0,r6,#2
ADD r0,r6,#2	ADD r3,r7,#4
ADD r3,r7,#4	SUB r1,r2,#4
fwd SUB r1,r2,#4	B bwd

Forward Branching

Backward Branching

Example for Branch Instructions

BL subroutine

CMP r1,#5

MOVEQ r1,#0

:

subroutine

<subroutine code>

MOV PC,LR

BX and BLX are branch instructions.

BX uses an absolute address stored in Register Rm

It is used to branch to and from Thumb code.

T bit in the CPSR is updated by the LSB of the Branch Register

Similarly BLX instruction updates the T bit of the CPSR with the LSB bit and
Additionally sets the LR with the Return Address

LOAD-STORE Instructions (Single Register Transfer)

LDR: reads a word from memory & writes to destination register $Rd \leftarrow \text{mem32}[\text{address}]$

$\text{LDR}\{\text{cond}\} \{B\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \leftarrow \text{mem8}[\text{address}]$

$\text{LDR}\{\text{cond}\} \{H\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \leftarrow \text{mem16}[\text{address}]$

$\text{LDR}\{\text{cond}\} \{SB\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \leftarrow \text{SignExtend}(\text{mem8}[\text{address}])$

$\text{LDR}\{\text{cond}\} \{SH\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \leftarrow \text{SignExtend}(\text{mem16}[\text{address}])$

STR: stores a single register to memory $Rd \rightarrow \text{mem32}[\text{address}]$

$\text{STR } \{\text{cond}\} \{B\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \rightarrow \text{mem8}[\text{address}]$

$\text{STR } \{\text{cond}\} \{H\} Rd, \langle \text{addressing_mode} \rangle$ $Rd \rightarrow \text{mem16}[\text{address}]$

Examples for LDR

Pre mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x00000000
 r1=0x00000000
 r2=0x00008000

 LDR r0,[r2]
 ADD r2,#4
 LDR r1,[r2]

Post mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x01010101
 r1=0x10101010
 r2=0x00008004

Pre mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x00000000
 r1=0x00000000
 r2=0x00008000

 LDR r0,[r2],#4
 LDR r1,[r2]

Post mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x01010101
 r1=0x10101010
 r2=0x00008004

Here r2 is the Base Register
4 is the offset from the base

Examples for STR

Pre mem32[0x00008000] = 0x00000000
 mem32[0x00008004] = 0x00000000

 r0=0x01010101
 r1=0x10101010
 r2=0x00008000

 STR r0,[r2]
 ADD r2,#4
 STR r1,[r2]

Post mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x01010101
 r1=0x10101010
 r2=0x00008004

Pre mem32[0x00008000] = 0x00000000
 mem32[0x00008004] = 0x00000000

 r0=0x01010101
 r1=0x10101010
 r2=0x00008000

 STR r0,[r2,#4] ←
 STR r1,[r2]

Post mem32[0x00008000] = 0x01010101
 mem32[0x00008004] = 0x10101010

 r0=0x01010101
 r1=0x10101010
 r2=0x00008004

Here r2 is the Base Register
4 is the offset from the base.
If no offset is mentioned then it is 0

Single Register LOAD-STORE Addressing Modes- INDEX METHODS

There are 3 Index Methods

Index Methods	Data	Base Address Register	Example
Pre-Index with Writeback	mem[base+offset]	base+offset	LDR r0,[r1,#4]!
Pre-Index	mem[base+offset]	Not updated	LDR r0,[r1,#4]!
Post-Index	mem[base]	base+offset	LDR r0,[r1],#4

Pre-Index with Writeback: Calculates an address from (Base Register+Offset) and then updates that value in the base register prior to using that address

Pre-Index: Calculates an address from (Base Register+Offset) prior to using that address but does not update that value in the base register.

