UNIT – I

OVERVIEW

- Introduction to microprocessors
- Evolution of microprocessors
- Features of 8085 microprocessor
- Pin Diagram of 8085 microprocessor
- Architecture of 8085
- Addressing modes of 8085
- Timing Diagrams
- Instruction set

UNIT-I

INTRODUCTION:

Microprocessor acts as a CPU in a microcomputer. It is present as a **single IC chip** in a microcomputer. Microprocessor is the heart of the machine.

A Microprocessor is a device, which is capable of

1. Receiving Input 2 Performing Computations 3. Storing data and instructions
4. Display the results 5. Controlling all the devices that perform the above 4 functions.

The device that performs tasks is called Arithmetic Logic Unit (ALU). A single chip called Microprocessor performs these tasks together with other tasks.

"A MICROPROCESSOR is a multipurpose programmable logic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to those instructions and provides results as output."

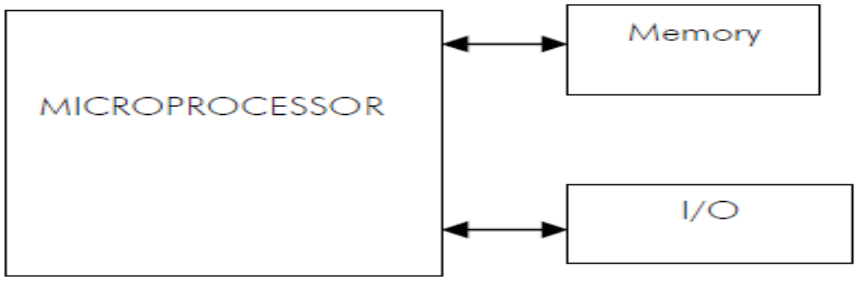


Figure 1.1

Figure shows a programmable machine, which consists of a microprocessor, memory, I/O. All these three components work together to perform a given task.

EVOLUTION OF MICROPROCESSORS:

The microprocessor age began with the advancement in the IC technology to put all necessary functions of a CPU into a single chip.

Intel started marketing its first microprocessor in the name of Intel 4004 in 1971. This was a4-bit microprocessor having 16-pins in a single chip of PMOS technology. This was called the **first generation microprocessor**. The Intel 4004 along with few other devices was used for making calculators. The 4004 instruction set contained only 45 instructions. Later in 1971, INTEL Corporation released the 8008 – an extended 8-bit version of the 4004 microprocessor. The 8008 addressed an expanded memory size (16KB) and 48 instructions.

Limitations of first generation microprocessors is small memory size, slow speed and instruction set limited its usefulness.

Second generation microprocessors:

The second generation microprocessor using NMOS technology appeared in the market in the year 1973. The Intel 8080, an 8-bit microprocessor, of NMOS technology was developed in the year 1974 which required only two additional devices to design a functional CPU.

The advantages of second generation microprocessors were

- Large chip size (170 x 200 mil) with 40-pins. More chips on decoding circuits.
- Ability to address large memory space (64-K Byte) and I/O ports (256).
- More powerful instruction sets. Dissipate less power.

- Better interrupt handling facilities.
- Sized 70x200 mil) with 40-pins.
- Used Single Power Supply

Cycle time reduced to half (1.3 to 9 m sec.)
Less Support Chips Required
Faster Operation

The 8080 microprocessor addresses more memory and execute additional instructions, but executes them 10 times faster than 8008.The 8080 has memory of 64 KB whereas for 8008 16 KB only. In 1977, INTEL, introduced 8085 which was an updated version of 8080 last 8-bit processor.

The main advantages of 8085 were its internal clock generator, internal system controller and higher clock frequency.

Third Generation Microprocessor:

In 1978, INTEL released the 8086 microprocessor, a year later it released 8088. Both devices were 16 bit microprocessors, which executed instructions in less than 400ns.The 8086 and 8088 addresses 1MB of memory and rich instruction set to 246.16-bit processors were designed using HMOS technology. The Intel 80186 and 80188 were the improved versions of Intel 8086 and8088, respectively. In addition to 16-bit CPU, the 80186 and 80188 had programmable peripheral devices integrated on the same package.

Fourth Generation Microprocessor:

The single chip 32-bit microprocessor was introduced in the year 1981 by Intel as iAPX 432. The other 4^{th}generation microprocessors were; Bell Single Chip Bellmac-32, Hewlett-Packard, National NSl 6032, Texas Instrument99000. Motorola 68020 and 68030. The Intel in the year 1985 announced the 32-bit microprocessor (80386). The 80486 has already been announced and is also a 32-bit microprocessor.

The 80486 is a combination 386 processor a math coprocessor, and a cache memory controller on a single chip.

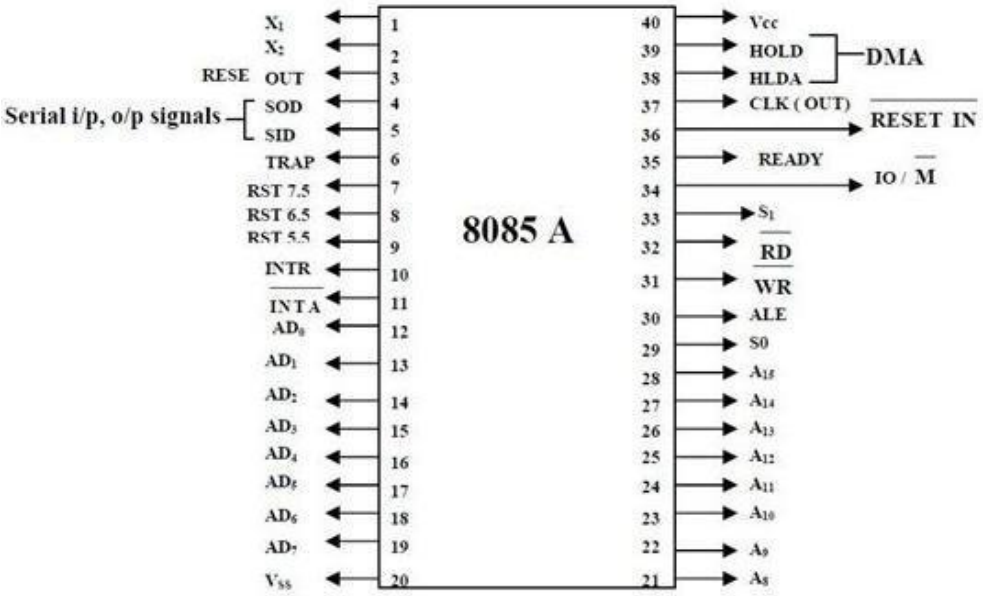
The Pentium is a 64-bit superscalar processor. It can execute more than one instruction at a time and has a full 64-bit data bus and 32-bit address bus. Its performance is double than 80486.

inition is: ALAB: MOV AX, COUNT

8085 Microprocessor

The salient features of 8085 µp are:

• It is a 8 bit microprocessor.
• It is manufactured with N-MOS technology.
• It has 16-bit address bus and hence can address up to 216 = 65536 bytes (64KB) memory locations through A_0-A_{15}.
• The first 8 lines of address bus and 8 lines of data bus are multiplexed $AD_0 – AD_7$.
• Data bus is a group of 8 lines $D_0 – D_7$.
• It supports external interrupt request.
• A 16 bit program counter (PC)
• A 16 bit stack pointer (SP)
• Six 8-bit general purpose register arranged in pairs: BC, DE, HL.
• It requires a signal +5V power supply and operates at 3.2 MHZ single phase clock.
• It is enclosed with 40 pins DIP (Dual in line package).



A8 - A15 (Output 3 State)

Address Bus:The most significant 8 bits of the memory address or the 8 bits of the I/0 address,3 stated during Hold and Halt modes.

AD0 - AD7 (Input/Output 3state)

Multiplexed Address/Data Bus; Lower 8 bits of the memory address (or I/0 address) appear on the bus during the first clock cycle of a machine state. It then becomes thedata bus during the second and third clock cycles. 3 stated during Hold and Halt modes.

ALE (Output)

Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information. ALE can also be used to strobe the status information. ALE is never 3stated.

RD (Output 3state)

READ: indicates the selected memory or 1/0 device is to be read and that the Data Bus is available for the data transfer.

WR (Output 3state)

WRITE: It indicates the data on the Data Bus is to be written into the selected memory or 1/0 location. Data is set up at the trailing edge of WR. Tri-stated during Hold and Halt modes.

READY (Input)

If Ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

HOLD (Input)

HOLD:indicates that another Master is requesting the use of the Address and DataBuses. The CPU, upon receiving the Hold request. will relinquish the use of buses as soon as the completion of the current machine cycle. Internal processing can continue.The processorcanregain the buses only after the Hold is removed. When the Hold is acknowledged, the Address, Data, RD, WR, and IO/M lines are 3stated.

HLDA (Output)

HOLD ACKNOWLEDGE:indicates that the CPU has received the Hold request and that it will relinquish the buses in the next clock cycle. HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycle after HLDA goes low.

INTR (Input)

INTERRUPT REQUEST is used as a general purpose interrupt. It is sampled onlyduring the next to the last clock cycle of the instruction. If it is active, the Program Counter (PC) will be inhibited from incrementing and an INTA will be issued. During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.

INTA (Output)

INTERRUPT ACKNOWLEDGE: is used instead of (and has the same timing as) RDduring the Instruction cycle after an INTR is accepted. It can be used to activate the 8259 Interrupt chip or some other interrupt port.

RESTART INTERRUPTS

These three inputs have the same timing as INTR except they cause an internal RESTART to be automatically inserted.

RST 7.5 ~~ Highest Priority RST 6.5

RST 5.5 Lowest Priority

TRAP (Input)

Trap interrupt is a non-maskable restart interrupt. It is recognized at the same time as INTR. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt.

RESET IN (Input)

Reset sets the Program Counter to zero and resets the Interrupt Enable and HLDA flip-flops. None of the other flags or registers (except the instruction register) are affected The CPU is held in the reset condition as long as Reset is applied.

RESET OUT (Output)

Indicates CPU is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.

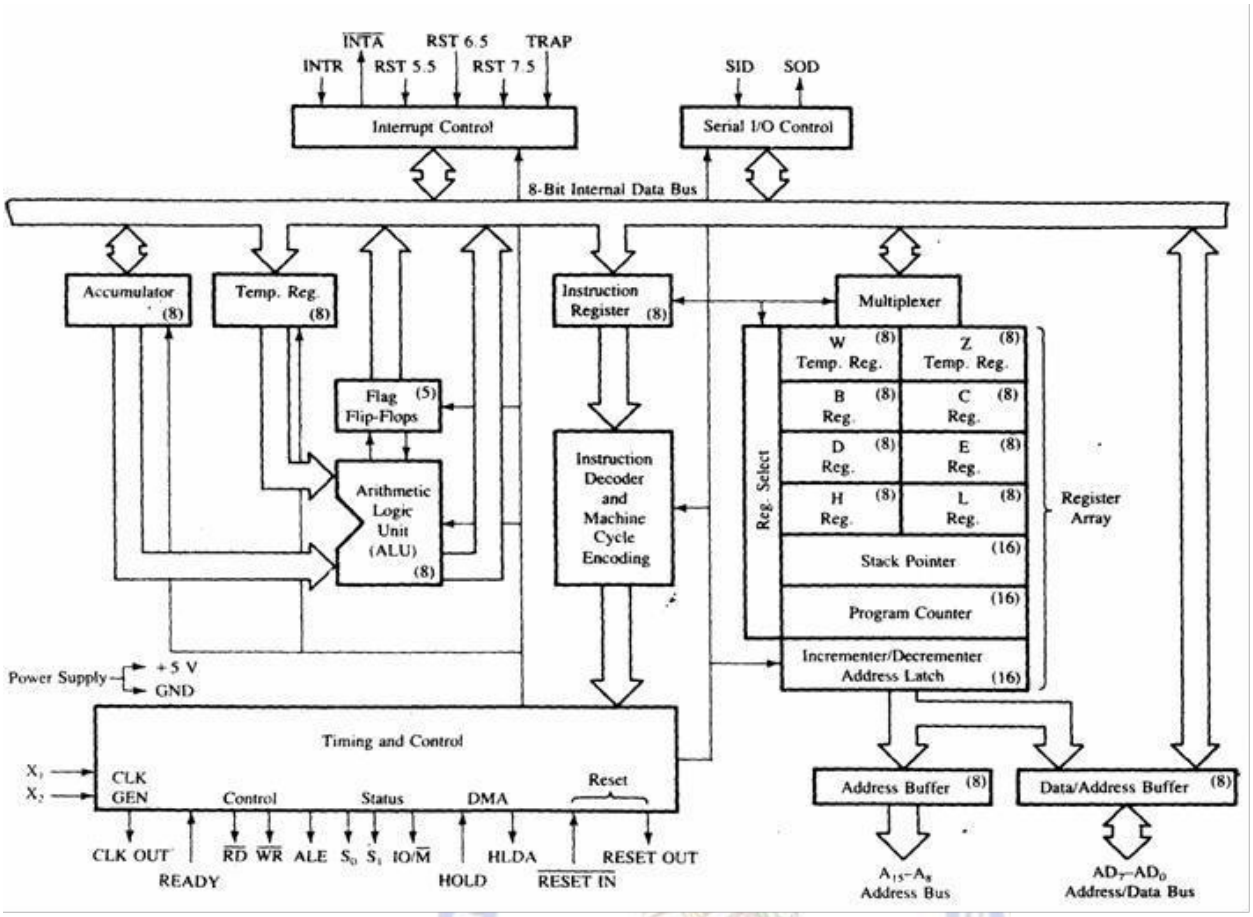
SO, S1 (Output)

Data Bus Status. Encoded status of the bus cycle:

S1 S0	OPERATION
0 0	HALT
0 1	WRITE
1 0	READ
1 1	FETCH

X1, X2 (Input)

Crystal or R/C network connections to set the internal clock generator X1 can also be



Accumulator				
A	(8)	PSW	(8)	Processor status word
B	(8)	C	(8)	
D	(8)	E	(8)	
H	(8)	L	(8)	Memory address
SP			(16)	Stack pointer
PC			(16)	Program counter

s	z		a		p		c

PSW

where
c = carry
z = zero
s = sign
p = parity
a = auxiliary carry (BCD arithmetic)

Control Unit

Generates signals within Microprocessor to carry out the instruction, which has been decoded. In reality causes certain connections between blocks of the uP to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

Arithmetic Logic Unit

The ALU performs the actual numerical and logic operation such as „add", „subtract", „AND", „OR", etc. Uses data from memory and from Accumulator to perform arithmetic. Always stores result of operation in Accumulator.

Registers

The 8085/8080A-programming model includes six registers, one accumulator, and one flag register, as shown in Figure. In addition, it has two 16-bit registers: the stack pointer and the program counter. The 8085/8080A has six general-purpose registers to store 8-bit data; these are identified as B,C, D, E, H, and L as shown in the figure. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.

For example, after an addition of two numbers, if the sum in the accumulator id larger than eight bits, the flip-flop uses to indicate a carry -- called the Carry flag (CY) – is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero (Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction. These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when CY flag is set.

Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a16-bit register.

The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location

Stack Pointer (SP)

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer.

Instruction Register/Decoder

Temporary store for the current instruction of a program. Latest instruction sent here from memory prior to execution. Decoder then takes instruction and decodes or interprets the instruction. Decoded instruction then passed to next stage.***Memory Address Register***

Holds address, received from PC, of next program instruction. Feeds the address bus with addresses of location of the program under execution.

Control Generator

Generates signals within uP to carry out the instruction which has been decoded. In reality causes certain connections between blocks of the uP to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

Register Selector

This block controls the use of the register stack in the example. Just a logic circuit which switches between different registers in the set will receive instructions from Control Unit.

8085 Addressing mode:

Addressing modes are the manner of specifying effective address. 8085 Addressing mode can be classified into:

1) Direct addressing mode: the instruction consist of three byte, byte for the op-code of the instruction followed by two bytes represent the address of the operand Low order bits of the address are in byte 2 High order bits of the address are in byte 3

Ex: **LDA 2000h;** this instruction load the Accumulator is loaded with the 8-bit content of memory location [2000h]

2) Register addressing mode The instruction specifies the register or register pair in which the data is located

Ex: **MOV A,B ;**Here the content of B register is copied to the Accumulator

3) Register indirect addressing mode The instruction specifies a register pair which contains the memory address where the data is located.

Ex. **MOV M , A ;**Here the **HL** register pair is used as a pointer to memory location. The content of Accumulator is copied to that location

4) Immediate addressing mode: The instruction contains the data itself. This is either an 8 bit quantity or 16 bit (the LSB first and the MSB is the second)

Ex: **MVI A , 28h LXI H , 2000h ;**First instruction loads the Accumulator with the 8-bit immediate data 28h Second instruction loads the **HL** register pair with 16-bit immediate data 2000h

5) Implicit addressing mode: Here the operands are implicitly in the instruction itself.

Ex: CMC –Complement carry

STC – Set Carry

Timing Diagrams of 8085:

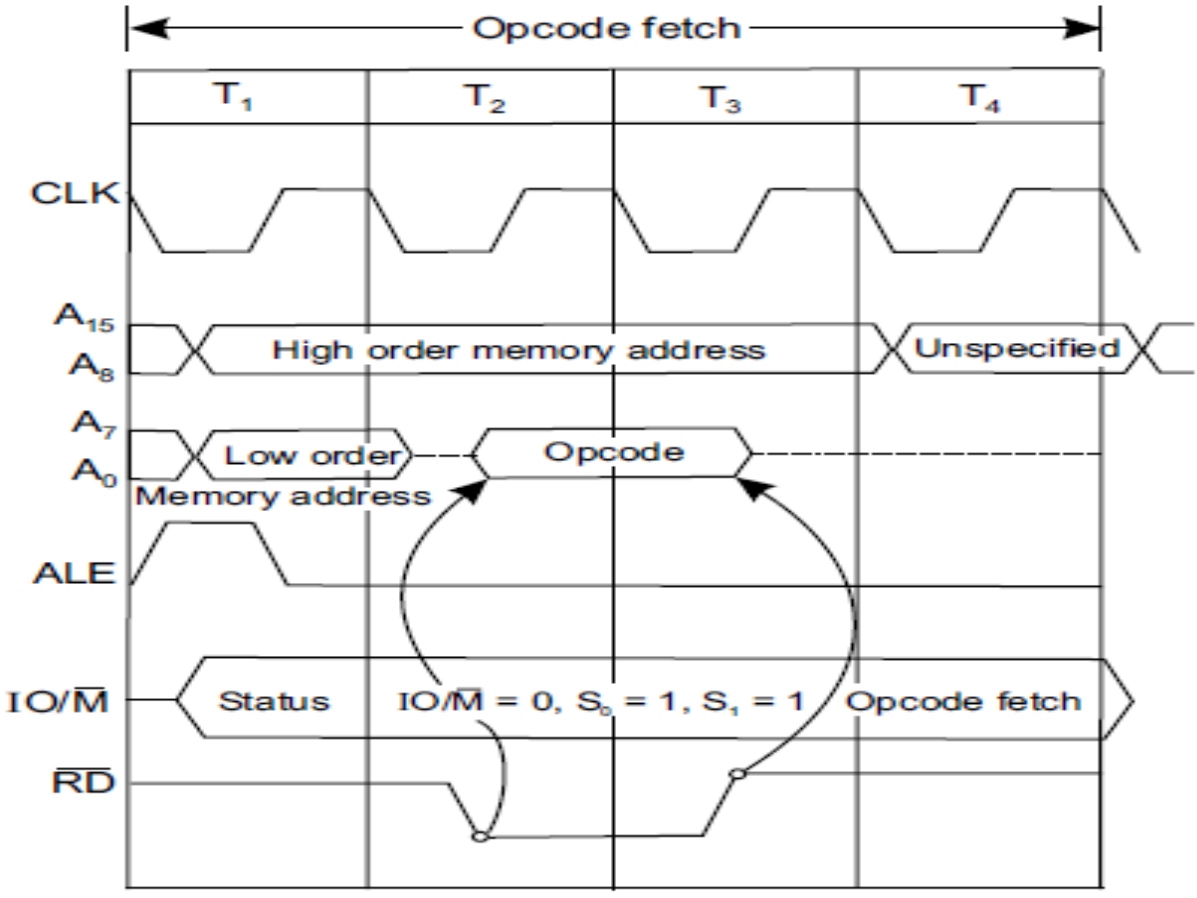
8085 has seven different machine cycles. These are:

(1) Opcode Fetch (2) Memory Read (3) Memory Write (4) I/O Read (5) I/O Write (6) Interrupt Acknowledge (7) Bus Idle.

Opcode Fetch Machine Cycle:

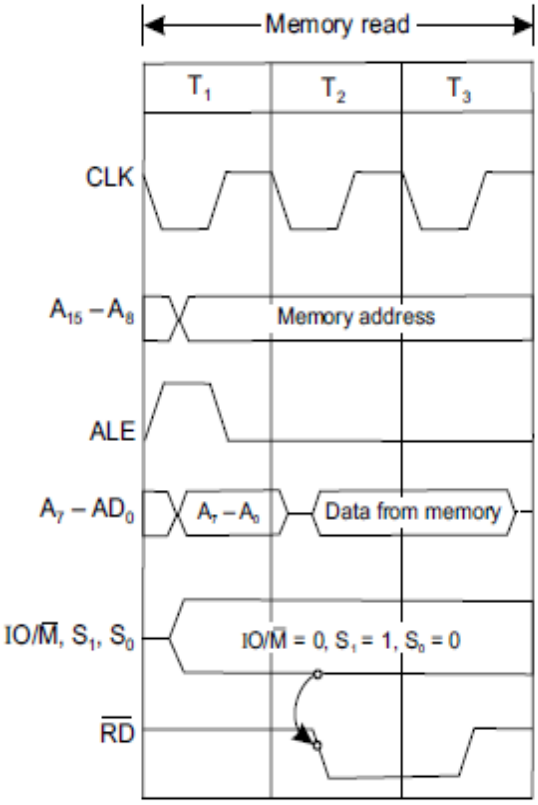
- The first machine cycle of every instruction is the Opcode Fetch. This indicates the kind of instruction to be executed by the system. The length of this machine cycle varies between 4T to 6T states—it depends on the type of instruction. In this, the processor places the contents of the PC on the address lines, identifies the nature of machine cycle (by IO/M, S0, S1) and activates the ALE signal. All these occur in T1 state.
- In T2 state, RD signal is activated so that the identified memory location is read from and places the content on the data bus (D0 – D7).
- In T3, data on the data bus is put into the instruction register (IR) and also raises the RD signal thereby disabling the memory.
- In T4, the processor takes the decision, on the basis of decoding the IR, whether to enter into T5 and T6 or to enter T1 of the next machine cycle.

One byte instructions that operate on eight bit data are executed in T4. Examples are ADD B, MOV C, B, RRC, DCR C, etc.

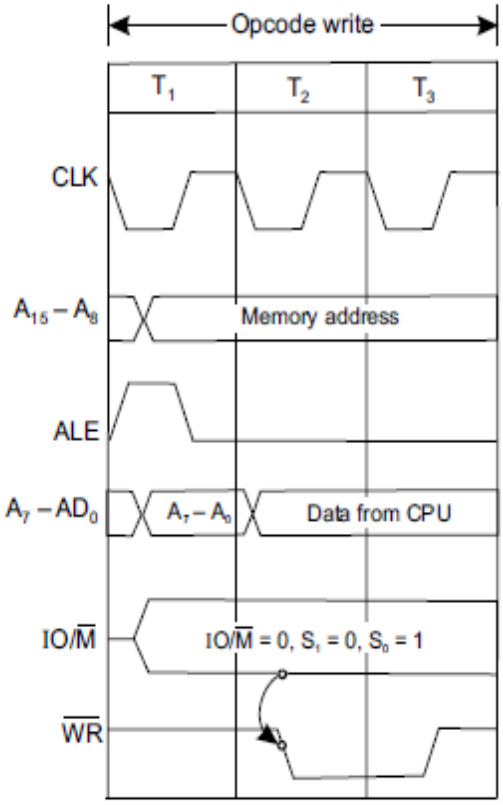


OPCODE FETCH TIMING DIAGRAM FOR 8085

Memory Read and Write Machine cycles:



(a) Memory read machine cycle



(b) Memory write machine cycle

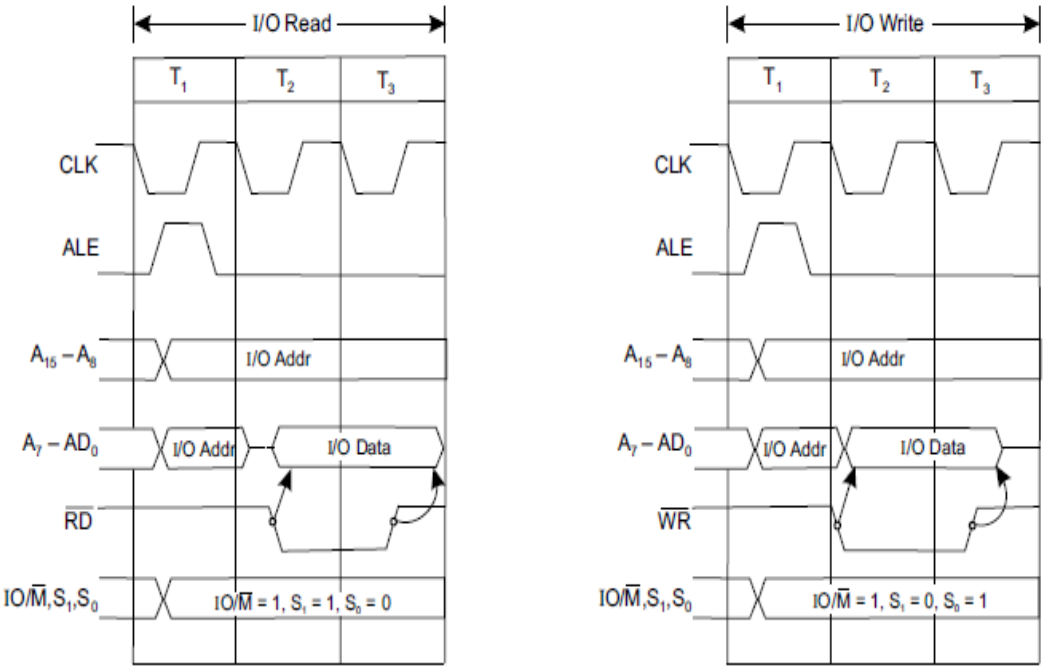
Both the Memory Read and Memory Write machine cycles are 3T states in length. In Memory Read the contents of R/W memory (including stack also) or ROM are read while in Memory Write, it stores data into data memory (including stack memory).

During T2 and T3 states data from either memory or CPU are made available in Memory Read or Memory Write machine cycles respectively. The status signal (IO/ M, S0, S1) states are complementary in nature in Memory Read and Memory Write cycles. Reading or writing operations are performed in T2.

In T3 of Memory Read, data from data bus are placed into the specified register (A, B, C, etc.) and raises RD so that memory is disabled while in T3 of Memory Write WR signal is raised which disables the memory.

IO Read and Write Machine cycles:

I/O Read and Write machine cycles are almost similar to Memory Read and Write machine cycles respectively. The difference here is in the IO/ M signal status which remains 1 indicating that these machine cycles are related to I/O operations. These machine cycles take 3T states. In I/O read, data are available in T2 and T3 states, while during the same time (T2 and T3) data from CPU are made available in I/O write.



(a) I/O read machine cycle (b) I/O write machine cycle

Instruction Set

An instruction is a command given to the microcomputer to perform a specific task or function on a given data. An instruction comprises of an operation code (called 'opcode') and the address of the data (called 'operand'), on which the opcode operates. This is the structure on which an instruction is based. The opcode specifies the nature of the task to be performed by an instruction. Symbolically, an instruction looks like

Operation code Address of data
opcode operand

An instruction set is a collection of instructions that the microprocessor is designed to perform.
Functionally, the instructions can be classified into five groups:

- Data transfer (copy) group
- Arithmetic group
- Logical group
- Branch group
- Stack, I/O and machine control group.

Data transfer (copy) group

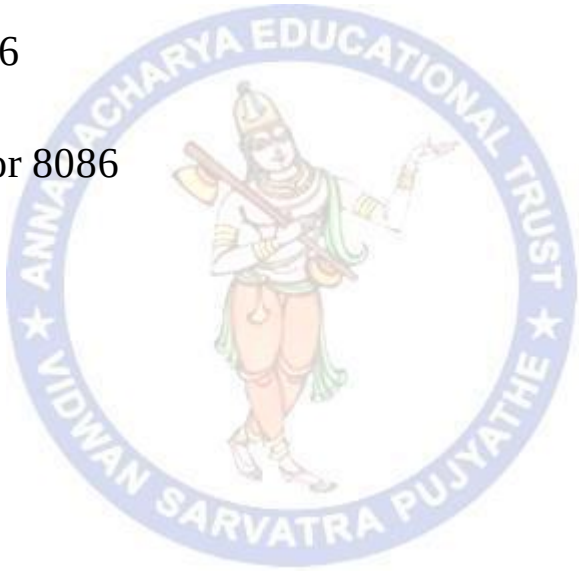
The different types of data transfer operations possible are cited below:

- Between two registers.
- Between a register and a memory location.
- A data byte can be transferred between a register and a memory location.
- Between an I/O device and the accumulator.
- Between a register pair and the stack.

UNIT – II

OVERVIEW

- Introduction to 8086 microprocessors
- Architecture of 8086 processors
- Register Organization of 8086
- Memory Segmentation of 8086
- Pin Diagram of 8086
- Timing Diagrams for 8086
- Interrupts of 8086



UNIT-II

Features of 8086:

•It is a 16-bit µp.
•8086 has a 20 bit address bus can access up to 2^20 memory locations (1 MB).
•It can support up to 64K I/O ports.
•It provides 14, 16 -bit registers.
•It has multiplexed address and data bus AD0- AD15 and A16 – A19.
•It requires single phase clock with 33% duty cycle to provide internal timing.
•8086 is designed to operate in two modes, Minimum and Maximum.
•It can pre-fetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
•It requires +5V power supply.
•A 40 pin dual in line package.

Architecture of 8086:

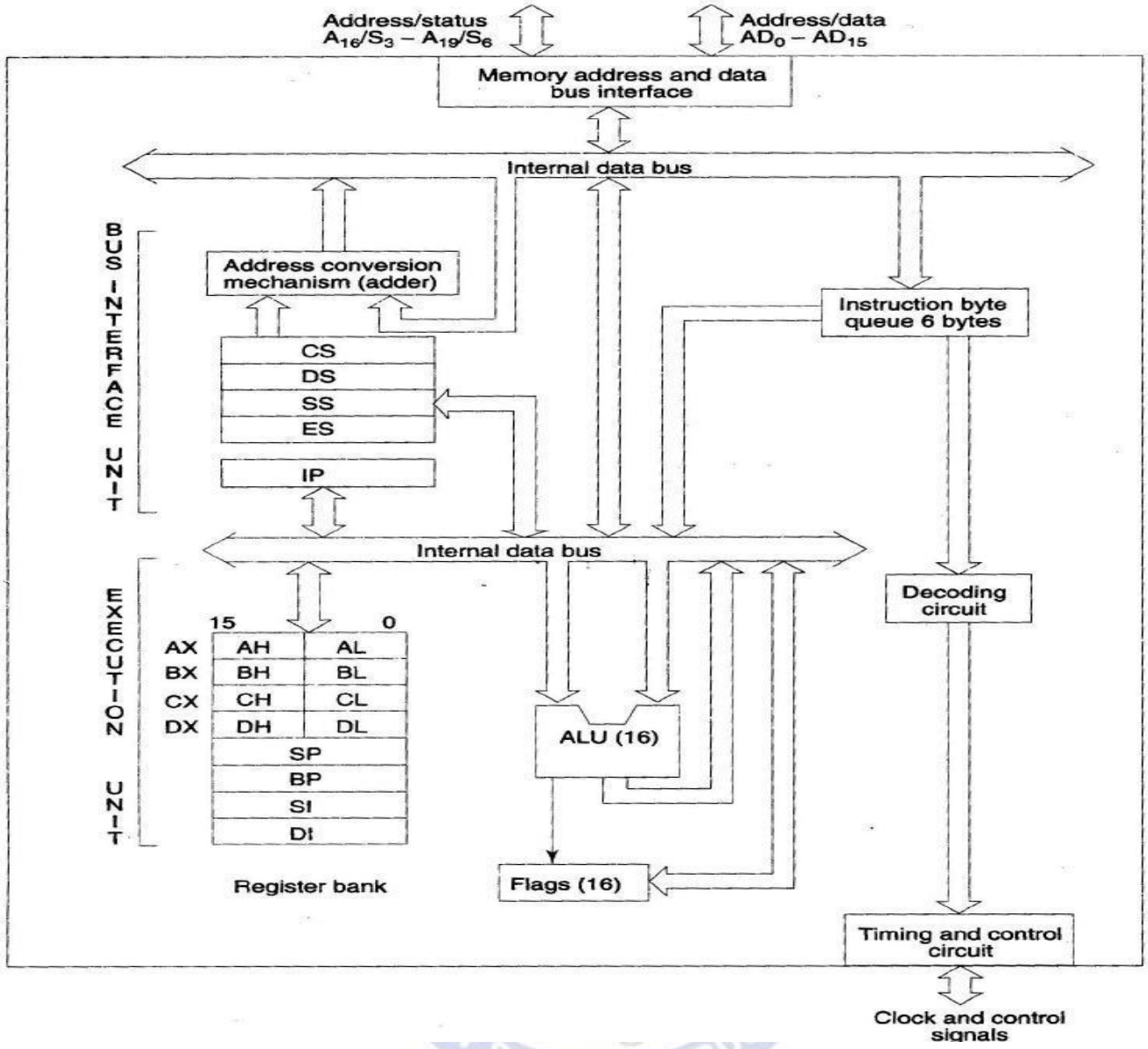
- 8086 has two blocks BIU and EU.
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.
- EU executes instructions from the instruction byte queue.
- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as **Pipelining.** This results in efficient use of the system bus and system performance.
- BIU contains Instruction queue, Segment registers, IP, address adder.
- EU contains control circuitry, Instruction decoder, ALU, Flag register.

Bus Interface Unit:

- It provides full 16 bit bidirectional data bus and 20 bit address bus.
- The BIU is responsible for performing all external bus operations.

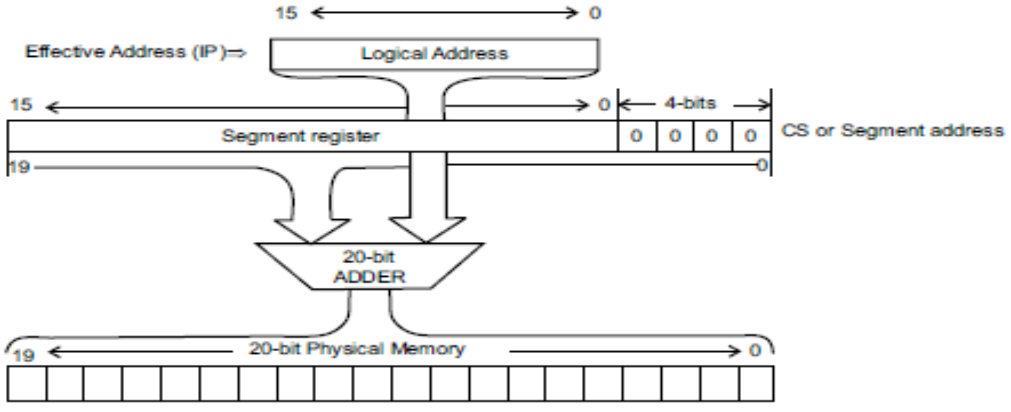
Specifically it has the following functions:

- Instructions fetch Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture.
- This queue permits pre-fetch of up to six bytes of instruction code. Whenever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by pre-fetching the next sequential instruction.
- These pre-fetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.
- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.
- These intervals of no bus activity, which may occur between bus cycles, are known as **idle state**.



• If the bus is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.

• The BIU also contains a dedicated adder which is used to generate the 20 bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.



Physical address generation

Thus, Physical Address = Segment Register content 16 D + Offset

• For example: The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.

• The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Execution Unit:

• The EU extracts instructions from top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands.

• During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.

• If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.

• When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.

• Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

Register organization of 8086:

The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointer and index register, four segment registers. The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register***, with 9 of bits implemented for status and control flags.

There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located the processor uses four segment registers:

•**Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

•**Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

•**Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment.DS register can be changed directly using POP and LDS instructions.

•**Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX. AL in this case contains the low order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations and string manipulation.

•**Base** register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

•**Count** register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.

