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

ANNAMACHARYA EDUCATIONAL TRUST ★ VIDWAN SARVATRA PUJYATHE ★

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."

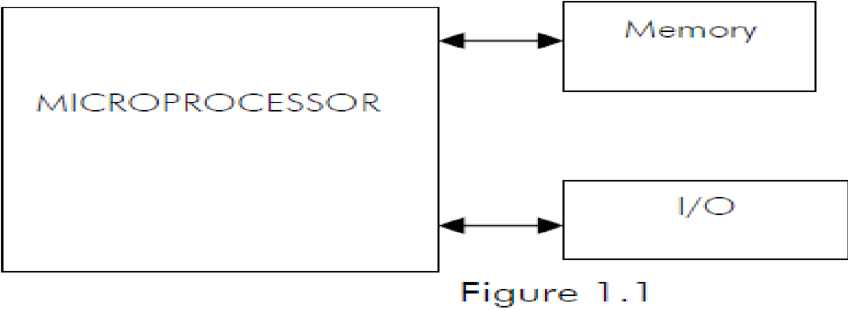


Figure 1.1

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 msec.)
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.

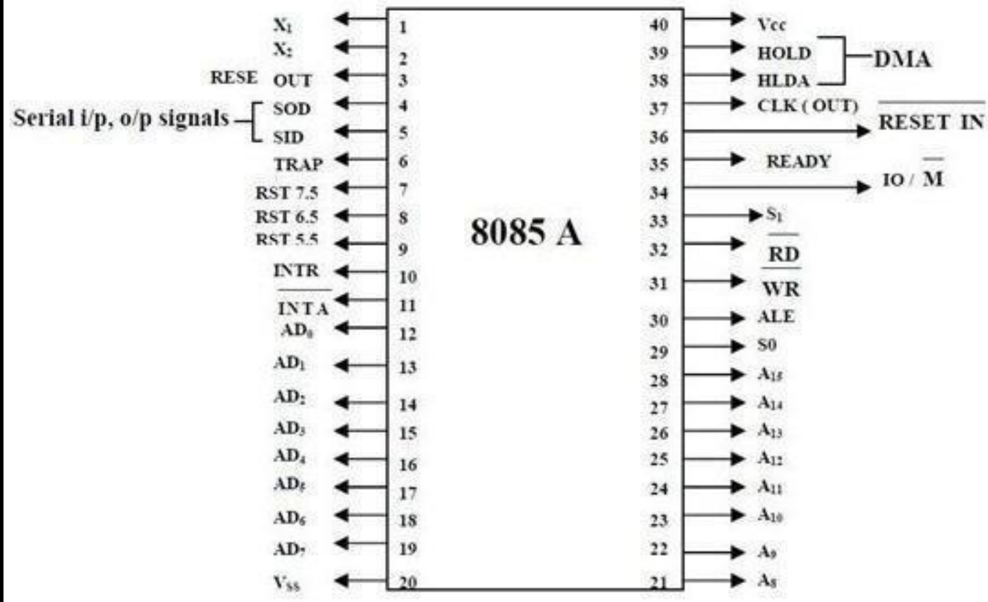
Fourth Generation Microprocessor:

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 a utomatically 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.

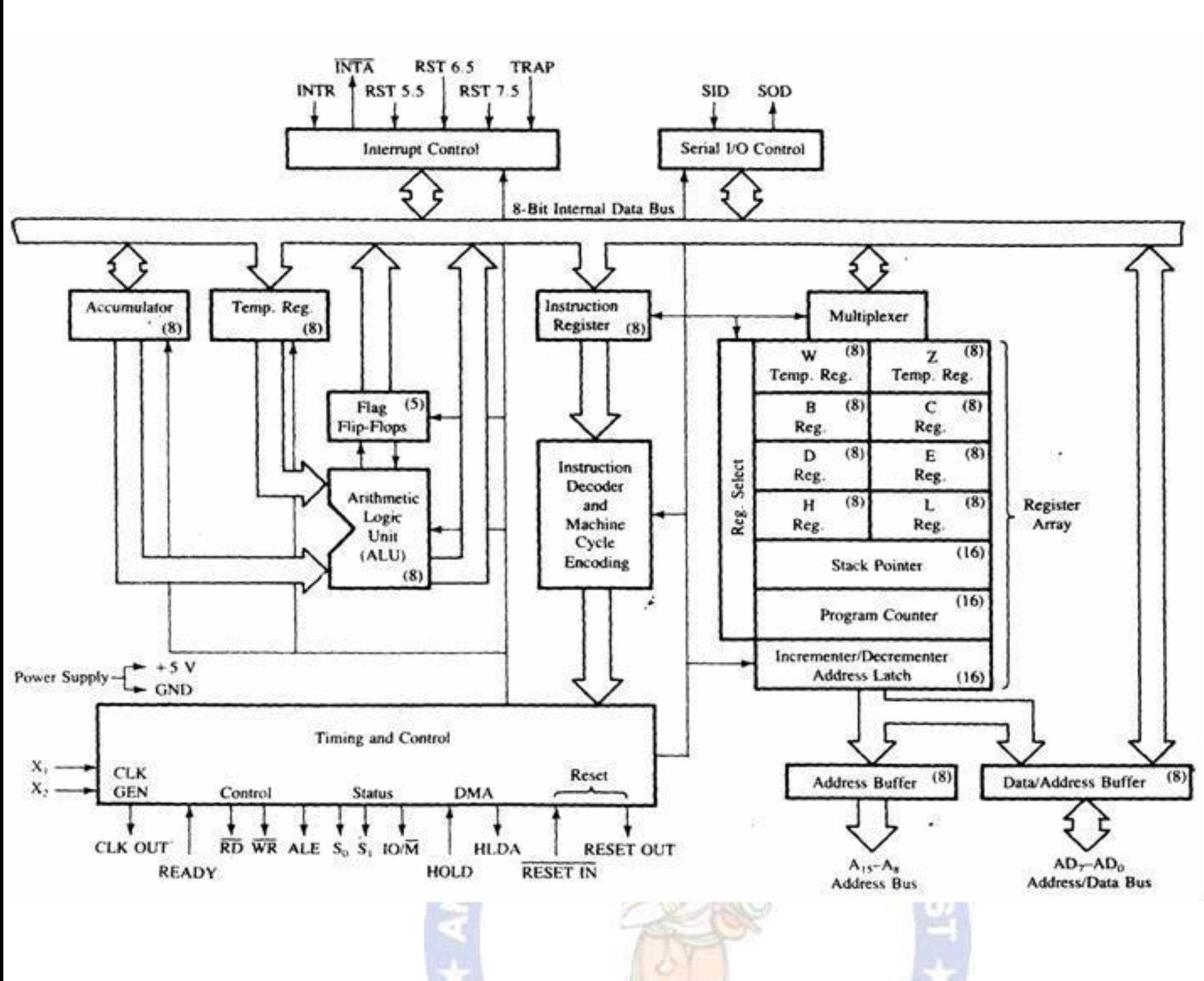
SO, S1 (Output)

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:

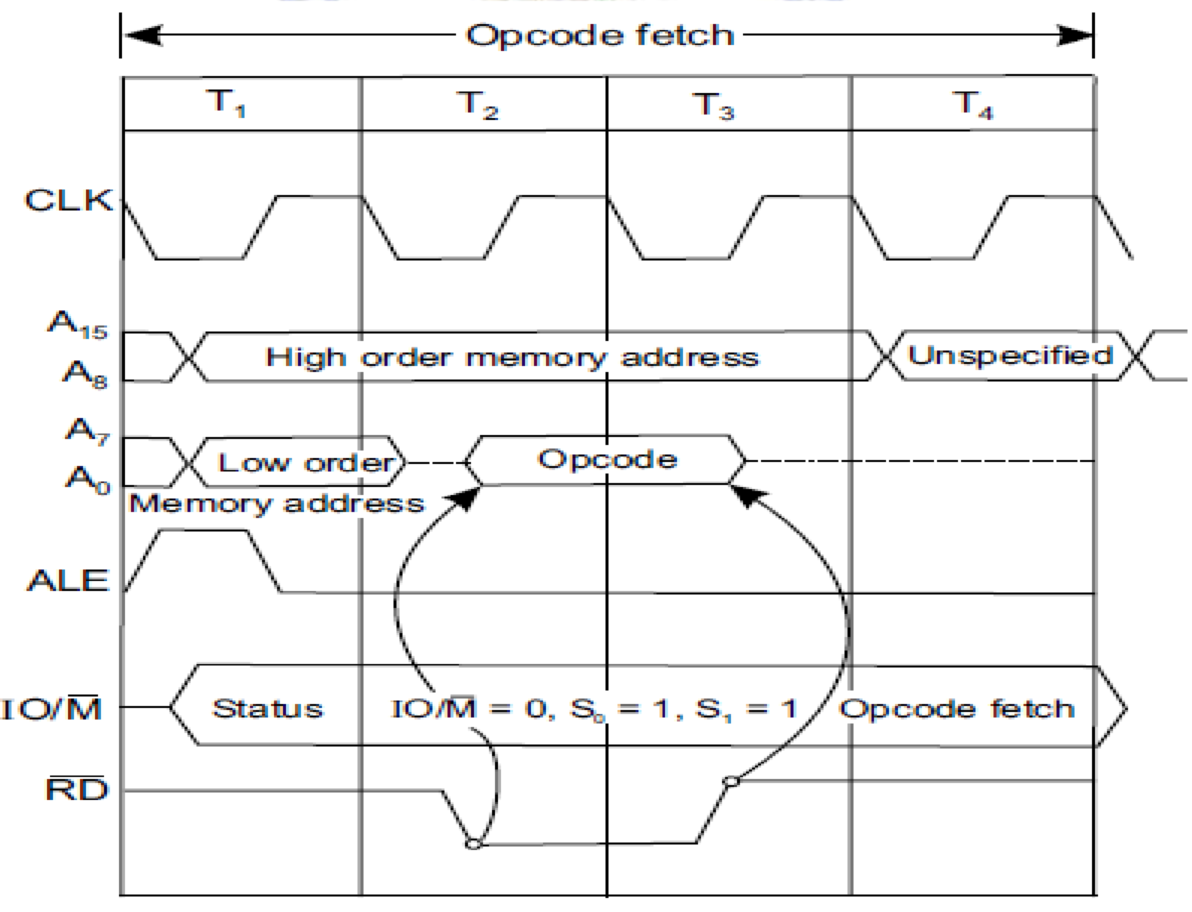
8085 has seven different machine cycles. These are:

(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:

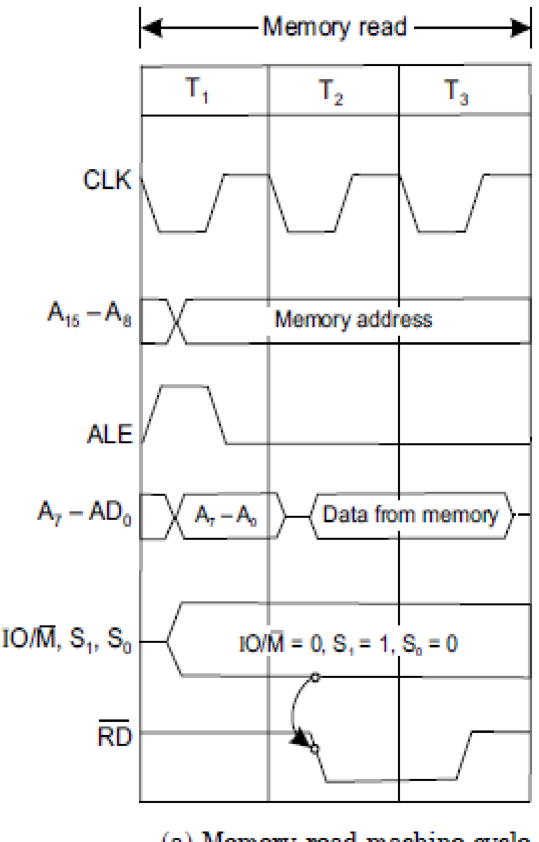
- The first machine cycle of every instruction is the Opcode Fetch. This indicates the kind of instruction to be executed by the system. The length of this machine cycle varies between 4T to 6T states—it depends on the type of instruction. In this, the processor places the contents of the PC on the address 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.

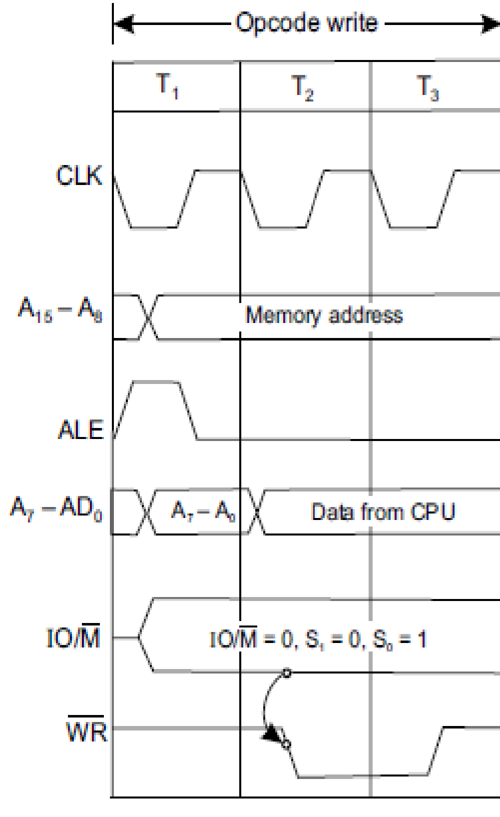


OPCODE FETCH TIMING DIAGRAM FOR 8085

Memory Read and Write Machine cycles:



(a) Memory read machine cycle



(b) Memory write machine cycle

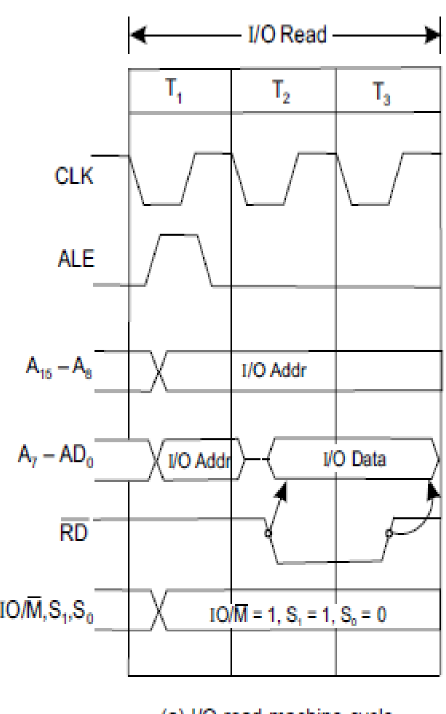
Both the Memory Read and Memory Write machine cycles are 3T states in length. In Memory Read the contents of R/W memory (including stack also) or ROM are read while in Memory Write, it stores data into data memory (including stack memory).

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.

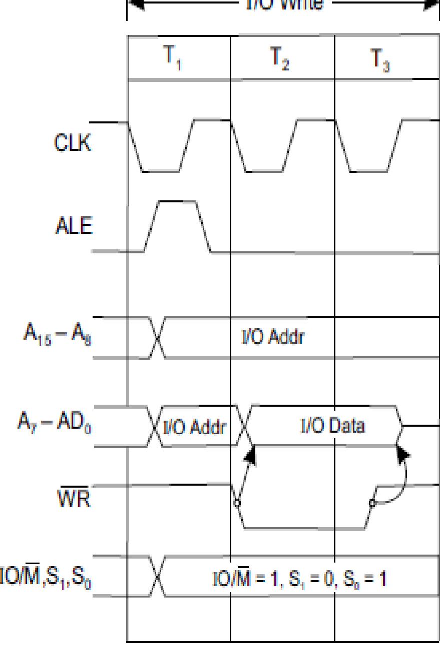
In T3 of Memory Read, data from data bus are placed into the specified register (A, B, C, etc.) and raises RD so that memory is disabled while in T3 of Memory Write WR signal is raised which disables the memory.

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:

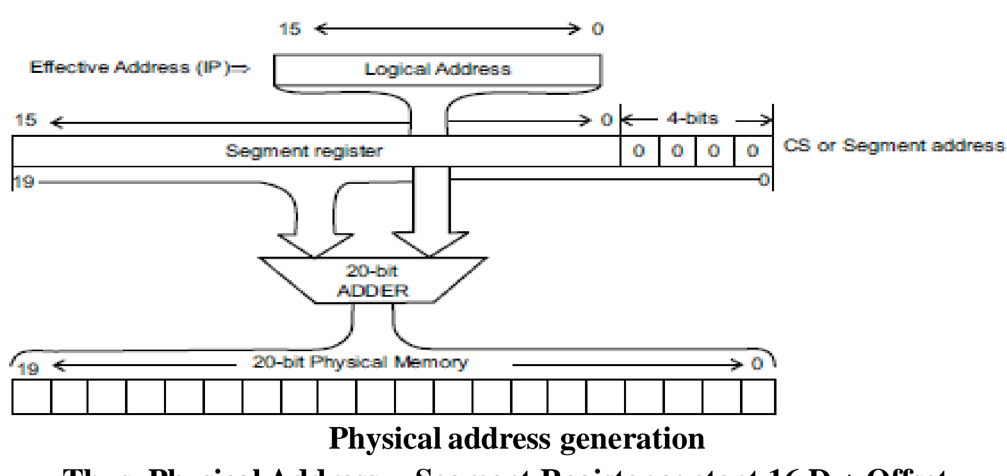
- It provides full 16 bit bidirectional data bus and 20 bit address bus.
- The BIU is responsible for performing all external bus operations.

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 contro l 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:

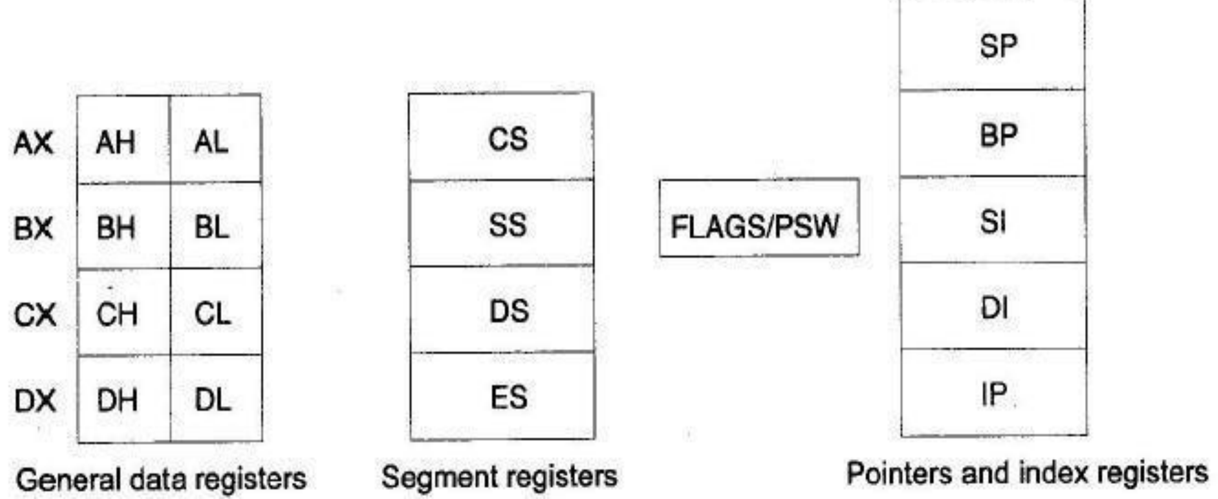
- **Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated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

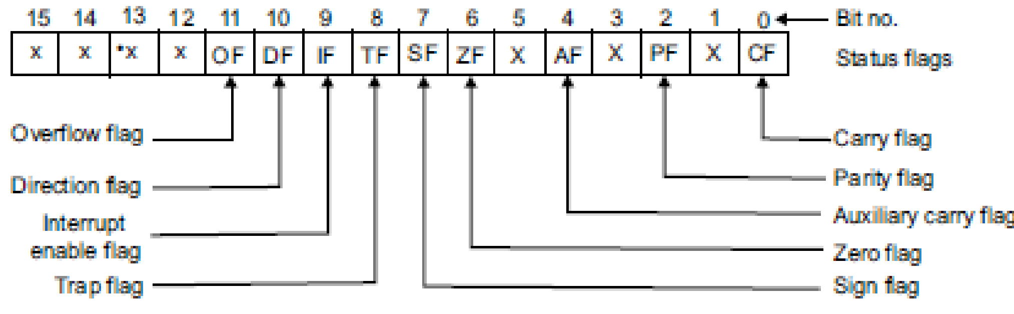
• **Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

• **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 comp ute 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 fla gs. 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 fromthe 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 t he 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**.