Post-Index: Calculates an address from (Base Register+Offset) and then updates that value in the base register after the address is used

Example for Pre-Indexing with Writeback, Pre-Indexing and Post-Indexing

Pre	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x00000000 r1=0x00009000 LDR r0,[r1,#4]!	Pre	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x00000000 r1=0x00009000 LDR r0,[r1,#4]	Pre	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x00000000 r1=0x00009000 LDR r0,[r1],#4
Post	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x02020202 r1=0x00009004	Post	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x02020202 r1=0x00009000	Post	mem32[0x00009000] = 0x01010101 mem32[0x00009004] = 0x02020202 r0=0x01010101 r1=0x00009004
Pre-Indexing with Writeback		Pre-Indexing		Post-Indexing	

Single Register LOAD STORE Addressing Modes

Addressing Modes & Index Method	Addressing Syntax
Pre-Index Writeback with Immediate Offset	[Rn,#+/-offset_12]! Word/Unsigned byte [Rn,#+/-offset_8]! Halfword /Signed Half Word/Signed Byte/Doubleword
Pre-Index Writeback with Register Offset	[Rn,+/-Rm]!
Pre-Index Writeback with Scaled Register Offset	[Rn,+/-Rm, shift #immediate]! Word/Unsigned byte (only)
Pre-Index with Immediate Offset	[Rn,#+/-offset_12] Word/Unsigned byte [Rn,#+/-offset_8] Halfword /Signed Half Word/Signed Byte/Doubleword
Pre-Index with Register Offset	[Rn,+/-Rm]
Pre-Index with Scaled Register Offset	[Rn,+/-Rm, shift #immediate] Word/Unsigned byte (only)
Immediate Post-Indexed	[Rn] ,#+/-offset_12 Word/Unsigned byte [Rn] ,#+/-offset_8 Halfword /Signed Half Word/Signed Byte/Doubleword
Register Post-Indexed	[Rn] ,+/-Rm
Scaled Register Postindexed	[Rn] ,+/-Rm, shift #immediate Word/Unsigned byte (only)

Single Register LOAD STORE Addressing Modes

A signed offset or register is denoted by +/- indicating a positive or a negative offset from base register Rn.

The base address register is a pointer to a byte in memory

Offset specifies the number of bytes. Eg: 4 indicates access of a 32 Bit Word.

Immediate means that the address is calculated using the base address register and a 12-bit offset encoded in the instruction

Register means that the address is calculated using the base address register and a specified register's contents

Scaled means that the address is calculated using the base address register and a barrel shifter operation on the contents of a specified register.

Example of LDR instructions using different Addressing Modes

Indexing	Instruction	r0=	r1+=
Preindex with writeback	LDR r0,[r1,#0x4]!	Mem32[r1+0x4]	0x4
	LDR r0,[r1,r2]!	Mem32[r1+r2]	r2
	LDR r0,[r1,r2,LSR #0x4]!	Mem32[r1+(r2 LSR 0x4)]	R2 LSR 0x4
	<i>LDR r0,[r1,-r2,LSR #0x4]!</i>	Mem32[r1-(r2 LSR 0x4)]	<i>-(R2 LSR 0x4)</i>
Preindex	LDR r0,[r1,#0x4]	Mem32[r1+0x4]	Not updated
	LDR r0,[r1,r2]	Mem32[r1+r2]	Not updated
	LDR r0,[r1,r2,LSR #0x4]	Mem32[r1+(r2 LSR 0x4)]	Not updated
	<i>LDR r0,[r1,-r2,LSR #0x4]</i>	<i>Mem32[r1-(r2 LSR 0x4)]</i>	<i>Not updated</i>
Postindex	LDR r0,[r1],#0x4	Mem32[r1]	0x4
	LDR r0,[r1],r2	Mem32[r1]	r2
	LDR r0,[r1],r2,LSR #0x4	Mem32[r1]	R2 LSR 0x4