•**Data** register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

•**The following registers are both general and index registers:**

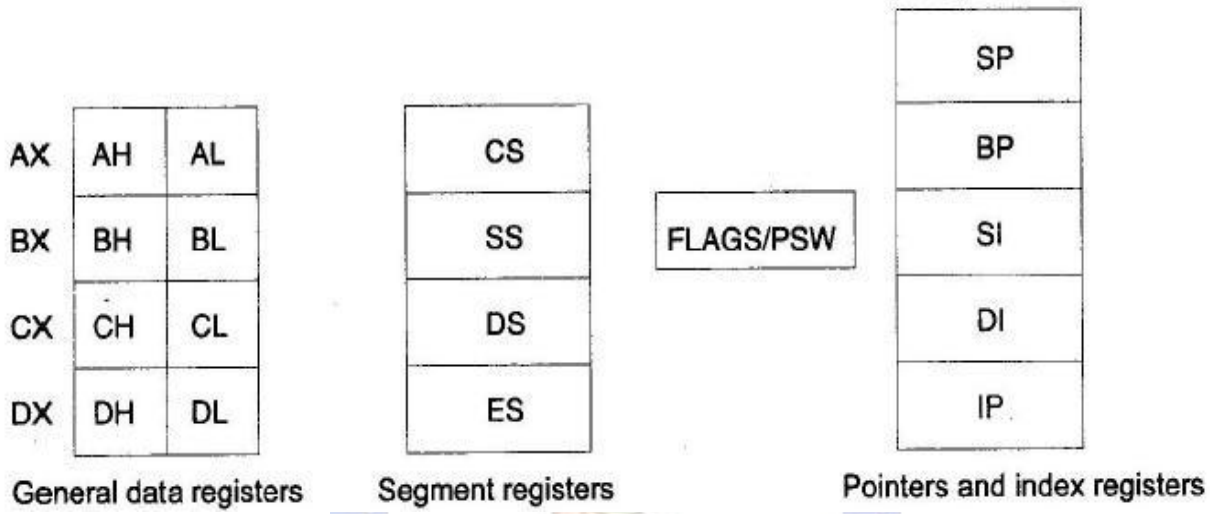
•**Stack Pointer** (SP) is a 16-bit register pointing to program stack.

•**Base Pointer** (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

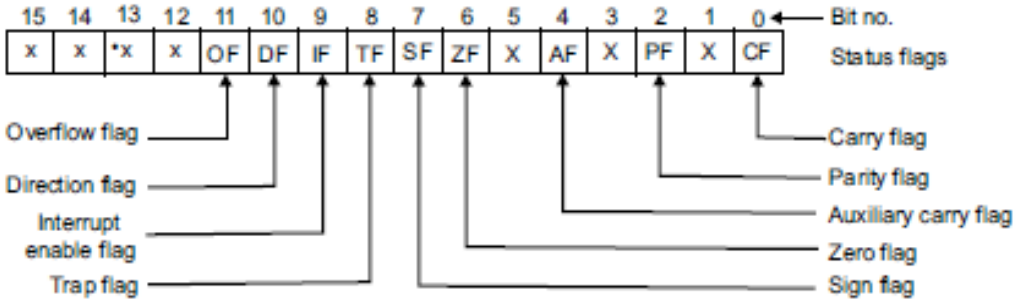
•**Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

•**Destination Index** (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data addresses in string manipulation instructions.

Instruction Pointer (IP) register acts as a program counter for 8086. It points to the address of the next instruction to be executed. Its content is automatically incremented when the program execution of a program proceeds further. The contents of IP and CS register are used to compute the memory address of the instruction code to be fetched.



Flag register of 8086: It is a 16-bit register, also called flag register or Program Status Word (PSW). Seven bits remain unused while the rest nine are used to indicate the conditions of flags. The status flags of the register are shown below in Fig.



Status flags of Intel 8086

- Out of nine flags, six are condition flags and three are control flags. The control flags
- are TF (Trap), IF (Interrupt) and DF (Direction) flags, which can be set/reset by the
- programmer, while the **condition flags [OF (Overflow), SF (Sign), ZF (Zero), AF (Auxiliary**
- **Carry), PF (Parity) and CF (Carry)]** are set/reset depending on the results of some arithmetic or logical operations during program execution.
- **CF is set** if there is a carry out of the MSB position resulting from an addition operation or if a borrow is needed out of the MSB position during subtraction.
- **PF is set** if the lower 8-bits of the result of an operation contains an even number of 1's. AF is set if there is a carry out of bit 3 resulting from an addition operation or borrow required from bit 4 into bit 3 during subtraction operation.
- **ZF is set** if the result of an arithmetic or logical operation is zero.

- **SF is set** if the MSB of the result of an operation is 1. SF is used with unsigned numbers.
- OF is used only for signed arithmetic operation and is set if the result is too large to be fitted in the number of bits available to accommodate it.

  The three control flags of 8086 are TF, IF and DF. These three flags are programmable, i.e., can be set/reset by the programmer so as to control the operation of the processor.
- **When TF (trap flag) is set (=1),** the processor operates in **single stepping mode**—i.e., pausing after each instruction is executed. This mode is very useful during program development or program debugging.
- When an interrupt is recognized, TF flag is cleared. When the CPU returns to the main program from ISS (interrupt service subroutine), by execution of IRET in the last line of ISS, TF flag is restored to its value that it had before interruption.
- TF cannot be directly set or reset. So indirectly it is done by pushing the flag register on the stack, changing TF as desired and then popping the flag register from the stack.
- **When IF (interrupt flag) is set,** the maskable interrupt INTR is enabled otherwise disabled (i.e., when IF = 0).
- **IF can be set by executing STI instruction and cleared by CLI instruction.** Like TF flag, when an interrupt is recognized, IF flag is cleared, so that INTR is disabled. In the last line of ISS when IRET is encountered, IF is restored to its original value. When 8086 is reset, IF is cleared, i.e., resetted.
- DF (direction flag) is used in string (also known as block move) operations. **It can be set by STD instruction and cleared by CLD.** If DF is set to 1 and MOVS instruction is executed, the contents of the index registers DI and SI are automatically decremented to access the string from the highest memory location down to the lowest memory location.

PIN DIAGRAM OF 8086

The **8086** is internally a **16-bit MPU** and **externally** it has a **16-bit data bus**. It has the ability to address up to **1 MB** of memory via its **20-bit address bus**. In addition, it can address up to **64K of byte-wide input/output ports**.

• It is manufactured using **high-performance metal-oxide semiconductor (HMOS) technology**, and the circuitry on its chip is equivalent to approximately **29,000 transistors**.

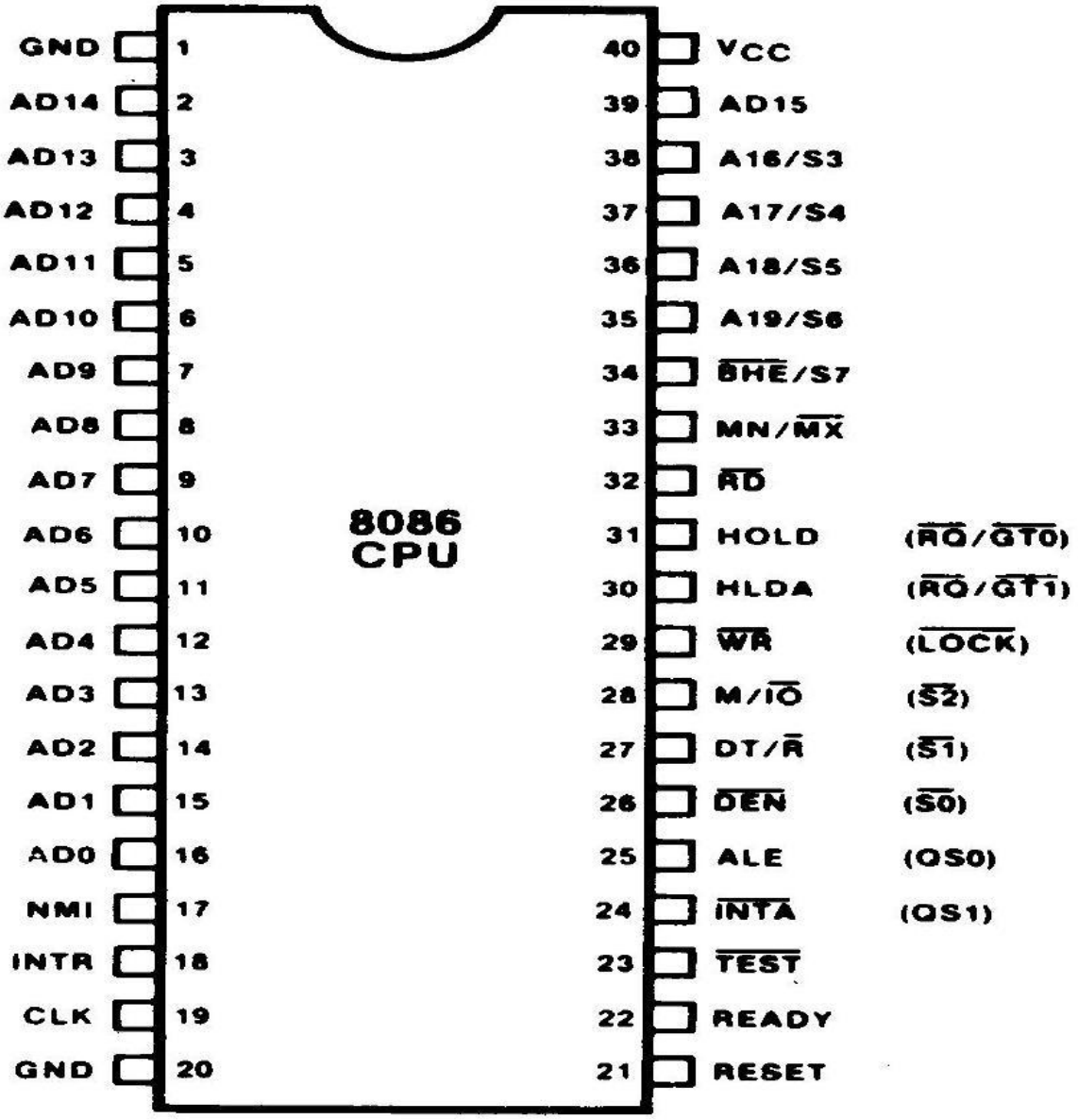
• The 8086 is housed in a **40-pin dual in-line package**. The signals pinned out to each lead are shown in figure.

The **address bus lines** A0 through A15 and **data bus lines** D0 through D15 are **multiplexed**. For this reason, these leads are labeled AD0 through AD15. By *multiplexed* we mean that the same physical pin carries an address bit at one time and the data bits at another time.

• The **8086** can be configured to work in either of **two modes**:

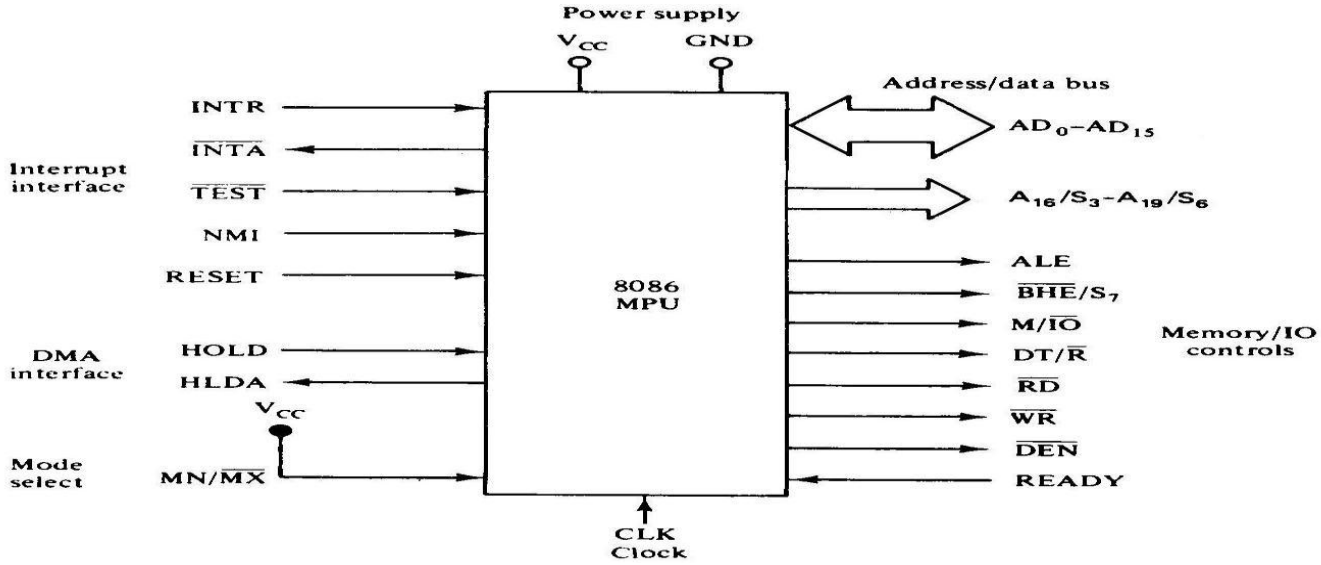
• The **minimum mode** is selected by applying **logic 1** to the **MN/MX** input lead. It is typically used for smaller **single microprocessor** systems.

• The **maximum mode** is selected by applying **logic 0** to the **MN/MX** input lead. It is typically used for larger **multiple microprocessor** systems.

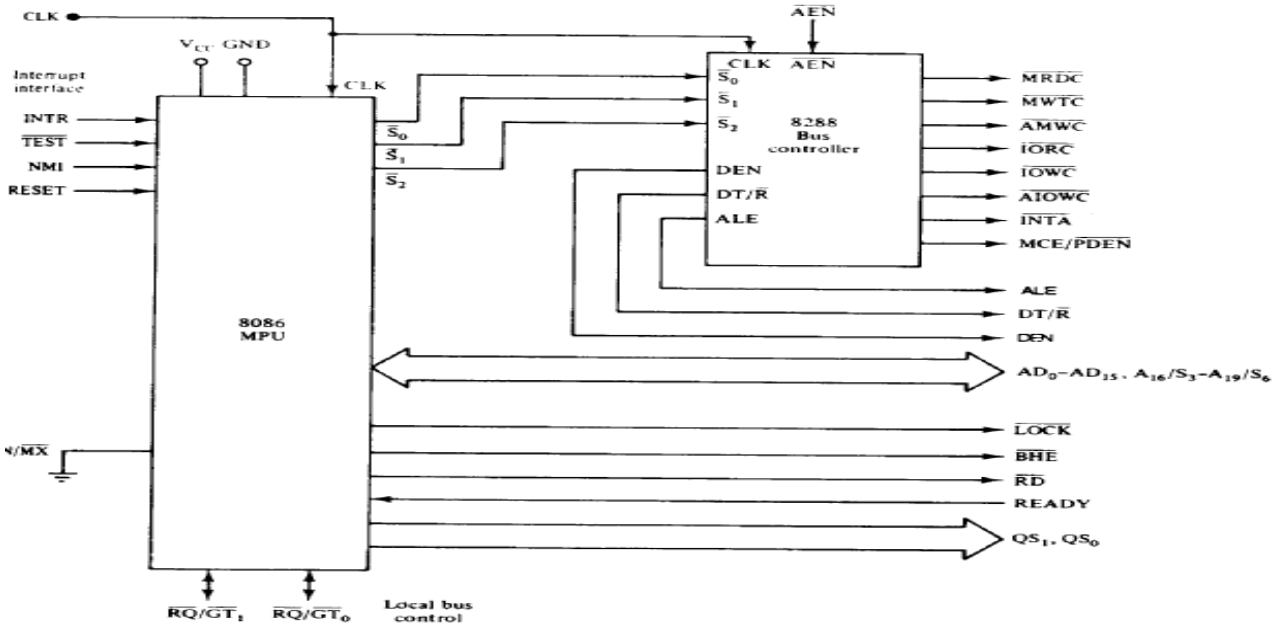


• Depending on the **mode** of operation selected, the **assignments** for a number of the **pins** on the microprocessor package are **changed**. The **pin functions** specified in **parentheses** pertain to the **maximum-mode**.

• In minimum mode, the **8086** itself **provides** all the **control signals** needed to implement the memory and I/O interfaces. In **maximum-mode**, a separate chip (the **8288 Bus Controller**) is used to help in sending control signals over the shared bus shown in figure.



MINIMUM MODE OF 8086



MAXIMUM MODE OF 8086

• **Address/Data Bus:** The address bus is **20 bits long** and consists of signal lines **A0** (LSB) through **A19** (MSB). However, only address lines **A0** through **A15** are used when accessing **I/O**.

• The **data bus** lines are **multiplexed** with address lines. For this reason, they are denoted as **AD0** through **AD15**. Data line **D0** is the LSB.

• **Status Signals:** The four most significant address lines **A16** through **A19** of the 8086 are multiplexed with **status signals S3** through **S6**. These status bits are output on the bus at the same time that data are transferred over the other bus lines.

The status of the Interrupt Enable Flag (IF) bit (displayed on S5) is updated at the beginning of each clock cycle.

S4, S3: together indicates which segment register is presently being used for memory access. These lines float at tristate off during the local bus hold acknowledge.

S6: It is always low.

S4	S3	Indications
0	0	Alternate data
0	1	Stack
1	0	Code or none
1	1	Data

BHE/S7-Bus High Enable/Status: The bus high enable signal is used to indicate the transfer of data over the higher order (D15-D8) data bus. It goes low for the data transfers over D15-D8 and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T1 for read, write and interrupt acknowledge cycles, when- ever a byte is to be transferred on the higher byte of the data bus.

BHE	A0	Indication
0	0	Whole word i.e AD15 – AD8
0	1	Upper byte from or to i.e AD15-AD8
1	0	Lower byte from or to even address i.e AD7-AD0
1	1	None

TEST: This input is examined by a 'WAIT' instruction. If the TEST input goes low, execution will continue, else, the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

RESET: This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronized.

VCC: +5V power supply for the operation of the internal circuit. GND ground for the internal circuit.

• Control Signals:

• When ***Address latch enable*** (**ALE**) is **logic 1** it signals that a **valid address** is on the bus. This address can be latched in external circuitry on the **1-to-0 edge** of the pulse at ALE.

• **M/IO** (*memory*/IO) tells external circuitry whether a memory or I/O transfer is taking place over the bus. **Logic 1** signals a **memory operation** and **logic 0** signals an **I/O operation**.

• **DT/R** (*data transmit/receive*) signals the **direction of data transfer** over the bus. **Logic 1** indicates that the bus is in the **transmit mode** (i.e., data are either written into memory or to an I/O device). **Logic 0** signals that the bus is in the **receive mode** (i.e., reading data from memory or from an input port).

• The *bank high enable* (**BHE**) signal is used as a **memory enable signal** for the **most significant byte** half of the data bus, **D8** through **D15**.

• **WR** (*write*) is switched to **logic 0** to signal external devices that **valid output data** are on the bus.

• **RD** (*read*) indicates that the MPU is performing a **read of data** off the bus. During read operations, one other control signal, **DEN** (*data enable*), is also supplied. It enables external devices to supply data to the microprocessor.

• The **READY** signal can be used to **insert wait states** into the bus cycle so that it is extended by a number of clock periods. This signal is supplied by a **slow** memory **or I/O subsystem** to signal the MPU when it is ready to permit the data transfer to be completed.

• Interrupt Signals:

• *Interrupt request* (**INTR**) is an **input** to the 8086 that can be used by an **external device** to **signal** that it needs to be **serviced**. **Logic 1** at INTR represents an active interrupt request.

• When the MPU **recognizes an interrupt request**, it indicates this fact to external circuits with logic 0 at the *interrupt acknowledge* (**INTA**) output.

• On the **0-to-1 transition** of *non maskable interrupt* (**NMI**), control is passed to a non maskable **interrupt service routine** at completion of execution of the current instruction. NMI is the interrupt request with highest priority and **cannot be masked by software**.

• The **RESET** input is used to provide a **hardware reset** for the MPU. Switching RESET to **logic 0** initializes the internal registers of the MPU and initiates a reset service routine.

• DMA Interface Signals:

• When an **external device** wants to **take control** of the **system bus**, it signals this fact to the MPU by switching HOLD to the **logic level 1**.

• When in the hold state, lines **AD0** through **AD15**, **A16/S3** through **A19/S6**, **BHE**, **M/IO**, **DT/R**, **WR, RD**, **DEN** and **INTR** are all put in the **high-Z** state. The MPU signals external devices that it is in this state by switching **HLDA** to **1**.

SYSTEM CLOCK:

• To **synchronize** the internal and external operations of the microprocessor a *clock* (**CLK**) **input signal** is used. The CLK can be generated by the **8284 clock generator IC**.

• The **8086** is manufactured in three speeds: **5 MHz**, **8 MHz** and **10 MHz**.

MAXIMUM MODE SIGNALS:

S2, S1, S0 (Status lines): These are the status lines which reflect the type of operation, being carried out by the processor. These lines active during T4 of the previous cycle & remain active during T1 & T2 of the current bus cycle.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Indication
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1	0	0	Code Access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

LOCK:

· This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the LOCK=0.

· The LOCK signal is activated by the LOCK prefix instruction and remains active until the completion of the next instruction.

· This floats to tri-state off during 'hold acknowledge'.

QS1, QS0 (Queue status):

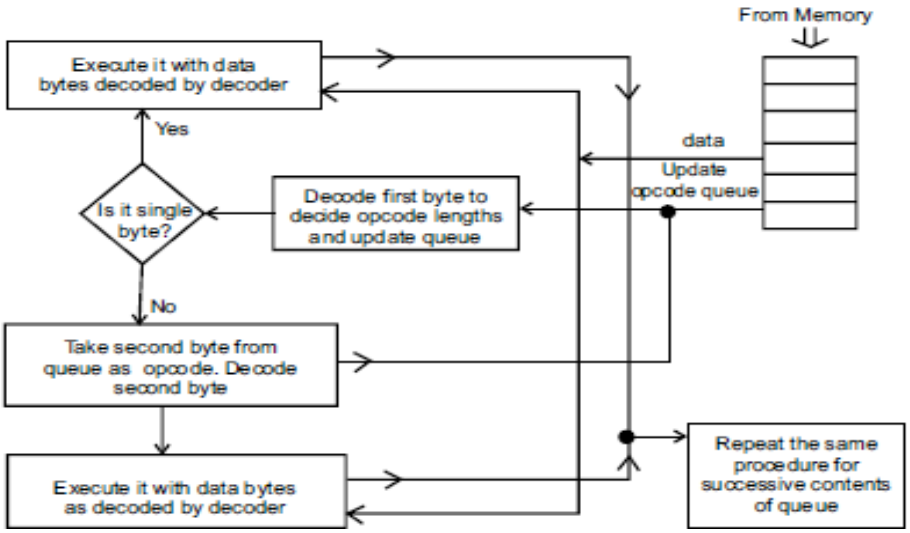
· These lines give information about the status of the code-prefetch queue.

· These are active during the CLK cycle after which the queue operation is performed.

· The 8086 architecture has a 6-byte instruction pre-fetch queue.

QS_1,	QS_0	Indication
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is **single opcode byte**, the next bytes are treated as data byte depending upon the decoded instruction length; otherwise, the next byte in the queue is treated as the **second byte** of the instruction opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least, two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.



RQ/GT0, RQ/GT1 (Request/Grant):

These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with RQ_0/GT_0 having higher priority than RQ_1/GT_1.

Minimum Mode 8086 System

•In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.

•In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.

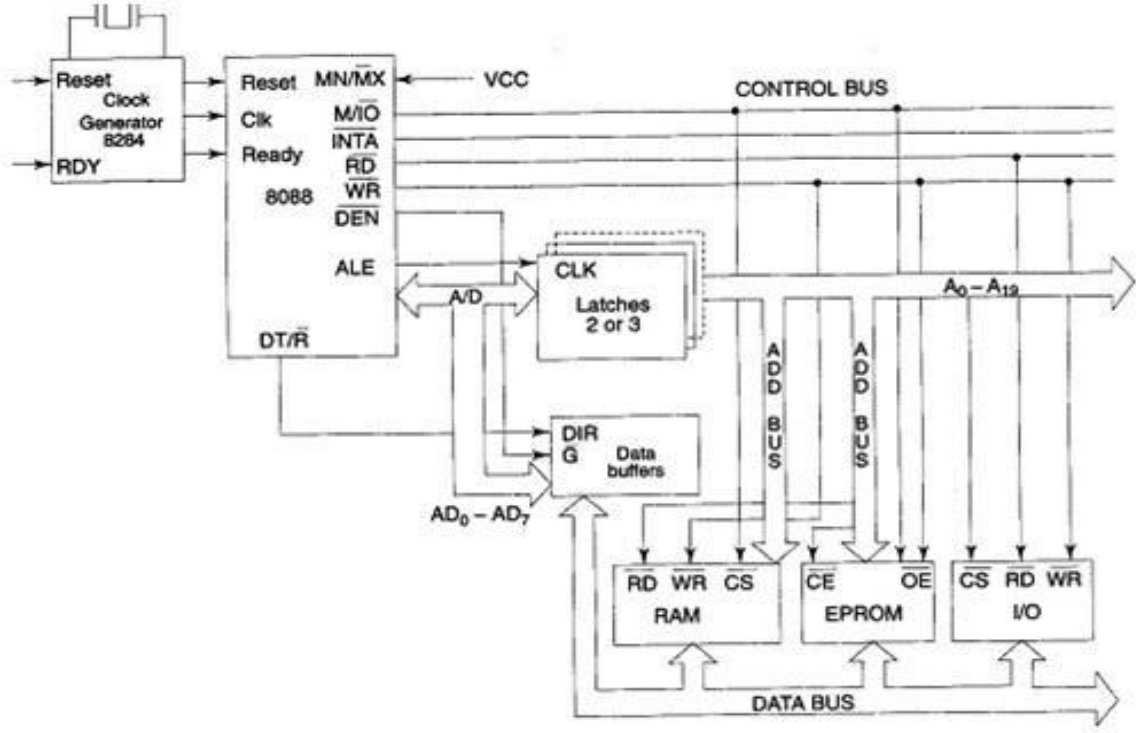
•The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

•Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

•Transreceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals.

•They are controlled by two signals namely, DEN and DT/R.

•The DEN signal indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage.



MINIMUM MODE SYSTEM

•Usually, EPROMs are used for monitor storage, while RAM for users program storage. A system may contain I/O devices.

•The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.

•The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.

•The BHE and A0 signals address low, high or both bytes. From T1 to T4, the M/IO signal indicates a memory or I/O operation.

•At T2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T2.

•The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.

•The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

•A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/O operation. In T2, after sending the address in T1, the processor sends the data to be written to the addressed location.

•The data remains on the bus until middle of T4 state. The WR becomes active at the beginning of T2 (unlike RD is somewhat delayed in T2 to provide time for floating).

•The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.

Maximum Mode 8086 System

•In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.

In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information.

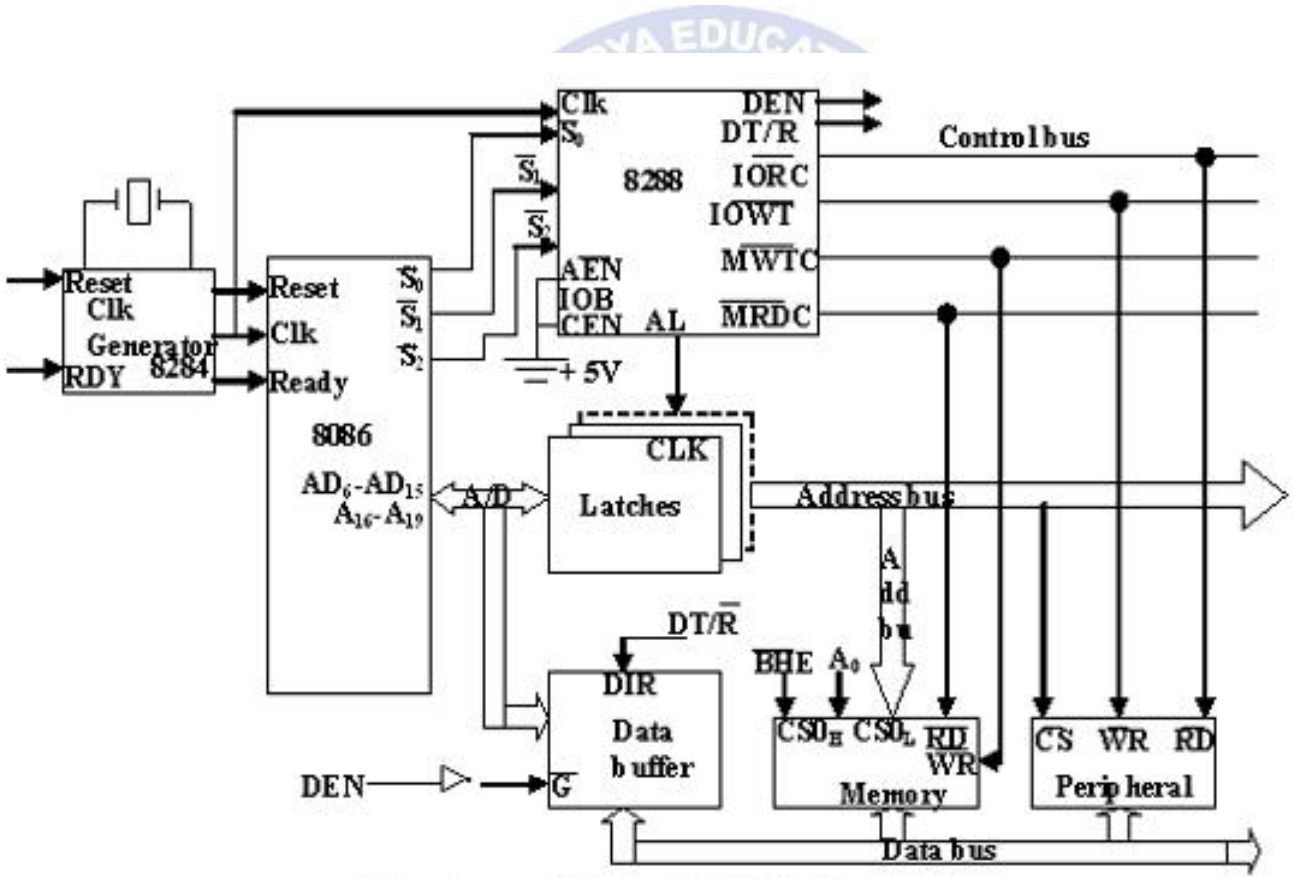
•In the maximum mode, there may be more than one microprocessor in the system configuration.

•The components in the system are same as in the minimum mode system.

•The basic function of the bus controller chip IC8288, is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.

•The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU.

•It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are specially useful for multiprocessor systems.



Maximum Mode 8086 System.

•AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/PDEN output depends upon the status of the IOB pin.

•If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations.

•INTA pin used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

•IORC, IOWC are I/O read command and I/O write command signals respectively.

These signals enable an IO interface to read or write the data from or to the address port.

•The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.

•All these command signals instructs the memory to accept or send data from or to the bus.

•For both of these write command signals, the advanced signals namely AIOWC and AMWTC are available.

•Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals.

•R0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.

•In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or

AIOWC is activated from T2 to T4 and MWTC or IOWC is activated from T3 to T4.

•The status bit S0 to S2 remains active until T3 and become passive during T3 and T4.

•If reader input is not activated before T3, wait state will be inserted between T3 and T4.

TIMING DIAGRAMS FOR 8086 IN MINIMUM MODE

BUS CYCLE AND TIME STATES

• A **bus cycle or machine cycle** defines the sequence of events when the MPU communicates with an external device, which starts with an address being output on the system bus followed by a read or write data transfer.

• Types of bus cycles:

Memory Read Bus Cycle

Memory Write Bus Cycle

Input/output Read Bus Cycle

Input/output Write Bus Cycle

One cycle of clock is called a state or t-state. The bus cycle of the 8086 microprocessor consists of at least four clock periods. These four time states are called T1, T2, T3 and T4. This group of states is called a **MACHINE CYCLE.**

The total time required to fetch and execute an instruction is called an **instruction cycle**. An instruction cycle consists of one or more machine cycle.

The following figure **shows** a **memory read cycle** of the **8086**:

• During **period T1**,

- The 8086 outputs the **20-bit address** of the memory location to be accessed on its multiplexed **address/data bus**. **BHE** is also output along with the address during T1.
- At the same time a pulse is also produced at **ALE**. The **trailing edge** or the **high level** of this pulse is used to **latch** the address in external circuitry.
- Signal **M/IO** is set to **logic 1** and signal **DT/R** is set to the **0 logic level** and both are maintained throughout all four periods of the bus cycle.

• Beginning with **period T2**,

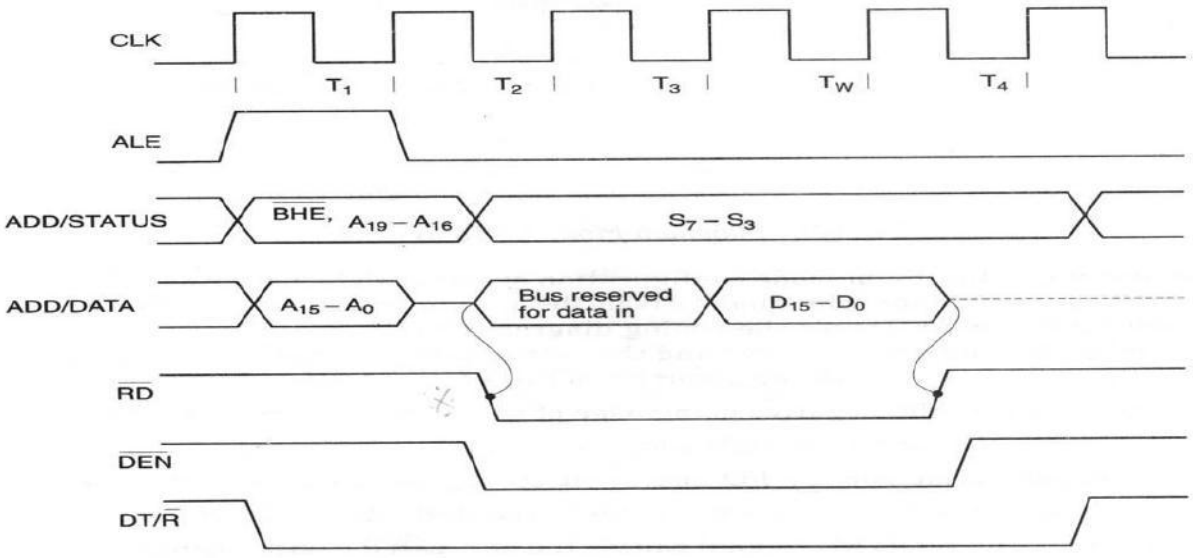
- Status bits **S3** through **S6** are output on the upper four address bus lines. This status information is maintained through periods **T3** and **T4**.
- On the other hand, address/data bus lines **AD0 through AD7** are put in the **high-Z state** during **T2**.
- Late in period **T2**, **RD** is switched to **logic 0**. This indicates to the memory subsystem that a read cycle is in progress. **DEN** is switched to **logic 0** to enable external circuitry to allow the data to move from memory onto the microprocessor's data bus.

• During **period T3**,

- The memory must provide **valid data** during **T3** and maintain it until after the processor terminates the read operation. The data read by the 8086 microprocessor can be carried over all **16 data bus** lines.

• During **T4**,

- The 8086 switches **RD** to the inactive **1 logic level** to terminate the read operation. **DEN** returns to its inactive logic level late during **T4** to disable the external circuitry.



MEMORY READ CYCLE FOR 8086 IN MINIMUM MODE

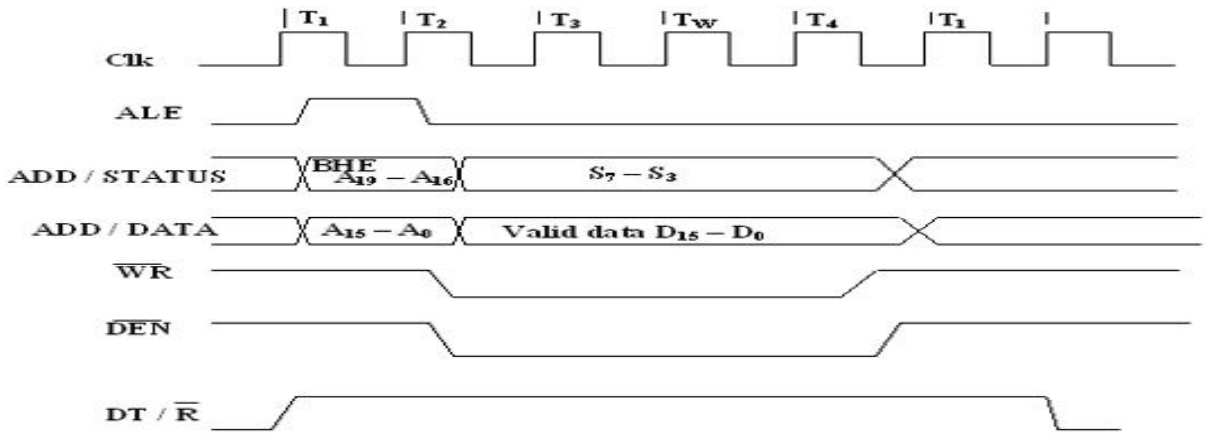
The following figure shows a **memory write cycle** of the 8086:

• During **period T1**,

- The **address** along with **BHE** is output and latched with the **ALE** pulse.
- **M/IO** is set to **logic 1** to indicate a memory cycle.
- However, this time **DT/R** is switched to **logic 1**. This signals external circuits that the 8086 is going to **transmit data** over the bus.