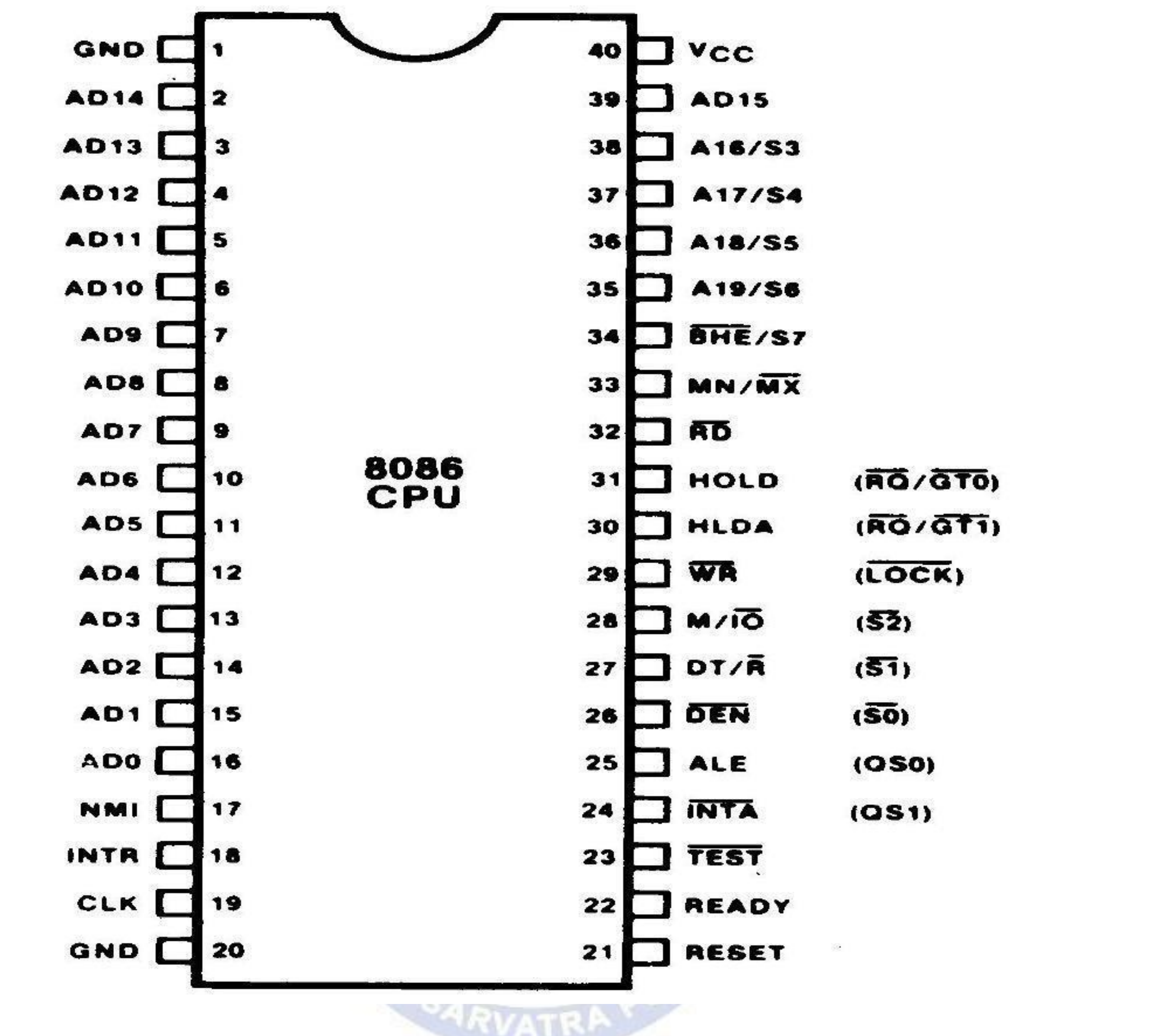
• The 8086 is housed in a **40-pin dual in-line package**. The signals pinned out to each lead are shown in figure.

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**:

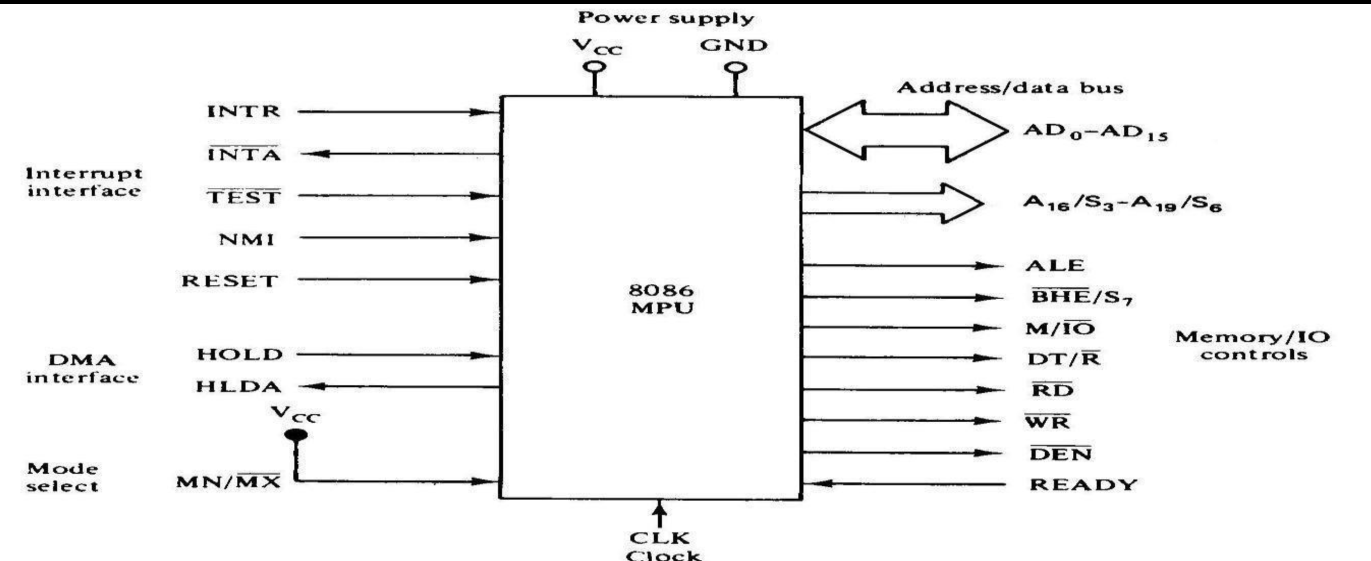
• The **minimum mode** is selected by applying **logic 1** to the **MN/MX** input lead. It is typically used for smaller **single microprocessor** systems.

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

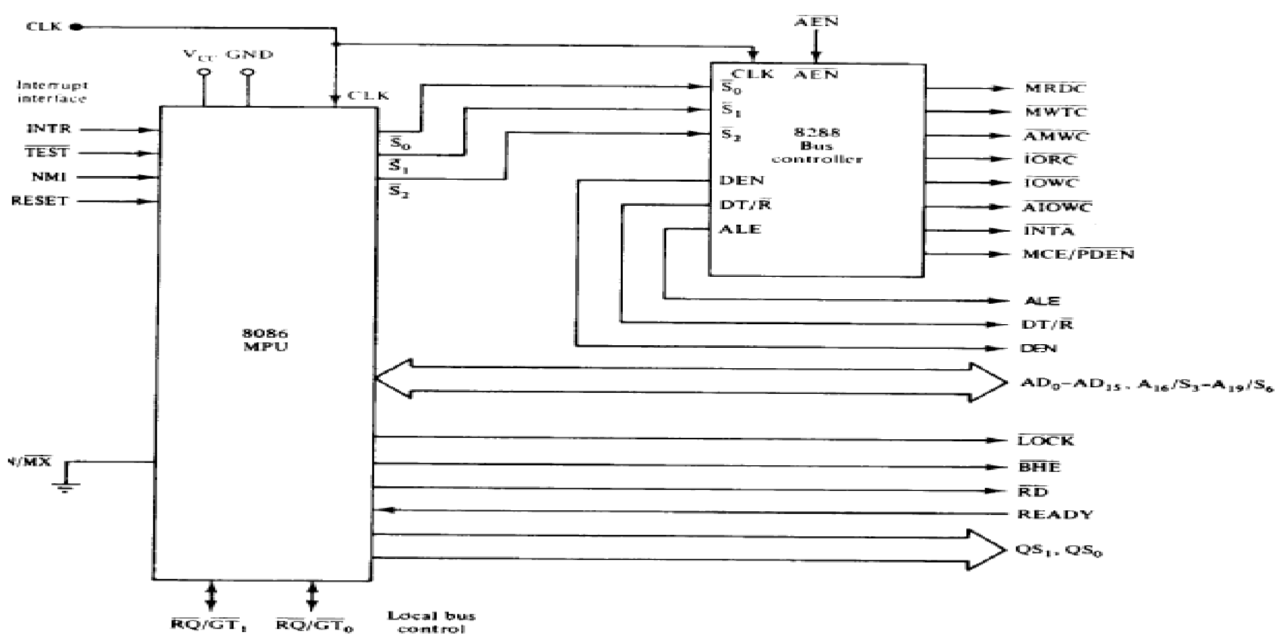


• Depending on the **mode** of operation selected, the **assignments** for a number of the **pins** on the microprocessor package are **changed**. The **pin functions** specified in **parentheses** pertain to the **maximum-mode**.