Example of STR instructions using different Addressing Modes

Indexing	Instruction	r0=	r1+=
Preindex with writeback	STR r0,[r1,#0x4]!	Mem32[r1+0x4]	0x4
	STR r0,[r1,r2]!	Mem32[r1+r2]	r2
	STR r0,[r1,r2,LSR #0x4]!	Mem32[r1+(r2 LSR 0x4)]	R2 LSR 0x4
	<i>STR r0,[r1,-r2,LSR #0x4]!</i>	Mem32[r1-(r2 LSR 0x4)]	<i>-(R2 LSR 0x4)</i>
Preindex	STR r0,[r1,#0x4]	Mem32[r1+0x4]	Not updated
	STR r0,[r1,r2]	Mem32[r1+r2]	Not updated
	STR r0,[r1,r2,LSR #0x4]	Mem32[r1+(r2 LSR 0x4)]	Not updated
	<i>STR r0,[r1,-r2,LSR #0x4]</i>	<i>Mem32[r1-(r2 LSR 0x4)]</i>	<i>Not updated</i>
Postindex	STR r0,[r1],#0x4	Mem32[r1]	0x4
	STR r0,[r1],r2	Mem32[r1]	r2
	STR r0,[r1],r2,LSR #0x4	Mem32[r1]	R2 LSR 0x4

Multiple Register Transfer

Can transfer multiple Register values between Processor and Memory in a single instruction.

Transfer occurs from a base register Rn pointing into memory

Efficient for transferring blocks of data and saving and restoring contents of stack.

Can increase interrupt latency(an instruction is not interrupted while it is executing).

Eg: a LOAD Multiple Instruction takes $2+Nt$ cycles where N is the number of Registers to load and t is the number of cycles required for each sequential access to memory. If an interrupt has been raised, then it has no effect until the execution of the LOAD multiple instruction is completed if the LOAD multiple instruction execution had started before the interrupt occurred. This latency can be controlled by compilers which implement restriction on the number of multiple registers used with LOAD and STORE Multiple instructions.

Multiple Register Transfer

LDM{<cond>}<addressing_mode>,Rn{!},<Registers>{^}

{Rd}*N \leftarrow mem32[start address+4*N] optional Rn updated

STM {<cond>}<addressing_mode>,Rn{!},<Registers>{^}

{Rd}*N \rightarrow mem32[start address+4*N] optional Rn updated

There are 4 Addressing Modes for LDM and STM

Rn is the Base Register

! - Indicates that the Register is updated after the instruction is executed.

Registers can be individually listed by using comma separator or can be listed as a range within { } .

Addressing Mode	Start Address	End Address	Rn!
IA (Increment After)	Rn	Rn+4*N - 4	Rn+4*N
IB (Increment Before)	Rn+4	Rn+4*N	Rn+4*N
DA (Decrement After)	Rn-4*N +4	Rn	Rn-4*N
DB (Decrement Before)	Rn-4*N	Rn-4	Rn-4*N