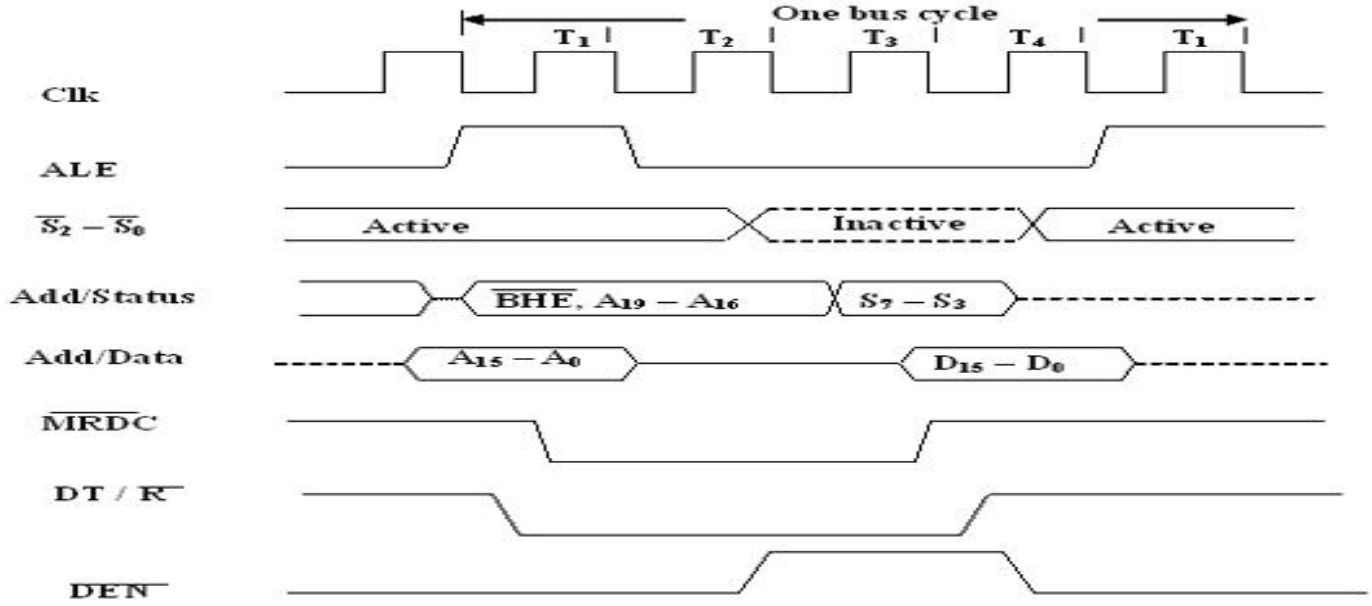
• Beginning with **period T2**,

- **WR** is switched to **logic 0** telling the memory subsystem that a write operation is to follow.
- The 8086 puts the **data** on the bus late in **T2** and maintains the data valid through **T4**. Data will be carried over all **16 data bus lines**.
- **DEN** enables the external circuitry to provide a path for data from the processor to the memory.

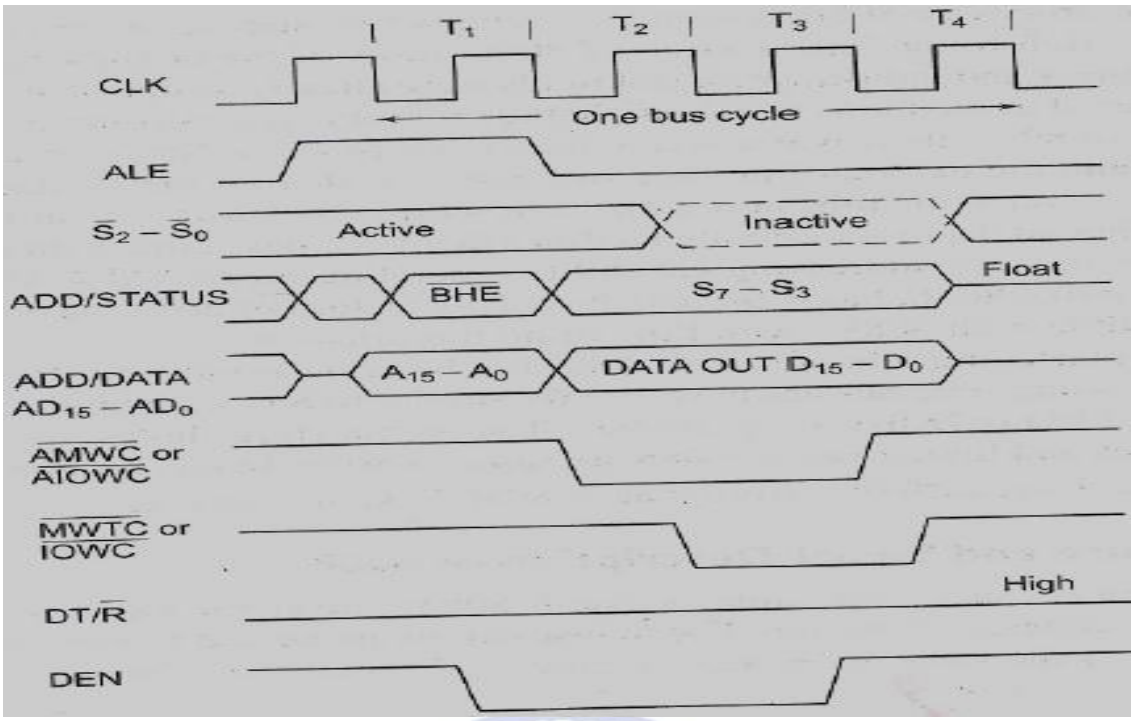


Write Cycle Timing Diagram for Minimum Mode

MAXIMUM MODE TIMING DIGRAMS



Memory Read Timing in Maximum Mode



WRITE CYCLE TIMING DIAGRAM FOR 8086

UNIT – III

OVERVIEW:

- Addressing Modes of 8086
- Assembler Directives
- Procedures and Macros
- Instruction Set of 8086
    - Data Transfer Group
    - Arithmetic Group
    - Logical Instructions
    - Rotate and Shift instructions
    - Loop Instructions
    - Conditional and Unconditional instructions
    - Machine Control and Flag Manipulation instructions
- Programming on 8086

UNIT III

ADDRESSING MODES OF 8086:

Addressing modes indicates way of locating data or operands. Depending upon the data types used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes. Thus the addressing modes describe the types of operands and the way they are accessed for executing an instruction.

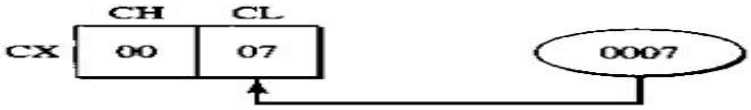
According to the flow of instruction execution, the instruction may be categorized as:

Sequential Control flow instructions Control Transfer instructions

- **Sequential Control flow instructions:** In this type of instruction after execution control can be transferred to the next immediately appearing instruction in the program.

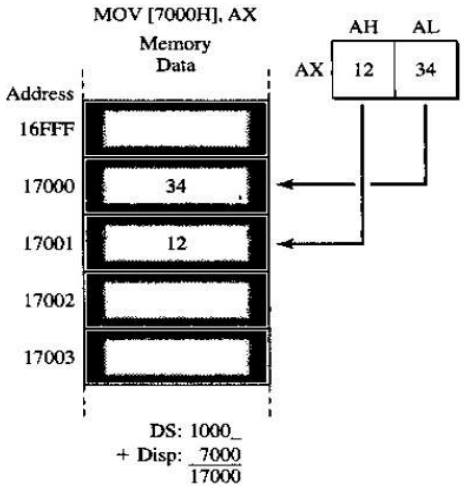
  The addressing modes for sequential control transfer instructions are as follows:
- **Immediate addressing mode:** In this mode, immediate is a part of instruction and appears in the form of successive byte or bytes.

  Example: MOV CX, 0007H; Here 0007 is the immediate data



- **Direct Addressing mode:** In this mode, the instruction operand specifies the memory address where data is located.

  Example: MOV AX, [5000H]; Data is available in 5000H memory location



Effective Address (EA) is computed using 5000H as offset address and content of DS as segment address.

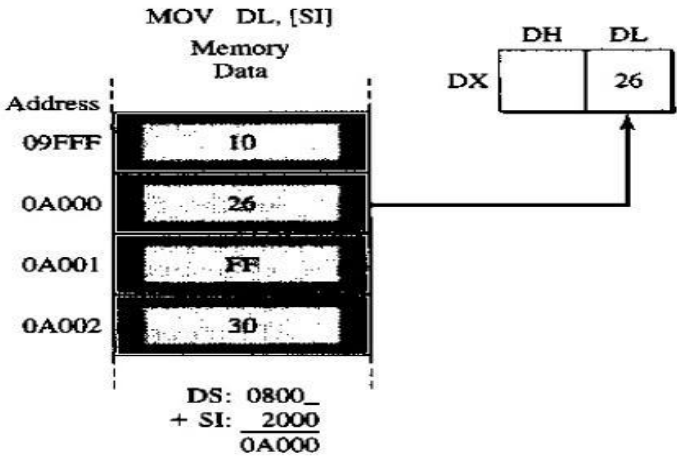
EA=10H * DS + 5000H

- **Register Addressing mode:** In this mode, the data is stored in a register and it is referred using particular register. All the registers except IP may be used in this mode.

  Example: MOV AX, BX;
- **Register Indirect addressing mode:** In this mode, instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.

  Example: MOV AX, [BX]; **EA=10H * DS + [BX]**
- **Indexed Addressing mode:** 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides. DS and ES are

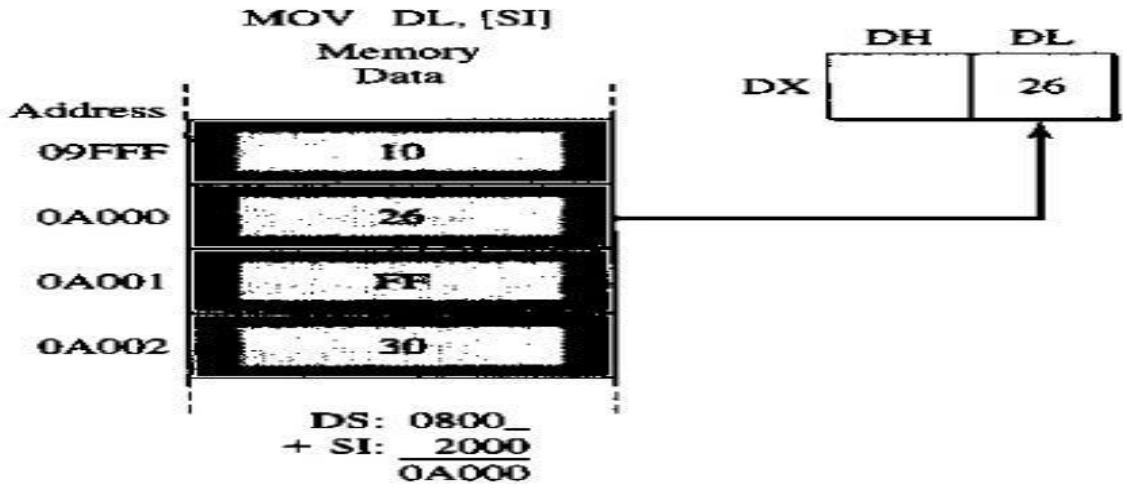
default segments for index registers SI and DI. DS=0800H, SI=2000H, MOV DL, [SI]



Example: MOV AX, [SI]; **EA=10H * DS + [SI]**

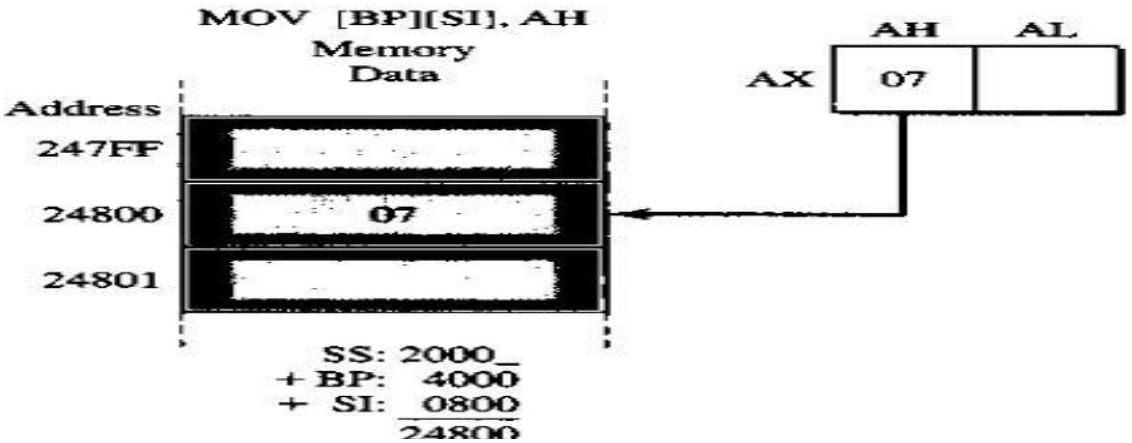
- **Register Relative Addressing mode:** In this mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI, DI in the default segments.

  Example: MOV AX, 50H [BX]; **EA=10H * DS + 50H + [BX]**

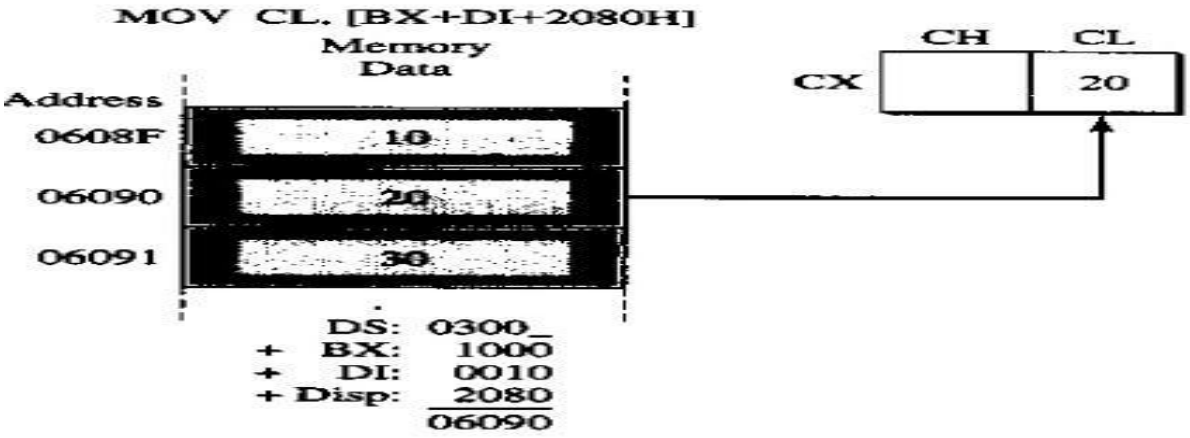


- **Based Indexed Addressing mode:** In this mode, the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.

  Example: MOV AX, [BX] [SI]; **EA=10H * DS + [BX] + [SI]**



- **Relative Based Indexed Addressing mode:** In this mode, 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.
  Example: MOV AX, 50H [BX] [SI]; **EA=10H * DS + 50H + [BX] + [SI]**



- **Control Transfer Instructions:** In control transfer instruction, the control can be transferred to some predefined address or the address somehow specified in the instruction after their execution.

  For the control transfer instructions, the addressing modes depend upon whether the destination location is within the segment or different segments. It also depends upon the method of passing the destination address to the processor. Depending on this control transfer instructions are categorized as follows:
- **Intra segment Direct mode:** In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.
- **Intra segment Indirect mode:** In this mode, the address to which control is to be transferred lies in the same segment in which control transfer instruction lies but it is passed to the instruction indirectly.
- **Inter segment Direct mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies and appears directly in the instruction as an immediate displacement value.
- **Inter segment Indirect mode:** In this mode, the address to which control is to be transferred lies in a different segment in which control transfer instruction lies but it is passed to the instruction indirectly.

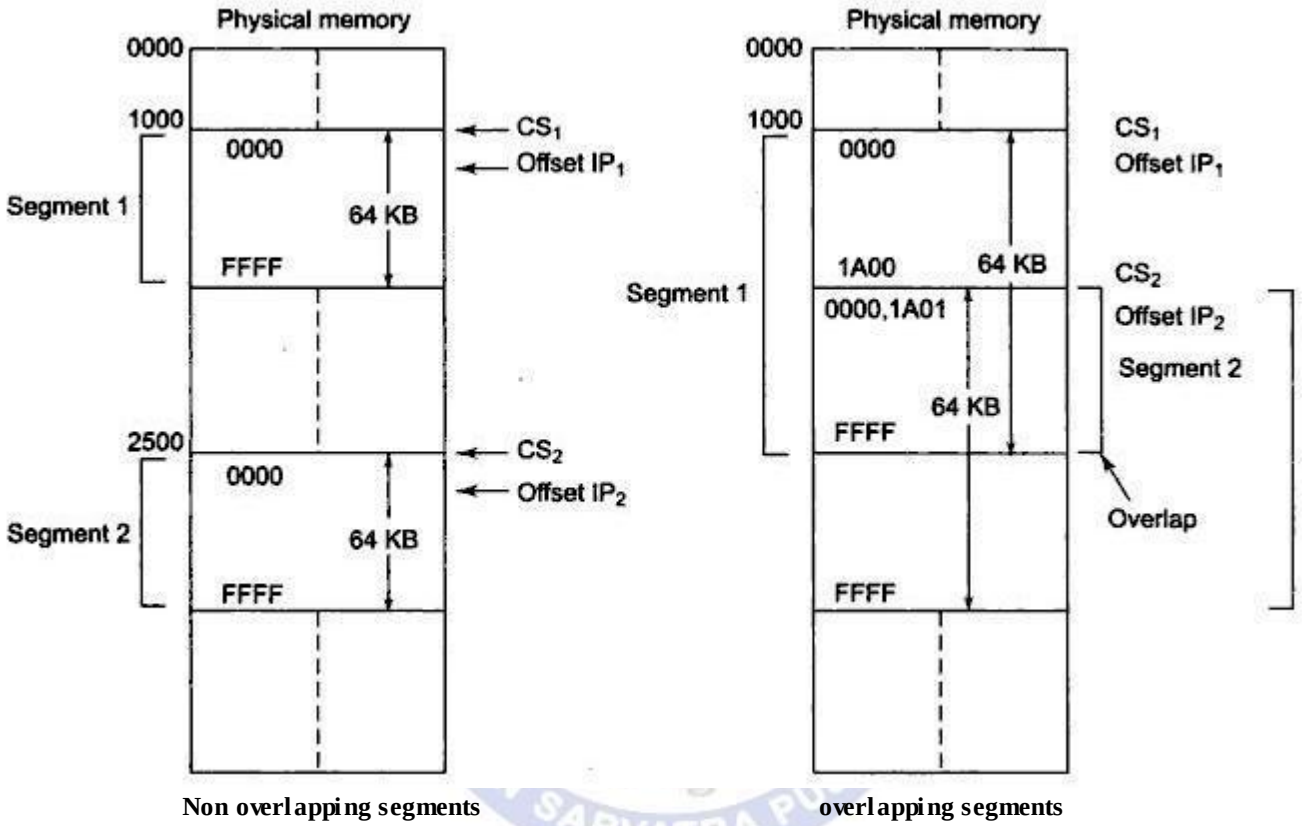
Memory Segmentation for 8086:

8086, via its 20-bit address bus, can address 220 = 1,048,576 or 1 MB of different memory locations. Thus the memory space of 8086 can be thought of as consisting of 1,048,576 bytes or 524,288 words. The memory map of 8086 is shown in Figure where the whole memory space starting from 00000 H to FFFFF H is divided into 16 blocks—each one consisting of 64KB.

1 MB memory of 8086 is partitioned into 16 segments—each segment is of 64 KB length. Out of these 16 segments, only 4 segments can be active at any given instant of time— these are code segment, stack segment, data segment and extra segment. The four memory segments that the CPU works with at any time are called currently active segments. Corresponding to these four segments, the registers used are Code Segment Register (CS), Data Segment Register (DS), Stack Segment Register (SS) and Extra

Segment Register (ES) respectively. Each of these four registers is 16-bits wide and user accessible—i.e., their contents can be changed by software.

The code segment contains the instruction codes of a program, while data, variables and constants are held in data segment. The stack segment is used to store interrupt and subroutine return addresses. The extra segment contains the destination of data for certain string instructions. Thus 64 KB are available for program storage (in CS) as well as for stack (in SS) while128 KB of space can be utilized for data storage (in DS and ES).One restriction on the base address (starting address) of a segment is that it must reside on a 16-byte address memory—examples being 00000 H, 00010 H or 00020 H, etc.



Memory segmentation of 8086

Memory segmentation, as implemented for 8086, gives rise to the following advantages:

- Although the address bus is 20-bits in width, memory segmentation allows one to work with registers having width 16-bits only.
- It allows instruction code, data, stack and portion of program to be more than 64 KB long by using more than one code, data, extra segment and stack segment.
- In a time-shared multitasking environment when the program moves over from one user's program to another, the CPU will simply have to reload the four segment registers with the segment starting addresses assigned to the current user's program.
- User's program (code) and data can be stored separately.
- Because the logical address range is from 0000 H to FFFF H, the same can be loaded at any place in the memory.

Instruction Set of 8086:

There are 117 basic instructions in the instruction set of 8086.The instruction set of 8086 can be divided into the following number of groups, namely:

1. Data copy / Transfer instructions
2. Arithmetic and Logical instructions
3. Branch instructions
4. Loop instructions
5. Machine control instructions
6. Flag Manipulation instructions
7. Shift and Rotate instructions
8. String instructions

Data copy / Transfer instructions: The data movement instructions copy values from one location to another. These instructions include **MOV, XCHG, LDS, LEA, LES, PUSH, PUSHF, PUSHFD, POP, POPF, LAHF, AND SAHF.**

MOV The MOV instruction copies a word or a byte of data from source to a destination. The destination can be a register or a memory location. The source can be a register, or memory location or immediate data. MOV instruction does not affect any flags. The mov instruction takes several different forms:

Mov reg, reg1; mov mem, reg; mov reg, mem; mov mem, immediate data; mov reg, immediate data; mov ax/al, mem; mov mem, ax/al; mov segreg, mem16; mov segreg, reg16; mov mem16, segreg; mov reg16, segreg

The MOV instruction cannot:

1. Set the value of the CS and IP registers.
2. Copy value of one segment register to another segment register (should copy to general register first). MOV CS, DS (Invalid)
3. Copy immediate value to segment register (should copy to general register first). MOV CS, 2000H (Invalid)

Example:

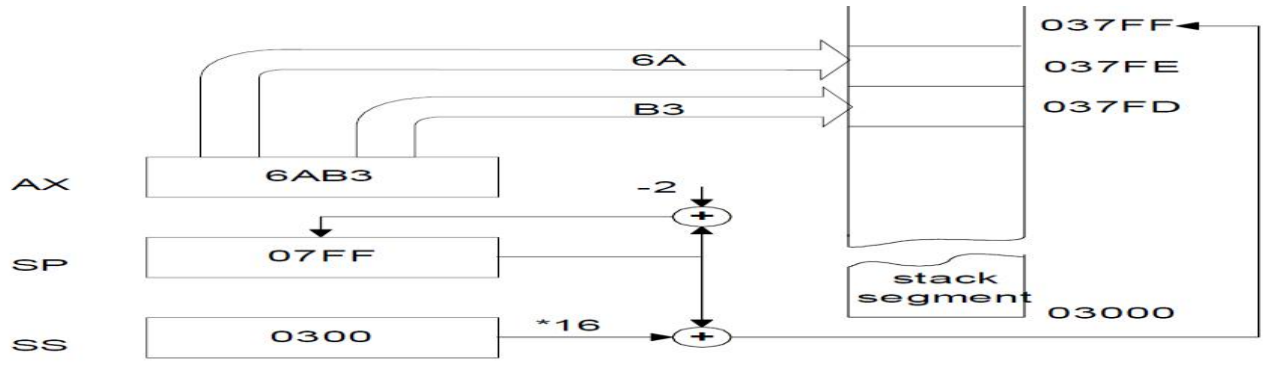
```
ORG 100h
MOV AX, 0B800h;          set AX = B800h
MOV DS, AX;              copy value of AX to DS.
MOV CL, 'A';          CL = 41h (ASCII code).
```

The XCHG Instruction: Exchange This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them, may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Example: MOV AL, 5; AL = 5
MOV BL, 2; BL = 2
XCHG AL, BL; AL = 2, BL = 5

PUSH: Push to stack; this instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and then stores the two byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremented stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

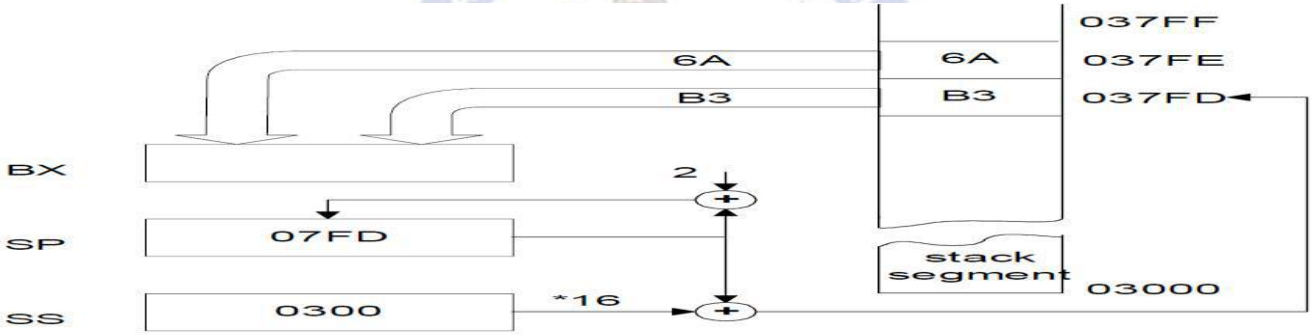
1. PUSH AX
2. PUSH DS
3. PUSH [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.



The effect of PUSH AX instruction

POP: Pop from Stack this instruction when executed loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.

1. POP BX
2. POP DS
3. POP [5000H]



The effect of POP BX instruction

PUSHF: Push Flags to Stack The push flag instruction pushes the flag register on to the stack; first the upper byte and then the lower byte will be pushed on to the stack. The SP is decremented by 2, for each push operation. The general operation of this instruction is similar to the PUSH operation.

POPF: Pop Flags from Stack The pop flags instruction loads the flag register completely (both bytes) from the word contents of the memory location currently addressed by SP and SS. The SP is incremented by 2for each pop operation.

LAHF: Load AH from Lower Byte of Flag This instruction loads the AH register with the lower byte of the flag register. This instruction may be used to observe the status of all the condition code flags (except overflow) at a time.

SAHF: Store AH to Lower Byte of Flag Register This instruction sets or resets the condition code flags (except overflow) in the lower byte of the flag register depending upon the corresponding bit positions in AH. If a bit in AH is 1, the flag corresponding to the bit position is set, else it is reset.

LEA: Load Effective Address The load effective address instruction loads the offset of an operand in the specified register. This instruction is similar to MOV, MOV is faster than LEA.

LEA cx, [bx+si]; CX (BX+SI) mod 64K **If bx=2f00 H; si=10d0H cx = 3fd0H**

The LDS AND LES instructions:

• LDS and LES load a 16-bit register with offset address retrieved from a memory location then load either DS or ES with a segment address retrieved from memory.

This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers.

• LDS and LES instructions obtain a new far address from memory.

– Offset address appears first, followed by the segment address

• This format is used for storing all 32-bit memory addresses.

• A far address can be stored in memory by the assembler.

LDS BX, DWORD PTR[SI]

BL [SI];

BH [SI+1]

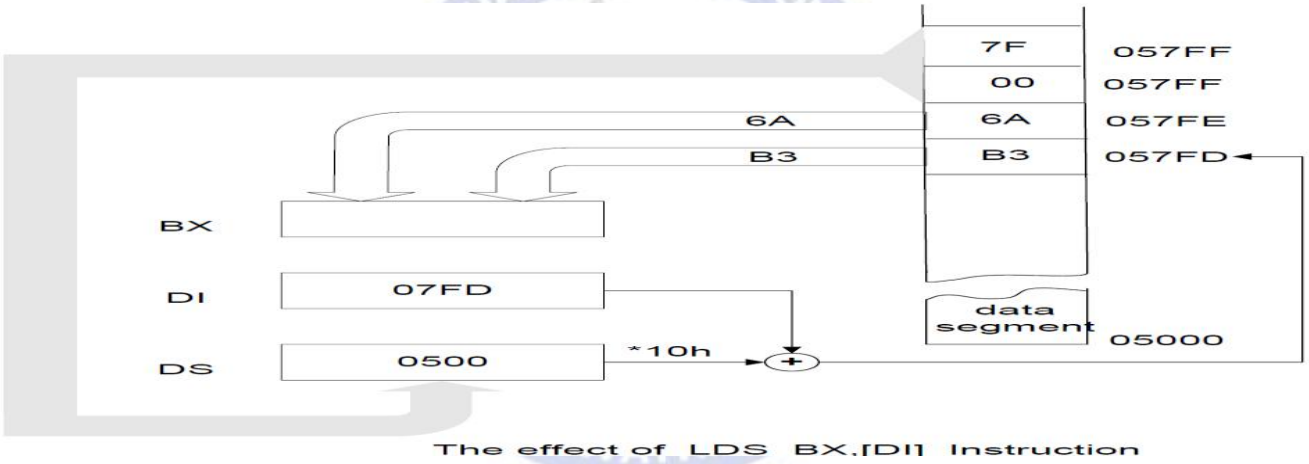
DS [SI+3: SI+2]; in the data segment

LES BX, DWORD PTR[SI]

BL [SI];

BH [SI+1]

ES [SI+3: SI+2]; in the extra segment



The effect of LDS BX,[DI] Instruction

I/O Instructions: The 80x86 supports two I/O instructions: in and out15. They take the forms:

In ax, port

in ax, dx

out port, ax

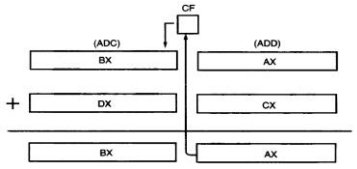
out dx, ax

port is a value between 0 and 255.

The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Arithmetic instructions: These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong to this type of instructions.

The ADD and ADC instructions: The add instruction adds the contents of the source operand to the destination operand. For example, **add ax, bx** adds bx to ax leaving the sum in the ax register. **Add computes dest: = dest + source while adc computes dest: = dest + source + C where C represents the value in the carry flag**. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.



Example:

CF=1

BX=25 AX=98

DX=78 CX=94

BX=9E AX=2C

Both instructions affect the flags identically. They set the flags as follows:

- The overflow flag denotes a signed arithmetic overflow.
- The carry flag denotes an unsigned arithmetic overflow.
- The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).
- The zero flag is set if the result of the addition is zero.
- The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.
- The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there is even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote even parity). If there is an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote odd parity).

The INC instruction: The increment instruction adds one to its operand. Except for carry flag, inc sets the flags the same way as Add ax, 1 same as inc ax. The inc operand may be an eight bit, sixteen bit. The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction.

The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. BCD values are decimal integer coded in binary form with one decimal digit (0...9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero (30 ….39).

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add the mas though they were binary numbers and then execute the daa instruction afterwards to correct the results.