• 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):

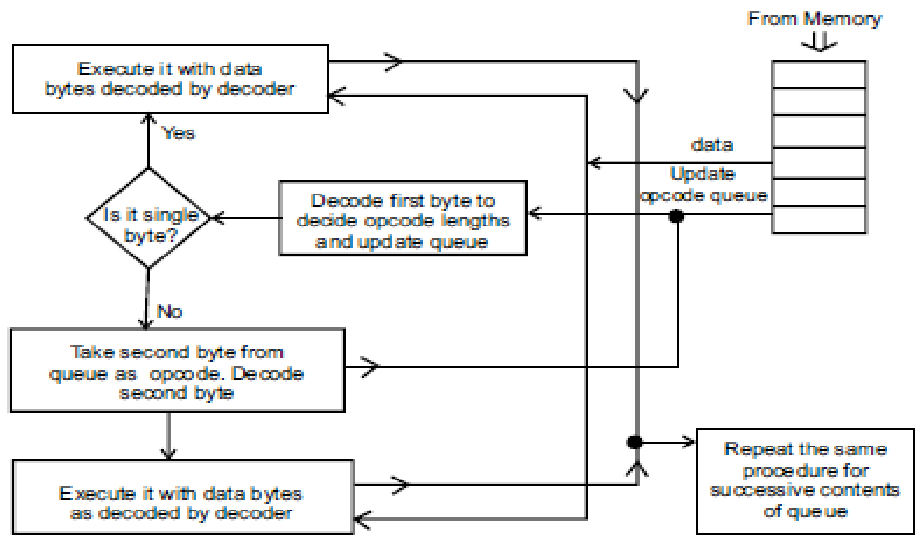
· These lines give information about the status of the code-prefetch queue.

· 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.

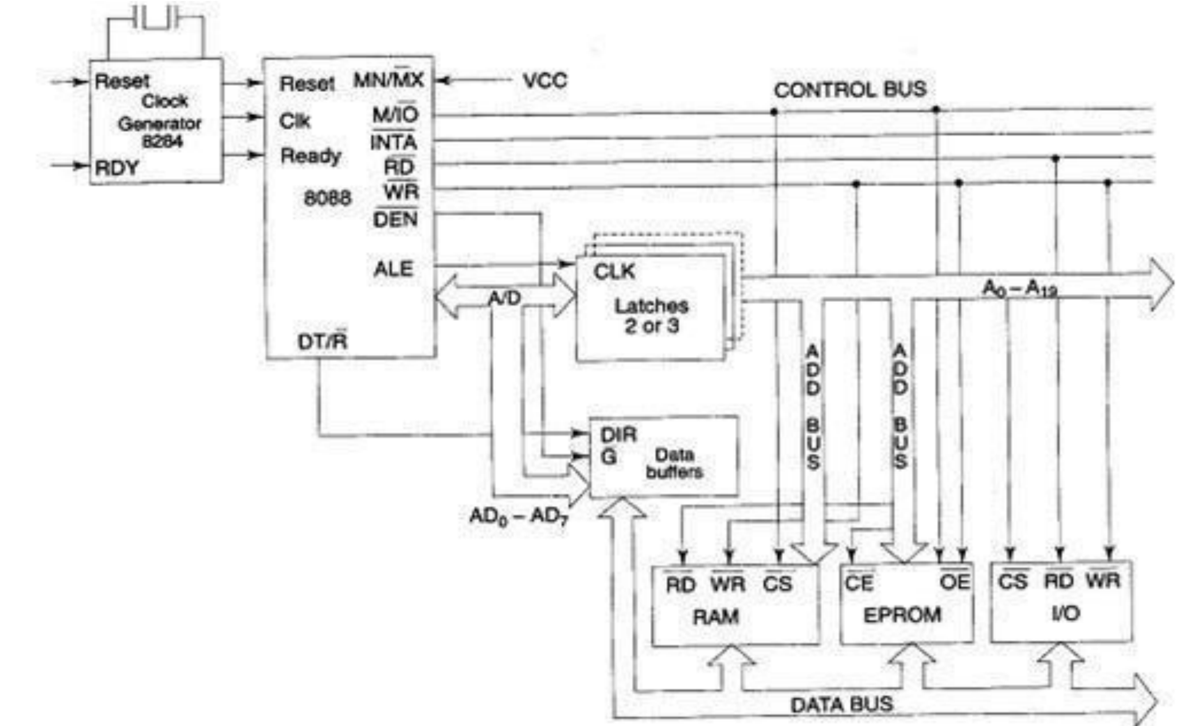
•The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

•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 se nds 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.

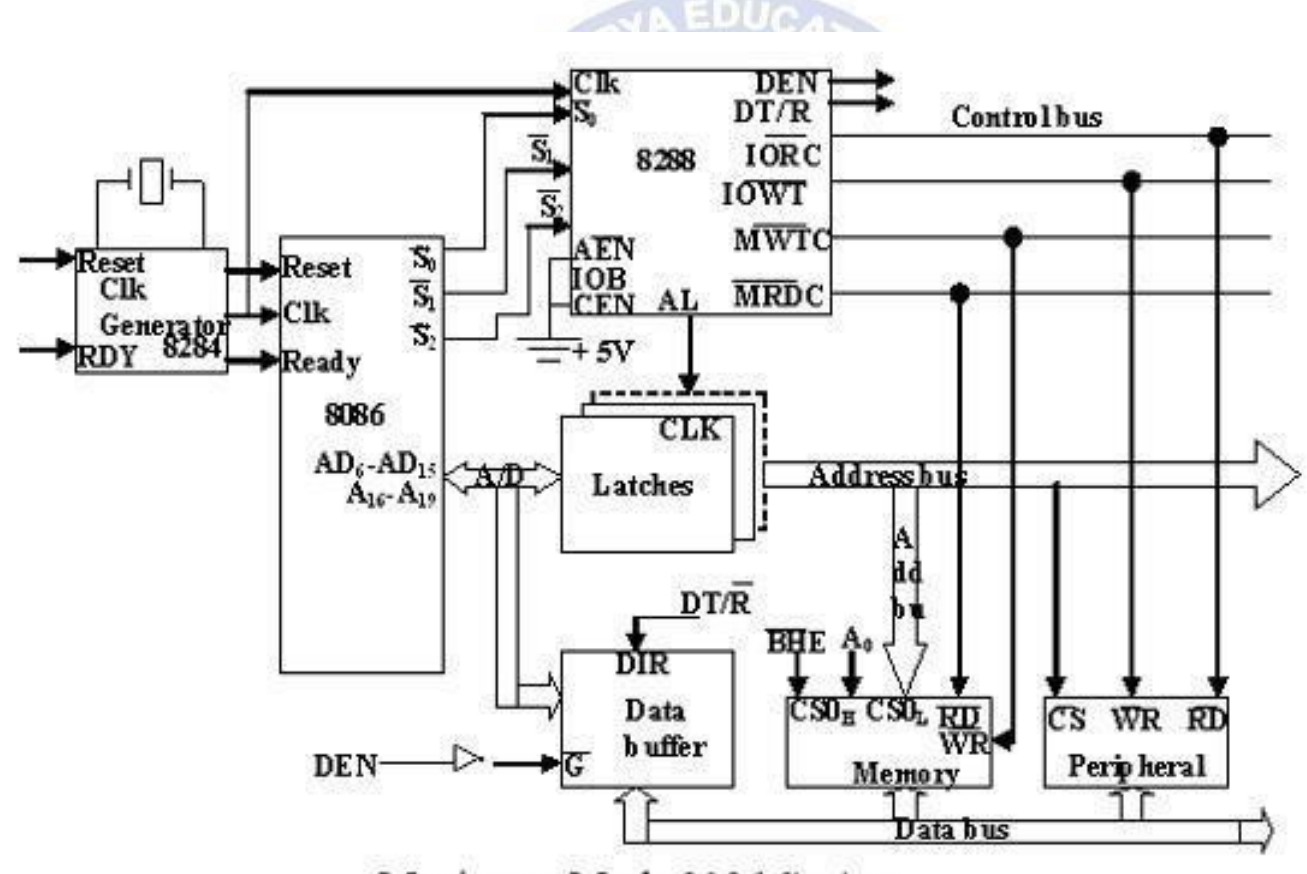
• In the maximum mode, there may be more than one microprocessor in the system configuration.

• 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**,

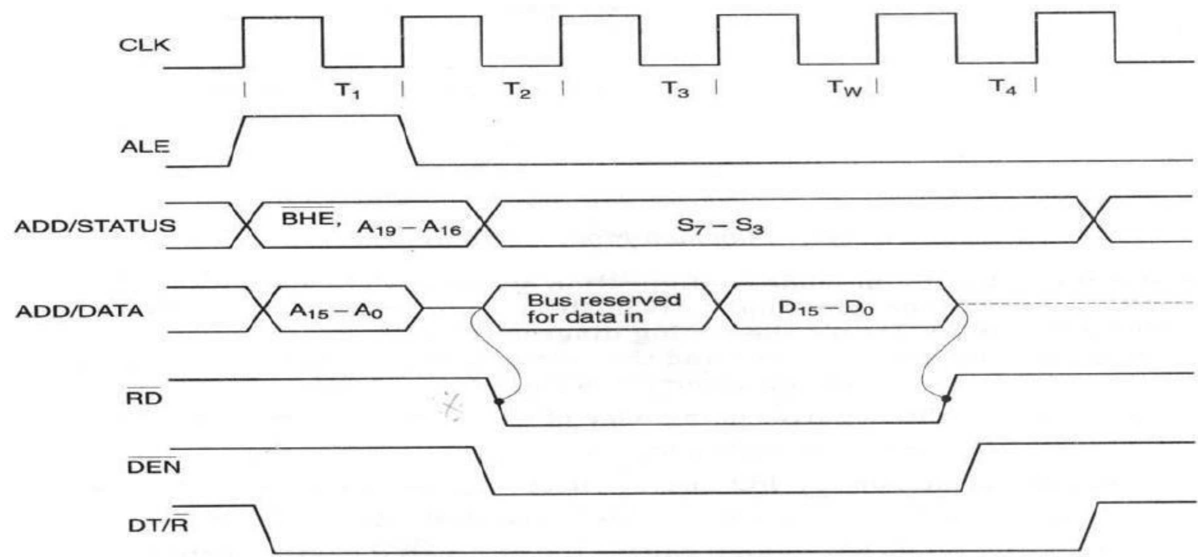
- Status bits **S3** through **S6** are output on the upper four address bus lines. This status information is maintained through periods **T3** and **T4**.
- On the other hand, address/data bus lines **AD0 through AD7** are put in the **high-Z state** during **T2**.
- Late in period **T2**, **RD** is 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**,