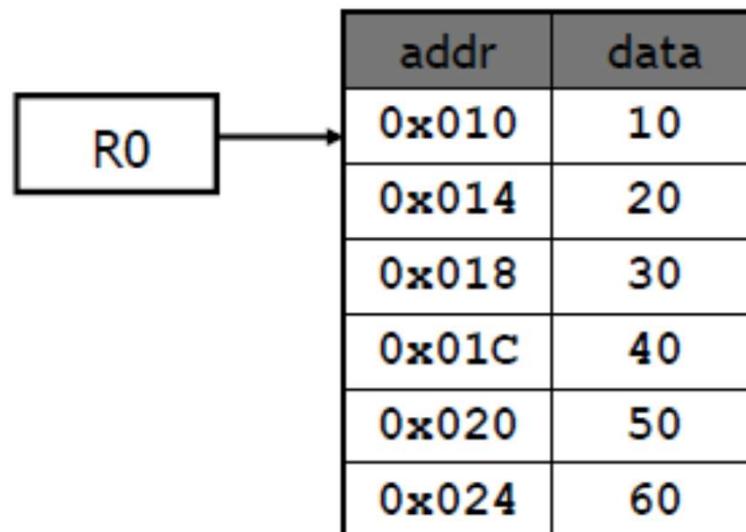
LDMIA r0,{r1-r3}

LDMIA R0 , {R1,R2,R3}

or

LDMIA R0 , {R1-R3}

R1: 10
R2: 20
R3: 30
R0: 0x10



LDMIA r0!,{r1-r3}

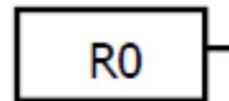
LDMIA R0!, {R1,R2,R3}

R1: 10

R2: 20

R3: 30

R0: **0x01C**



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

LDMIB r0!,{r1-r3}

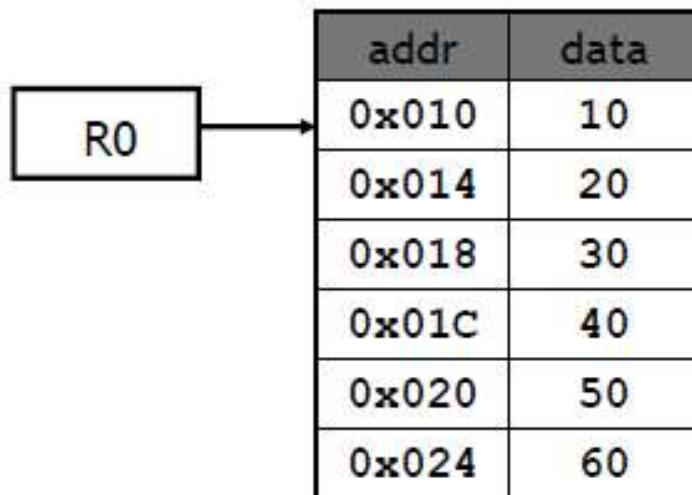
LDMIB R0!, {R1,R2,R3}

R1: 20

R2: 30

R3: 40

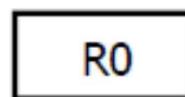
R0: 0x01C



LDMDA r0!,{r1-r3}

LDMDA R0!, {R1,R2,R3}

R1: 40
R2: 50
R3: 60
R0: 0x018

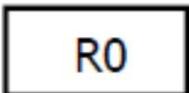


addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

LDMDB r0!,{r1-r3}

LDMDB R0!, {R1,R2,R3}

R1: 30
R2: 40
R3: 50
R0: 0x018



addr	data
0x010	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

Examples for LDMIA

Pre mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080010

r1=0x00000000

r2=0x00000000

r3=0x00000000

LDMIA r0!,{r1-r3}

Post mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x0008001C

r1=0x00000001

r2=0x00000002

r3=0x00000003

Examples for LDMIB

Pre mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080010

r1=0x00000000

r2=0x00000000

r3=0x00000000

LDMIA r0!,{r1-r3}

Post mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x0008001C

r1=0x00000002

r2=0x00000003

r3=0x00000004

Examples for LDMDA

Pre mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080020

r1=0x00000000

r2=0x00000000

r3=0x00000000

LDMDA r0!,{r1-r3}

Post mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080014

r1=0x00000003

r2=0x00000004

r3=0x00000005

Examples for LDMDB

Pre mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080020

r1=0x00000000

r2=0x00000000

r3=0x00000000

LDMDB r0!,{r1-r3}

Post mem32[0x80020] = 0x05

mem32[0x8001C] = 0x04

mem32[0x80018] = 0x03

mem32[0x80014] = 0x02

mem32[0x80010] = 0x01

r0=0x00080014

r1=0x00000002

r2=0x00000003

r3=0x00000004

Multiple Register Transfer

LDM

LDM{<cond>}<addressing_mode>,Rn{!},<Registers>{^}

STM {<cond>}<addressing_mode>,Rn{!},<Registers>{^}

Addressing Modes (Multiple Register Transfer)

IA

IB

DA

DB

LDMIA, LDMIB, LDMDA, LDMDB

STMIA, STMIB, STMDA, STMDB

Stack Operation using STM & LDM

LDM

LDM{<cond>}<addressing_mode>,Rn{!},<Registers>{^}

STM {<cond>}<addressing_mode>,Rn{!},<Registers>{^}

Addressing Modes (Multiple Register Transfer)

FA

FD

EA

ED

STMEA, STMED, STMFA, STMFD

LDMEA,LDMED,LDMFA,LDMFD

STACK Operations

ARM architecture uses LOAD-STORE MULTIPLE instructions to carry out STACK Operations.

PUSH operation uses STORE MULTIPLE instruction

POP operation uses LOAD MULTIPLE instruction

When using stack it can grow upwards (towards higher address) or downwards (towards lower address) in memory

Upward growing stack – ASCENDING

Downward growing stack – DESCENDING

FULL STACK → SP points to the LAST ITEM on the stack

EMPTY STACK → SP Point to the First EMPTY LOCATION on the stack

LDFA,LDMFD,LDMEA,LDMED

STMFA,STMFD,STMEA,STMED

Stack Operations

ATPCS-ARM Thumb Procedure Call Standard

Stacks defined as FULL DESCENDING.