Note: These two instructions assume that the add operands were proper decimal or ASCII values. If you add binary (non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC =1) ) then        add al=08 +06; al=0E > 9
al := al + 6                                     al=0E + 06=04
else
ax := ax + 6
end if
ah := ah + 1                                     ah=00+01=01
AuxC := 1 ;Set auxilliary carry
```

Carry := 1 ; and carry flags.
Else
AuxC := 0 ;Clear auxilliary carry
Carry := 0 ; and carry flags.
endif
al := al and 0Fh

al=04+03=08, now al<9, so only clear ah=0

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers.

The **daa instruction** functions like aaa except it handles packed BCD values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

if ((AL and 0Fh) > 9 or (AuxC = 1)) then
al := al + 6
AuxC: = 1 ; Set Auxilliary carry.
End if
if ((al > 9Fh) or (Carry = 1)) then
al := al + 60h
Carry: = 1; Set carry flag.
End if

al=24+77=9B, as B>9 add 6 to al
al=9B+06=A1, as higher nibble A>9, add 60 to al, al=A1+60=101
Note: if higher or lower nibble of AL <9 then no need to add 6 to AL

EXAMPLE:
Assume AL = 0 0 1 1 0 1 0 1, ASCII 5
BL = 0 0 1 1 1 0 0 1, ASCII 9
ADD AL, BL Result: AL= 0 1 1 0 1 1 1 0 = 6EH, which is incorrect BCD
AAA Now AL = 00000100, unpacked BCD 4.
CF = 1 indicates answer is 14 decimal
NOTE: OR AL with 30H to get 34H, the ASCII code for 4. The AAA instruction works only on the AL register. The AAA instruction updates AF and CF, but OF, PF, SF, and ZF are left undefined.

EXAMPLES:

AL = 0101 1001 = 59 BCD; BL = 0011 0101 = 35 BCD
ADD AL, BL AL = 1000 1110 = 8EH
DAA Add 01 10 because 1110 > 9 AL = 1001 0100 = 94 BCD
AL = 1000 1000 = 88 BCD BL = 0100 1001 = 49 BCD
ADD AL, BL AL = 1101 0001, AF=1
DAA Add 0110 because AF =1, AL = 11101 0111 = D7H
1101 > 9 so add 0110 0000
AL = 0011 0111= 37 BCD, CF =1
The DAA instruction updates AF, CF, PF, and ZF. OF is undefined after a DAA instruction.

The SUBTRACTION instructions: SUB, SBB, DEC, AAS, and DAS

The sub instruction computes the value dest: =dest - src. The sbb instruction computes dest: =dest - src - C.

The sub, sbb, and dec instructions affect the flags as follows:

• They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.
• These instructions set the sign flag if the result is negative.

• These instructions set the overflow flag if signed overflow/under flow occurs.
• They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.
• They set the parity flag according to the number of one bits appearing in the result value.
• The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0...9) per byte. This instruction uses the following algorithm:

if ((al and 0Fh) > 9 or AuxC = 1) then
al := al - 6
ah := ah - 1
AuxC: = 1; Set auxilliary carry
Carry: = 1; and carry flags.
else
AuxC: = 0; Clear Auxilliary carry
Carry: = 0; and carry flags.
End if
al := al and 0Fh

The das instruction handles the same operation for BCD values, it uses the following Algorithm:

if ((al and 0Fh) > 9 or (AuxC = 1)) then
al := al -6
AuxC = 1
End if
if (al > 9Fh or Carry = 1) then
al := al - 60h
Carry: = 1; Set the Carry flag.
End if

EXAMPLE:

ASCII 9-ASCII 5 (9-5)
AL = 00111001 = 39H = ASCII 9
BL = 001 10101 = 35H = ASCII 5
SUB AL, BL Result: AL = 00000100 = BCD 04 and CF = 0
AAS Result: AL = 00000100 = BCD 04 and CF = 0
no borrow required
ASCII 5-ASCII 9 (5-9)
Assume AL = 00110101 = 35H ASCII 5
and BL = 0011 1001 = 39H = ASCII 9
SUB AL, BL Result: AL = 11111100 = - 4 in 2s complement and CF =1
AAS Result: AL = 00000100 = BCD 04 and CF = 1, borrow needed

EXAMPLES:

AL 1000 0110 86 BCD ; BH 0101 0111 57 BCD

SUB AL,BH AL 0010 1111 2FH, CF = 0
DAS Lower nibble of result is 1111, so DAS automatically

Subtracts 0000 0110 to give AL = 00101001 29 BCD
AL 0100 1001 49 BCD BH 0111 0010 72 BCD
SUB AL, BH AL 1101 0111 D7H, CF = 1
DAS Subtracts 0110 0000 (- 60H) because 1101 in upper nibble > 9
AL = 01110111= 77 BCD, CF=1 CF =1 means borrow was needed

The CMP Instruction: The cmp (compare) instruction is identical to the sub instruction with one crucial difference– it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub; the generic form is **cmpdest, src**

Consider the following cmp instruction: cmp ax, bx

This instruction performs the computation ax-bx and sets the flags depending up on the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflows or underflow.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow.This occurs only when ax is less than bx where ax and bx are both unsigned values.

The Multiplication Instructions: MUL, IMUL, and AAM: This instruction multiplies an unsigned byte or word by the contents of AL. The unsigned byte or word may be in any one of the general-purpose registers or memory locations. The most significant word of the result is stored in DX, while the least significant word of the result is stored in AX.

The mul instruction, with an **eight bit operand**, multiplies the al register by the operand and **stores the 16 bit result in ax.** So

mul operand	(Unsigned)	MUL BL	i.e. AL * BL; Al=25 * BL=04; AX=00 (AH) 64 (AL)
imul operand	(Signed)	IMUL BL	i.e. AL * BL; AL=09 * BL=-2; AL * 2's comp(BL) AL=09 * BL (0EH) =7E; 2's comp (7e) =-82

The aam (ASCII Adjust after Multiplication) instruction, adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81,the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al: mul bl; al=9, bl=9 al*bl=9*9=51H; AX=00(AH) 51(AL); AAM ; first hexadecimal value is converted to decimal value i.e. 51 to 81; al=81D; second convert packed BCD to unpacked BCD, divide AL content by 10 i.e. 81/10 then AL=01, AH =08; AX = 0801

EXAMPLE:

AL 00000101 unpacked BCD 5
BH 00001001 unpacked BCD 9
MUL BH AL x BH; result in AX
AX = 00000000 00101101 = 002DH
AAM AX = 00000100 00000101 = 0405H, which is unpacked BCD for 45.
If ASCII codes for the result are desired, use next instruction OR AX, 3030H Put 3 in upper nibble of each byte.
AX = 0011 0100 0011 0101 = 3435H, which is ASCII code for 45

The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16division or a 16/8 division. These instructions take the form:

Div reg For unsigned division

Div mem

Idiv reg For signed division

Idiv mem

The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in .

Note: If an overflow occurs (or you attempt a division by zero) then the80x86 executes an INT 0 (interrupt zero).

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation.It splits apart unpacked binary coded decimal values before an ASCII division operation. The aad instruction is useful forother operations. The algorithm that describes this instruction is

al := ah*10 + al AX=0905H; BL=06; AAD; AX=AH*10+AL=09*10+05=95D;
convert decimal to hexadecimal; 95D=5FH; al=5f;
DIV BL; AL/BL=5F/06; AX=05(AH) 0F (AL)

ah := 0

EXAMPLE:

AX = 0607H unpacked BCD for 67 decimal CH = 09H, now adjust to binary

AAD Result: AX = 0043 = 43H = 67 decimal

DIV CH Divide AX by unpacked BCD in CH

Quotient: AL = 07 unpacked BCD Remainder:

AH = 04 unpacked BCD Flags undefined after DIV

NOTE: If an attempt is made to divide by 0, the 8086 will do a type 0 interrupt.

CBW-Convert Signed Byte to Signed Word: This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be the sign extension of AL. The CBW operation must be done beforea signed byte in AL can be divided by another signed byte with the IDIV instruction. CBW affects no flags.

EXAMPLE:

AX = 00000000 10011011 155 decimal

CBW Convert signed byte in AL to signed word in AX

Result: AX = 11111111 10011011 155 decimal

CWD-Convert Signed Word to Signed Double word: CWD copies the sign bit of a word in AX to all the bits of the DX register. In other words it extends the sign of AX into all of DX. The CWD operation must be done before a signed word in AX can be divided by another signed word with the IDIV instruction. CWD affects no flags.

EXAMPLE:

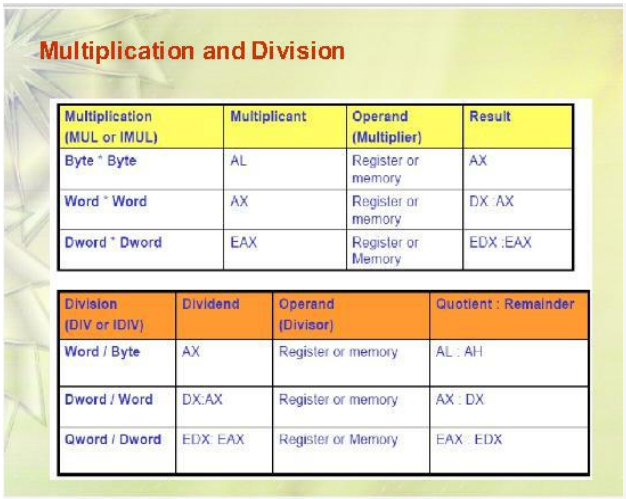
DX = 00000000 00000000

AX = 11110000 11000111 3897 decimal

CWD Convert signed word in AX to signed doubleword in DX:AX

Result DX = 11111111 11111111

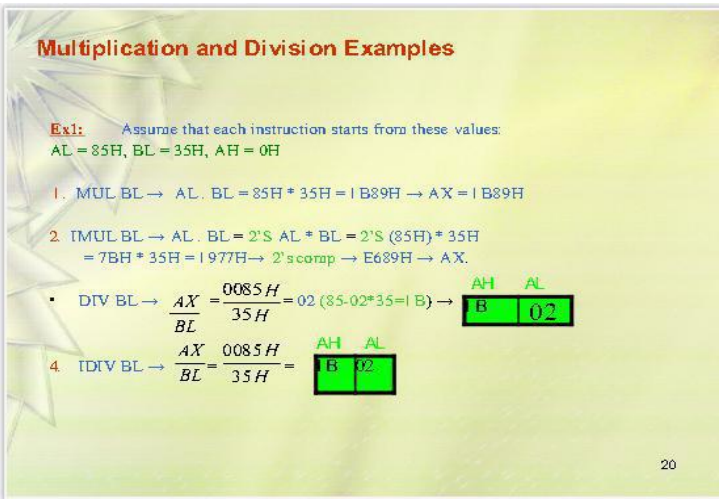
AX = 11110000 11000111 3897 decimal



Multiplication and Division

Multiplication (MUL or IMUL)	Multiplicant	Operand (Multiplier)	Result
Byte * Byte	AL	Register or memory	AX
Word * Word	AX	Register or memory	DX :AX
Dword * Dword	EAX	Register or Memory	EDX :EAX

Division (DIV or IDIV)	Dividend	Operand (Divisor)	Quotient : Remainder
Word / Byte	AX	Register or memory	AL : AH
Dword / Word	DX:AX	Register or memory	AX : DX
Qword / Dword	EDX: EAX	Register or Memory	EAX : EDX



Multiplication and Division Examples

Ex1: Assume that each instruction starts from these values:
AL = 85H, BL = 35H, AH = 0H

1. MUL BL → AL . BL = 85H * 35H = 1B89H → AX = 1B89H

2. IMUL BL → AL . BL = 2'S AL * BL = 2'S (85H) * 35H
= 7BH * 35H = 1977H→ 2'scomp → E689H → AX.

- DIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} = 02$ (85-02*35=1B) → AH = 1B, AL = 02

4. IDIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} =$ AH = 1B, AL = 02

20

Logical, Shift, Rotate and Bit Instructions: The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror,rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar.

The Logical Instructions: AND, OR, XOR, and NOT: The 80x86 logical instructions operate on a bit-by-bit basis. Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

The **AND** instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions. **AND AX, BX**

The **OR** instruction will only set the zero flag if both operands contain zero. **OR AX, BX**

The **XOR** instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form xor reg16, reg16; XOR AX, AX is shorter than the comparable mov reg, 0 instruction.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; Likewise, you can use the or instruction to force certain bits to one in the destination operand;

The Shift Instructions: SHL/SAL, SHR, SAR: The 80x86 supports three different shift instructions (shl and sal are the same instruction): shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right). The general format for a shift instruction is

Shl dest, count sal dest, count shr dest, count sar dest, count

SHL/SAL: These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag.

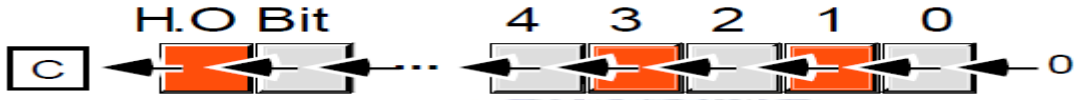
The shl/sal instruction sets the condition code bits as follows:

• If the shift count is zero, the shl instruction doesn't affect any flags.
• The carry flag contains the last bit shifted out of the H.O. bit of the operand.
• The overflow flag will contain one if the two H.O. bits were different prior to a single bit shift. The overflow flag is undefined if the shift count is not one.
• The zero flag will be one if the shift produces a zero result.
• The sign flag will contain the H.O. bit of the result.
• The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
• The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

```
shl ah, 4 ;
or al, ah ; Merge in H.O. four bits.
```

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction).



SHL OPERATION

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

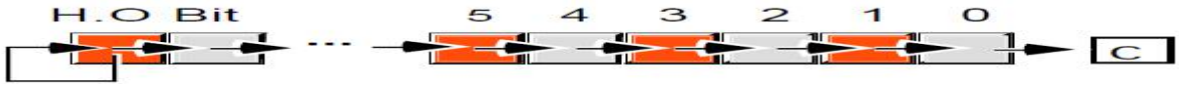
```
shl ah, 4 ;Move L.O. bits to H.O. position.
and al, 0Fh ;Clear H.O. four bits.
or al, ah ;Merge the bits.
```

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the **shift left instruction for multiplication by powers of two**:

```
shl ax, 1 ;Equivalent to AX*2
shl ax, 2 ;Equivalent to AX*4
shl ax, 3 ;Equivalent to AX*8
```

SAR: Thesar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

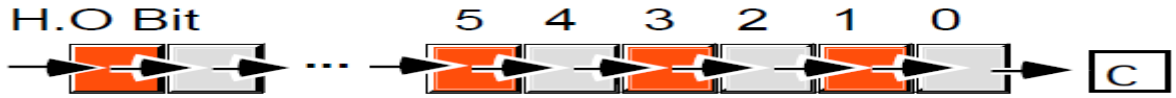


SAR OPERATION

```
sar ax, 1 ;Signed division by 2
sar ax, 2 ;Signed division by 4
sar ax, 3 ;Signed division by 8
sar ax, 4 ;Signed division by 16
sar ax, 5 ;Signed division by 32
sar ax, 6 ;Signed division by 64
sar ax, 7 ;Signed division by 128
sar ax, 8 ;Signed division by 256
```

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity.

SHR: The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit



SHR OPERATION

The shift right instruction is especially useful for unpacking data. shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

shr ax, 1 ;Equivalent to AX/2
shr ax, 2 ;Equivalent to AX/4
shr ax, 3 ;Equivalent to AX/8
shr ax, 4 ;Equivalent to AX/16

The Rotate Instructions: RCL, RCR, ROL, and ROR

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the operand. They include rcl (rotate through carry left), rcr(rotate through carry right), rol(rotate left), And ror (rotate right). These instructions all take the forms:

```
rcl dest, count      rol dest, count      rcr dest, count      ror dest, count
```

RCL: The rcl (rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right. The rcl instruction sets the flag bits as follows:

• The carry flag contains the last bit shifted out of the H.O. bit of the operand.
• If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
• The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

RCL OPERATION

RCR: The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit. This instruction sets the flags in a manner analogous to rcl:

• The carry flag contains the last bit shifted out of the L.O. bit of the operand.
• The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

RCR OPERATION

ROL: The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit, rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag. The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction.

Like shl, the rol instruction is often useful for packing and unpacking data.

ROL OPERATION

ROR: The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit.

ROR OPERATION

String Instructions: A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes or words on 8086.**All members of the 80x 86 families support five different string instructions: MOVS, CMPS, SCAS, LODS, AND STOS.**

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, we need a source address, a destination address, and a count (the number of string elements to move). The operands for the string instructions include:

- **the SI (source index) register,**
- **the DI (destination index) register,**
- **the CX (count) register,**
- **the AX register, and**
- **the direction flag in the FLAGS register.**

The REP/REPE/REPZ and REPNZ/REPNE Prefixes: The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

Field:

Label repeat mnemonic operand; comment

For MOVS:

Rep movs {operands}

For CMPS:

Repe cmps {operands} repz cmps {operands} repne cmps {operands} repnz cmps {operands}

For SCAS:

Repe scas {operands} repz scas {operands} repnescas {operands} repnzscas {operands}

For STOS:

Rep stos {operands}

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times. Without the repeat prefix, the instruction operates only on a single byte,word, or double word.

If the direction flag is clear, the CPU increments si and di after operating upon each string element. If the direction flag is set, then the 80x86 decrements si and di after processing each string

element. The direction flag may be set or cleared using the cld (clear direction flag) and std (setdirection flag) instructions.

The MOVS Instruction: The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es :di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds: si to es: di and decrements the cx register. This process repeats until cx becomes zero. The syntax is:

{REP} MOVSB **{REP} MOVSW**

The CMPS Instruction: The cmps instruction compares two strings. The CPU compares the string referenced by es: di to the string pointed at by ds: si. CX contains the length of the two strings (when using the rep prefix). The syntax is: **{REPE} CMPSB** **{REPE} CMPSW**

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match or length of the string cx=0. The **repe** prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero.

The SCAS Instruction: The scas instruction, by itself, compares the value in the accumulator (al or ax) against the value pointed at by es:di and then increments (or decrements) di by one or two. The CPU sets the flags according to the result of the comparison. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator. The scas instruction takes the following forms: **{REPNE} SCASB** **{REPNE} SCASW**

The STOS Instruction: The stos instruction stores the value in the accumulator at the location specified by es: di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Its primary use is to initialize arrays and strings to a constant value. **{REP} STOSB**
{REP} STOSW

The LODS Instruction: The lods instruction copies the byte or word pointed at by ds:si into the al or ax register, after which it increments or decrements the si register by one or two.**{REP} LODSB**
{REP} LODSW

Flag Manipulation and Processor Control Instructions: These instructions control the functioning of the available hardware inside the processor chip. These are categorized into two types; (a) flag manipulation instructions and (b) machine control instructions.

The flag manipulation instructions directly modify some of the flags of 8086. The machine control instructions control the bus usage and execution. The flag manipulation instructions and their functions are as follows:

CLC - Clear carry flag **CMC - Complement carry flag** **STC - Set carry flag**
CLD - Clear direction flag **STD - Set direction flag** **CLI - Clear interrupt flag**
STI - Set interrupt flag

These instructions modify the carry (CF), direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be modified using the flag manipulation instructions, further control the processor operation; like interrupt responses and auto increment or auto decrement modes.

The machine control instructions supported by 8086 and 8088 are listed as follows along with their functions. These machine control instructions do not require any operand.

WAIT - Wait for Test input pin to go low **HLT - Halt the processor** **NOP - No operation** **ESC - Escape to external device like NDP (numeric co-processor)** **LOCK - Bus lock instruction prefix.**

After executing the **HLT instruction**, the processor enters the halt state. The two ways to pull it out of the halt state are to reset the processor or to interrupt it.

When **NOP instruction** is executed, the processor does not perform any operation till 4 clock cycles, except incrementing the IP byone. It then continues with further execution after 4 clock cycles.

ESC instruction when executed, frees the bus for an external master like a coprocessor or peripheral devices.

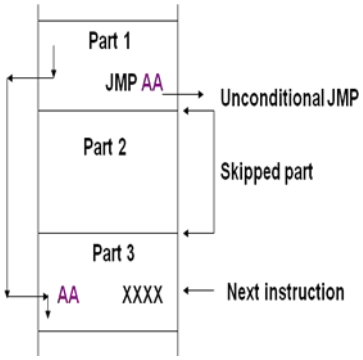
The **LOCK prefix** may appear with another instruction. When it is executed, the bus access is not allowed for another master till the lock prefixed instruction is executed completely. This instruction is used in case of programming for multiprocessor systems.

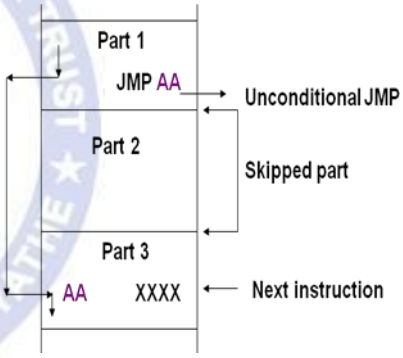
The **WAIT instruction** when executed holds the operation of processor with the current status till the logic level on the TEST pin goes low. The processor goes on inserting WAIT states in the instruction cycle, till the TEST pin goes low. Once the TEST pin goes low, it continues further execution.

Program Flow Control Instructions: The control transfer instructions are used to transfer the control from one memory location to another memory location. In 8086 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions.

Unconditional Jumps: The jmp (jump) instruction unconditionally transfers control to another point in the program. Intra segment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

JMP Address





Unconditional jump **Conditional jump**

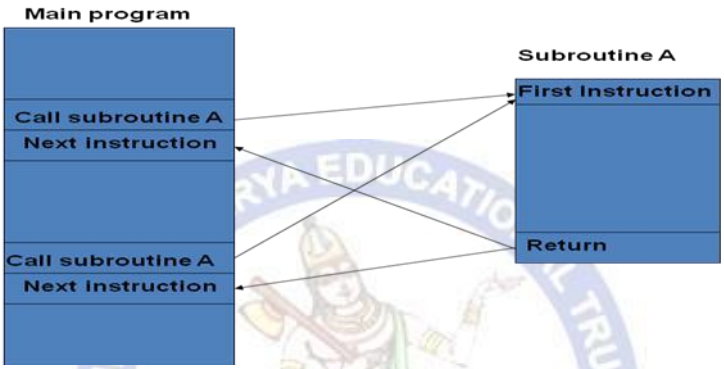
Conditional Jump: The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if….then statement. The conditional jumps test one or more bits in the status register to see if they match some particular pattern. If the pattern matches, control transfers to the target location. If the condition fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow and zero flags.

Definition	Description	Condition[1]
Jump Based on Unsigned Data		
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JA / JNBE	Jump above or jump not below/ equal	C=0 & Z=0
JAE / JNB	Jump above/ equal or jump not below	C=0
JB / JNAE	Jump below or jump not above/ equal	C=1
JBE / JNA	Jump below/ equal or jump not above	C=1 or Z=1
Jump Based on Signed Data		
JE / JZ	Jump equal or jump zero	Z=1
JNE / JNZ	Jump not equal or jump not zero	Z=0
JG / JNLE	Jump greater or jump not less/ equal	N=0 & Z=0
JGE / JNL	Jump greater/ equal or jump not less	N=0
JL / JNGE	Jump less or jump not greater/ equal	N=1
JLE / JNG	Jump less/ equal or jump not greater	N=1 or Z=1
Arithmetic Jump		
JS	Jump sign set	N=1
JNS	Jump no sign set	N=0
JC	Jump carry set	C=1
JNC	Jump no carry set	C=0
JO	Jump overflow set	O=1
JNO	Jump not overflow set	O=0
JP / JPE	Jump parity even	P=1
JNP / JPO	Jump parity odd	P=0

Loop Instruction:

- These instructions are used to repeat a set of instructions several times.
- Format: LOOP Short-Label
- Operation: (CX) ← (CX)-1
- Jump is initialized to location defined by short label if CX≠0. Otherwise, execute next sequential instruction.
- Instruction LOOP works with respect to contents of CX. CX must be preloaded with a count that represents the number of times the loop is to be repeat.
- Whenever the loop is executed, contents at CX are first decremented then checked to determine if they are equal to zero.
- If CX=0, loop is complete and the instruction following loop is executed.
- If CX ≠ 0, content return to the instruction at the label specified in the loop instruction.
- **LOOP AGAIN is <u>almost same</u> as: DEC CX, JNZ AGAIN**

SUBROUTINE & SUBROUTINE HANDILING INSTRUCTIONS: CALL, RET



- A subroutine is a special segment of program that can be called for execution from any point in a program.
- An assembly language subroutine is also referred to as a "procedure".
- Whenever we need the subroutine, a single instruction is inserted in to the main body of the program to call subroutine.
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
- To branch a subroutine the value in the IP or CS and IP must be modified.
- After execution, we want to return the control to the instruction that immediately follows the one called the subroutine i.e., the original value of IP or CS and IP must be preserved.
- Execution of the instruction causes the contents of IP to be saved on the stack. (this time (SP) ← (SP) -2)
- A new 16-bit (near-proc, mem16, reg16 i.e., Intra Segment) value which is specified by the instructions operand is loaded into IP.
- Examples: CALL 1234H
  CALL BX
  CALL [BX]

Return Instruction: RET instruction removes an address from the stack so the program returns to the instruction following the CALL

- Every subroutine must end by executing an instruction that returns control to the main program. This is the return (RET) instruction.

- By execution the value of IP or IP and CS that were saved in the stack to be returned back to their corresponding registers. (this time (SP) ← (SP)+2)

MACROS: The macro directive allows the programmer to write a named block of source statements, then use that name in the source file to represent the group of statements. During the assembly phase, the assembler automatically replaces each occurrence of the macro name with the statements in the macro definition.

Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if used repeatably. Procedures or subroutines take up less space, but the increased overhead of saving and restoring addresses and parameters can make them slower. In summary, the advantages and disadvantages of macros are,

Advantages

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages

In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Modular Programming: Instead of writing a large program in a single unit, it is better to write small programs—which are parts of the large program. Such small programs are called program modules or simply modules. Each such module can be separately written, tested and debugged. Once the debugging of the small programs is over, they can be linked together. Such methodology of developing a large program by linking the modules is called modular programming.

Assembler Directives:

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ, assume and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, to generate address.

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and byte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

ASSUME: The ASSUME directive tell the assembler the name of the logical segment it should use for a specified segment. Ex: ASSUME CS: Code, DS: Data, SS: Stack; or ASSUME CS: Code

Data Directives: The directives DB, DW, DD, DR and DT are used to (a) define different types of variables or (b) to set aside one or more storage locations in memory-depending on the data type:

DB — Define Byte DW — Define Word DD — Define Double word

DQ — Define Quad word DT — Define Ten Bytes

The **DB directive** is used to declare a byte-type variable or to set aside one or more storage locations of type byte in memory (Define Byte)

Example: Temp DB 42H; Temp is a variable allotted 1byte of memory location assigned with data 42H

The **DW directive** is used to declare a variable of type word or to reserve memory locations which can be accessed as type double word (Define word)

Example: N2 DW 427AH; N2 variable is initialized with value 427AH when it is loaded into memory to run.

The **DD directive** is used to declare a variable of type double word or to reserve memory locations which can be accessed as type double word (Define double word)

Example: Big DD 2456756CH; Big variable is initialized with 4 bytes

The **DQ directive** is used to tell the assembler to declare a variable 4 words in length or to reverse 4 words of storage in memory (Define Quad word)

Example: Big DQ 2456756C88464567H; Big variable is initialized with 4 words (8 bytes)

The **DT directive** is used to tell the assembler to declare a variable 10 bytes in length or to reverse 10bytes of storage in memory (Define Ten bytes)

Example: Packed BCD DT 1122334455667788900H; 10 byte data is initialized to variable packed BCD

DUP: This directive operator is used to initialize several locations and to assign values to these locations. Its format is: Name Data-Type Num DUP (value)

Example: TABLE DB 20 DUP (0); Reserve an array of 20 bytes of memory and initialize all 20 bytes with 0. Array is named TABLE

END: The **END** directive is placed after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statement after an end directive.

The **ENDP** directive is used with the name of the procedure to indicate the end of a procedure to the assembler.

SQUARE NUM PROC

….

….

SQUARE NUM ENDP

The **ENDS** directive is used with the name of the segment to indicate the end of a segment to the assembler.

CODE SEGMENT

…

…

CODE ENDS

EQU: The **EQU** directive is used to give a name to some value or to a symbol. Each time assembler finds the name in the program it will replace the name with the value.

FACTOR EQU 03H; This statement should be written at the start

ADD AL, FACTOR; The assembler converts this instruction as ADD AL, 03H

EVEN: The **EVEN** directive instructs the assembler to increment the location of the counter to the next even address if it is not already in the even address. If the word starts at an odd address, 8086 will take 2 bus cycles to get the 2 byte of the word. *"A series of words can read much more quickly if they are at even address".*

```
DATA HERE SEGMENT     ; Location counter will point to 0009H after assembler reads next
statement
SALES DB 9 DUP (?)    ; Declare an array of 9 bytes
EVEN                  ; Increment location counter to 000AH
RECORD DW 100 DUP (?) ; Array of 100 words starting on even address for quicker read
DATA HERE ENDS        ;
```

GLOBAL: This **GLOBAL** directive can be used in place of PUBLIC directive or in place of an EXTRN directive. The GOLBAL directive is used to make the symbol available to other modules.

PUBLIC: The **PUBLIC** directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive.

EXTRN: This **EXTRN** directive is used to tell the assembler that the names or labels following the directive are in some other assembly module.

GROUP: This **GROUP** directive is used to tell the assembler to group the logical segments named after the directive into one logical group segment.

Example: SMALL SYSTEM GROUP CODE, DATA, STACK

ASSUME CS: SMALL SYSTEM, DS: SMALL SYSTEM, SS: SMALL SYSTEM

OFFSET—Is an operator which tells the assembler to determine the offset or the displacement of a named data item (variable) or procedure from start of the segment which contains it. This operator is used to load the offset of a variable into a register so that the variable can be accessed with one of the indexed addressing modes. MOV AL, OFFSET N1

ORG – This **ORG** directive allows to set the location counter to a desired value at any point in the program. The statement ORG 100H tells the assembler to set the location counter to 0100H.

PROCEDURE: A PROC directive is used to define a label and to delineate a sequence of instructions that are usually interpreted to be a subroutine, that is, CALLed either from within the same physical segment (near) or from another physical segment (far).

Syntax:

```
name PROC [type]          P1 PROC NEAR
                          MOV AX, 1 5
                          ADD OX, AX
…..                       ENDP
name ENDP
```

Labels: A label, a symbolic name for a particular location in an instruction sequence, maybe defined in one of three ways. The first way is the most common. The format is shown below: **label: [instruction]**

where "label" is a unique ASM86 identifier and "instruction" is an*8086/8087/8088* instruction. This label will have the following attributes:

1. Segment-the current segment being assembled.
2. Offset-the current value of the location counter.
3. Type-will be NEAR.

An example of this form of label definition is: ALAB: MOV AX, COUNT

PROGRAM: 8 – BIT ADDITION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DB 00H N2 DB 00H	

	RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, N1	Copy the content of data from N1 memory location
	MOV BL, N2	Copy the content of data from N2 memory location
	ADD AL, BL	Perform addition on AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**25** **34**	**3002**	**59**

PROGRAM: 8 – BIT SUBTRACTION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DB 00H N2 DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, N1	Copy the content of data from N1 memory location
	MOV BL, N2	Copy the content of

		data from N2 memory location
	SUB AL, BL	Perform subtraction on AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT		**OUTPUT**	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**35** **14**	**3002**	**21**

PROGRAM: 8 – BIT MULTIPLICATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DB 00H N2 DB 00H RES1 DB 00H RES2 DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, N1	Copy the content of data from N1 memory location
	MOV BL, N2	Copy the content of data from N2 memory location
	MUL BL	Perform multiplication on AL and BL registers and store result in AL and AH
	MOV RES1, AL	Copy the content of AL to RES1 memory location
	MOV RES2, AH	Copy the content of AH to RES2 memory

		location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**98** **C5**	**3002** **3003**	**F8** **74**

PROGRAM: 8 – BIT DIVISION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DB 00H N2 DB 00H RES1 DB 00H RES2 DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, N1	Copy the content of data from N1 memory location
	MOV BL, N2	Copy the content of data from N2 memory location
	DIV BL	Perform division on AL and BL registers and store quotient in AL and remainder in AH
	MOV RES1, AL	Copy the content of AL to RES1 memory location
	MOV RES2, AH	Copy the content of AH to RES2 memory

		location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**98** **15**	**3002** **3003**	**07** **05**

PROGRAM: 16 – BIT ADDITION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DW 00H N2 DW 00H RES1 DW 00H RES2 DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV CX, 0000	Clear CX register
	MOV AX, N1	Copy the content of data from N1 memory location
	MOV BX, N2	Copy the content of data from N2 memory location
	ADD AX, BX	Perform addition on AX and BX registers and store result in AX
	JNC **L1**	Jump if CF is not zero to L1

	INC CX	Increment count
L1:	MOV RES1, AX	Copy the content of AX to RES1 memory location
	MOV RES2, CX	Copy the content of AX to RES2 memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

WITH CARRY

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3004**	**10**
3001	**C5**	**3005**	**23**
3002	**78**	**3006**	**00**
3003	**5D**	**3007**	**01**

WITH OUT CARRY

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3004**	**3E**
3001	**C5**	**3005**	**D8**
3002	**A6**	**3006**	**00**
3003	**12**	**3007**	**00**

PROGRAM: 16 – BIT SUBTRACTION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DW 00H N2 DW 00H RES1 DW 00H RES2 DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV CX, 0000	Clear CX register
	MOV AX, N1	Copy the content of data from N1 memory location

	MOV BX, N2	Copy the content of data from N2 memory location
	SUB AX, BX	Perform subtraction on AX and BX registers and store result in AX
	JNC **L1**	Jump if CF is not zero to L1
	INC CX	Increment count
L1:	MOV RES1, AX	Copy the content of AX to RES1 memory location
	MOV RES2, CX	Copy the content of AX to RES2 memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

WITH BORROW

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3004**	**20**
3001	**C5**	**3005**	**E8**
3002	**78**	**3006**	**00**
3003	**DD**	**3007**	**01**

WITH OUT BORROW

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3004**	**20**
3001	**C5**	**3005**	**68**
3002	**78**	**3006**	**00**
3003	**5D**	**3007**	**00**

PROGRAM: 16 – BIT MULTIPLICATION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DW 00H N2 DW 00H RES1 DW 00H RES2 DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data

		segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AX, N1	Copy the content of data from N1 memory location
	MOV BX, N2	Copy the content of data from N2 memory location
	MUL BX	Perform multiplication on AX and BX registers and store lower word in AX and higher word in DX
	MOV RES1, AX	Copy the content of AX to RES1 memory location
	MOV RES2, DX	Copy the content of DX to RES2 memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATION

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3004**	**40**
3001	**C5**	**3005**	**D7**
3002	**78**	**3006**	**24**
3003	**5D**	**3007**	**48**

PROGRAM: 2H 16 – BIT DIVISION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H N1 DW 00H N2 DW 00H N3 DW 00H RES1 DW 00H RES2 DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator

	MOV AX, N1	Copy the content of data from N1 memory location to AX
	MOV DX, N2	Copy the content of data from N2 memory location to DX
	MOV BX, N3	Copy the content of data from N3 memory location to BX
	DIV BX	Perform division on AX DX by BX registers and store quotient in AX and remainder in DX
	MOV RES1, AX	Copy the content of AX to RES1 memory location
	MOV RES2, DX	Copy the content of DX to RES2 memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATION

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**98**	**3006**	**F1**
3001	**C5**	**3007**	**9C**
3002	**78**	**3008**	**A0**
3003	**5D**	**3009**	**1C**
3004	**78**		
3005	**98**		

PROGRAM: MULTI BYTE ADDITION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H IP1 DD 1223445566H IP2 DD 7788557733H RES DD 0000000000H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	SUB AX, AX	Clear garbage value
	MOV SI, OFFSET IP1	Copy address of IP1 in to SI
	MOV DI, OFFSET IP2	Copy address of IP2 in to DI

	MOV BX, OFFSET RES	Copy address of RES in to BX
	MOV CX, 03H	Copy data 03H TO CX register
	MOV AL, [SI]	Copy the content of memory location of SI to AL register
	MOV DL, [DI]	Copy the content of memory location of DI to DL register
	ADD AL, DL	Perform addition on AL and DL register
	MOV [BX], AL	Copy AL register content to memory location of BX register
BACK:	INC SI	Increment SI register
	INC DI	Increment DI register
	INC BX	Increment BX register
	MOV AL, [SI]	Copy the content of memory location of SI to AL register
	MOV DL, [DI]	Copy the content of memory location of DI to DL register
	ADC AL, DL	Perform addition with carry on AL and DL register
	MOV [BX], AL	Copy AL register content to memory location of BX register
	LOOP **BACK**	Decrement CX register, jump if CL is not zero to BACK
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATION

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**66**	**300A**	**99**
3001	**55**	**300B**	**CC**
3002	**44**	**300C**	**99**
3003	**23**	**300D**	**AB**
3004	**12**	**300E**	**89**
3005	**33**		
3006	**77**		
3007	**55**		
3008	**88**		
3009	**77**		

PROGRAM: MULTI BYTE SUBTRACTION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H IP1 DD 7788557733H IP2 DD 1223445566H RES DD 0000000000H DATA ENDS	

	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	SUB AX, AX	Clear garbage value
	MOV SI, OFFSET IP1	Copy address of IP1 in to SI
	MOV DI, OFFSET IP2	Copy address of IP2 in to DI
	MOV BX, OFFSET RES	Copy address of RES in to BX
	MOV CX, 03H	Copy data 03H TO CX register
	MOV AL, [SI]	Copy the content of memory location of SI to AL register
	MOV DL, [DI]	Copy the content of memory location of DI to DL register
	SUB AL, DL	Perform subtraction on AL and DL register
	MOV [BX], AL	Copy AL register content to memory location of BX register
BACK:	INC SI	Increment SI register
	INC DI	Increment DI register
	INC BX	Increment BX register
	MOV AL, [SI]	Copy the content of memory location of SI to AL register
	MOV DL, [DI]	Copy the content of memory location of DI to DL register
	SUBB AL, DL	Perform subtract with borrow on AL and DL register
	MOV [BX], AL	Copy AL register content to memory location of BX register
	LOOP **BACK**	Decrement CX register, jump if CL is not zero to BACK
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATION

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**33**	**300A**	**CD**
3001	**77**	**300B**	**21**
3002	**55**	**300C**	**11**
3003	**88**	**300D**	**65**
3004	**77**	**300E**	**65**
3005	**66**		
3006	**55**		
3007	**44**		
3008	**23**		
3009	**12**		

ASCII ARITHMETIC OPERATIONS

PROGRAM: ASCII ADDITION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H ASC1 DB 00H ASC2 DB 00H RES DW 0000H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	XOR AX, AX	Clear the garbage
	MOV AL, ASC1	Copy the content of data from N1 memory location
	MOV BL, ASC2	Copy the content of data from N2 memory location
	ADD AL, BL	Perform addition on AL and BL registers and store result in AL
	AAA	Perform ASCII adjustment after addition
	OR AX, 3030H	
	MOV RES, AX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**34** **38**	**3002** **3003**	**31H** **32H**

PROGRAM: ASCII SUBTRACTION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H	

	ASC1 DB 00H ASC2 DB 00H RES DW 0000H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	XOR AX, AX	Clear the garbage
	MOV AL, ASC1	Copy the content of data from N1 memory location
	MOV BL, ASC2	Copy the content of data from N2 memory location
	SUB AL, BL	Perform subtraction on AL and BL registers and store result in AL
	AAS	Perform ASCII adjustment after subtraction
	OR AX, 3030H	
	MOV RES, AX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**39** **34**	**3002** **3003**	**30H** **35H**

PROGRAM: ASCII MULTIPLICATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H ASC1 DB 00H ASC2 DB 00H RES DW 0000H DATA ENDS	
	CODE SEGMENT ORG 4000H	

START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	XOR AX, AX	Clear the garbage
	MOV AL, ASC1	Copy the content of data from N1 memory location
	MOV BL, ASC2	Copy the content of data from N2 memory location
	MUL BL	Perform multiplication on AL and BL registers and store result in AL
	AAM	Perform ASCII adjustment after addition
	OR AX, 3030H	
	MOV RES, AX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**06** **02**	**3002** **3003**	**31H** **32H**

PROGRAM: ASCII DIVISION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H ASC1 DW 00H ASC2 DB 00H RES DW 0000H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data

		segment
	MOV DS, AX	
	XOR AX, AX	Clear the garbage
	MOV AX, ASC1	Copy the content of data from N1 memory location
	MOV BL, ASC2	Copy the content of data from N2 memory location
	AAD	Perform ASCII adjustment before division
	DIV BL	Perform division on AX and BL registers and store result in AX
	OR AX, 3030H	
	MOV RES, AX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT		**OUTPUT**	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001** **3002**	**06** **03** **05**	**3002** **3003**	**37H** **31H**

LOGICAL OPERATIONS

PROGRAM: LOGICAL AND OPERATION

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT	

		ORG 3000H OP1 DB 00H OP2 DB 00H RES DB 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 0000	Clear the accumulator
		MOV AL, OP1	Copy the content of data from OP1 memory location
		MOV BL, OP2	Copy the content of data from OP2 memory location
		AND AL, BL	Perform AND on AL and BL registers and store result in AL
		MOV RES, AL	Copy the content of accumulator to RES memory location
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**35** **0F**	**3002**	**05**

PROGRAM: LOGICAL OR OPERATION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H OP2 DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location

	MOV BL, OP2	Copy the content of data from OP2 memory location
	OR AL, BL	Perform OR on AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**25** **46**	**3002**	**67**

PROGRAM: LOGICAL XOR OPERATION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H OP2 DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV BL, OP2	Copy the content of data from OP2 memory location
	XOR AL, BL	Perform XOR on AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**25** **25**	**3002**	**00**

PROGRAM: SHIFT ARITHMETIC LEFT OPERATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location
	SAL AL, CL	Perform shift arithmetic left AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**34** **04**	**3002**	**40**

PROGRAM: SHIFT ARITHMETIC RIGHT OPERATION

OFFSET ADDRESS	LABEL	MNEMONICS	COMMENTS
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT	

		ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 0000	Clear the accumulator
		MOV AL, OP1	Copy the content of data from OP1 memory location
		MOV CL, COUNT	Copy the content of data from count memory location
		SAR AL, CL	Perform shift arithmetic right AL and BL registers and store result in AL
		MOV RES, AL	Copy the content of accumulator to RES memory location
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**34** **04**	**3002**	**03**

PROGRAM: SHIFT LOGICAL LEFT OPERATION

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment

	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	SHL AL, CL	Perform shift arithmetic left AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**34** **04**	**3002**	**40**

PROGRAM: SHIFT LOGICAL RIGHT OPERATIONS

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	SHR AL, CL	Perform shift logical right AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES

		memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT		OUTPUT	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**35** **04**	**3002**	**03**

PROGRAM: ROTATE LEFT WITHOUT CARRY OPERATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	ROL AL, CL	Perform rotate left without carry AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT		OUTPUT	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**23** **04**	**3002**	**32**

PROGRAM: ROTATE RIGHT WITHOUT CARRY OPERATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	ROR AL, CL	Perform rotate right without carry AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**23** **04**	**3002**	**32**

PROGRAM: ROTATE LEFT WITH CARRY OPERATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT	

	ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	RCL AL, CL	Perform rotate left with carry AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**23** **04**	**3002**	**32**

PROGRAM: ROTATE RIGHT WITH CARRY OPERATION

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H OP1 DB 00H COUNT DB 00H RES DB 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, OP1	Copy the content of data from OP1 memory location
	MOV CL, COUNT	Copy the content of data from count

		memory location to CL
	RCR AL, CL	Perform rotate right with carry AL and BL registers and store result in AL
	MOV RES, AL	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**35** **04**	**3002**	**46**

PACKED AND UNPACKED BCD NUMBERS

PROGRAM: PACKED TO UNPACKED BCD NUMBERS

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H IP1 DB 00H COUNT DB 00H RES DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, IP1	Copy the content of data from IP1 memory location
	MOV DL, AL	Move data from AL to DL register
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	AND AL, 0F0H	Perform AND operation to hide the data of higher nibble
	ROR AL, CL	Perform rotate right without carry AL by CL registers and store

		result in AL
	MOV BH, AL	Copy AL register content to BH register
	AND DL, 0F	Mask the lower nibble of DL register using AND
	MOV BL, DL	Copy DL register to BL register
	MOV RES, DX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT		**OUTPUT**	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**56** **04**	**3002** **3003**	**06** **05**

PROGRAM: UNPACKED TO PACKED BCD NUMBERS

LABEL	**MNEMONICS**	**COMMENTS**
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H IP1 DB 00H IP2 DB 00H COUNT DB 00H RES DW 00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV AX, 0000	Clear the accumulator
	MOV AL, IP1	Copy the content of data from IP1 memory location to AL
	MOV BL, IP2	Copy the content of data from IP1 memory location to BL
	MOV CL, COUNT	Copy the content of data from count memory location to CL
	AND AL, 0F0H	Perform AND operation to hide the data of higher

		nibble
	ROR AL, CL	Perform rotate right without carry AL by CL registers and store result in AL
	AND BL, 0F	Mask the lower nibble of DL register using AND
	OR AL,BL	Perform OR operation on AL and BL registers
	MOV RES, AX	Copy the content of accumulator to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**06** **04**	**3002** **3003**	**64**

SORTING THE GIVEN NUMBERS

PROGRAM: ASCENDING ORDER

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H COUNT EQU 04H LIST DB 00H,00H,00H DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	XOR AX, AX	Clear the garbage data
	MOV CX, COUNT-1	Decremented count is loaded to CX register
	MOV SI, OFFSET LIST	LIST address is copied to SI
L3:	MOV AL, [SI]	SI register address content is copied to AL register
	MOV DX, CX	CX register is loaded to DX register
L2:	INC SI	Increment SI register
	MOV BL, [SI]	Move SI register memory location

		content to BL register
	CMP AL, BL	Perform comparison on AL and BL
	JB **L1**	If CF is zero, jump to L1
	XCHG AL, [SI]	Exchange the contents of AL and SI register address contents
L1:	LOOP **L2**	Decrement CX register and check CX is zero or not, if CX ≠ 0, jump to L2
	SUB SI, DX	Perform subtract on SI and DX registers
	INC SI	Increment SI register
	MOV CX, DX	DX register is loaded to CX register
	LOOP **L3**	Decrement CX register and check CX is zero or not, if CX ≠ 0, jump to L3
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**06**	**3000**	**04**
3001	**04**	**3001**	**06**
3002	**25**	**3002**	**12**
3003	**12**	**3003**	**25**

PROGRAM: DESCENDING ORDER

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H COUNT EQU 04H LIST DB 00H,00H,00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		XOR AX, AX	Clear the garbage data
		MOV CX, COUNT-1	Decremented count is

			loaded to CX register
		MOV SI, OFFSET LIST	LIST address is copied to SI
	L3:	MOV AL, [SI]	SI register address content is copied to AL register
		MOV DX, CX	CX register is loaded to DX register
	L2:	INC SI	Increment SI register
		MOV BL, [SI]	Move SI register memory location content to BL register
		CMP AL, BL	Perform comparison on AL and BL
		JNB **L1**	If CF is not zero, jump to L1
		XCHG AL, [SI]	Exchange the contents of AL and SI register address contents
	L1:	LOOP **L2**	Decrement CX register and check CX is zero or not, if CX ≠ 0, jump to L2
		SUB SI, DX	Perform subtract on SI and DX registers
		INC SI	Increment SI register
		MOV CX, DX	DX register is loaded to CX register
		LOOP **L3**	Decrement CX register and check CX is zero or not, if CX ≠ 0, jump to L3
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT		OUTPUT	
MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**06**	**3000**	**88**
3001	**88**	**3001**	**25**

3002 **3003**	**25** **12**	**3002** **3003**	**12** **06**

STRING OPERATIONS

PROGRAM: LENGTH OF THE STRING

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H STRING1 DB 'EMPTY VESSELS $' STRLEN EQU ($-STRING1) DATA ENDS	
	CODE SEGMENT ORG 4000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	SUB CL, CL	Clear the CL register
	MOV BL, STRLEN	Copy the string length to BL
	MOV SI, OFFSET STRING1	STRING1 offset address is copied to SI register
BACK:	LODSB	Load string byte
	INC CL	Increment CL
	CMP AL, '$'	Compare AL with '$'
	JNE **BACK**	Jump if AL ≠ '$', to BACK
	MOV RES, CL	Copy CL to RES memory location
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**45**	**300E**	**0E**
3001	**4D**		
3002	**50**		
3003	**54**		
3004	**59**		
3005	**20**		
3006	**56**		
3007	**45**		
3008	**53**		

3009	**53**		
300A	**45**		
300B	**4C**		
300C	**53**		
300D	**20**		

PROGRAM: MOVING A STRING

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 1000H S1 DB 'MSGRCVE' ORG 2000H S2 DB 07 DUP (0) DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data, extra and code segment
	MOV DS, AX	
	MOV ES, AX	
	MOV CL, 07H	Copy data 07H to CL
	LEA SI, S1	Load effective address of S1 to SI
	LEA DI, S2	Load effective address of S2 to DI
	CLD	Clear the direction flag i.e. DF=0
REP:	MOVSB	Repeat the copy of data byte by byte from SI to DI registers
	NOP	Perform no operation
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**4D**	**2000**	**4D**
3001	**56**	**2001**	**56**
3002	**47**	**2002**	**47**
3003	**52**	**2003**	**52**
3004	**43**	**2004**	**43**
3005	**53**	**2005**	**53**
3006	**45**	**2006**	**45**

PROGRAM: REVERSING A STRING

LABEL	MNEMONICS	COMMENTS

	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 2000H STR1 DB 'EMPTY $' STRLEN EQU ($-STR1) ORG 3000H STR2 DB 05H DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data segment
	MOV DS, AX	
	MOV SI, OFFSET STR1	Move address of str1 to SI register
	MOV CX, STRLEN	Copy the length of the string1
	MOV DI, OFFSET STR2	Copy STR2 offset to DI register
	ADD DI, 06	Addition of DI to 05H
	MOV CX, STRLEN	Move strlen data to cx
L1:	LODSB	Load data byte from SI to AL and increment SI
	MOV [DI], AL	Copy data of AL to extra segment address in DI register
	DEC DI	Decrement DI
	LOOP **L1**	Decrement CX, check CX≠0
	NOP	Repeat copying of data from SI to DI in extra register
	INT 03H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
2000	**45**	**3000**	**24**
2001	**4D**	**3001**	**59**
2002	**50**	**3002**	**54**
2003	**54**	**3003**	**50**
2004	**59**	**3004**	**4D**
2005	**24**	**3005**	**45**

PROGRAM: COMPARING TWO STRINGS

LABEL	**MNEMONICS**	**COMMENTS**

	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT ORG 3000H STR1 DB 'EMPTY1 $' STRLEN EQU ($-STR1) ORG 2000H RES DB 00H ORG 1500H STR2 DB 'EMPTY $' DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data, extra and code segment
	MOV DS, AX	
	MOV ES, AX	
	MOV BX, OFFSET STR1	Copy STR1 offset to BX register
	MOV SI, OFFSET STR1	Copy STR1 offset to SI register
	MOV DI, OFFSET STR2	Copy STR2 offset to DI register
	CLD	Clear the direction flag i.e. DF=0
	MOV CX, STRLEN	Move strlen data to CX
REPNE:	CMPSB	Compare every byte of a string
	JZ **NEXT**	If ZF=0, jump to NEXT
	MOV AH, 09H	Copy 09 to AH register
	MOV DX, 'A'	Copy 'A' to DX register
	INT 21H	Interrupt 21H to run function from DOS
	JMP **EXIT**	
NEXT:	MOV AH, 09H	Copy 09 to AH register
	MOV DX, 'E'	Copy 'E' to DX register
	INT 21H	Interrupt 21H to run function from DOS
EXIT:	NOP	No operation
	MOV RES, DL	Copy DL to res, 4CH to AH
	MOV AH, 4CH	
	INT 21H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
2100	**45**	**2000**	**65H**
2101	**4D**		
2102	**50**		
2103	**54**		
2104	**59**		
2105	**24**		

READING DATA FROM KEYBOARD USING DOS

PROGRAM: READING CHARACTER WITHOUT ECHO

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT MSG DB 'ENTER CHARACTERS FROM KEYBOARD:', '$' DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data, and code segment
	MOV AH, 09H	Copy 09 to AH register
	MOV DX, OFFSET MSG	Offset MSG to DX register
	INT 21H	Interrupt 21H to run function from DOS
NEXT:	MOV AH, 08H	Copy 08 to AH register
	INT 21H	Interrupt 21H to run function from DOS
	CMP AL, '#'	Compare AL with '#'
	JNE **NEXT**	Jump to Next if not equal i.e. ZF≠0
	MOV AH, 4CH	Copy 4C to AH register
	MOV AL, 00H	Copy 00H to AL register
	INT 21H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

Enter character from keyboard: H (ZF ≠ 1)

Enter character from keyboard: H (ZF = 1)

PROGRAM: READING CHARACTER WITH ECHO

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT MSG DB 'ENTER CHARACTERS FROM KEYBOARD:', '$' DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data, and code segment
	MOV AH, 09H	Copy 09 to AH register
	MOV DX, OFFSET MSG	Offset MSG to DX register
	INT 21H	Interrupt 21H to run function from DOS
NEXT:	MOV AH, 08H	Copy 08 to AH register
	INT 21H	Interrupt 21H to run function from DOS
	MOV AH, 02H	Copy 02H to AH register
	MOV DL, AL	Copy AL to DL register
	INT 21H	Interrupt 21H to run function from DOS
	CMP AL, '#'	Compare AL with '#'
	JNE **NEXT**	Jump to Next if not equal i.e. ZF≠0
	MOV AH, 4CH	Copy 4C to AH register
	MOV AL, 00H	Copy 00H to AL register
	INT 21H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

Enter character from keyboard: H (ZF ≠ 1)

Enter character from keyboard: H (ZF = 1)

PROGRAM: DISPLAYING A MESSAGE ON SCREEN

LABEL	MNEMONICS	COMMENTS
	ASSUME CS:CODE, DS:DATA	
	DATA SEGMENT MSG DB 'SV	

	COLLEGE OF ENGINEERING', '$' DATA ENDS	
	CODE SEGMENT ORG 3000H	
START:	MOV AX, DATA	Initialize the data, and code segment
	MOV AH, 09H	Copy 09 to AH register
	MOV DX, OFFSET MSG	Offset MSG to DX register
	INT 21H	Interrupt 21H to run function from DOS
	MOV AH, 08H	Copy 08 to AH register
	INT 21H	Interrupt 21H to run function from DOS
	MOV AH, 4CH	Copy 4C to AH register
	MOV AL, 00H	Copy 00H to AL register
	INT 21H	Return control to OS
	CODE ENDS	
	END START	

OBSERVATIONS:

Displaying message: "SV COLLEGE OF ENGINEERING"

Program: Fibonacci series

OFFSET ADDRESS	LABEL	MNEMONICS	COMMENTS
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H CNT DB 00H LIST DB 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		XOR AX, AX	Clear the garbage data
		MOV CL, CNT	Count is loaded to CL register
		MOV SI, OFFSET LIST	LIST address is copied to SI
		MOV AL, 00H	Copy 00H to AL

			register
		MOV BL, 01H	Copy 01H to AL register
		MOV [SI], AL	AL register content is copied to address of SI register
	BACK:	INC SI	Increment SI register
		MOV [SI], BL	BL register content is copied to address of SI register
		ADD AL, BL	Perform addition on AL and BL
		XCHG AL, BL	Exchange the contents of AL and BL register
		LOOP **BACK**	Decrement CL register and check CX is zero or not, if CL ≠ 0, jump to BACK
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**08H**	**3001** **3002** **3003** **3004** **3005** **3006** **3007** **3008** **3009**	**00** **01** **01** **02** **03** **05** **08** **0D** **15**

PROGRAM: FACTORIAL OF A NUMBER

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H N1 DB 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		XOR AX, AX	Clear the garbage data
		MOV CL, N1	N1 is loaded to CL

			register
		MOV AX, 01H	Copy 01H to AX register
	L1:	MUL CL	Perform multiplication on CL and AL
		LOOP **L1**	Decrement CL register and check CL is zero or not, if CL ≠ 0, jump to BACK
		MOV RES, AX	Copy the data of AX to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**05H**	**3001** **3002**	**00** **78**

PROGRAM: SUM OF 'N' NUMBERS

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H CNT DB 00H LIST DB 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 00H	Copy 00H to AX register
		MOV BX, 00H	Copy 00H to BX register
		MOV CL, CNT	CNT is loaded to CL register
		MOV SI, OFFSET LIST	Copy address of LIST to SI address
	L1:	MOV AL, [SI]	Copy content of SI to AL register
		ADD BX, AX	Perform addition of

			AX and BX register
		INC SI	Increment SI register
		LOOP **L1**	Decrement CL register and check CL is zero or not, if CL ≠ 0, jump to BACK
		MOV RES, BX	Copy the data of BX to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**04**	**3005**	**19**
3001	**05**	**3006**	**00**
3002	**09**		
3003	**08**		
3004	**03**		

PROGRAM: SUM OF SQUARES FOR 'N' NUMBERS

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H CNT DB 00H LIST DB 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 00H	Copy 00H to AX register
		MOV BX, 00H	Copy 00H to BX register
		MOV CL, CNT	CNT is loaded to CL register
		MOV SI, OFFSET LIST	Copy address of LIST to SI address
	L1:	MOV AL, [SI]	Copy content of SI to AL register

		MUL AL	Perform multiplication on AL and AL
		ADD BX, AX	Perform addition of AX and BX register
		INC SI	Increment SI register
		LOOP **L1**	Decrement CL register and check CL is zero or not, if CL ≠ 0, jump to BACK
		MOV RES, BX	Copy the data of BX to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001** **3002** **3003** **3004**	**04** **05** **09** **08** **03**	**3005** **3006**	**0B3** **00**

PROGRAM: SUM OF CUBES FOR 'N' NUMBERS

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H CNT DB 00H LIST DB 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 00H	Copy 00H to AX register
		MOV BX, 00H	Copy 00H to BX register
		MOV CL, CNT	CNT is loaded to CL register

		MOV SI, OFFSET LIST	Copy address of LIST to SI address
	L1:	MOV AL, [SI]	Copy content of SI to AL register
		MOV DL, AL	
		MUL AL	Perform multiplication on AL and AL
		MUL DL	Perform multiplication on DL and AL
		ADD BX, AX	Perform addition of AX and BX register
		INC SI	Increment SI register
		LOOP **L1**	Decrement CL register and check CL is zero or not, if CL ≠ 0, jump to BACK
		MOV RES, BX	Copy the data of BX to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**04**	**3005**	**00**
3001	**05**	**3006**	**02**
3002	**09**		
3003	**08**		
3004	**03**		

PROGRAM: TO FIND EVEN OR ODD NUMBER (DL=00 'EVEN', DL=01 'ODD')

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H N1 DB 00H RES DB 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	

		MOV AX, 00H	Copy 00H to AX register
		MOV DL, 00H	Copy 00H to DL register
		MOV CL, 01	01H is loaded to CL register
	L1:	MOV AL, N1	Copy content of N1 to AL register
		ROR AL, CL	Perform RIGHT rotation by CL times i.e. CF=LSB
		JNC **L1**	Perform addition of AX and BX register
		INC DL	Increment DL register
		MOV RES, DL	Copy DL to RES address
		JMP **L2**	Jump to label L2
	L1:	MOV RES, DL	Copy the data of DL to RES address
	L2:	INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**47** **86**	**3001** **3001**	**01 (ODD DL)** **00 (EVEN DL)**

PROGRAM: TO FIND POSITVE OR NEGATIVE NUMBER (DL=00 'POSITIVE', DL=01 'NEGATIVE')

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H N1 DB 00H RES DB 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, 00H	Copy 00H to AX

			register
		MOV DL, 00H	Copy 00H to DL register
		MOV CL, 01	01H is loaded to CL register
	L1:	MOV AL, N1	Copy content of N1 to AL register
		ROL AL, CL	Perform LEFT rotation by CL times i.e. CF=LSB
		JNC **L1**	Perform addition of AX and BX register
		INC DL	Increment DL register
		MOV RES, DL	Copy DL to RES address
		JMP **L2**	Jump to label L2
	L1:	MOV RES, DL	Copy the data of DL to RES address
	L2:	INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001**	**95** **28**	**3001** **3001**	**01 (NEGATIVE DL)** **00 (POSITIVE DL)**

PROGRAM: GCD OF TW0 16-BIT NUMBERS

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H N1 DW 00H N2 DW 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment

		MOV DS, AX	
		MOV AX, N1	Copy N1 to AX register
		MOV BX, N2	Copy N2 to BX register
	AGAIN:	CMP AX, BX	Perform comparison on AX and BX
		JE **EXIT**	
		JB **BIG**	Jump if below to label Big
	ABOVE:	MOV DX, 00H	Copy 00H to DX register
		DIV BX	Perform division with BX register
		CMP DX, 00H	Compare DX with 00H
		JE **EXIT**	Jump if equal i.e. ZF=1 to label Exit
		MOV AX, DX	Copy the contents of DX to AX register
		JMP **AGAIN**	Jump to the label again
	BIG:	XCHG AX, BX	Exchange the contents of AX and BX
		JMP **ABOVE**	Jump to the label above
	EXIT:	MOV RES, BX	Copy BX data to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000	**88**	**3004**	**04**
3001	**00**	**3005**	**00**
3002	**24**		
3003	**00**		

PROGRAM: FINDING THE 16 - BIT PRIME NUMBER

OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H	

		N1 DW 00H N2 DW 00H RES DW 00H DATA ENDS	
		CODE SEGMENT ORG 4000H	
	START:	MOV AX, DATA	Initialize the data segment
		MOV DS, AX	
		MOV AX, N1	Copy N1 to AX register
		MOV BX, AX	Copy AX to BX register
	AGAIN:	MOV AX, N1	Copy N1 to AX register
		DEC BX	Decrement DX register
		XOR DX, DX	Clear DX register
		XOR CX, CX	Clear CX register
		DIV BX	Perform comparison on AX by BX
		CMP DX, 00H	Perform comparison on DX and 00 (check remainder is 0 or data)
		JZ **EXIT**	Jump to label Exit if ZF=1
		CMP BX, 0002H	Perform comparison on BX and 02
		JNZ **AGAIN**	Jump to label Again if ZF is not 1
		INC CX	Increment CX
	EXIT:	MOV RES, CX	Copy CX data to RES address
		INT 03H	Return control to OS
		CODE ENDS	
		END START	

OBSERVATIONS:

INPUT **OUTPUT**

MEMORY LOCATION	**DATA VALUE**	MEMORY LOCATION	**DATA VALUE**
3000 **3001** **3000** **3001**	**07** **00** **08** **00**	**3002** **3003** **3002** **3003**	**01 (cx=1; 00prime)** **00** **00 (cx=0; not prime)**

ANNAMACHARYA EDUCATIONAL TRUST
VIDWAN SARVATRA PUJYATHE

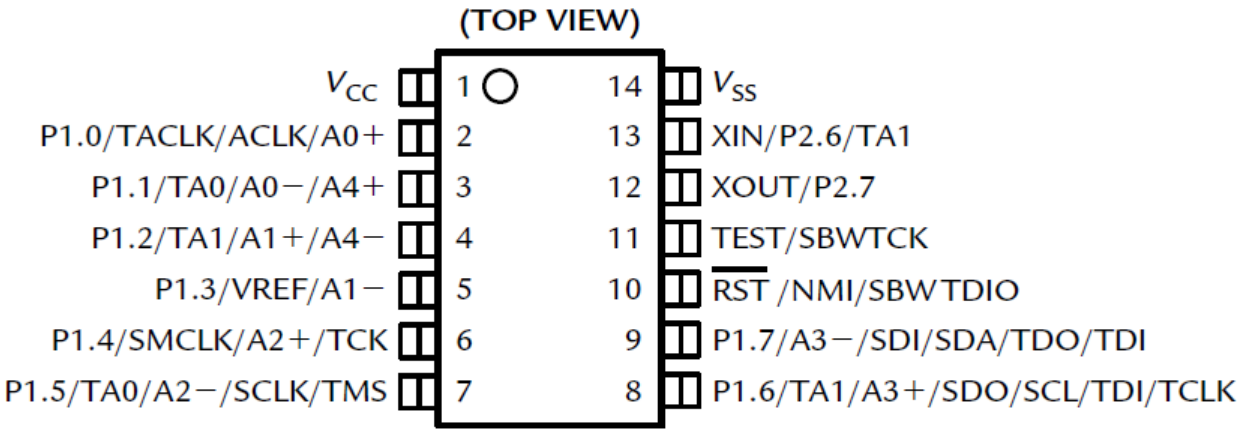
UNIT – IV

Low Power RISC MSP430:

Features:

- It is introduced in the late 1990s by Texas Instruments
- It is a 16-bit RISC Based Microcontroller with Von-Neumann Architecture
- It is Low cost and Low power consumption
- It is suitable for low-power and portable applications
- Its CPU is small and efficient, with a large number of registers
- It has set of intelligent peripherals like I/O Ports, Timers, ADC, DAC, Flexible Clock and USCI-I2C, SPI, UART
- It has 16-bit Data bus and 16-bit Address bus
- It can address 64 KB memory with Flash ROM and RAM
- It has 16 Registers in its CPU and each register is 16-bits wide can be used for either data or address
- It has only 27 Instructions
- It has 7 addressing modes
- Its CPU can run at16 MHz
- It has several low-power modes of operation
- It operates at Low voltage i.e. from 1.8V to 3.6V
- It is extremely easy to put the device into a low-power mode. No special instruction is needed**:** The mode is controlled by bits in the status register. TheMSP430 is awakened by an interrupt and returns automatically to its low-power mode after handling the interrupt.
- It can wake from a standby mode rapidly, perform its tasks, and return to a low-power mode.
- A wide range of peripherals is available, many of which can run autonomously without the CPU for most of the time.
- Many portable devices include liquid crystal displays, which the MSP430 can drive directly.
- It has Prioritized nested interrupts
- Ultralow-power optimization extends battery life
  - – 0.1 µA for RAM Data Retention
  - – 0.8 µA for RTC mode
  - – 250 µA/MIPS for active mode
- Zero-power Brown-Out -Reset (BOR)
- The order of storing the bytes in memory is little endian

Pin Diagram of MSP430F2003 and F2013:

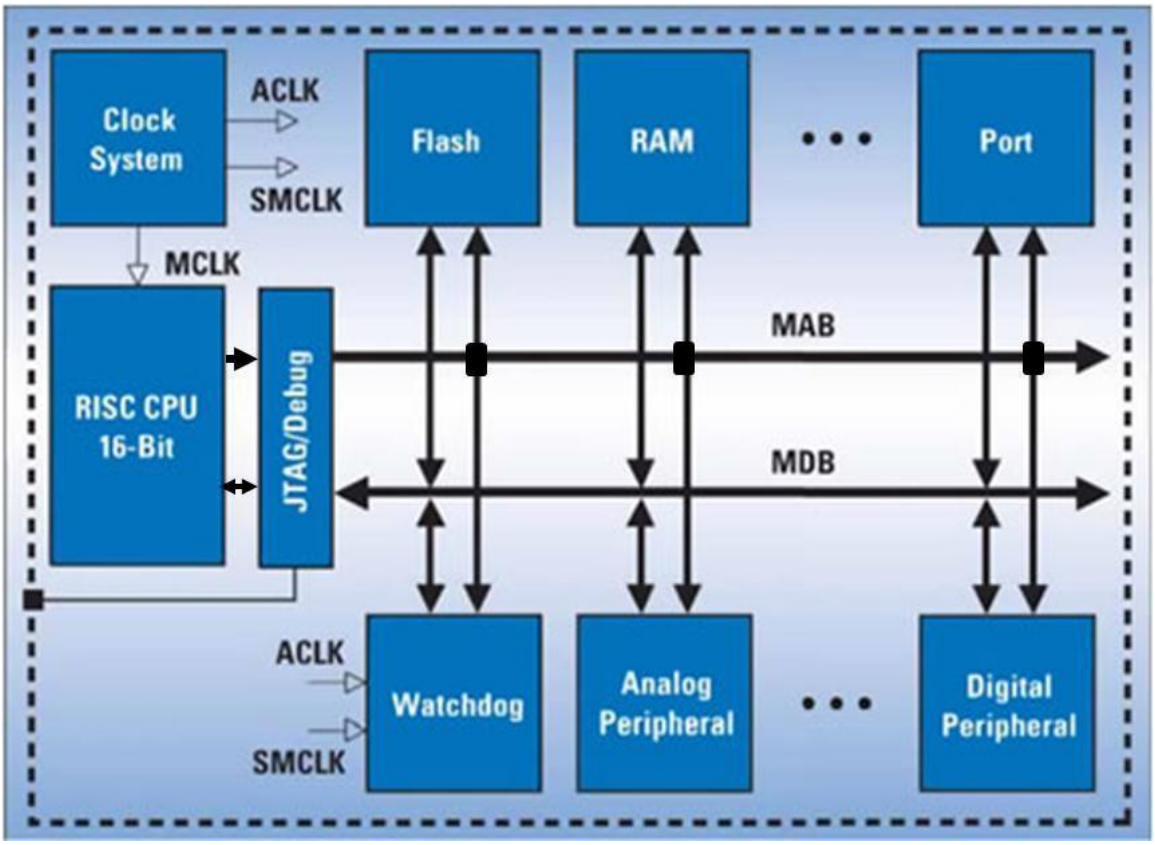


Pin-out of the MSP430F2003 and F2013

- V_{CC} and V_{SS} are the power supply voltage and ground pins for the whole device
- P1.0–P1.7, P2.6, and P2.7 are for digital input and output, grouped into ports P1and P2.
- TACLK, TA0, and TA1 are associated with Timer_A; TACLK can be used as the clock input to the timer, while TA0 and TA1 can be either inputs or outputs. These can be used on several pins because of the importance of the timer.
- A0−, A0+, and so on, up to A4±, are inputs to the analog-to-digital converter. It has four differential channels, each of which has negative and positive inputs. VREF is the reference voltage for the converter.
- ACLK and SMCLK are outputs for the microcontroller's clock signals. These can be used to supply a clock to external components or for diagnostic purposes.
- SCLK, SDO, and SCL are used for the universal serial interface, which communicates with external devices using the serial peripheral interface (SPI) or inter-integrated circuit (I^2C) bus.
- XIN and XOUT are the connections for a crystal, which can be used to provide an accurate, stable clock frequency.
- $\overline{RST}$ is an active low reset signal. Active low means that it remains high near V_{CC} for normal operation and is brought low near V_{SS} to reset the chip.
- NMI is the non-maskable interrupt input, which allows an external signal to interrupt the normal operation of the program.
- TCK, TMS, TCLK, TDI, TDO, and TEST form the full JTAG interface, used to program and debug the device.

- SBWTDIO and SBWTCK provide the Spy-Bi-Wire interface, an alternative to theusual JTAG connection that saves pins.

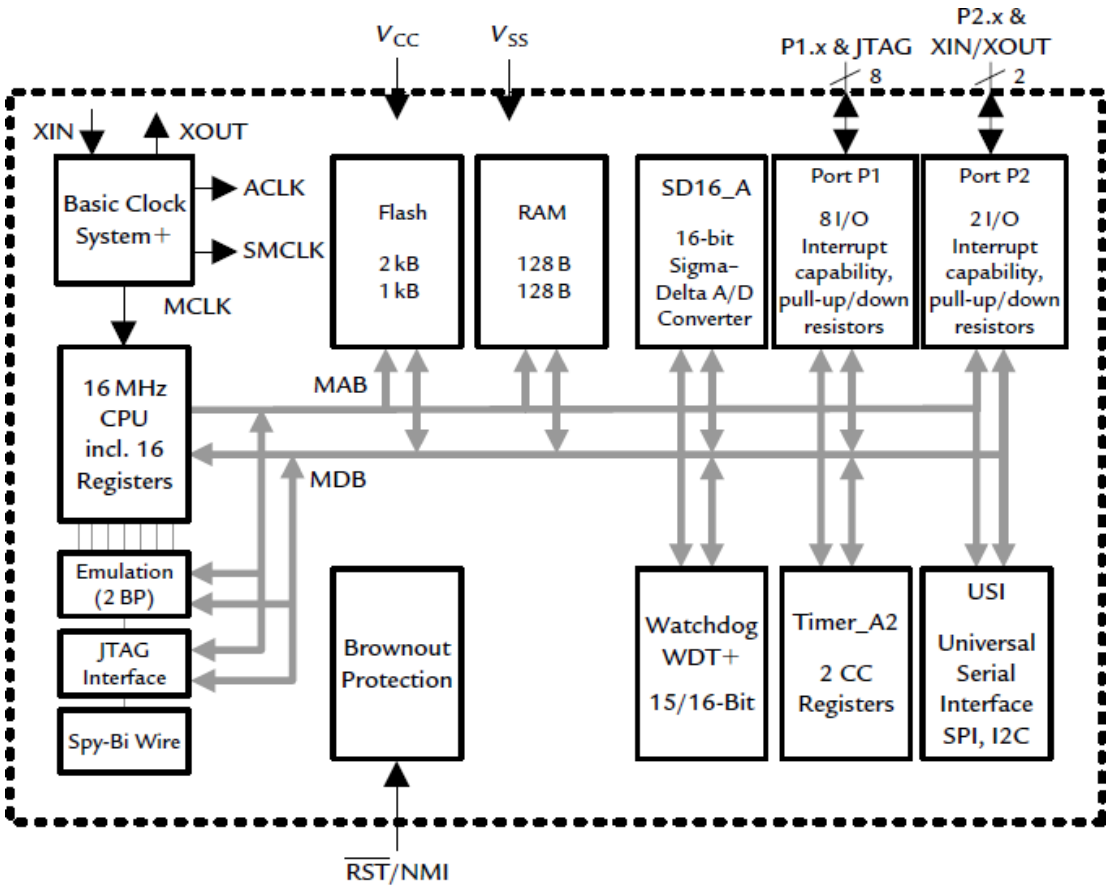
General Block Diagram of MSP430:



Architecture of MSP430F2003 and F2013:

The Architecture of MSP430F2003 and F2013 in shown below:

- On the left are the CPU and its supporting hardware, including the clock generator. The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.
- The main blocks are linked by the memory address bus (MAB) and memory data bus (MDB).
- These devices have flash memory, 1KB in the F2003 or 2KB in the F2013, and128 bytes of RAM.
- The brownout protection comes into action if the supply voltage drops to a dangerous level.
- There are ground and power supply connections. Ground is labeled V_{SS} and is taken to define 0V. The supply connection is V_{CC}.A range of 1.8–3.6V is specified for the F2013.
- Six blocks are shown for peripheral functions (there are many more in larger devices). All MSP430s include input/output ports, Timer_A, and a watchdog



Block diagram of the MSP430F2003 and F2013

timer. The universal serial interface (USI) and sigma–delta analog-to-digital converter (SD16_A) are particular features of this device.

Memory Organization:

- ❑ MSP430 consists of 64K memory which includes Flash/ROM and RAM
- ❑ The memory data bus is 16 bits wide and can transfer either a word of 16 bits or a byte of 8 bits.
- ❑ Memory Addresses are 16-bit
- ❑ Bytes may be accessed at any address but words need more care.
- ❑ Even Address access for word
- ❑ The address of a word is defined to be the address of the byte with the lower address, which must be even. Thus the two bytes at 0x0200 and 0x0201 can be considered as a valid word with address 0x0200, which may be fetched in a single cycle of the bus.
- ❑ On the other hand, it is not possible to treat the two bytes at 0x0201 and 0x0202 as a single word because their address would be 0x0201, which is odd and therefore invalid. These two bytes straddle the boundary of two words.

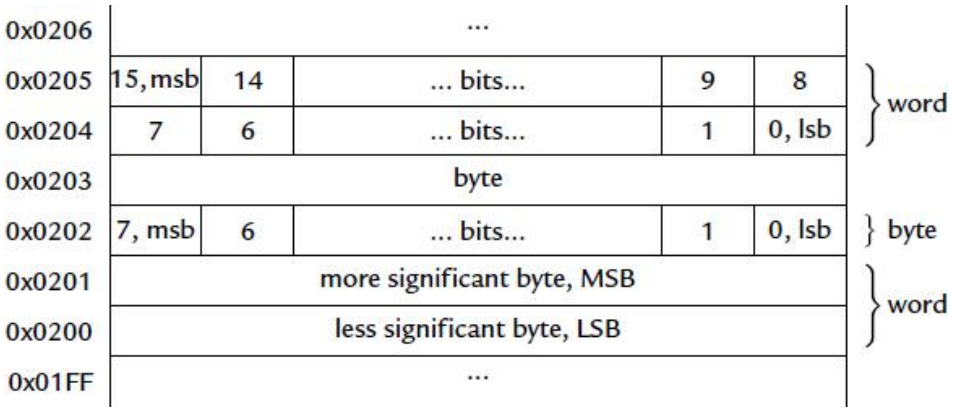


Figure: Ordering of bits, bytes, and words in Memory

Little-endian ordering may appear more logical but has one awkward outcome. A debugger usually displays the contents of memory by showing the value of each byte by default. Addresses increase from left to right across each line. This means that the low-order byte is displayed first, followed by the high-order byte. Thus our value of **0x1234** is displayed as **34 12**. It is easy to be puzzled by this. A simple solution is to display the contents of memory in words instead.

The following figure shows the **Memory Map of the F2013:**

Address	Type of memory
0xFFFF 0xFFC0	interrupt and reset vector table
0xFFBF 0xF800	flash code memory (lower boundary varies)
0xF7FF 0x1100	
0x10FF 0x1000	flash information memory
0x0FFF 0x0C00	*bootstrap loader* (*not in F20xx*)
0x0BFF 0x0280	
0x027F 0x0200	RAM (upper boundary varies)
0x01FF 0x0100	peripheral registers with word access
0x00FF 0x0010	peripheral registers with byte access
0x000F 0x0000	special function registers (byte access)

Figure: Memory map of the MSP430F2013

Here is a brief description of each region:

- **Special function registers:** Mostly concerned with enabling functions of some modules and enabling and signaling interrupts from peripherals.
- **Peripheral registers with byte access and Peripheral registers with word access:** Provide the main communication between the CPU and peripherals. Some must be accessed as words and others as bytes. They are grouped in this way to avoid wasting addresses
- **Random access memory:** Used for variables. This always starts at address 0x0200 and the upper limit depends on the size of the RAM. The F2013 has 128 B.
- **Bootstrap loader:** Contains a program to communicate using a standard serial protocol, often with the COM port of a PC. This can be used to program the chip. All MSP430s had a bootstrap loader until the F20xx.
- **Information memory:** A 256B block of flash memory that is intended for storage of nonvolatile data. This might include serial numbers to identify equipment—an address for a network, for instance—or variables that should be retained even when power is removed.
- **Flash Code memory:** Holds the program, including the executable code itself and any constant data. The F2013 has 2KB but the F2003 only 1KB.
- **Interrupt and reset vectors:** Used to handle "exceptions," when normal operation of the processor is interrupted or when the device is reset. This table was smaller and started at 0xFFE0

Central Processing Unit:

The central processing unit (CPU) executes the instructions stored in memory. It steps through the instructions in the sequence in which they are stored in memory until it encounters a branch or when an exception occurs (interrupt or reset). It includes the arithmetic logic unit (ALU), which performs computation, a set of 16 registers designated R0–R15 and the logic needed to decode the instructions and implement them. The CPU can run at a maximum clock frequency f_{MCLK} of 16MHz in the MSP430F2xx family.

Registers of MSP430 CPU:

The CPU of MSP 430 includes a 16-bit ALU and a set of 16 Registers R0 – R15. In these registers four are special Purpose and 12 are general purpose registers. All the registers can be addressed in the same way.

The special Purpose Registers are:

PC (Program Counter), SP (Stack Pointer), SR (Status Register), CGx (Constant Generator)

The MSP430 CPU includes an arithmetic logic unit (ALU) that handles addition, subtraction, comparison and logical (AND, XOR) operations. ALU operations can affect the overflow, zero, negative, and carry flags in the status register.
The following figure shows the register organization of MSP430 CPU.

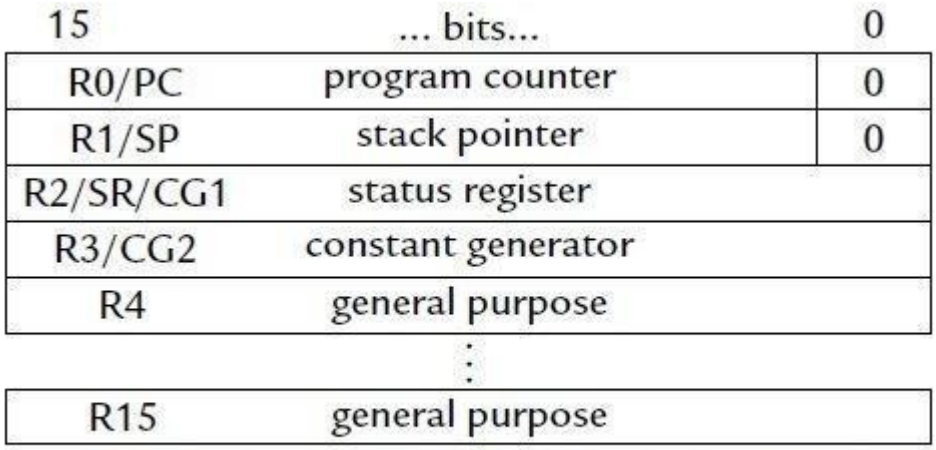


Figure: Registers in the CPU of the MSP430

R0: Program Counter (PC):

This contains the address of the next instruction to be executed. The Program counter is incremented by 2. It is important to note that the PC is aligned at even addresses, because the instructions are 1-3 words.

Subroutines and interrupts also modify the PC but in these cases the previous value (Next line of current instruction which is executing) is saved on the stack and restored later.

R1: Stack Pointer (SP):

- The Stack Pointer (SP/R1) is located in R1.
- The Stack Pointer holds the address of the top of the stack
- Stack can be used by user to store data for later use(instructions: store by PUSH, retrieve by POP)
- The stack pointer is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a pre-decrement, post-increment scheme.
- The stack is allocated at top of RAM and grows down towards the low address. SP holds the address of top of the stack.
- The stack pointer holds the address of the most recently added word
- Stack can be used by subroutine calls to store the program counter value for return at subroutine's end (RET)

- Used by interrupt - system stores the actual PC value first, then the actual status register content (on top of stack) on return from interrupt (RETI) the system get the same status as just before the interrupt happened (as long as none has changed the value on TOS) and the same program counter value from stack.

The operation of the stack is illustrated in below Figure.

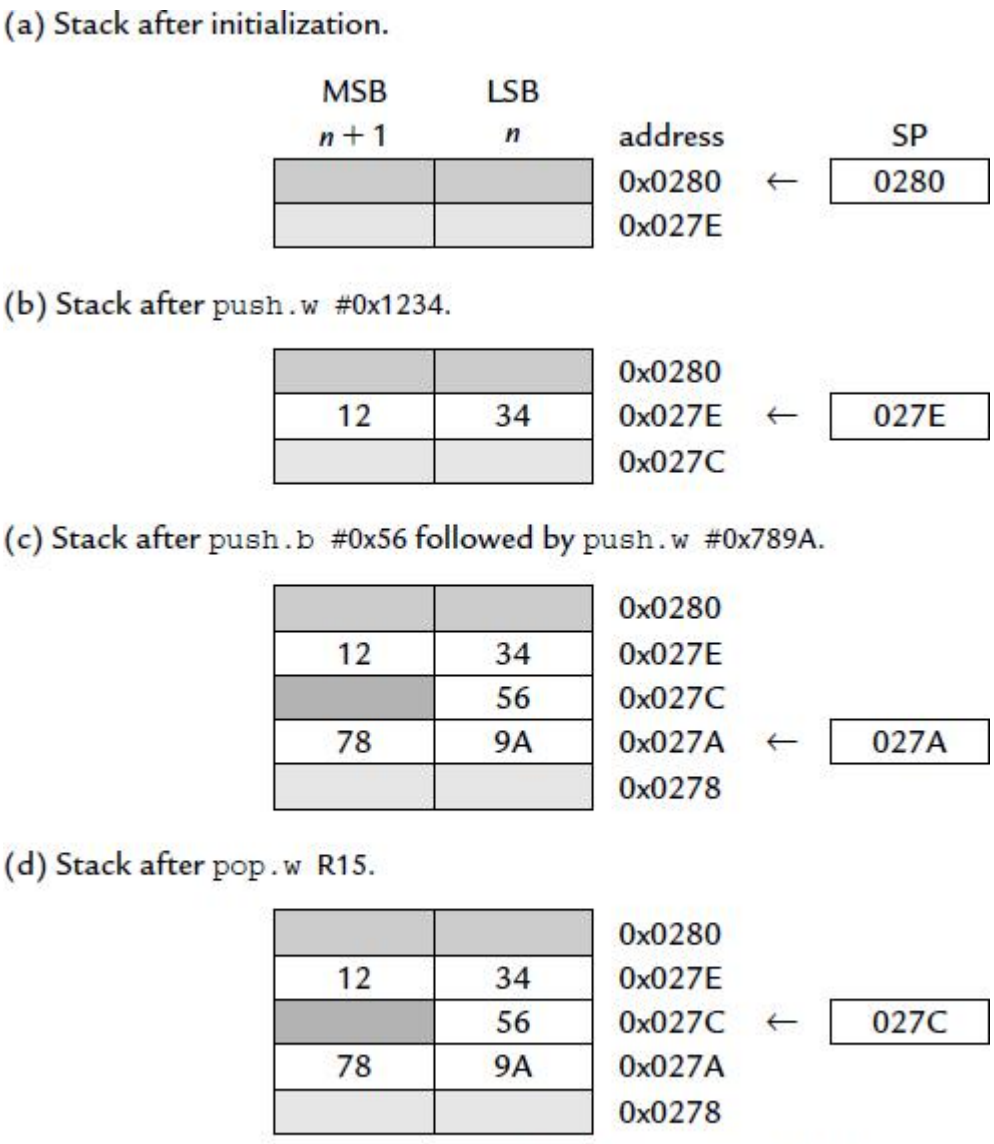


Figure: Operation of the stack in the MSP430F2013, whose RAM lies from 0x0200 to 0x027F

Note: For programs written in C, the compiler initializes the stack automatically as part of the startup code, which runs silently before the program starts, but you must initialize SP yourself in assembly language.

R2: Status Register (SR):

The Status Register (SR/R2) is a 16 bit register, and it stores the state and control bits. The system flags are changed automatically by the CPU depending on the result of an operation in a register. The reserved bits are not used in the

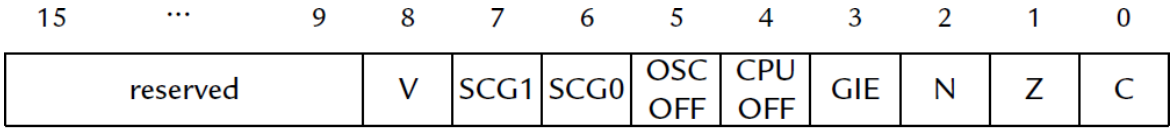


Figure: Individual bits in the status register

MSP430.

- The **Carry flag C** is set when the result of an arithmetic operation is too large to fit in the space allocated. In other words, an overflow occurred.
- The **Zero flag Z** is set when the result of an operation is 0.
- The **Negative flag N** is made equal to the msb of the result, which indicates a negative number if the values are signed.
- The **Signed Over Flow flag V** is set when the result of a signed operation has overflowed, even though a carry may not be generated

❖ Remember that a byte can hold the values 0 to 0xFF if it is unsigned or −0x80 to 0x7F if it is signed.

- **Enable Interrupts:** Setting the **General Interrupt Enable–GIE** bit enables maskable interrupts, provided that the individual sources of interrupts have themselves been enabled. Clearing the bit disables all maskable interrupts. There are also non-maskable interrupts, which cannot be disabled with GIE.
- **Control of Low-Power Modes:** The **CPUOFF, OSCOFF, SCG0 (**System Clock Generator**),** and **SCG1** bits control the mode of operation of the MCU. All systems are fully operational when all bits are clear. Setting combinations of these bits puts the device into one of its low-power modes

R2/R3: Constant Generator Registers (CG1/CG2):

This provides the six most frequently used values so that they need not be fetched from memory whenever they are needed. It uses both R2 and R3 to provide a range of useful values by using the CPU's addressing modes.

R4 - R15: General–Purpose Registers:

The remaining 12 registers R4–R15 have no dedicated purpose and may be used as general working registers. They may be used for either data or addresses because both are 16-bit values, which simplify the operation significantly.
The following figure shows the MSP430 CPU Block Diagram.

The CPU features include:

- RISC architecture with 27 instructions and 7 addressing modes.
- Orthogonal architecture with every instruction usable with every addressing mode.
- Full register access including program counter, status registers, and stack pointer.
- Single-cycle registers operations.
- Large 16-bit register file reduces fetches to memory.
- 16-bit address bus allows direct access and branching throughout entire memory range.
- 16-bit data bus allows direct manipulation of word-wide arguments.
- Constant generator provides six most used immediate values and reduces code size.
- Direct memory-to-memory transfers without intermediate register holding.
- Word and byte addressing and instruction formats.

➢ An **orthogonal** instruction set is an instruction set architecture where all instruction types can use all addressing modes. It is "orthogonal" in the sense that the instruction type and the addressing mode vary independently.

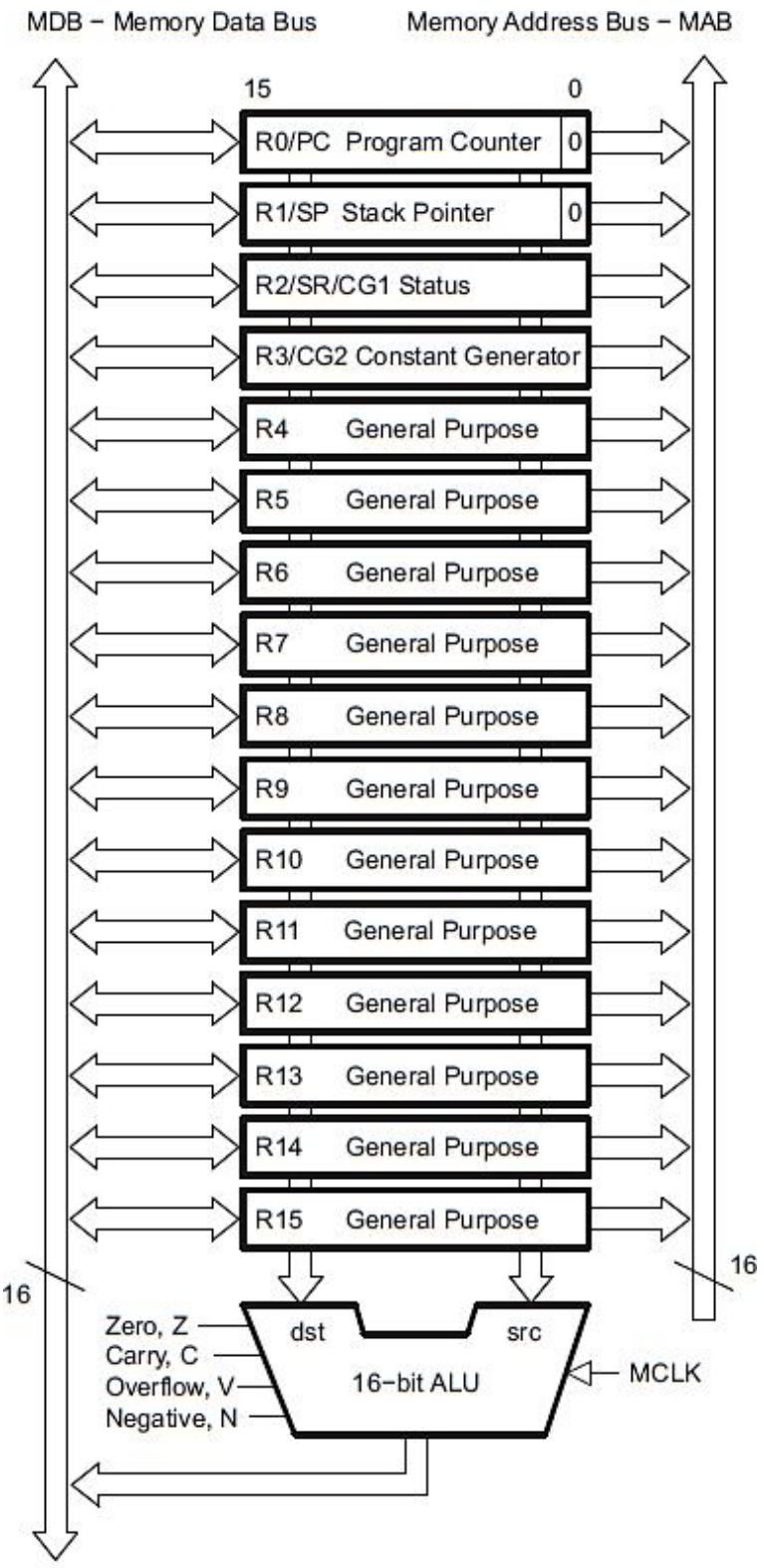


Figure: CPU Block Diagram

Addressing modes:

The MSP430 supports seven addressing modes. They are:

1) Register mode
2) Indexed mode
3) Symbolic mode
4) Absolute mode
5) Indirect register mode
6) Indirect auto increment mode
7) Immediate mode

1) Register Mode:

Register mode operations work directly on the processor registers, R4 through R15, or on special function registers, such as the program counter or status register. They are very efficient in terms of both instruction speed and code space.

Ex: MOV.b R4, R5 *; move (copy) byte from R5 to R6*

MOV.w R4, R5*; move (copy) word from R5 to R6*

Operation: Move (copy) the contents of source (register R4) to destination (register R5), Register R4 is not affected. **.b →** for byte operation& **.w →** for word operation

2) Indexed mode:

The Indexed mode commands are formatted as X(Rn), where X is a constant and Rn is one of the CPU registers. The absolute memory location X+Rn is addressed.

Indexed mode addressing is useful for applications such as lookup tables

Ex: MOV.b 4(R5), 6(R6) ; move data from address **4 + (R5)**

; to address **6 + (R6)**

Operation: Move the contents of the source address (contents of R5 + 4) to the destination address (contents of R6 + 6). Thesource and destination registers (R5 and R6) are not affected.

3) Symbolic mode(PC Relative):

Symbolic mode allows the assignment of labels to fixed memory locations, so that those locations can be addressed. This is useful for the development of embedded programs.

In this case the program counter PC is used as the base address, so the constant is the offset to the data from the PC. TI calls this the symbolic mode although it is usually described as PC-relative addressing. It is used by writing the symbol for a memory location without any prefix.

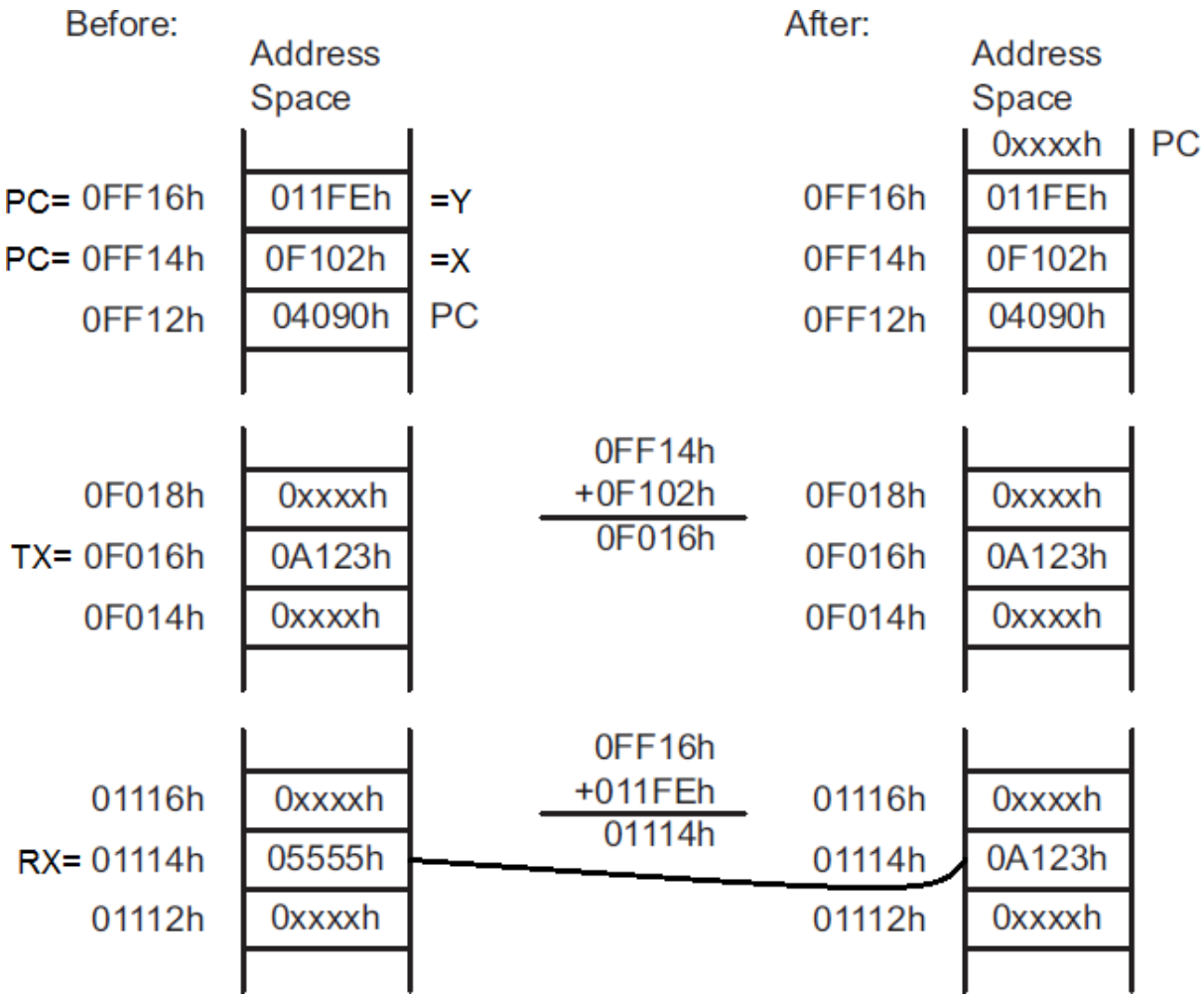
Ex: MOV.w TX, RX ; move data from src address **TX**
; to dst address **RX**

The assembler replaces the above instruction by the indexed form

MOV X(PC),Y(PC) ; where X=TX-PC ⇒ TX=PC+X **&**
; Y=RX-PC ⇒ RX=PC+Y

Operation: Move the contents of the source address TX (contents of PC + X) to the destination address RX (contents of PC + Y). The words after the instruction contains the differences between the PC and the source address i.e. X or destination address i.e. Y. The assembler computes and inserts offsets X and Y automatically.

Ex: MOV.wTX, RX ; Src. address TX = 0F016h
;Dst. address RX = 01114h



4) Absolute mode:

Similar to Symbolic mode, with the difference that the label is preceded by "**&**". This mode is used for special function and peripheral registers, whose addresses are fixed in the memory map.

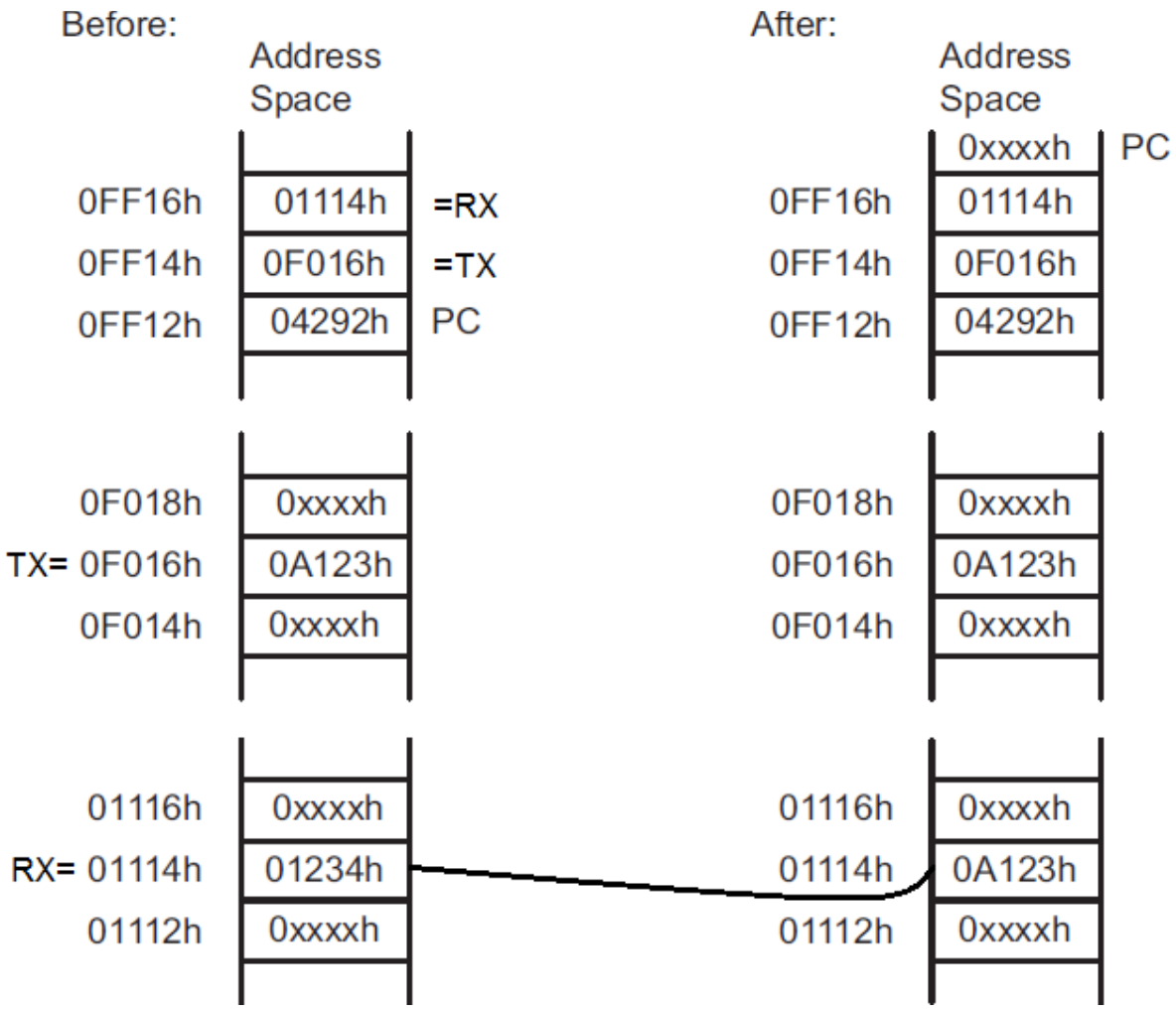
Ex: MOV.w &TX, &RX ; move data from src address **TX**
; to dst address **RX**

The assembler replaces the above instruction by the indexed form

MOV X(SR),Y(SR) MOV X(0),Y(0)
; Where X=TX-0 TX=X Absolute Address**&**
; Y=RX-0 RX=Y Absolute Address

Operation: Move the contents of the source address TX to the destination address RX. The words after the instruction contain the absolute address of the source and destination addresses.

Ex: MOV.w&TX, &RX ; Src. address TX = 0F016h
;Dst. addressRX = 01114h



Indirect register mode:

This is available only for the source and is shown by the symbol @ in front of a register,such as @R5. It means that the contents of R5 are used as the address of the operand. In other words, R5 holds a pointer rather than a value.

Indirect addressing cannot be used for the destination so indexed addressing must be used instead Indirect addressing i.e. the substitute for destination operand is 0(Rd).

Ex: MOV.w @R10, 0(R11)

Operation: Move the contents of the source address (contents of R10) to the destination address (contents of R11). The registers are not modified.

6) Indirect Autoincrement Mode:

Again this is available only for the source and is shown by the symbol **@** in front of a register with a **+** sign after it, such as @Rn+. It uses the value in Rn as a

pointer and automatically increments it afterward by 1 for a byte operation or by 2 for a word operation after the fetch.

Ex: MOV.w @R10+, 0(R11)

Operation: Move the contents of the source address (contents of R10) to the destination address (contents of R11). After that R10 is incremented by 1 for a byte operation, or 2 for a word operation.

7) Immediate Mode:

Immediate mode is used to assign constant values to registers or memory locations.

Ex: MOV.b #45h, R5

Operation: Move the immediate constant 45h to the destination (register R5).

Instruction Set:

- The complete MSP430 instruction set consists of 27 core instructions and 24 emulated instructions.
- The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves; instead they are replaced automatically by the assembler with an equivalent core instruction. There is no code or performance penalty for using emulated instruction.
- The instruction set is orthogonal withfew exceptions, meaning that all addressing modes can be used with all instructions andregisters.

Movement Instructions:

There is only the one **"mov"** instruction to move data. It can address all of memory as either source or destination, including both registers in the CPU and the whole memory map.