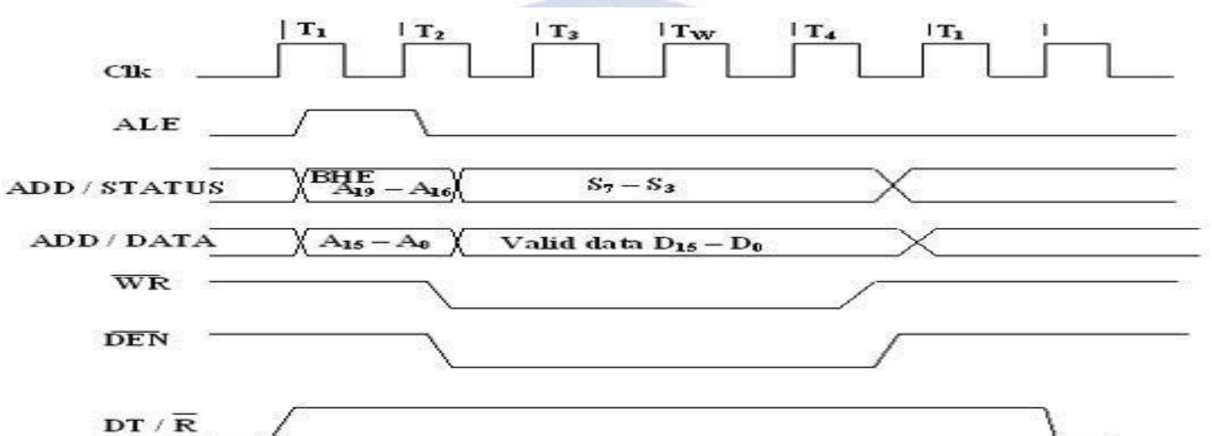
- The 8086 switches **RD** to the inactive **1 logic level** to terminate the read operation. **DEN** returns to its inactive logic level late during **T4** to disable the external circuitry.



MEMORY READ CYCLE FOR 8086 IN MINIMUM MODE

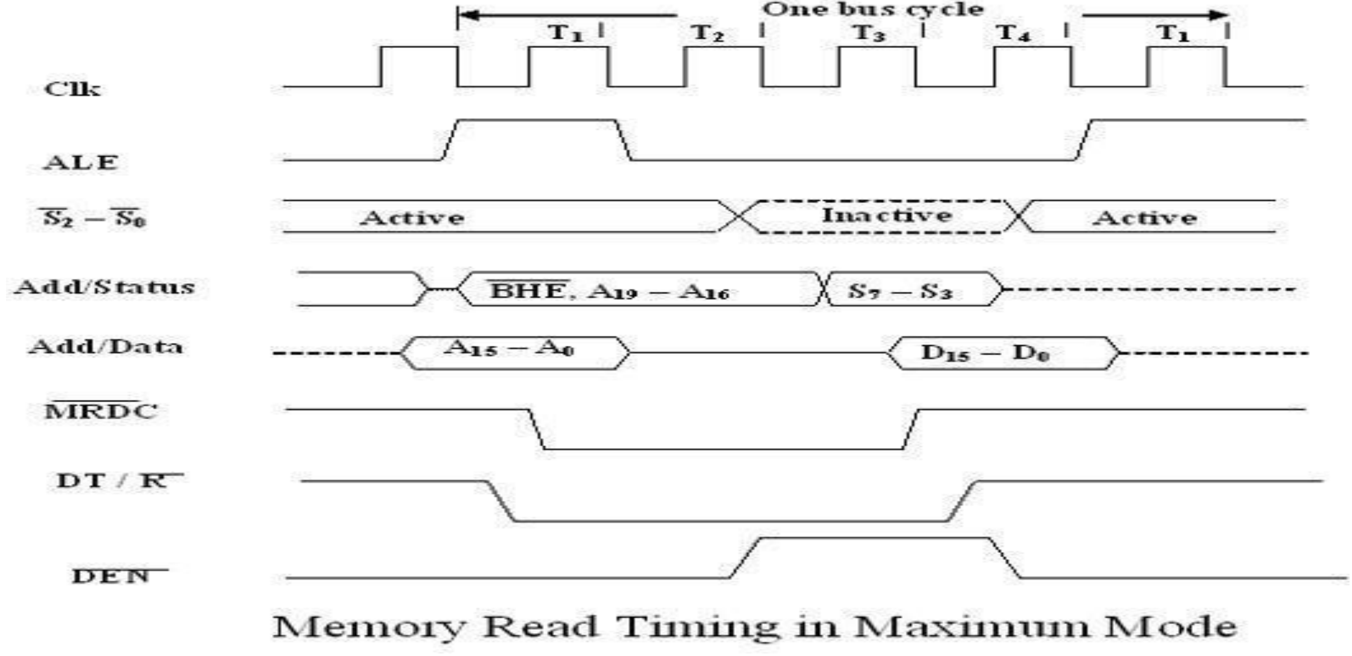
The following figure shows a **memory write cycle** of the 8086:

- 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.

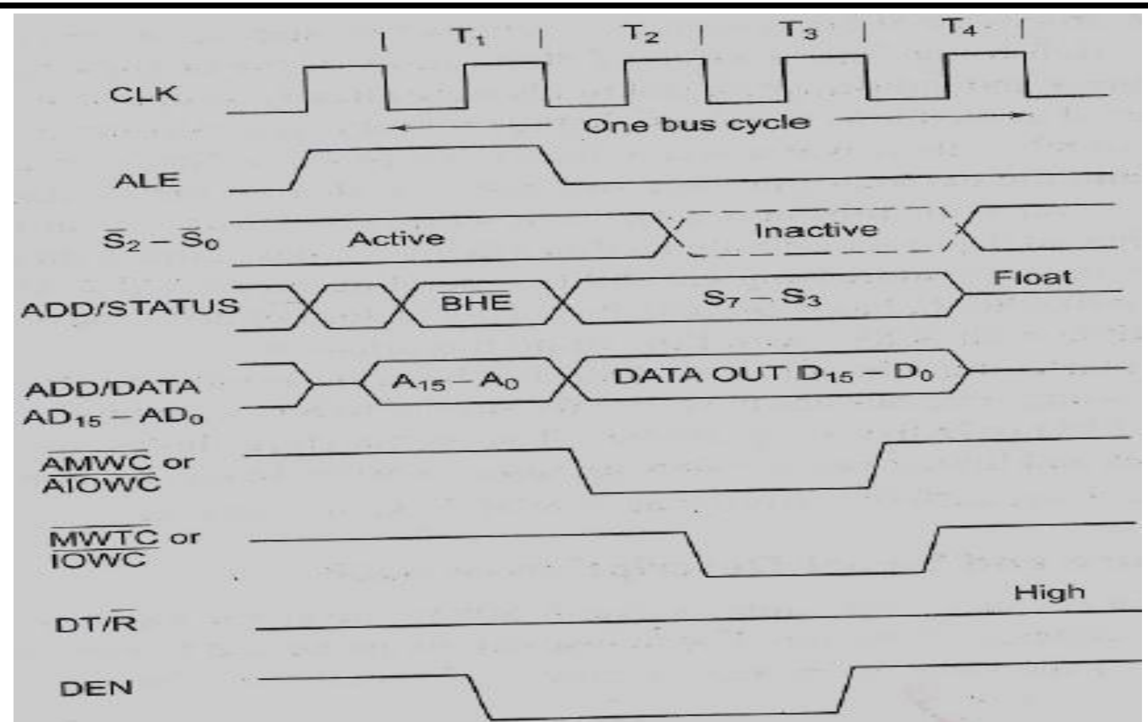


Write Cycle Timing Diagram for Minimum Mode

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

ANNAMACHARYA EDUCATIONAL TRUST ★ VIDYAYA AMRUTHAM ASNUTHE

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

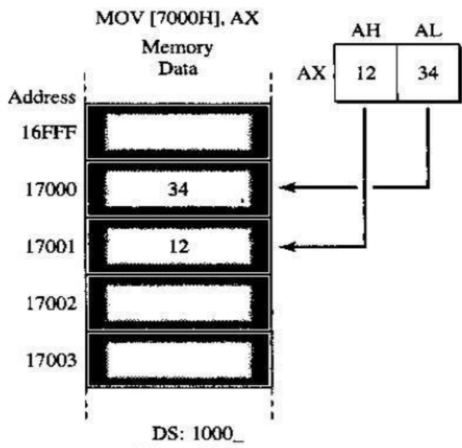
- **Sequential Control flow instructions:** In this type of instruction after execution control can be transferred to the next immediately appearing instruction in the program.

  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.

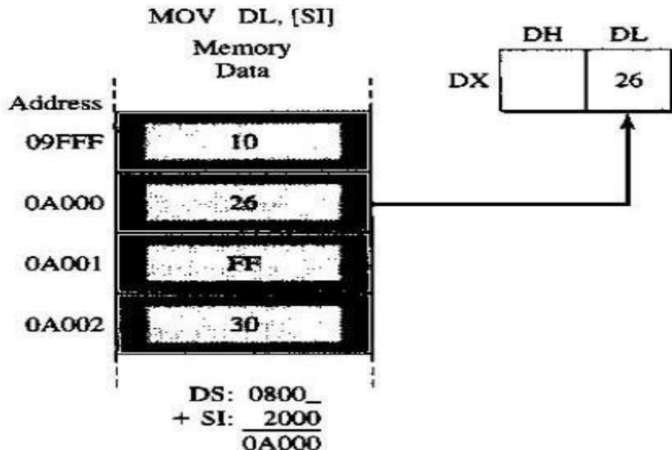
EA=10H * DS + 5000H

- **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.