Therefore

PUSH → STMFD

POP → LDMFD

STACK OPERATIONS

Addressing Mode	POP	=LDM	PUSH	=STM
FA	LDMFA	LDMDA	STMFA	STMIB
FD	LDMFD	LDMIA	STMFD	STMDB
EA	LDMEA	LDMDB	STMEA	STMIA
ED	LDMED	LDMIB	STMED	STMDA

PRE r1=0x00000002

r4=0x00000003

sp=0x00080014

STMFD sp!,{r1,r4}

POST r1=0x00000002

r4=0x00000003

sp=0x0008000C

PRE Address Data

sp →	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	Empty
	0x8000c	Empty

STMFD sp!, {r1,r4}

POST Address Data

sp →	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002

Stack Limit

ATPCS defines r10 as the Stack Limit

Stack Overflow Error

Swap instruction

SWP – Special Case of LOAD-STORE. Swaps the contents of the memory with the contents of the Register.

ATOMIC OPERATION- Reads from and Writes to a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

SWP{B}{<cond>} Rd,Rm,[Rn]

SWP –

tmp =mem32[Rn]

mem32[Rn]=Rm

Rd=tmp

SWPB –

tmp =mem8[Rn]

mem8[Rn]=Rm

Rd=tmp

SWP Example

PRE mem32[0x9000]=0x12345678

r0=0x00000000

Useful when implementing Semaphores & Mutual Exclusion in an OS

r1=0x11112222

r2=0x00009000

SWP r0,r1,[r2]

POST mem32[0x9000]=0x11112222

r0=0x12345678

r1=0x11112222

r2=0x00009000

Semaphore Example

SPIN

```
MOV r1,=semaphore  
MOV r2,#1  
SWP r3,r2,[r1]  
CMP r3,#1  
BEQ SPIN
```

Address pointer at by Sempahore either contains value 0 or 1

When the semaphore equals 1, then the service in question is being used by another process.

The Routine will continue to loop around until the service is released by the other process in other words when the semaphore address location contains value 0.

Software Interrupt Instruction

SWI – causes a software interrupt exception, which provides a mechanism for applications to call OS routines.

SWI{<cond>} SWI_number

lr_svc=address of instruction following SWI

spsr_svc=cpsr

pc=vectors+0x8

cpsr mode=SVC

cpsr I=1 (MASK IRQ interrupts)

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0xFFFF0000
Undefined Instruction	UNDEF	0x00000004	0xFFFF0004
Software Interrupt	SWI	0x00000008	0xFFFF0008
Prefetch Abort	PABT	0x0000000C	0xFFFF000C
Data Abort	DABT	0x00000010	0xFFFF0010
Reserved	---	0x00000014	0xFFFF0014
Interrupt Request	IRQ	0x00000018	0xFFFF0018
Fast Interrupt Request	FIQ	0x0000001C	0xFFFF001C

Coprocessor instructions

CDP

CDP{<cond>} cp,opcode1,Cd,Cn {,opcode2}

STC

STC{<cond>} cp,Cd,<addressing_mode>

MRC

MRC{<cond>} cp,opcode1,Rd,Rn, Cm {,opcode2}

MCR

MCR{<cond>} cp,opcode1,Rd,Rn, Cm {,opcode2}

LDC

LDC{<cond>} cp,Cd,<addressing_mode>

Loading constants

LDR

LDR Rd, =constant

ADR

ADR Rd,label

Example

Pre

r5=5

r7=8

Instruction

mov r7,r5

Post

r5= ?

r7= ?

example

Pre

r5=5

r7=8

Instruction

mov r7,r5,LSL #2

Post

r5=5

r7=20

Example to demonstrate effect on condition flags

Pre

cpsr=nzcvqiFt_USER

r0=0x00000000

r1=0x80000004

Instruction

MOVS r0,r1,LSL #1

Post

r0=0x00000008

r1=0x80000004