```
mov.wsrc, dst        ; move (copy)        dst = src
```

Stack Operations

These instructions either push data onto the stack or pop them off.

```
push.wsrc      ; push data onto stack--SP = src
pop.w dst              ; pop data off stack.   dst = SP++ emulated
```

The pop operation is emulated using post-increment addressing but push requires a special instruction because pre-decrement addressing is not available.

1) Arithmetic and Logic Instructions:

Arithmetic Instructions with Two Operands

add.w src ,dst	; add	dst += src
addc.w src ,dst	; add with carry	dst += (src + C)
adc.w dst	; add carry bit	dst += C **emulated**
sub.w src ,dst	; subtract	dst -= src
subc.wsrc ,dst	; subtract with borrow	dst -= (src + ~C)
sbc.w dst	; subtract borrow bit	dst -= ~C **emulated**
cmp.w src ,dst	; compare, set flags only	(dst - src)

Note: The compare operation **"cmp"** is the same as subtraction except that only the bits in SR are affected; the result is not written back to the destination.

Arithmetic Instructions with One Operand

All these are emulated, which means that the operand is always a destination:

clr.wdst	; clear	dst = 0 **emulated**
dec.wdst	; decrement	dst -- **emulated**
decd.wdst	; double decrement	dst -= 2 **emulated**
inc.wdst	; increment	dst++ **emulated**
incd.wdst	; double increment	dst += 2 **emulated**
tst.wdst	; test (compare with 0) (dst - 0)	**emulated**

Decimal Arithmetic

These instructions are used when operands are binary-coded decimal (BCD) rather than ordinary binary values.

dadd.w src ,dst	; decimal add with carry	dst += src + C
dadc.w dst	; decimal add carry bit	dst += C **emulated**

Logic Instructions with Two Operands

and.w src ,dst	; bitwise and	dst &= src
xor.w src ,dst	; bitwise xor	dst ^= src
bit.w src ,dst	; bitwise test, set flags only	(dst & src)
bis.w src ,dst	; bit set	dst |= src
bic.w src ,dst	; bit clear	dst &= ~src

Note: The **and** &**bitwise test** operations are identical except that bit is only a test and does not change its destination.

Logic Instructions with One Operand

There is only one of these, the invert **"inv"** instruction, also known as ones complement, which changes all bits of 0 to 1 and those of 1 to 0:

inv.w dst	; invert bits	dst = ~dst	**emulated**

Byte Manipulation

These instructions do not need a suffix because the size of the operands is fixed:

swpb src	; swap upper and lower bytes (word only)
sxt src	; extend sign of lower byte (word only)

- The swap bytes instruction "swpb" swaps the two bytes in a word.
- The sign extend instruction "sxt" is used to convert a signed byte into a signed word.

Operations on Bits in Status Register

There is a set of emulated instructions to set or clear the four lowest bits in the status register, those that can be masked using the constant generator:

clrc	; clear carry bit c = 0	**emulated**
clrn	; clear negative bit n = 0	**emulated**
clrz	; clear zero bit z = 0	**emulated**
setc	; set carry bit c = 1	**emulated**
setn	; set negative bit n = 1	**emulated**
setz	; set zero bit z = 1	**emulated**
dint	; disable general interrupts GIE=0	**emulated**
eint	; enable general interrupts GIE =1	**emulated**

2) Shift and Rotate Instructions:

There are three types of shifts
(i) logicalshift (ii) arithmetic shift (iii) rotation.

- **Logical shift** inserts zeroes for both right and left shifts.
- **Arithmetic shift** inserts zero for left shifts at **lsb**but for the right shifts the **msb** is replicated.
- **Rotation** does not introduce or lose any bits; bits that are moved out of one end ofthe register are passed around to the other.

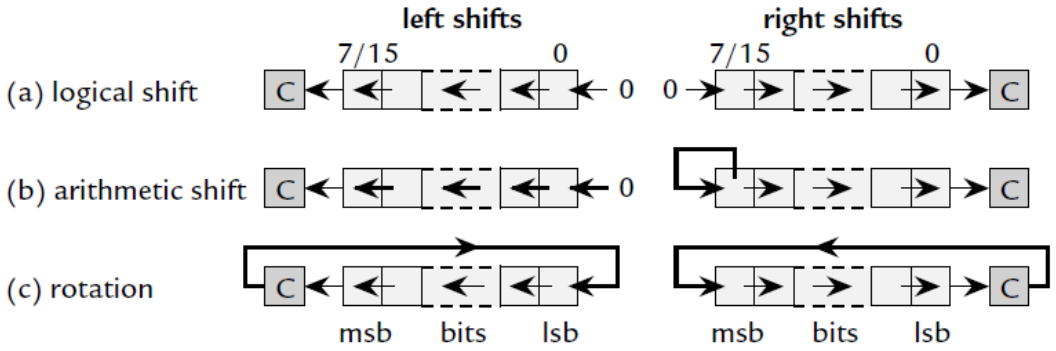


Figure: Left and right logical shifts, arithmetic shifts, and rotations on an 8- or 16-bit register.

- The MSP430 has arithmetic shifts and rotations, all of which use the carry bit.
  The right-shifts are native instructions but the left shifts are emulated

rla dst	; arithmetic shift left	**emulated**
rra src	; arithmetic shift right	
rlc dst	; rotate left through carry	**emulated**
rrc src	; rotate right through carry	

3) Flow of Control:

Subroutines, Interrupts, and Branches

brdst	; branch (go to)	PC = dst	**emulated**
call src	; call subroutine		
ret	; return from subroutine		**emulated**
reti	; return from interrupt		
nop	; no operation (consumes single cycle)		**emulated**

Jumps Unconditional and Conditional

- The unconditional jump instruction is

```
jmp label          ; unconditional jump
```

- **jmp** fits in a single word, including the offset, but its range is limited to about±1KB from the current location.
- **br** can go anywhere in the address space and use any addressing mode but isslower and requires an extra word of program storage.

- The conditional jumps are the "decision-making" instructions and test certain bits or combinations in the status register.