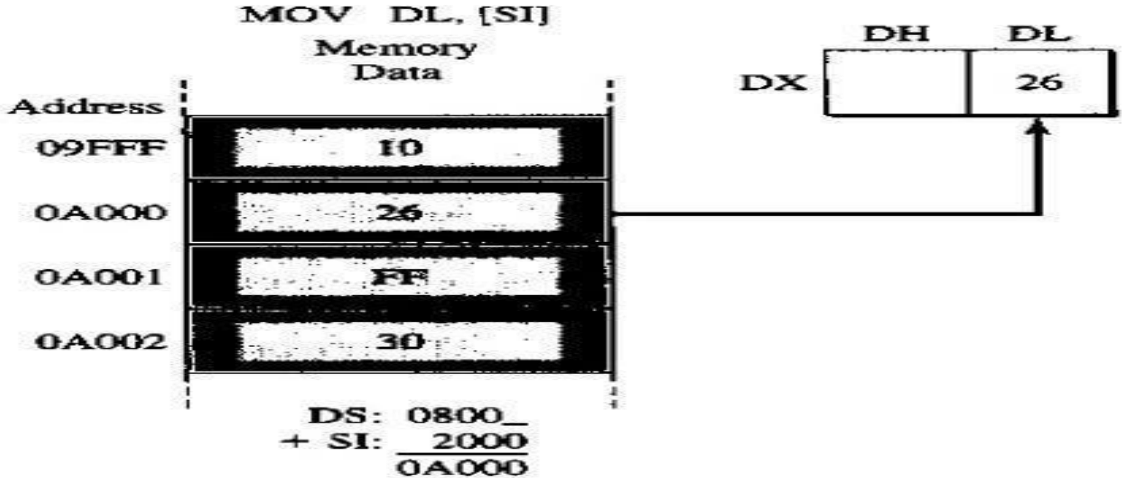
  Example: MOV AX, [BX]; **EA=10H * DS + [BX]**
- **Indexed Addressing mode:** 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides. DS and ES are

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

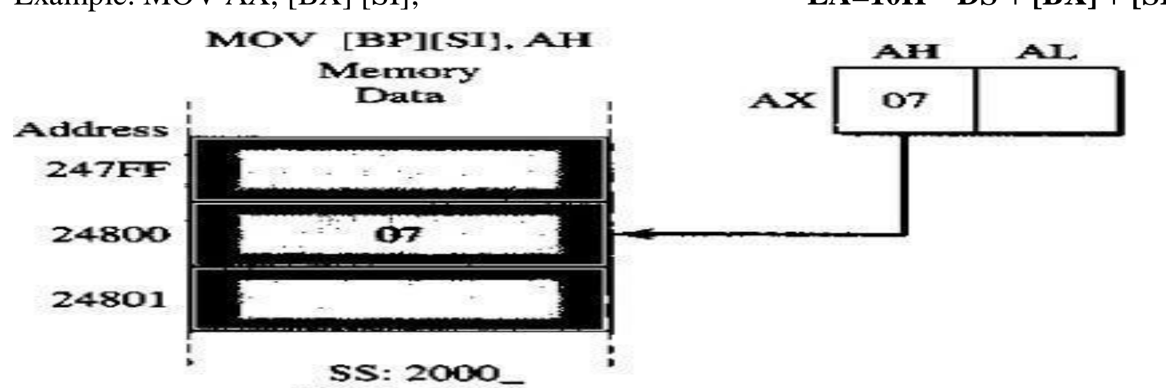


Example: MOV AX, [SI]; **EA=10H * DS + [SI]**

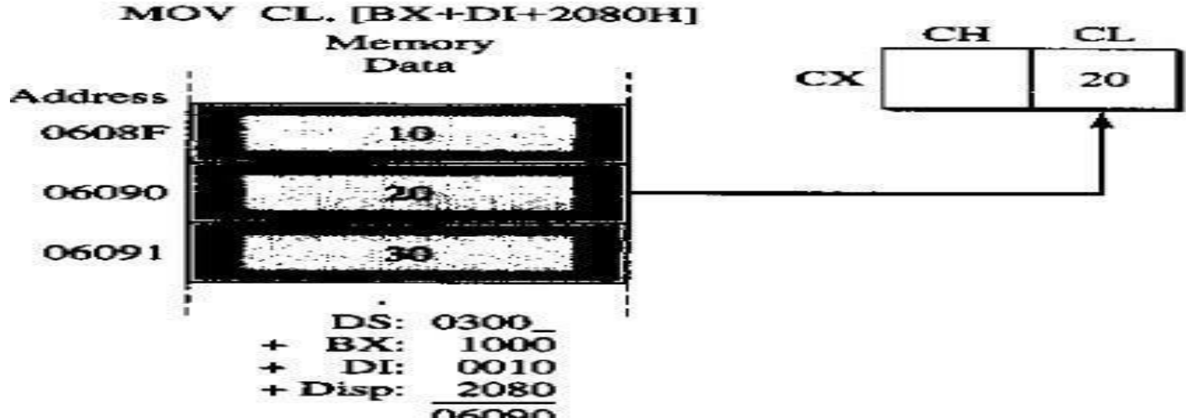
- **Register Relative Addressing mode:** In this mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI, DI in the default segments.
  Example: MOV AX, 50H [BX]; **EA=10H * DS + 50H + [BX]**



- **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.

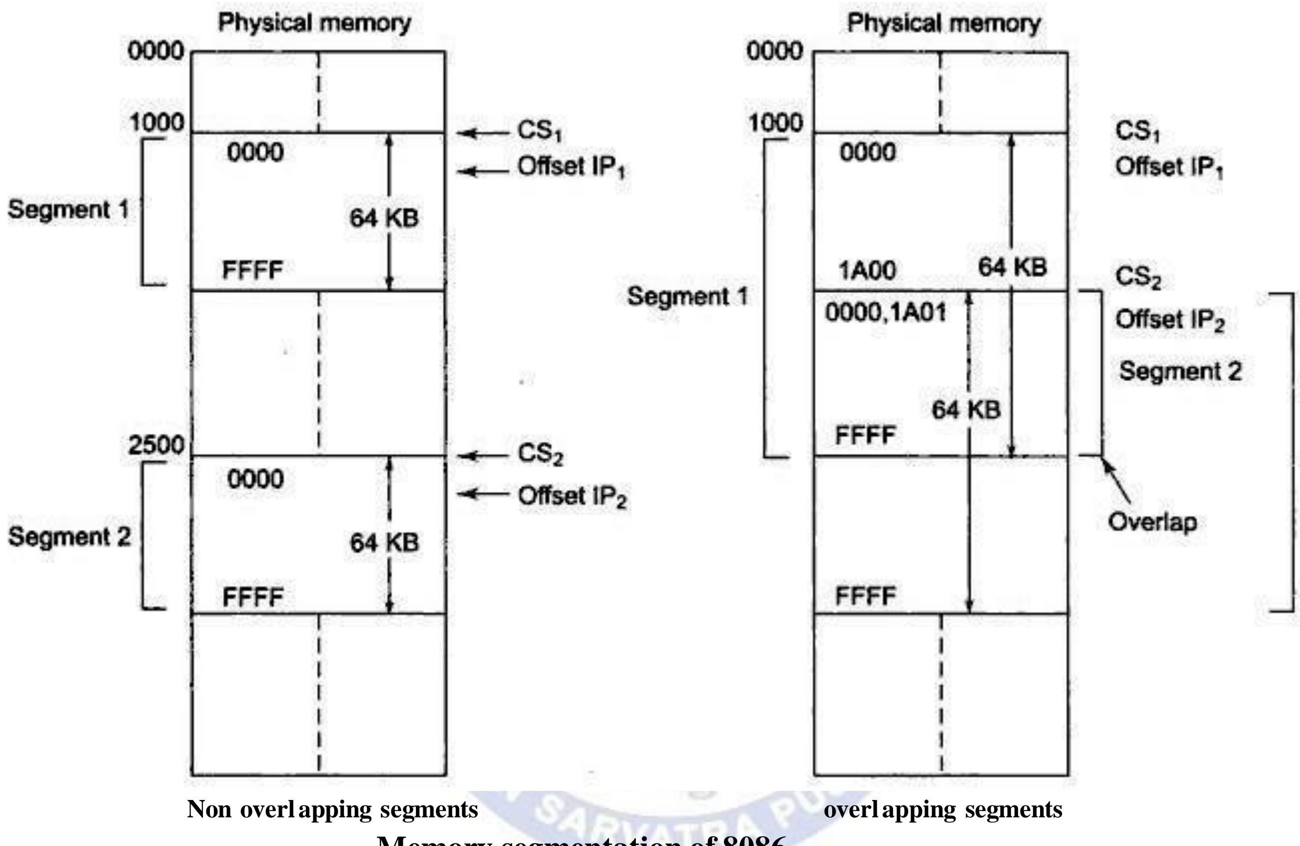
Memory Segmentation for 8086:

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:

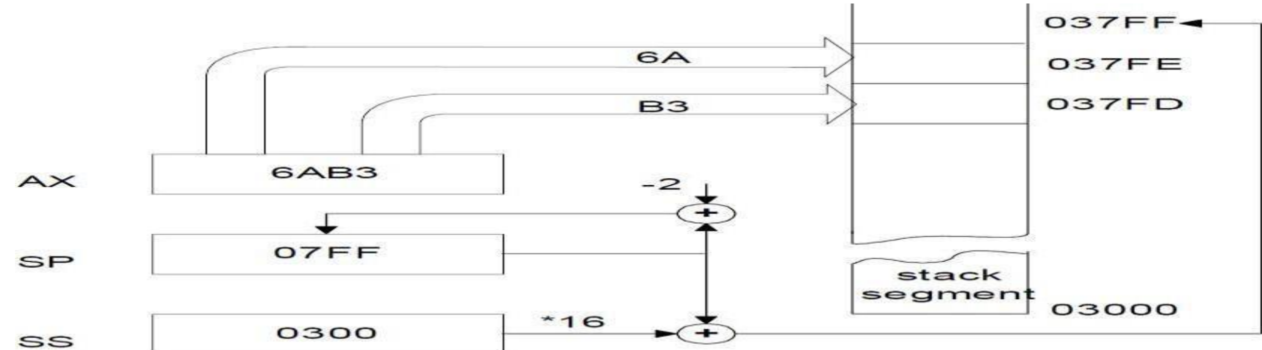
ORG 100h
MOV AX, 0B800h; set AX = B800h
MOV DS, AX; copy value of AX to DS.
MOV CL, 'A'; CL = 41h (ASCII code).