```
jc label     ; jump if carry set,      C = 1    same as jhs
jnc label    ; jump if carry not set,C = 0    same as jlo
jn label     ; jump if negative,               N = 1
jz label     ; jump if zero,           Z = 1   same as jeq
jnz label    ; jump if nonzero,                Z = 0    same as jne
```

```
jeq label    ; jump if equal,                  dst = src          same as jz
jne label    ; jump if not equal,     dst!= src same as jnz
jhs label    ; jump if higher or same,         dst>= src same as jc
jlo label    ; jump if lower,                  dst < src same as jnc
```

```
jge label    ; jump if greater or equal,       dst >= src         signed values
jl(t) label  ; jump if less than,              dst < src          signed values
```

Many branches have two names to reflect different usage. For example, it is clearer to use**jc** if the carry bit is used explicitly—after a rotation, for instance—but **jhs** is more appropriate after a comparison.

Assume that the "comparison" jumps follow **cmp.wsrc,dst**, which sets the flags according to the difference **dst-src**. Alternatively, **tst.wdst** sets the flags for **dst – 0.**

Both mnemonics jl and jlt are used. It is up to the programmer to select the correct instruction. For example, suppose that two bytes contain 0x99 and 0x01. They are related by 0x99 > 0x01 if the values are unsigned but 0x99 < 0x01 if they

are signed, twos complement numbers because 0x99 is the representation of −0x67.

The following table shows the list of 27 core instructions of MSP430:

S.No.	Mnemonic	S-Reg, D- Reg	Operation	Status Bits			
				V	N	Z	C
1	MOV	src,dst	src → dst	-	-	-	-
2	ADD	src,dst	src + dst → dst	*	*	*	*
3	ADDC	src,dst	src + dst + C → dst	*	*	*	*
4	SUB	src,dst	dst + .not.src + 1 → dst	*	*	*	*
5	SUBC	src,dst	dst + .not.src + C → dst	*	*	*	*
6	CMP	src,dst	dst → src	*	*	*	*
7	DADD	src,dst	src + dst + C → dst (decimally)	*	*	*	*
8	BIT	src,dst	src .and. dst	0	*	*	Z
9	BIC	src,dst	not src .and. dst → dst	-	-	-	-
10	BIS	src,dst	src .or. dst → dst	-	-	-	-
11	XOR	src,dst	src .xor. dst → dst	*	*	*	Z
12	AND	src,dst	src .and. dst → dst	0	*	*	Z
13	RRC	dst	C → MSB → ..... LSB → C	*	*	*	*
14	RRA	dst	MSB → MSB →... LSB → C	0	*	*	*
15	PUSH	src	SP - 2 → SP, src → SP	–	–	–	–
16	SWPB	dst	bit 15...bit 8 ↔ bit 7...bit 0	–	–	–	–
17	CALL	dst	Call subroutine in lower 64KB	–	–	–	–
18	RETI		TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SP	*	*	*	*
19	SXT	dst	Register mode: bit 7 → bit 8...bit 19 Other modes: bit 7 → bit 8...bit 15	0	*	*	Z
20	JEQ/JZ	Label	Jump to label if zero bit is set	Status bits are not affected			
21	JNE/JNZ	Label	Jump to label if zero bit is reset				
22	JC	Label	Jump to label if carry bit is set				
23	JNC	Label	Jump to label if carry bit is reset				
24	JN	Label	Jump to label if negative bit is set				
25	JGE	Label	Jump to label if (N .XOR. V) = 0				
26	JL	Label	Jump to label if (N .XOR. V) = 1				

27	JMP	Label	Jump to label unconditionally	

Note:

*=Statusbitisaffected.

–=Statusbitisnotaffected.

0=Statusbitiscleared.

1=Statusbitisset.

The following table shows the list of 24 Emulated Instructions:

Emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves. Instead, they are replaced automatically by the assembler with a core instruction. There is no code or performance penalty for using emulated instructions.

S.No.	Instruction	Explanation	Emulation	Status Bits			
				V	N	Z	C
1	ADC dst	Add Carry to dst	ADDC #0,dst	*	*	*	*
2	BR dst	Branch indirectly dst	MOV dst,PC	–	–	–	–
3	CLR dst	Clear dst	MOV #0,dst	–	–	–	–
4	CLRC	Clear Carry bit	BIC #1,SR	–	–	–	0
5	CLRN	Clear Negative bit	BIC #4,SR	–	0	–	–
6	CLRZ	Clear Zero bit	BIC #2,SR	–	–	0	–
7	DADC dst	Add Carry to dst decimally	DADD #0,dst	*	*	*	*
8	DEC dst	Decrement dst by 1	SUB #1,dst	*	*	*	*
9	DECD dst	Decrement dst by 2	SUB #2,dst	*	*	*	*
10	DINT	Disable interrupt	BIC #8,SR	–	–	–	–
11	EINT	Enable interrupt	BIS #8,SR	–	–	–	–
12	INC dst	Increment dst by 1	ADD #1,dst	*	*	*	*
13	INCD dst	Increment dst by 2	ADD #2,dst	*	*	*	*
14	INV dst	Invert dst	XOR #–1,dst	*	*	*	*
15	NOP	No operation	MOV R3,R3	–	–	–	–
16	POP dst	Pop operand from stack	MOV @SP+,dst	–	–	–	–
17	RET	Return from subroutine	MOV @SP+,PC	–	–	–	–
18	RLA dst	Shift left dst arithmetically	ADD dst,dst	*	*	*	*
19	RLC dst	Shift left dst logically through Carry	ADDC dst,dst	*	*	*	*
20	SBC dst	Subtract Carry from dst	SUBC #0,dst	*	*	*	*
21	SETC	Set Carry bit	BIS #1,SR	–	–	–	1
22	SETN	Set Negative bit	BIS #4,SR	–	1	–	–
23	SETZ	Set Zero bit	BIS #2,SR	–	–	1	–
24	TST dst	Test dst (compare with 0)	CMP #0,dst	0	*	*	1

Note:

*=Statusbitisaffected.

–=Statusbitisnotaffected.

0=Statusbitiscleared.

1=Statusbitisset.

Instruction Formats:

There are three core-instruction formats:

1) Double operand (Format I)
2) Single operand (Format II)
3) Jump (Format III)

Note: The Instruction Formats can be used to find the Machine codes manually for assembly language instructions

- **opcode-** is the operation code
- **src-**The source operand defined by As and S-Reg
- **dst-** The destination operand defined by Ad and D-Reg
- **As** (2 bits-addressing bits) gives the mode of addressing for the source, which has four basic modes.
- **Ad** (1 bit-addressing bits) similarly gives mode of addressing for the destination, which has only two basic modes.
- **S-Reg** and **D-Reg** specify the CPU registers associated with the source and destination, the registers either contain the data or addresses.
- **B/W** (1 bit) Byte or Word operation:
  - 0: word operation, 1: byte operation

As Bits		*Addressing Mode*
0	**0**	**Register**
0	**1**	**Indexed**
1	**0**	**Indirect Reg.**
1	**1**	**Indirect Auto-Increment /Immediate**

Ad Bit	*Addressing Mode*
0	**Register**
1	**Indexed**

Double-Operand (Format I) Instructions:

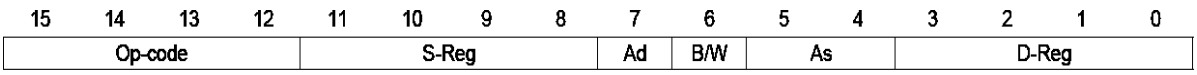


Figure: Double Operand Instruction Format

Single-Operand (Format II) Instructions:

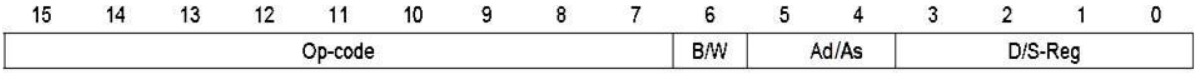


Figure: Single Operand Instruction Format

Jump Instruction Format:

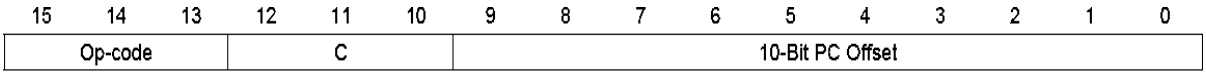


Figure: Jump Instruction Format

Here is an example of a move from register to register with the resulting machine code:

```
MOV.w R5, R6          ; 4506
```

The instruction can be broken into its fields of opcode = 4, S-reg = 5, Ad = 0, B/W = 0,As= 0, D-reg = 6. What do these mean?

- The opcode of 4 represents a move.
- The bit B/W = 0 shows that the operand is a word.
- The addressing mode for the source is As = 0, which is register. The register is S-reg = 5, which is R5 as expected.
- Similarly, the addressing mode for the destination is Ad = 0, which again means register. The register is D-reg = 6 = R6.

Here is another example addition rather than a move:

```
ADD.w R5, R6          ; 5506
```

The machine code is identical except for the opcode which is 5 rather than 4. The specification of the operands is unchanged. This is because of the orthogonality: All instructions use the same addressing modes.

Let us move an immediate value instead of a register:

```
MOV.w #5, R6          ; 4036 0005
```

Now there are two words. The fields of the instruction are opcode = 4, S-reg = 0, Ad = 0, B/W = 0, As = 3 = 11b, D-reg = 6. The difference is in the specification of the source, which means immediate operand. The register is S-Reg = 0. The value itself is contained in the second word in the machine code.

The following table shows the **opcodes** for core instructions:

OPCODE (HEX)	CORE INSTRUCTION
4	MOV
5	ADD
6	ADDC
7	SUB

8	SUBC
9	CMP
A	DADD
B	BIT
C	BIC
D	BIS
E	XOR
F	AND

OPCODE (BINARY)	**CORE INSTRUCTION**
000100000	RRC
000100001	SWPB
000100010	RRA
000100011	SXT
000100100	PUSH
000100101	CALL
000100110	RETI

OPCODE (BINARY)	**CONDITION (C) (BINARY)**	**CORE INSTRUCTION**
001	000	JNE/JNZ
001	001	JEQ/JZ
001	010	JNC/JLO
001	011	JC/JHS
001	100	JN
001	101	JGE
001	110	JL
001	111	JMP

Instruction Timing:

- It takes one cycle to fetch the instruction word itself. This is all if both source and destination are in CPU registers.
- One more cycle is needed to fetch the source if it is given indirectly as @Rn or @Rn+, in which case the address is already in the CPU. This includes immediate data.
- Alternatively, two more cycles are needed if one of the indexed modes is used. The first is to fetch the base address, which is added to the value in a CPU register to get the address of the source. A second cycle is necessary to fetch the operand itself. This includes absolute and symbolic modes.
- Two more cycles are needed to fetch the destination in the same way if it is indexed.
- A final cycle is needed to write the destination back to memory if required; no allowance is needed for a register in the CPU.

Table: Number of MCLK cycles required for typical instructions. It applies only to logical and arithmetic instructions and when the destination is not PC.

Format I Destination	Source		
	Rs	@Rs, @Rs+	S(Rs)
Rd	1	2	3
D(Rd)	4	5	6
Format II	1	3	4

(a) Two operands (Format I), destination is register.

`add.w Rs, Rd`	`add.w @Rs, Rd`	`add.w S(Rs), Rd`
fetch instruction	fetch instruction	fetch instruction
	fetch source @Rs	fetch S
		fetch source S(Rs)

(b) Two operands (Format I), destination is indexed.

`add.w Rs, D(Rd)`	`add.w @Rs, D(Rd)`	`add.w S(Rs), D(Rd)`
fetch instruction	fetch instruction	fetch instruction
fetch D	fetch source @Rs	fetch S
fetch destination D(Rd)	fetch D	fetch source S(Rs)
write destination D(Rd)	fetch destination D(Rd)	fetch D
	write destination D(Rd)	fetch destination D(Rd)
		write destination D(Rd)

(c) One operand (Format II)

`rra.w Rs`	`rra.w @Rs`	`rra.w S(Rs)`
fetch instruction	fetch instruction	fetch instruction
	fetch source @Rs	fetch S
	write source @Rs	fetch source S(Rs)
		write source S(Rs)

Figure: Cycle-by-cycle operation of typical instructions, showing the traffic with memory.

Variants of the MSP430 Family:

MSP430x1xx:

- Provides a wide range of general purpose devices from simple versions to complete systems for processing signals
- There is a broad selection of peripherals and some include a hardware multiplier, which can be used as rudimentary digital signal processor
- Packages have 20–64 pins

MSP430x2xx:

- Introduced in 2005.
- CPU can run at 16 MHz, double the speed of earlier devices, while consuming only half the current at the same speed.
- 14 pin PDIP package.
- Pull-up or pull-down resistors are provided on the inputs to reduce the number of external components needed.
- Even the smallest,14-pin devices offer a 16-bit sigma–delta ADC

MSP430x3xx:

- The original family, which includes drivers for LCDs. It is now obsolescent.

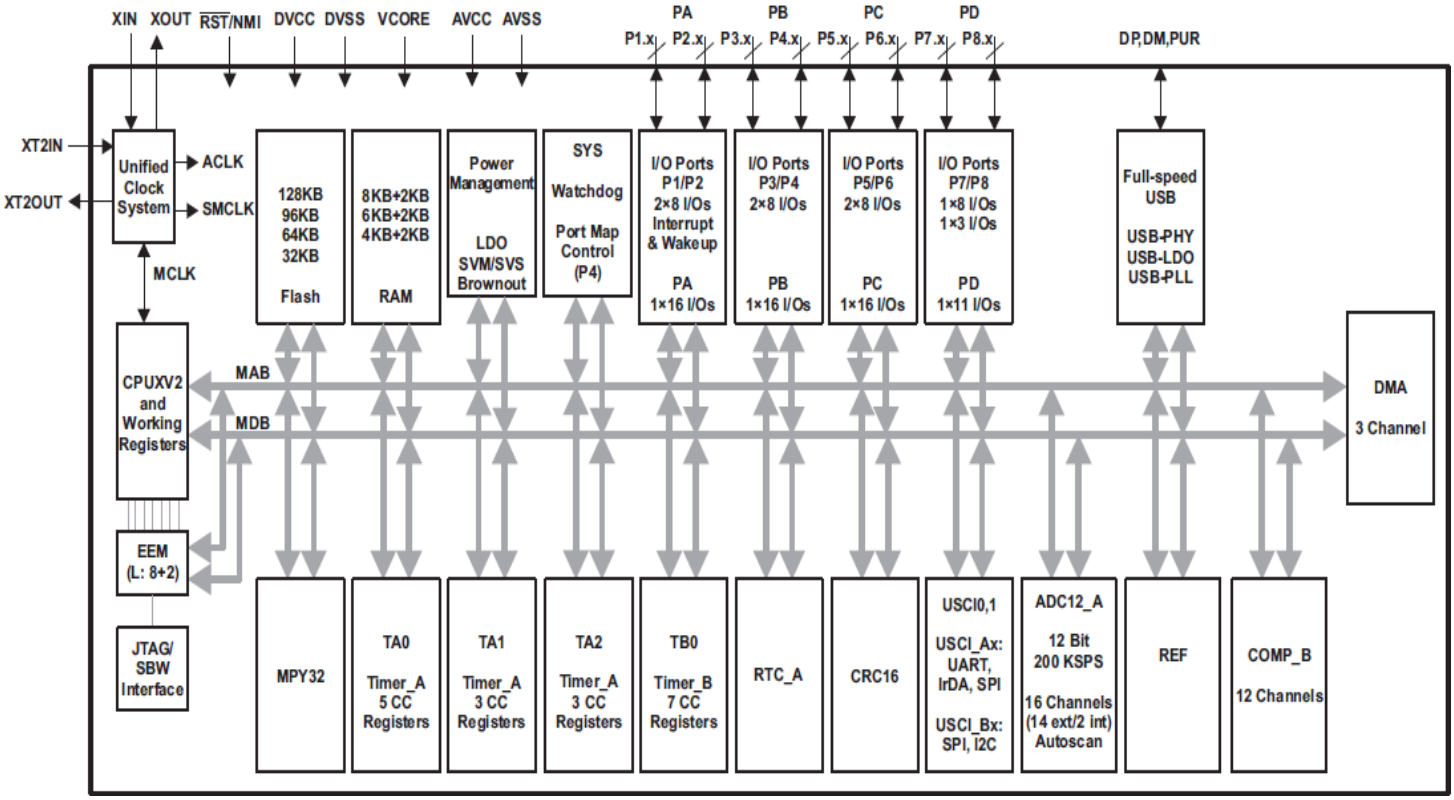
MSP430x4xx:

- Can drive LCDs with up to 160 segments. Many of them are ASSPs (application-specific standard product), but there are general-purpose devices as well. Their packages have 48–113 pins, many of which are needed for the LCD.

MSP430x5xx:

- It is Next Generation of MSP430 Family
- Advanced Ultra Low Power features
- Increased Performance, Functionality and ease-of-use
- Significantly longer battery life
- It contains almost all Peripherals like PORTS (P1-P8), ADC, DAC, TIMER_A0-2 & B0, DMA, COMPARATOR, USCI: UART, SPI, I2C IRDA, USB etc.
- Lowest Active Current/MHz:
  - <200uA/MHz

MSP430x5xx Series Block Diagram:



CPUX:

The MSP430X CPU is RISC architecture with 51 instructions and 7 addressing modes. It is integrated with 16 registers (each 20-bit wide except SR) that provide reduced instruction execution time. The register-to-register operation execution time is one cycle of the CPU clock. Peripherals are connected to the CPU using data, address, and control buses, and can be handled with all instructions. It has 20-bit address bus allows direct access and branching throughout the entire memory range without paging.

JTAG (Joint Test Action Group):

The MSP430 family supports the standard JTAG interface which requires four signals for sending and receiving data. The JTAG signals are shared with general-purpose I/O. It is used to program and debug the device.

SBW (Spy-Bi-Wire) Interface:

In addition to the standard JTAG interface, the MSP430 family supports the two wire Spy-Bi-Wire interface. Spy-Bi-Wire can be used to interface with MSP430 development tools and device programmers.

Flash Memory:

The flash memory can be programmed through the JTAG port, Spy-Bi-Wire (SBW), the BSL, or in-system by the CPU. The CPU can perform single-byte, single-word, and long-word writes to the flash memory.

The RAM is made up of n sectors. Each sector can be completely powered down to save leakage; however; all data is lost.

RAM has 5 sectors. The size of a each sector is 2KB. In that 5 sectors one is for USB & RAM (Both) and remaining 4 sectors only for RAM.

Peripherals:

Peripherals are connected to the CPUX through data, address, and control buses. Peripherals can be handled using all instructions.

On-Chip Peripherals (Analog and Digital):

- Digital I/O PORTs (GPIO)
- Port Mapping Controller
- Power Management Module (PMM)
- Hardware Multiplier (MPY32)
- Real-Time Clock (RTC_A)
- Watchdog Timer (WDT_A)
- System Module (SYS)
- DMA Controller
- Universal Serial Communication Interface (USCI: UART Mode, SPI Mode, I2C Mode)
- Timers (TA0, TA1, TA2, TB0)
- Comparator_B
- Analog to Digital Convertor (ADC12_A)
- Cyclic Redundancy Check (CRC16)
- Universal Serial Bus (USB)
- Embedded Emulation Module (EEM)

Digital I/O PORTs:

- There are up to eight 8-I/O ports P1- P8 each Port is 8 bit wide
- All individual I/O bits are independently programmable.
- Any combination of input, output, and interrupt conditions is possible.
- Pull-up or Pull-down on all ports is programmable.
- Read and write access to port-control registers is supported by all instructions.
- Ports can be accessed byte-wise (P1 through P8) or word-wise in pairs (PA through PD).
- Independent input and output data registers

Port Mapping Controller:

The port mapping controller allows the flexible and reconfigurable mapping of digital functions to port P4.

Power Management Module (PMM):

- The PMM includes an integrated voltage regulator that supplies the core voltage to the device and contains programmable output levels to provide for power optimization.
- The PMM also includes supply voltage supervisor (SVS) and supply voltage monitoring (SVM) circuitry, as well as brownout protection.
- The brownout circuit is implemented to provide the proper internal reset signal to the device during power on and power off.
- The SVS and SVM circuitry detects if the supply voltage drops below a user-selectable level and supports both supply voltage supervision (SVS) (the device is automatically reset) and supply voltage monitoring (SVM) (the device is not automatically reset).

Hardware Multiplier:

- The multiplication operation is supported by a dedicated peripheral module. The module performs operations with 32-, 24-, 16-, and 8-bit operands. The module supports signed and unsigned multiplication as well as signed and unsigned multiply-and-accumulate operations.

Real-Time Clock (RTC_A):

- The RTC_A module can be used as a general-purpose 32-bit counter (counter mode) or as an integrated real-time clock (RTC) (calendar mode).
- In counter mode, the RTC_A also includes two independent 8-bit timers that can be cascaded to form a 16-bit timer/counter. Both timers can be read and written by software.
- Calendar mode integrates an internal calendar which compensates for months with less than 31 days and includes leap year correction. The RTC_A also supports flexible alarm functions and offset calibration hardware.

Watchdog Timer (WDT_A):

The primary function of the WDT_A module is to perform a controlled system restart after a software problem occurs. If the selected time interval expires, a system reset is generated. If the watchdog function is not needed in an application, the module can be configured as an interval timer and can generate interrupts at selected time intervals.

System Module (SYS):

The SYS module handles many of the system functions within the device.

These include power-on reset and power-up clear handling, NMI source selection and management, reset interrupt vector generators, bootstrap loader entry mechanisms, and configuration management.

DMA Controller:

The DMA controller allows movement of data from one memory address to another without CPU intervention.

Universal Serial Communication Interface (USCI: UART Mode, SPI Mode, I2C Mode):

The USCI modules are used for serial data communication. The USCI module supports synchronous communication protocols such as SPI (3-pin or 4-pin) and I2C, and asynchronous communication protocols such as UART, enhanced UART with automatic baud rate detection, and IrDA.

Timers (TA0, TA1, TA2, TB0):

Timers are 16-bit timer and counter (Timer_A/B type) with 5/3/3/7 capture/compare registers. It can support multiple capture/compare registers, PWM outputs, and interval timing. It also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Comparator_B:

The primary function of the Comparator_B module is to support precision slope analog-to-digital conversions, battery voltage supervision, and monitoring of external analog signals.

ADC12_A:

The ADC12_A module supports fast 12-bit analog-to-digital conversions. It has 16 independent channels to be convert and store without any CPU intervention.

CRC16:

A Cyclic Redundancy Check (CRC) is an error-detecting code commonly used in digital networks and storage devices for data errors checking purpose

REF Voltage Reference:

The REF voltage module generates the Reference Voltage which is used by ADC as a reference mark or point to convert analog voltage from 0v to REF voltage.

Universal Serial Bus (USB):

The USB module is a fully integrated USB interface that is compliant with the USB 2.0 specification. The module supports full-speed operation of control, interrupt, and bulk transfers. The module includes an integrated LDO, PHY, and PLL.

Embedded Emulation Module (EEM):

The EEM supports real-time in-system debugging.

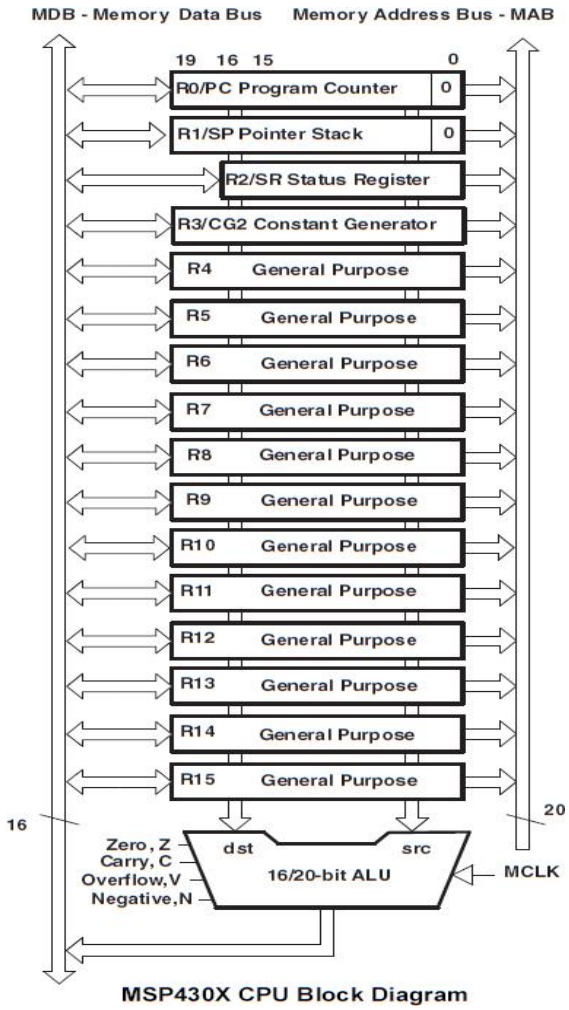
Features of EEM:

- Eight hardware triggers or breakpoints on memory access
- Two hardware triggers or breakpoints on CPU register write access
- Up to 10 hardware triggers can be combined to form complex triggers or breakpoints

- Two cycle counters
- Sequencer
- State storage
- Clock control on module level

MSP430X CPU (CPUX) – Features:

The MSP430X CPU features include:



MSP430X CPU Block Diagram

- RISC architecture
- Orthogonal architecture
- Full register access including program counter, status register and stack pointer
- Single-cycle register operations
- Large register file reduces fetches to memory
- It has 51 instructions
- 20-bit address bus allows direct access and branching throughout the entire memory range without paging
- 16-bit data bus allows direct manipulation of word-wide arguments
- Constant generator provides the six most often used immediate values and reduces code *size*
- *Direct memory-to-memory transfers without intermediate register* holding
- Byte, word, and 20-bit address-word addressing.

➢ An **orthogonal** instruction set is an instruction set architecture where all instruction types can use all addressing modes. It is "orthogonal" in the sense that the instruction type and the addressing mode vary independently.

Registers of MSP430 CPUX:

The CPUX of MSP 430 includes a 16/20-bit ALU and a set of 16 Registers R0 – R15. In these registers four are special Purpose and 12 are general purpose registers. All the registers can be addressed in the same way.

The special Purpose Registers are:

PC (Program Counter), SP (Stack Pointer), SR (Status Register), CGx (Constant Generator)

The MSP430 CPU includes an arithmetic logic unit (ALU) that handles addition, subtraction, comparison and logical (AND, XOR) operations. ALU operations can affect the overflow, zero, negative, and carry flags in the status register.

The following figure shows the register organization of MSP430 CPUX.

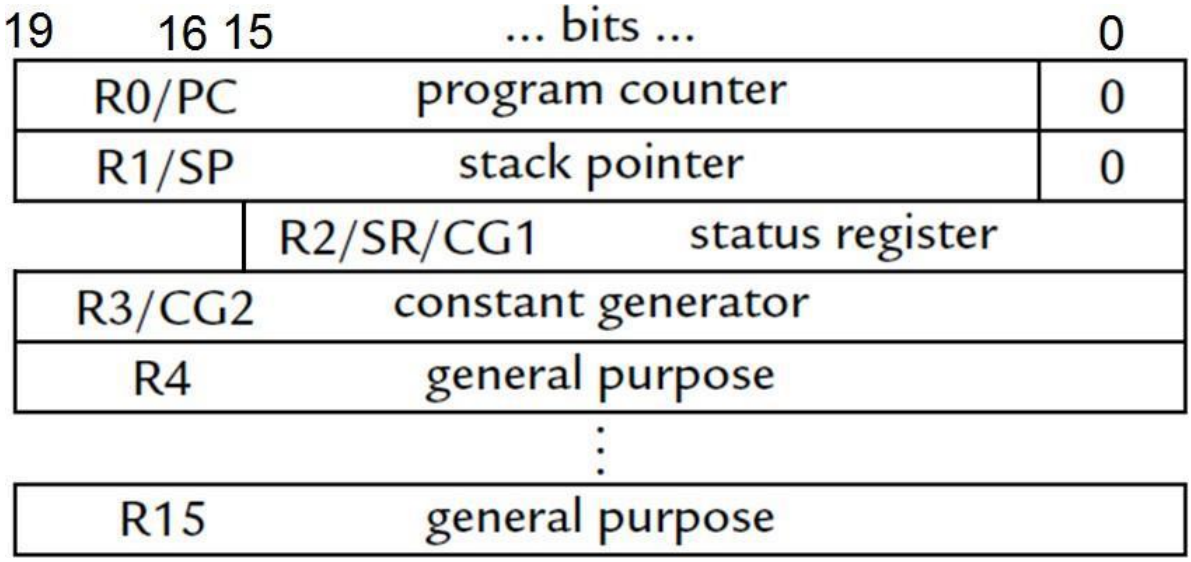


Figure: Registers in the CPUX of MSP430x5xx

R0: Program Counter (PC):

The 20-bit PC (PC/R0) points to the next instruction to be executed. Each instruction uses an evennumber of bytes (2, 4, 6, or 8 bytes), and the PC is incremented accordingly. Instruction accesses areperformed on word boundaries, and the PC is aligned to even addresses.Following Figure shows the PC structure.

Subroutines and interrupts also modify the PC but in these cases the previous value (Next line of current instruction which is executing) is saved on the stack and

restored later.

Figure: Program Counter

The PC is automatically stored on the stack with CALL (or CALLA) instructions and during an interruptservice routine. Following Figure shows the storage of the PC with the return address after a CALLA instruction.A CALL instruction stores only bits 15:0 of the PC.

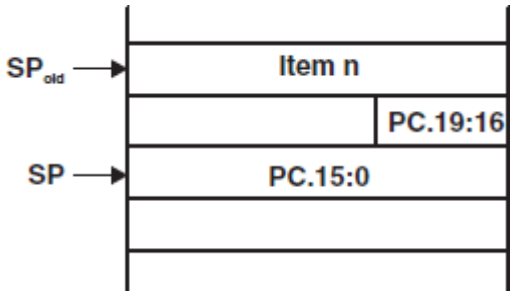
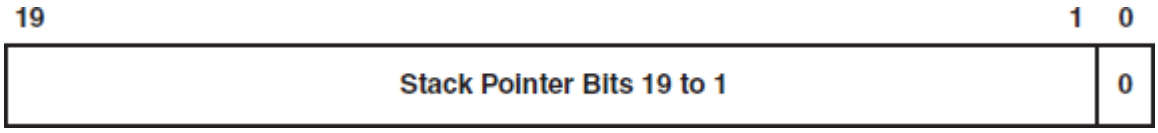


Figure: PC Storage on the Stack for CALLA

The RETA instruction restores bits 19:0 of the PC and adds 4 to the stack pointer (SP). The RET instruction restores bits 15:0 to the PC and adds 2 to the SP.

R1: Stack Pointer (SP):

The 20-bit SP (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a predecrement, postincrement scheme. In addition, the SP can be used by software with all instructions and addressing modes.



Following Figure shows the SP. The SP is initialized into RAM by the user,and is always aligned to even addresses.

Figure: Stack Pointer

The Following Figure shows the stack usage.

```
PUSH    #0123h          ; Put 0123h on stack
POP     R8              ; R8 = 0123h
```

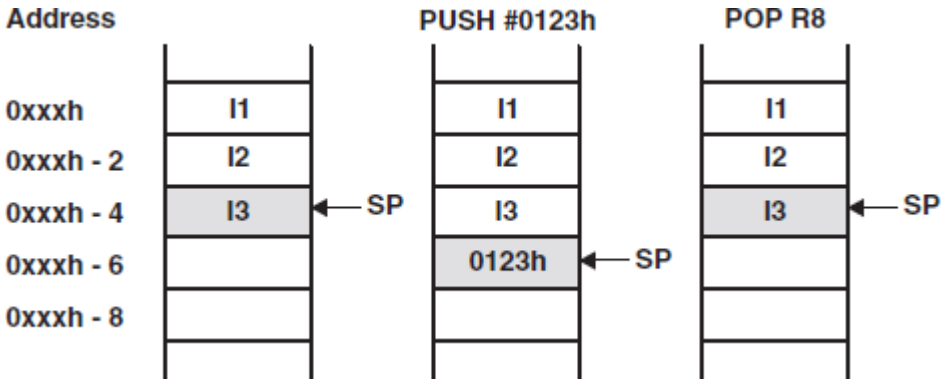
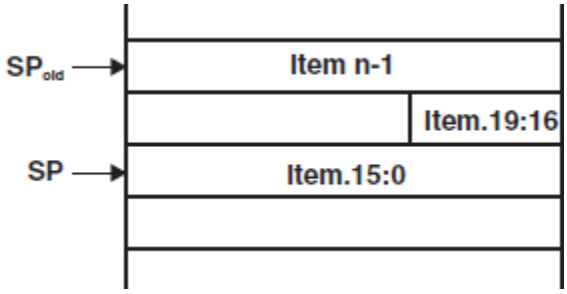


Figure: Stack Usage



The following Figure shows the stack usage when 20-bit address words arepushed.

Figure: PUSHX.A Format on the Stack

Note: For programs written in C, the compiler initializes the stack automatically as part of the startup code, which runs silently before the program starts, but you must initialize SP yourself in assembly language.

R2: Status Register (SR):

The 16-bit SR (SR/R2), used as a source or destination register, can only be used in register modeaddressed with word instructions. The remaining combinations of addressing modes are used to supportthe constant generator. Figure 4-9 shows the SR bits. Do not write 20-bit values to the SR. Unpredictable operation can result.

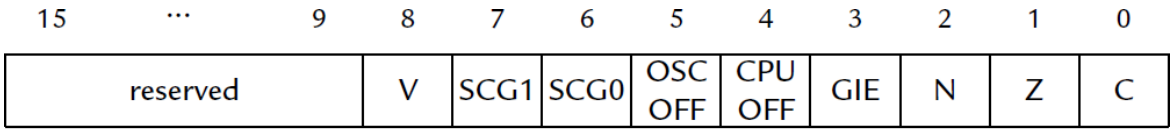


Figure: Individual bits in the status register

The reserved bits are not used in the MSP430.

Table: SR Bit Description

Bit	Description	
Reserved	Reserved	
V	Overflow.Thisbitissetwhentheresultofanarithmeticoperationoverflowsthesigned-variablerange.	
	ADD(.B), ADDX(.B,.A), ADDC(.B), ADDCX(.B.A), ADDA	Set when: positive+ positive=negative negative+ negative=positive otherwise reset
	SUB(.B), SUBX(.B,.A), SUBC(.B), SUBCX(.B,.A), SUBA, CMP(.B), CMPX(.B,.A), CMPA	Set when: positive– negative=negative negative–positive=positive otherwise reset
SCG1	Systemclockgenerator1.Thisbitmaybetoenable/disablefunctionsintheclocksystemdependingonthedevice family; for example, DCO bias enable/disable	
SCG0	Systemclockgenerator0.Thisbitmaybeusedtoenable/disablefunctionsintheclocksystemdependingonthe device family; for example, FLL disable/enable	
OSCOFF	Oscillator off. This bit, when set, turns off the LFXT1 crystal oscillator when LFXT1CLK is not used for MCLK or SMCLK.	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative.Thisbitissetwhentheresultofanoperationisnegativeandclearedwhentheresultispositive.	

Z	Zero.Thisbitissetwhentheresultofanoperationis0andclearedwhentheresultisnot0.
C	Carry.Thisbitissetwhentheresultofanoperationproducedacarryandclearedwhennocarryoccurred.

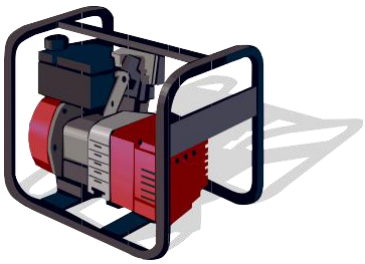
R2/R3: Constant Generator Registers (CG1/CG2):

Six commonly-used constants are generated with the constant generator registers R2

Table: Values of Constant

Register	As	Constant	Remarks
R2	00	–	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4,bitprocessing
R2	11	00008h	+8,bitprocessing
R3	00	00000h	0,wordprocessing
R3	01	00001h	+1
R3	10	00002h	+2,bitprocessing
R3	11	FFh,FFFFh,FFFFFh	–1,wordprocessing

Generators CG1, CG2