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 b yte 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.

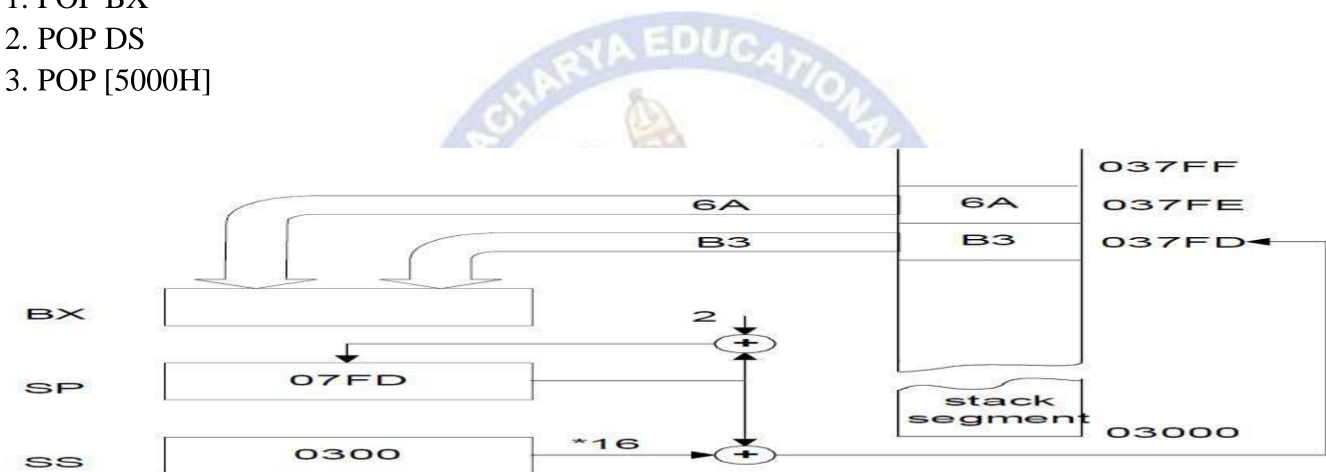
1. PUSH AX
2. PUSH DS
3. PUSH [500OH] ; Content of location 5000H and 5001 H in DS are pushed onto the stack.



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.

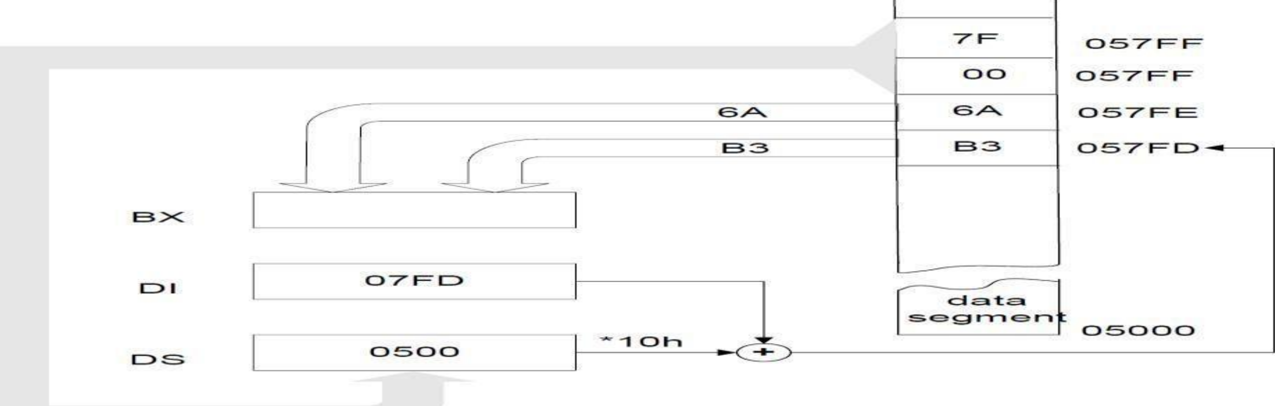
• LDS and LES instructions obtain a new far address from memory.

– 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

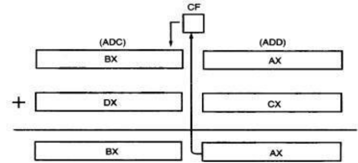
I/O Instructions: The 80x86 supports two I/O instructions: in and out15. They take the forms:

In ax, port
in ax, dx
out port, ax
out dx, ax
port is a value between 0 and 255.

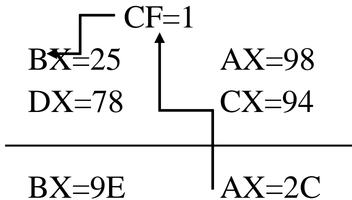
The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

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. e ight 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 co rrect 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:

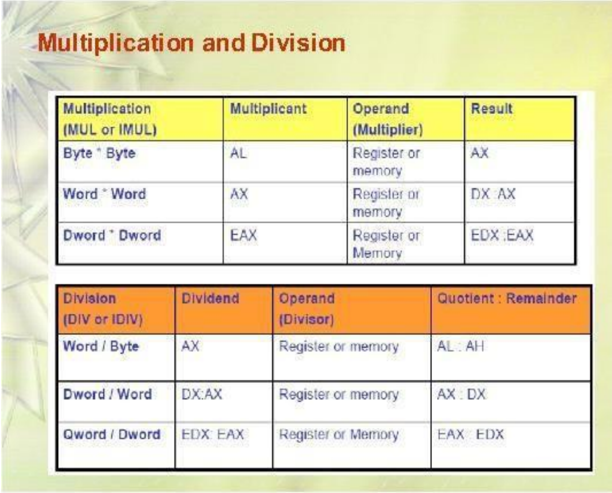
DX = 00000000 00000000

AX = 11110000 11000111 3897 decimal

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

Result DX = 11111111 11111111

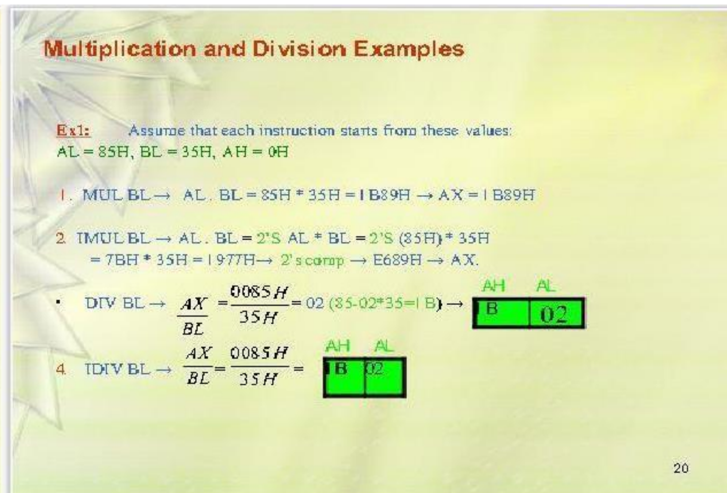
AX = 11110000 11000111 3897 decimal



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



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 aland 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

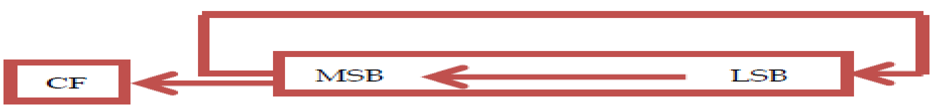
RCR: The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit. This instruction sets the flags in a manner analogous to rcl:

- 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

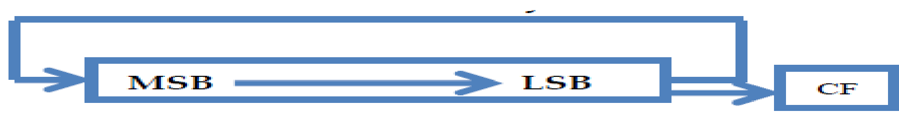
ROL: The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit, rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag. The 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: diand 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**

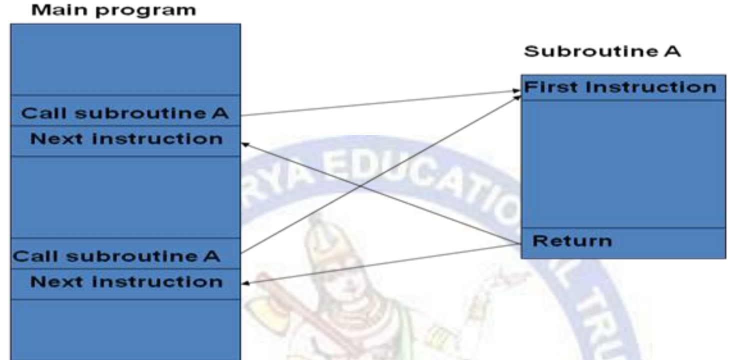
Conditional Jump: The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if…..then statement. The conditional jumps test one or more bits in the status register to see if they ma tch some particular pattern. If the pattern matches, 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 conditio ns 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 11223344556677889900H; 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	

ANNAMACHARYA EDUCATIONAL TRUST SARVATRA PUJYATHE

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 **3001** **3002** **3003**	**06** **04** **25** **12**	**3000** **3001** **3002** **3003**	**04** **06** **12** **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 **3001**	**06** **88**	**3000** **3001**	**88** **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 **3001** **3002** **3003** **3004** **3005** **3006** **3007** **3008**	**45** **4D** **50** **54** **59** **20** **56** **45** **53**	**300E**	**0E**

3009 **300A** **300B** **300C** **300D**	**53** **45** **4C** **53** **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**	**00**
		3002	**01**
		3003	**01**
		3004	**02**
		3005	**03**
		3006	**05**
		3007	**08**
		3008	**0D**
		3009	**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 **3001** **3002** **3003** **3004**	**04** **05** **09** **08** **03**	**3005** **3006**	**00** **02**

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 **3001** **3002** **3003**	**88** **00** **24** **00**	**3004** **3005**	**04** **00**