```
4314            mov.w #0002h, R4   ; With CG
4034 1234       mov.w #1234h, R4   ; Without CG
```

- Constant (Immediate) values -1,0,1,2,4,8 generated in hardware

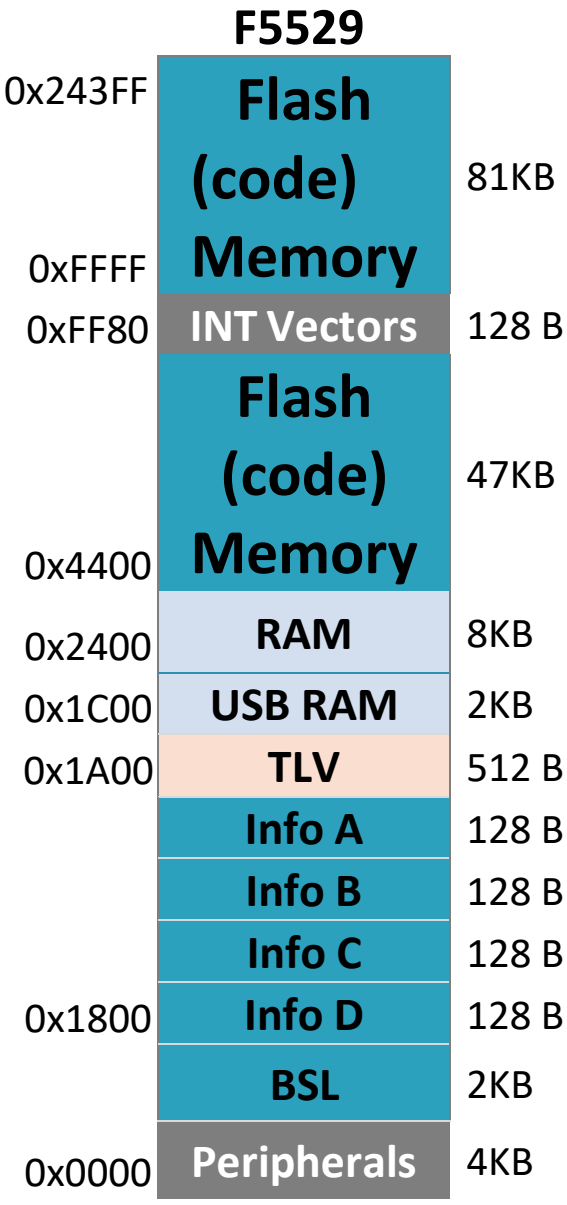
The constant generator advantages are:

- No special instructions required
- No additional code word for the six constants
- No code memory access required to retrieve the constant
- Reduces code size and cycles
- Completely Automatic

R4 - R15: General–Purpose Registers:

The remaining 12 registers R4–R15 have no dedicated purpose and may be used as general working registers. They may be used for either data or addresses because both are 16-bit values, which simplify the operation significantly.

Address Space (Memory Organization):



MEMORY TYPE		MSP430F5529 MSP430F5528 MSP430F5519
Unused Memory		0FFFFFh–024400h
Interrupt Vector	Size	128 B 00FFFFh–00FF80h
Flash (code) Memory	Total Size	128KB
	Bank D	32KB 0243FFh–01C400h
	Bank C	32KB 01C3FFh–014400h
	Bank B	32KB 0143FFh–00C400h
	Bank A	32KB 00C3FFh–004400h
RAM	Sector 3	2KB 0043FFh–003C00h
	Sector 2	2KB 003BFFh–003400h
	Sector 1	2KB 0033FFh–002C00h
	Sector 0	2KB 002BFFh–002400h
USB RAM #	Sector 7	2KB 0023FFh–001C00h
Information Memory (flash)	Info A	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h
Bootstrap loader (BSL) memory (flash)	BSL 3	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h
Peripherals	Size	4KB 000FFFh–0h

1MB

- **Info** – Information Memory (flash)
- **TLV** – Contents of the Device Descriptor **Tag Length Value** (TLV)
- **BSL**– Bootstrap Loader Memory (flash)

Sample Embedded System onMSP430 Microcontroller:

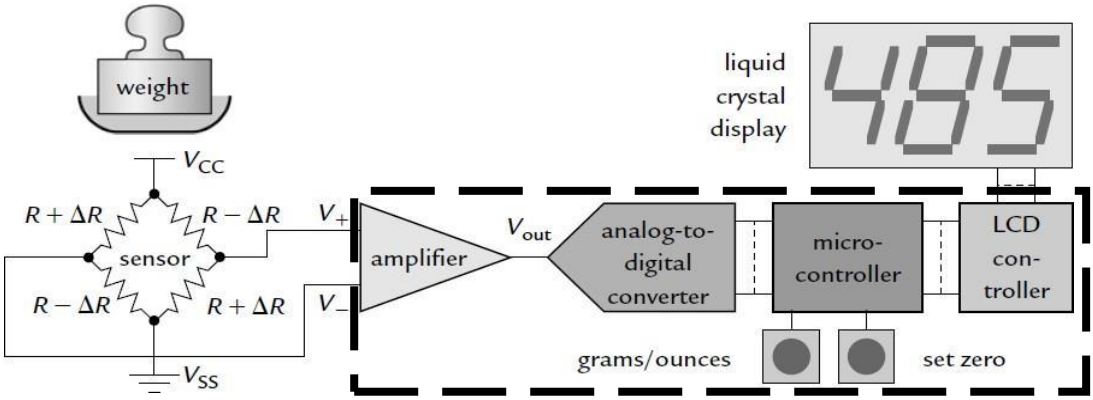


Fig: Weighing Machine with a liquid crystal display, broken down intoindividual functions.

*V*out = *A*(*V*+ − *V*−), where *A* is the gain

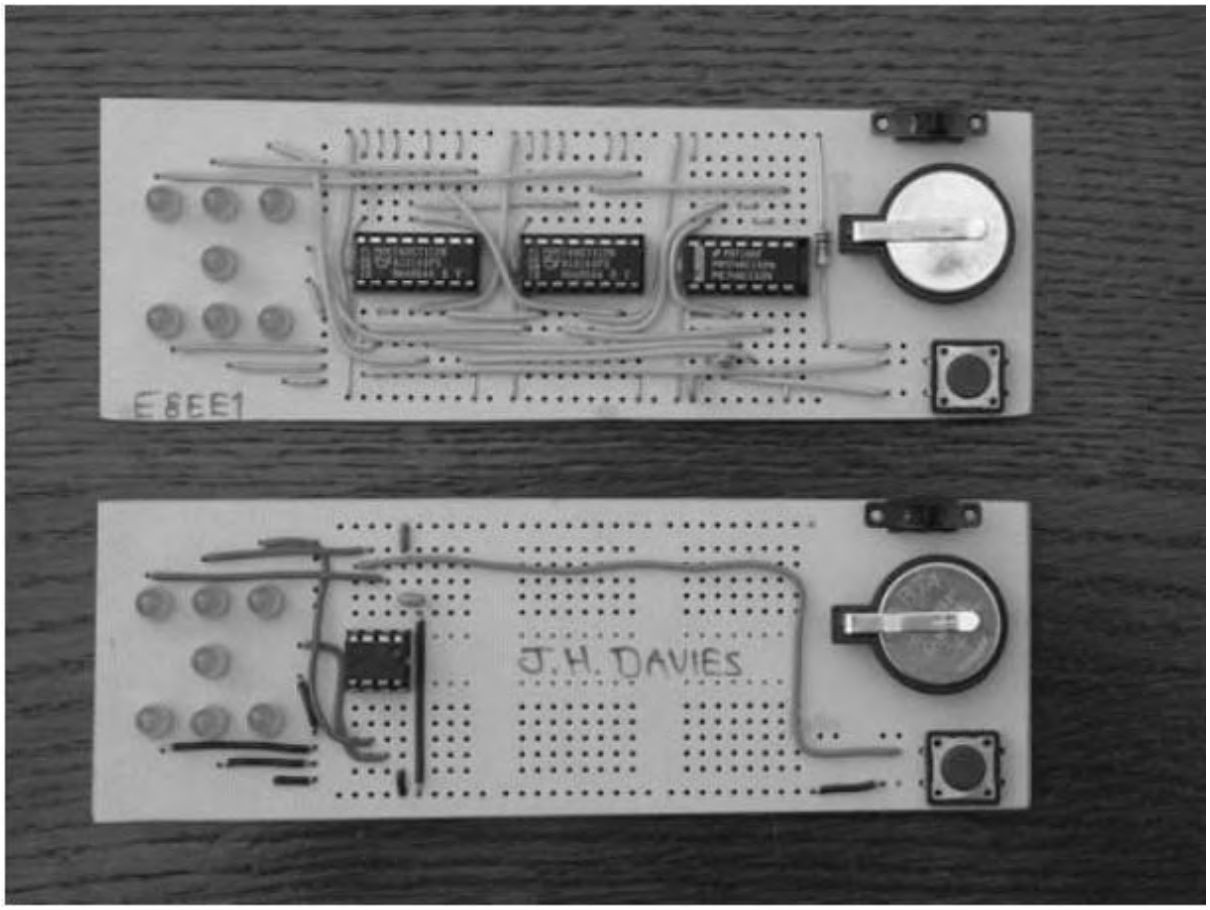


Figure: Electronic Dice built using (top) *JK* flip-flops and gates and (bottom) an eight-pin microcontroller.

INTRODUCTION to MSP430 MICROCONTROLLER

Features of MSP430

These are some features of MSP430.

It is available in a 20 pin plastic small outline widebody package.

Its operating voltage range is 2.5v to 5.5 v. Its active mode is 330 µA at 1 MHz, 3 V.

Its stands by mode are 1.5 µA. It's off mode (Ram Retention) is 0.1 µA.

It has serial onboard programming.

Applications of MSP430

These are some applications of MSP430.

It is used in Factory Control & Automation Applications.

It is used in Buildings & Home Automation systems.

It is used in Grid Infrastructure & Metering networks.

It is used in Portable Test & Measurement Equipment.

It is used in Health, Medical & Fitness Applications.

It also used in Consumer Electronics.

So, friends that were all about MSP430, if you have any question about it please ask in comments. I will resolve your problems. Will meet you guys in the next tutorial. Till then take care and have fun.

Basics of MSP430

This module provides features of 16-bit registers, 16-bit RISC CPU and constant generators.

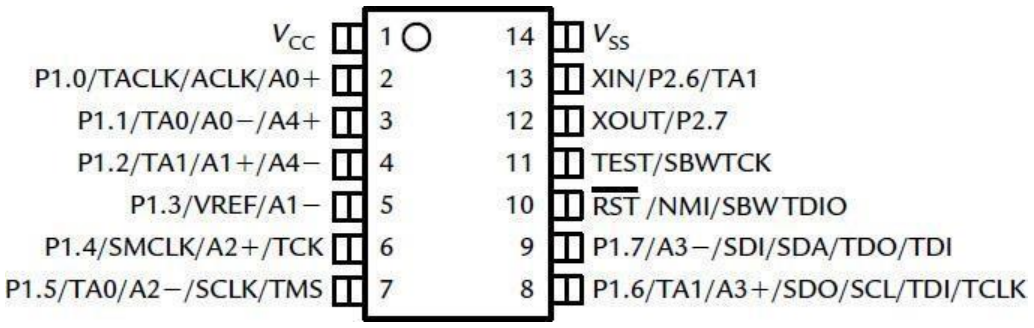
This module has five low power modes which enhance battery life in portable measurement applications.

This module changes its state from low power mode to active mode within 6us, by a digitally controlled oscillator (DCO).

The MSP430x11x series consists of a 16-bit timer and fourteen input and output pinouts.

Now let's have a look at MSP430 Pinout.

Pin diagram of the MSP430F2003 and F2013



1. VCC and VSS are the supply voltage and ground for the whole device (the analog and digital supplies are separate in the 16-pin package).
2. P1.0–P1.7, P2.6, and P2.7 are for digital input and output, grouped into ports P1 and P2.
3. TACLK, TA0, and TA1 are associated with Timer_A; TACLK can be used as the clock input to the timer, while TA0 and TA1 can be either inputs or outputs. These can be used on several pins because of the importance of the timer.
4. A0−, A0+, and so on, up to A4±, are inputs to the analog-to-digital converter. It has four differential channels, each of which has negative and positive inputs. VREF is the reference voltage for the

converter.

5. ACLK and SMCLK are outputs for the microcontroller's clock signals. These can be used to supply a clock to external components or for diagnostic purposes.
6. SCLK, SDO, and SCL are used for the universal serial interface, which communicates with external devices using the serial peripheral interface (SPI) or inter-integrated circuit (I2C) bus.
7. XIN and XOUT are the connections for a crystal, which can be used to provide an accurate, stable clock frequency.
8. RST is an active low reset signal. *Active low* means that it remains high near *V*CC for normal operation and is brought low near *V*SS to reset the chip. Alternative notations to show the active low nature are _RST and /RST.
9. NMI is the non-maskable interrupt input, which allows an external signal to interrupt the normal operation of the program.
10. TCK, TMS, TCLK, TDI, TDO, and TEST form the full JTAG interface, used to program and debug the device.
11. SBWTDIO and SBWTCK provide the Spy-Bi-Wire interface, an alternative to the usual JTAG connection that saves pins.

MSP430 Pinout

- There are main twenty pinouts of MSP430, which are described below.

Pin#	Type	Parameters
Pin#13	P1.0/TACLK	It is general-purpose digital I/O pin/Timer_A, clock signal TACLK input.
Pin#14	P1.1/TA0	It is general-purpose digital I/O pin/Timer_A, Capture: CCI0A input, Compare: Out0 output.
Pin#15	P1.2/TA1	It is general-purpose digital I/O pin/Timer_A, Capture: CCI1A input, Compare: Out1 output.
Pin#16	P1.3/TA2	It is general-purpose digital I/O pin/Timer_A, Capture: CCI2A input, Compare: Out2 output.
Pin#17	P1.4/SMCLK/TCK	It is general-purpose digital I/O pin/SMCLK signal output/Test clock, an input terminal for device programming and test.
Pin#18	P1.5/TA0/TMS	It is general-purpose digital I/O pin/Timer_A, Compare: Out0 output/test mode select, an input terminal for device programming and test.
Pin#19	P1.6/TA1/TDI	It is general-purpose digital I/O pin/Timer_A, Compare: Out1 output/test data input terminal.
Pin#20	P1.7/TA2/TDO/TDI	It is general-purpose digital I/O pin/Timer_A, Compare: Out2 output/test data output terminal or data input during programming.
Pin#8	P2.0/ACLK	It is general-purpose digital I/O pin/ACLK output.

Pin#9 P2.1/INCLK	It is general-purpose digital I/O pin/Timer_A, a clock signal at INCLK.
Pin#10P2.2/TA0	It is general-purpose digital I/O pin/Timer_A, Capture: CCI0B input, Compare: Out0 output.
Pin#11P2.3/TA1	It is general-purpose digital I/O pin/Timer_A, Capture: CCI1B input, Compare: Out1 output.
Pin#12P2.4/TA2	It is general-purpose digital I/O pin/Timer_A, Compare Out2 output.
Pin#3 P2.5/ROSC	It is general-purpose digital I/O pin/Input for an external resistor that defines the DCO nominal frequency.
Pin#7 RST/NMI	It is Reset or non-maskable interrupt input.
Pin#1 TEST/VPP	It is selected test mode for JTAG pins on Port1/programming voltage input during EPROM programming.
Pin#2 VCC	It is a Supply voltage.
Pin#4 VSS	It is Ground reference.
Pin#6 XIN	It is an Input terminal of the crystal oscillator.
Pin#5 XOUT/TCLK	The output terminal of a crystal oscillator or test clock input.

Architecture of MSP 430

Block diagram of the MSP430F2003 and F2013, taken from data sheet.

The main features of the MSP RISC CPU architecture are,

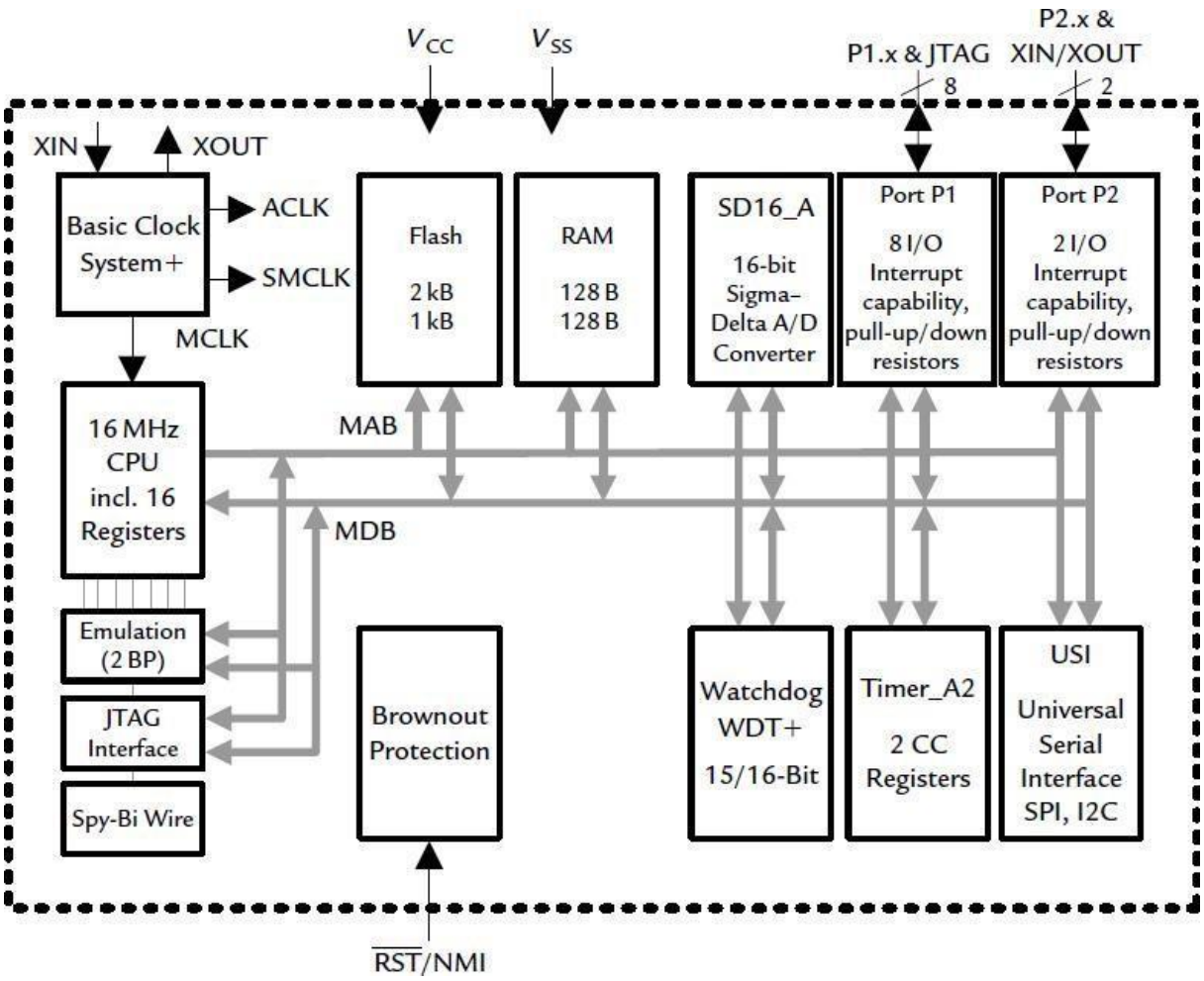
1. On the left is the CPU and its supporting hardware, including the clock generator. The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging
2. Clock generator generates up to three different clocks (MCLK, ACLK & SMCLK) using four different sources (VCO, DCO, LFXT1 and XT2).
3. The main blocks are linked by the *memory address bus* (MAB) and *memory data bus* (MDB).
4. These devices have flash memory, 1KB in the F2003 or 2KB in the F2013, and 128 bytes of RAM.
5. Six blocks are shown for peripheral functions (there are many more in larger devices).
    a. Input/output ports,

b. Timer_A,

c. Watchdog timer (resets the processor if program becomes stuck in the infinite loop).

d. The universal serial interface (USI) (SPI, I²C, RS232, USB, CAN etc…)

e. Sigma–delta analog-to-digital converter (SD16_A)



6. The brownout protection comes into action if the supply voltage drops to a dangerous level. Most devices include this but not some of the MSP430x1xx family.
7. There are ground and power supply connections. Ground is labeled *V*SS and is taken to define 0V. The supply connection is *V*CC which is mostly in the range of 1.8–3.6V.

REGISTERS OF MSP 430

MSP 430 has sixteen 16-bit registers. These registers do not have address in the main memory map. First four registers have dedicated alternate functions and the remaining 12 registers are used as working registers for general purposes.

R0/PC (PROGRAM COUNTER)
R1/SP (STACK POINTER)
R2/SR (STATUS REGISTER)
R3/CG (CONSTANT GENERATOR)
R4 (GENERAL PURPOSE)
R5 (GENERAL PURPOSE)
R6 (GENERAL PURPOSE)
R7 (GENERAL PURPOSE)

R8 (GENERAL PURPOSE)
R9 (GENERAL PURPOSE)
R10 (GENERAL PURPOSE)
R11 (GENERAL PURPOSE)
R12 (GENERAL PURPOSE)
R13 (GENERAL PURPOSE)
R14 (GENERAL PURPOSE)
R15 (GENERAL PURPOSE)

Program counter, PC: This contains the address of the next instruction to be executed

Stack pointer, SP: MSP430 uses the top (high addresses) of the main RAM as stack memory. The stack pointer holds the address of the most recently added word and is automatically adjusted as the stack grows downward in memory or shrinks upward.

Status register, SR: This contains a set of flags (single bits), whose functions fall into three categories.
The most commonly used flags are C, Z, N, and V, which give information about the result of the last arithmetic or logical operation. The Z flag is set if the result was zero and cleared if it was nonzero, for instance. Setting the GIE bit enables maskable interrupts. The final group of bits is CPUOFF, OSCOFF, SCG0, and SCG1, which control the mode of operation of the MCU. All systems are active when all bits are clear.

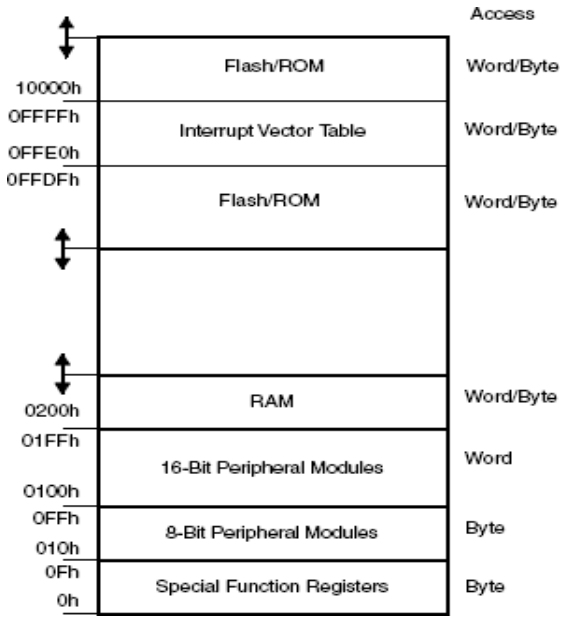
Constant generator: This provides the six most frequently used values so that they need not be fetched from memory whenever they are needed. It uses both R2 and R3 to provide a range of useful values by exploiting the CPU's addressing modes.

General purpose registers: The remaining 12 registers, R4–R15, are general working registers. They may be used for either data or addresses because both are 16-bit values, which simplify the operation significantly.

COMPILER FRIENDLY FEATURES

MSP430 stems from its recent introduction is that it is designed with compilers in mind. Most small microcontrollers are now programmed in C, and it is important that a compiler can produce compact, efficient code. The MSP430 has 16 registers in its CPU, which enhances efficiency because they can be used for local variables, parameters passed to subroutines, and either addresses or data. This is a typical feature of a RISC, but unlike a "pure" RISC, it can perform arithmetic directly on values in main memory. Microcontrollers typically spend much of their time on such operations.

MEMORY ADDRESS SPACE



- The MSP430 von Neumann architecture has one address space shared with
  - special function registers (SFRs),
  - peripherals,
  - RAM, and
  - Flash/ROM memory
- Code access are always performed on even addresses.
- Data can be accessed as bytes or words.
- The addressable memory space is 64 KB

Flash/ROM

- The start address depends on the amount of Flash/ROM present and varies by device.
- The end address is 0FFFFh for devices with less than 60kB of Flash/ROM; otherwise, it is device dependent.
- Flash can be used for both code and data.
- Word or byte tables can be stored and used without the need to copy the tables to RAM before using them.
- The interrupt vector table is mapped into the upper 16 words of address space, with the highest priority interrupt vector at address (0FFFEh).

RAM

- RAM starts at 0200h.
- End address depends on the amount of RAM present and varies by device.
- RAM can be used for both code and data.

Peripheral Modules

- 0100 to 01FFh is reserved for 16-bit peripheral modules.
- Accessed with word instructions.
- If Byte instructions are used ,then high byte of the result is always 0.
- 010h to 0FFh is reserved for 8-bit peripheral modules.
- These modules should be accessed with byte instructions.
- Accessed using word instructions results in unpredictable data in the high byte.
- If word data is written to a byte module only the low byte is written into the peripheral register, ignoring the high byte.

SFRs

- Peripheral functions are configured in the SFRs.
- Located in the lower 16 bytes of the address space and are organized by byte.
- SFRs must be accessed using byte instructions only

ADDRESSING MODES

1. **Register addressing mode.** The address is formed by adding a constant base address to the contents of a CPU register; the value in the register is not changed.

   Eg: **MOV R10, R11**

   Length: One or two words

   Operation: Move the content of R10 to R11. R10 is not affected.

Before:	**After:**
R10 - 0A023h	R10 - 0A023h
R11 - 0FA15h	R11 - 0A023h
PC - PC old	PC - PC old + 2

2. **Indexed addressing mode.** In this case the program counter PC is used as the base address, so the constant is the offset to the data from the PC.

   Eg: MOV 2(R5),6(R6)

   Length: 2 or 3 words

   Operation: Move the contents of the source address (contents of R5 + 2) to the destination address (contents of R6 + 6).

3. **Symbolic Mode (PC Relative)**
   In this case the program counter PC is used as the base address, so the constant is the offset to the data from the PC
   Eg: **MOV EDE,TONI**
   Length: Two or three words
   Operation: Move the contents of the source address EDE (contents of PC + X) to the destination address TONI (contents of PC + Y).

4. **Absolute Mode:** The constant in this form of indexed addressing is the absolute address of the data. This is already the complete address required so it should be added to a register that contains 0. Absolute addressing is shown by the prefix & and should be used for special function and peripheral registers, whose addresses are fixed in the memory map.
   Eg: mov.b &P1IN ,R6 ; copies the port 1 input register into register R6

5. **Indirect Register Mode:**
   Eg: MOV @R10,0(R11)
   Operation: Move the contents of the source whose address is in (R10) to the destination address (R11). Indirect addressing cannot be used for the destination.
6. **Indirect Auto increment Mode:** This is available only for the source and is shown by the symbol @ in front of a register with a + sign after it, such as @R5+. It uses the value in R5 as a pointer and automatically increments it afterward by 1 if a byte has been fetched or by 2 for a word.
   Eg: MOV @R10+,0(R11)

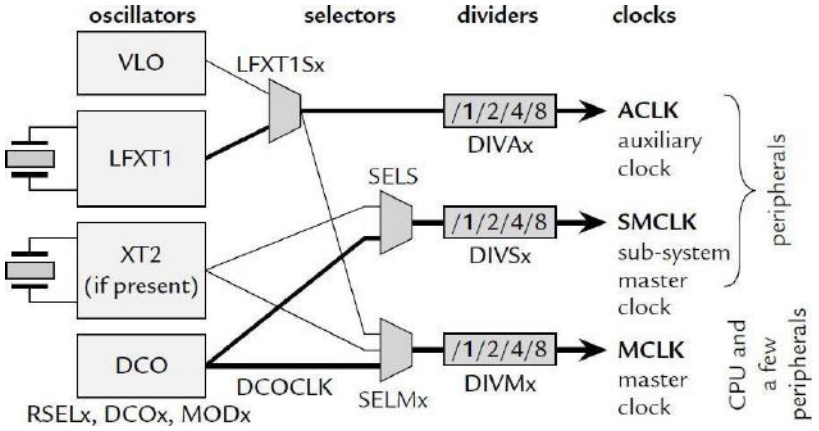
7. **Immediate Mode**
   Eg: MOV #45h,TONI: Operation: Move the immediate constant 45h, which is contained in the word following the instruction, to destination address TONI. When fetching the source, the program counter points to the word following the instruction and moves the contents to the destination.

CLOCK SYSTEM

Figure below shows a simplified diagram of the Basic Clock Module+ (BCM+) for the MSP430F2xx family. The clock module provides three outputs:

- Master clock, MCLK is used by the CPU and a few peripherals.
- Sub-system master clock, SMCLK is distributed to peripherals.
- Auxiliary clock, ACLK is also distributed to peripherals.

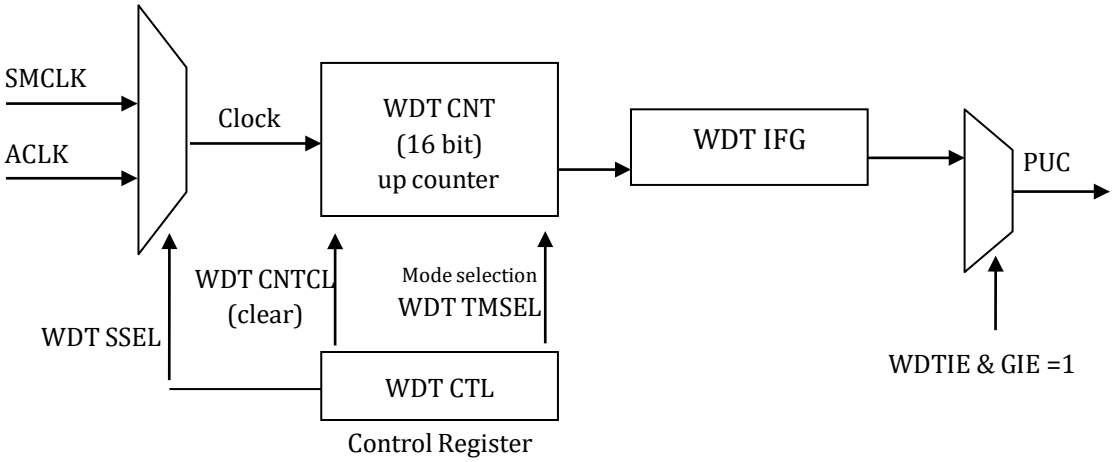
Most peripherals can choose either SMCLK, which is often the same as MCLK and in the megahertz range, or ACLK, which is typically much slower and usually 32 KHz. A few peripherals, such as analog-to-digital converters, can also use MCLK and some, such as timers, have their own clock inputs. The frequencies of all three clocks can be divided in the BCM+ as shown in figure.



Up to four sources are available for the clock, depending on the family and variant:
Low- or high-frequency crystal oscillator, LFXT1: Available in all devices. It is usually used with a low-frequency crystal (32 KHz) but can also run with a high-frequency crystal (typically a few MHz) in most devices. An external clock signal can be used instead of a crystal if it is important to synchronize the MSP430 with other devices in the system.

High-frequency crystal oscillator, XT2: Similar to LFXT1 except that it is restricted to high frequencies. It is available in only a few devices and LFXT1 (or VLO) is used instead if XT2 is missing.
Internal very low-power, low-frequency oscillator, VLO: Available in only the more recent MSP430F2xx devices. It provides an alternative to LFXT1 when the accuracy of a crystal is not needed.
Digitally controlled oscillator, DCO: Available in all devices and one of the highlights of the MSP430. It is basically a highly controllable RC oscillator that starts in less than 1µs in newer devices.

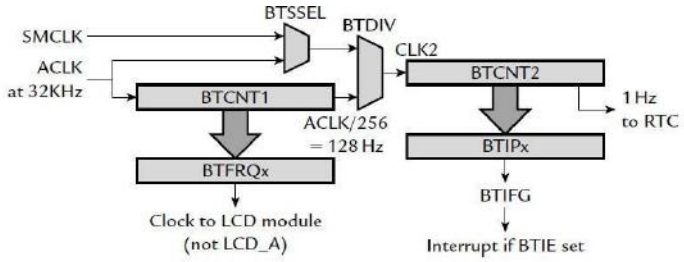
WATCH DOG TIMERS.
The main purpose of the watchdog timer is to protect the system against failure of the software, such as the program becoming trapped in an unintended, infinite loop. Watchdog counts up and resets the MSP430 when it reaches its limit. The code must therefore keep clearing the counter before the limit is reached to prevent a reset. The operation of the watchdog is controlled by the 16-bit register WDTCTL



The watchdog counter is a 16-bit register WDTCNT, which is not visible to the user. It is clocked from either SMCLK (default) or ACLK, according to the WDTSSEL bit. The watchdog is always active after the MSP430 has been reset. By default the clock is SMCLK, which is in turn derived from the DCO at about 1 MHz. The default period of the watchdog is the maximum value of 32,768 counts, which is therefore around 32 ms. We must clear, stop, or reconfigure the watchdog before this time has elapsed. If the watchdog is left running, the counter must be repeatedly cleared to prevent it counting up as far as its limit. This is done by setting the WDTCNTCL bit in WDTCTL. The watchdog timer sets the WDTIFG flag in the special function register IFG1. This is cleared by a power-on reset but its value is preserved during a PUC. Thus a program can check this bit to find out whether a reset arose from the watchdog.

BASIC TIMER.

Basic Timer1 is present in all MSP430xF4xx devices. It provides the clock for the LCD module and generates periodic interrupts. A simplified block diagram of basic timer is shown in figure below. Newer devices contain a real-time clock driven by a signal at 1Hz from Basic Timer1. The register BTCTL controls most of the functions of Basic Timer1 but there are also bits in the special function registers IFG2 and IE2 for interrupts.



Simplified block diagram of Basic Timer1.

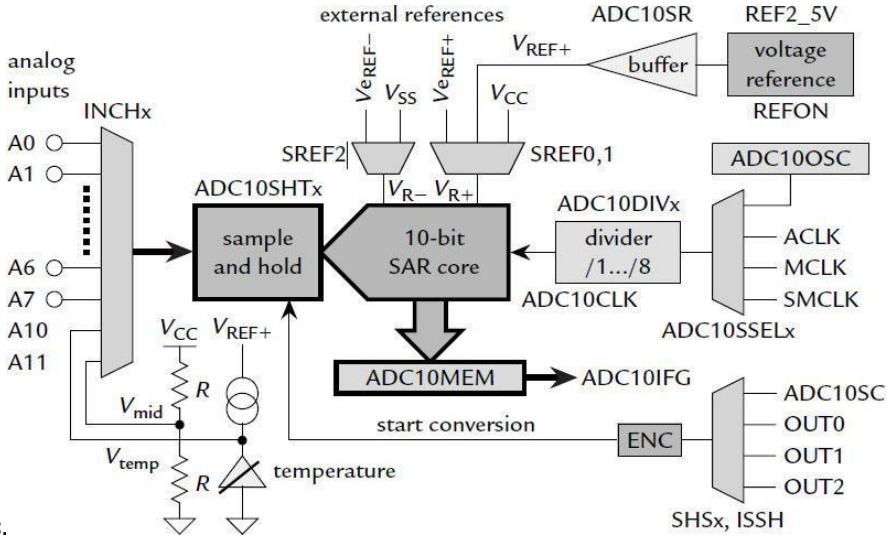
7	6	5	4	3	2	1	0
BTSSEL	BTHOLD	BTDIV	BTFRFQx		BTIPx		

The Basic Timer1 control register BTCTL.

REAL TIME CLOCK.

ADC10 SAR PERIPHERAL MODULE

Figure below shows a simplified block diagram of the ADC10 in the F20x2; there are more inputs in



larger devices.

The ADC10 module of the MSP430F2274 supports fast 10 bit analogue-to-digital conversions;
The module contains:

- **10-bit SAR core;** The ADC10ON bit enables the core and a flag ADC10BUSY is set while sampling and conversion is in progress. The result is written to ADC10MEM in a choice of two formats, selected with the ADC10DF bit.
- **Clock;** This can be taken from MCLK, SMCLK, ACLK, or the module's internal oscillator ADC10OSC, selected with the ADC10SSELx bits.
- **Sample-and-Hold Unit:** This is shown separately in the block diagram. The time is chosen with the ADC10SHTx bits, which allow 4, 8, 16, or 64 cycles of ADC10CLK.
- **Input Selection:** A multiplexer selects the input from eight external pins A0–A7 (more in larger MSP430s) and four internal connections.
- **Conversion Trigger;** A conversion can be triggered in two ways provided that the ENC bit is set. The first is by setting the ADC10SC bit from software (it clears again automatically).

DIGITAL I/O PORTS

There are 10 to 80 input/output pins on different devices in the current portfolio of MSP430s; the F20xx has one complete 8-pin port and 2 pins on a second port, while the largest devices have ten full ports. Almost all pins can be used either for digital input/output or for other functions and their operation must be configured when the device starts up.
Up to eight registers are associated with the digital input/output functions for each pin. Here are the registers for port P1 on a MSP430F2xx, which has the maximum number. Each pin can be configured and controlled individually; thus some pins can be digital inputs, some outputs, some used for analog functions, and so on.

- **Port P1 input, P1IN:** reading returns the logical values on the inputs if they are configured for digital input/output. This register is read-only and volatile. It does not need to be initialized because its contents are determined by the external signals.
- **Port P1 output, P1OUT:** writing sends the value to be driven to each pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in a buffer and appears on the pin if it is later switched to be an output. This register is not initialized and you should therefore write to P1OUT before configuring the pin for output.

- **Port P1 direction, P1DIR:** clearing a bit to 0 configures a pin as an input, which is the default in most cases. Writing a 1 switches the pin to become an output. This is for digital input and output; the register works differently if other functions are selected using P1SEL.
- **Port P1 resistor enable, P1REN:** setting a bit to 1 activates a pull-up or pull-down resistor on a pin. Pull-ups are often used to connect a switch to an input as in the section "Read Input from a Switch" on page 80. The resistors are inactive by default (0). When the resistor is enabled (1), the corresponding bit of the P1OUT register selects whether the resistor pulls the input up to VCC (1) or down to VSS (0).
- **Port P1 selection, P1SEL:** selects either digital input/output (0, default) or an alternative function (1). Further registers may be needed to choose the particular function.
- **Port P1 interrupt enable, P1IE:** enables interrupts when the value on an input pin changes. This feature is activated by setting appropriate bits of P1IE to 1. Interrupts are off (0) by default. The whole port shares a single interrupt vector although pins can be enabled individually.
- **Port P1 interrupt edge select, P1IES:** can generate interrupts either on a positive edge (0), when the input goes from low to high, or on a negative edge from high to low (1). It is not possible to select interrupts on both edges simultaneously but this is not a problem because the direction can be reversed after each transition. Care is needed if the direction is changed while interrupts are enabled because a spurious interrupt may be generated. This register is not initialized and should therefore be set up before interrupts are enabled.

- **Port P1 interrupt flag, P1IFG:** a bit is set when the selected transition has been detected on the input. In addition, an interrupt is requested if it has been enabled. These bits can also be set by software, which provides a mechanism for generating a software interrupt (SWI).