PROGRAM: FINDING THE 16 - BIT PRIME NUMBER

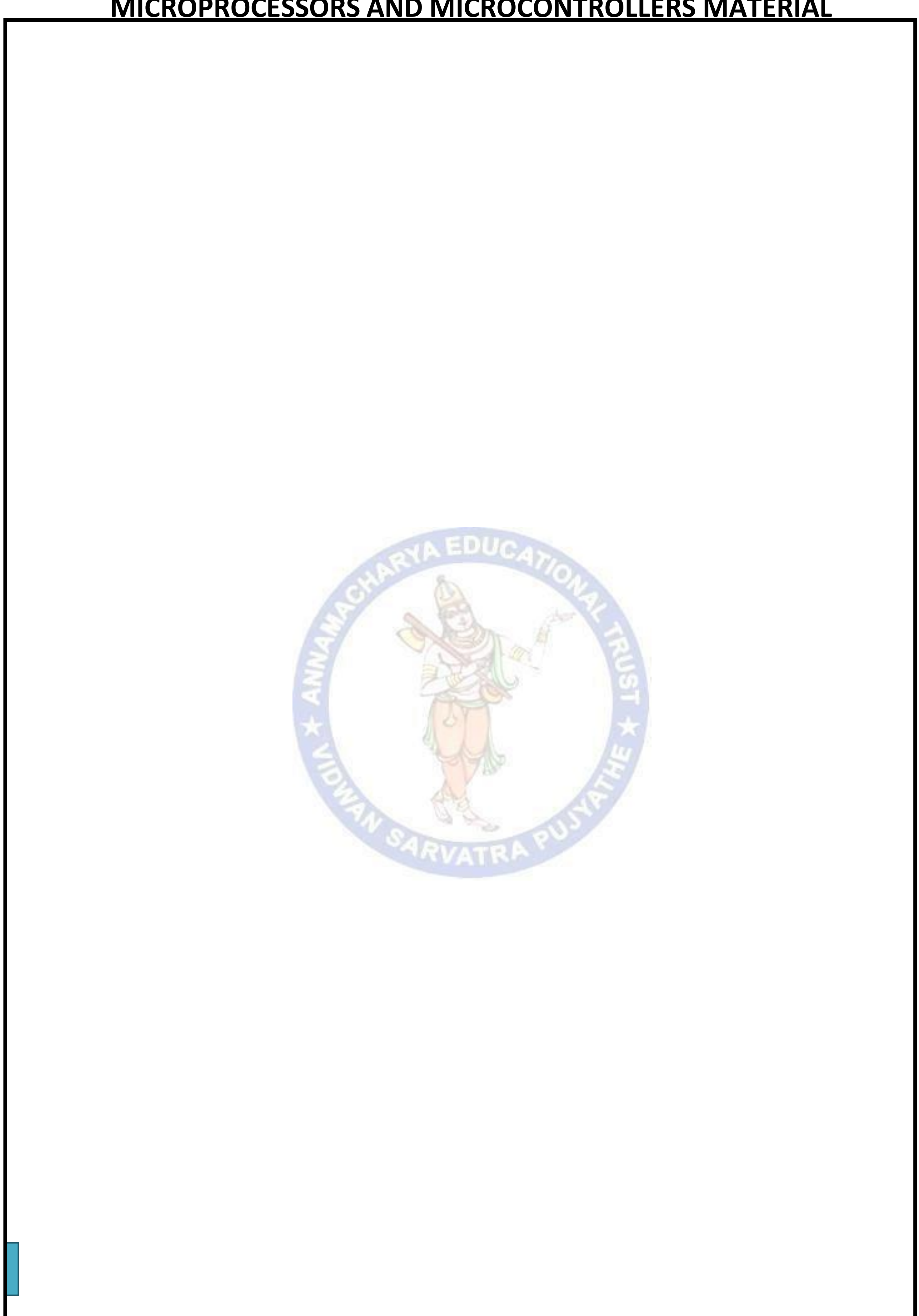
OFFSET ADDRESS	**LABEL**	**MNEMONICS**	**COMMENTS**
		ASSUME CS:CODE, DS:DATA	
		DATA SEGMENT ORG 3000H	

		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)**



UNIT-3: OVERVIEW OF MSP430 MICROCONTROLLER

1. Microprocessors Vs Microcontrollers
2. Introduction to MSP430 microcontrollers
   - *Features, Variants and Part numbering*
3. Block diagram of MSP430
4. Memory organization of MSP430
5. CPU architecture and Registers of MSP430
6. Block diagram of MSP430x5xx
7. Memory organization of MSP430x5xx
8. CPU architecture and Registers of MSP430x5xx
9. Addressing modes of MSP430
10. Instruction formats and Timings of MSP430
11. Instruction set of MSP430
12. Sample Embedded system using MSP430

MICROPROCESSORS Vs MICROCONTROLLERS

S.No	Microprocessors	Microcontrollers
1	ALU, Registers, Timing &Control	ALU, Reg., Timing & Control, RAM, ROM, Timers/ Counters, I/O Ports, Interrupt Control, Serial Ports
2	Microprocessor is the heart of the computer system, which fetches the instructions from memory and executes them.	Microcontroller is the heart of the embedded system, designed to perform a specific task by executing a fixed program stored in ROM
3	µp is a central processing unit (CPU) on a single chip, which consists of ALU, Registers, and Timing and control units.	µc has a CPU, in addition with RAM, ROM and other peripherals like I/O ports, Serial port, Timers/Counters, Interrupt controller, etc all embedded on a single chip
4	Less no. of on-chip registers	More no. of on-chip registers
5	Less no. of Bit manipulation instructions	More no. of Bit manipulation instructions
6	Size, power and cost of the system increases due to external components.	Low size, Low power and Low cost, due to less no. of external components.
7	Microprocessors are used for executing general purpose applications	Microcontrollers are used for executing specific applications.
8	Used in desktop PC's, Laptops, Notepads etc	Used in embedded applications like Washing machine, DVD player, AC, etc

3.2. INTRODUCTION TO MSP430 FAMILY

Introduction to MSP430 :

- MSP430 family microcontrollers from Texas Instruments (TI), are designed for low cost, low power and portable embedded applications
- MSP430 has **16-bit RISC based processor architecture**
- It supports different **Low power modes**
- It has 16 registers (R0-R15)
- All registers are 16-bit wide
- It has 16-bit Address bus and 16-bit data bus
- Supports 27 core instructions, 24 emulated instructions and 7 addressing modes
- It is capable of wake-up time below 1 microsecond
- Extensive vectored-interrupt capability
- A wide range of on-chip peripherals are available

Features of MSP430 :

Although there are variants in devices in the family, a MSP430 microcontroller can be characterized by:

Device parameters

- Flash/ ROM options: 1 KB – 60 KB
- RAM options: 128 B– 8 KB
- GPIO options: 14 - 80 pins

Clock and Power Specifications

- CPU clock : 8/16/25 MHz
- Operating voltage : 1.8–3.6 V
- Active operation : 160 - 250 µA/MIPS
- RTC mode operation : 0.7 µA
- RAM retention : 0.1 µA
- Fast wake-up from standby mode in less than 1 µs

Other integrated peripherals

* Basic Clock system
* I/O ports
* Serial Port : SPI, I2C, UART
* Timers
* WDT (Watch Dog Timer)
* RTC (Real Time Clock)
* Multiplier
* DMA
* LCD driver
* Supply Voltage Supervisor (SVS)
* Brown out Reset
* The emulator and JTAG interface
* 10/12/16-bit ADC
* 12-bit dual DAC
* Op-Amp
* Comparator_A
* Temp. sensor

Advantages of MSP430 family :

- 16-bit RISC architecture
- High-performance - High speed of execution
- Low power consumption
- Fast wake-up from standby mode in less than 1 µs
- Variety of models with integrated memories, multiple programmable GPIO and Integrated application-specific peripherals
- Cost-effective

Applications of MSP430

- Low power, hand-held smart devices
- Test and measurement equipment
- Smart Energy/Smart Grid solutions
- Factory automation
- Home and commercial site monitoring and control
- Medical instrumentation
- Fire and security
- Intelligent lighting control
- Transportation
- Motion control
- Automobiles
- Gaming equipment

VARIANTS OF MSP430 FAMILY

(i) MSP430x1xx

- 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 a basic digital signal processor.
- The integrated peripherals include - 12-bit DAC, two 16-bit timers, WDT, brown-out reset, SVS, USART module (UART, SPI), DMA, 16×16 multiplier, Comparator_A, Temp. Sensor

(ii) MSP430x2xx

- The MSP430F2xx Series are similar to the '1xx generation, but operate at even lower power, support up to 16 MHz operation**,** double the speed of earlier devices, while consuming only half the current at the same speed. It has On-chip clock (VLO) that makes it easier to operate without an external crystal. Pull-up or pull-down resistors are provided on the inputs to reduce the number of external components needed.
- The integrated peripherals include -operational amplifiers, 12-bit DAC, two 16-bit timers, watchdog timer, brown-out reset, SVS, USI module (I²C, SPI), USCI module, DMA, 16×16 multiplier, Comparator_A+, Temperature sensor

(iii) MSP430x3xx

- The MSP430x3xx Series is the oldest generation, designed for portable instrumentation with an embedded LCD controller. This also includes a frequency-locked loop oscillator that can automatically synchronize to a low-speed (32 kHz) crystal.

- This generation does not support EEPROM memory, only mask ROM and UV- eraseable and one-time programmable EPROM. The integrated peripherals include - LCD controller and multiplier

(iv) MSP430x4xx

- MSP430x4xx series can drive LCDs with up to 160 segments. Many of them are ASSPs, but there are general-purpose devices as well. These devices are used for low power metering and medical applications.
- The integrated peripherals include - 12-bit DAC, Op Amps, RTC, up to two 16-bit timers, watchdog timer, basic timer, brown-out reset, SVS, USART module (UART, SPI), USCI module, LCD Controller, DMA, 16×16 & 32x32 multiplier, Comparator_A, Temp. sensor

(v) MSP430x5xx

The MSP430x5xx Series are able to run up to 25 MHz, have up to 512 KB flash memory and up to 66 KB RAM. It includes an innovative power management module for optimal power consumption and integrated USB. The other integrated peripherals include - High resolution PWM, 5 V I/O's, backup battery switch, up to 4 16-bit timers, watchdog timer, Real-Time Clock, brown-out reset, SVS, USCI module, DMA, 32x32 multiplier, Comp B, temperature sensor

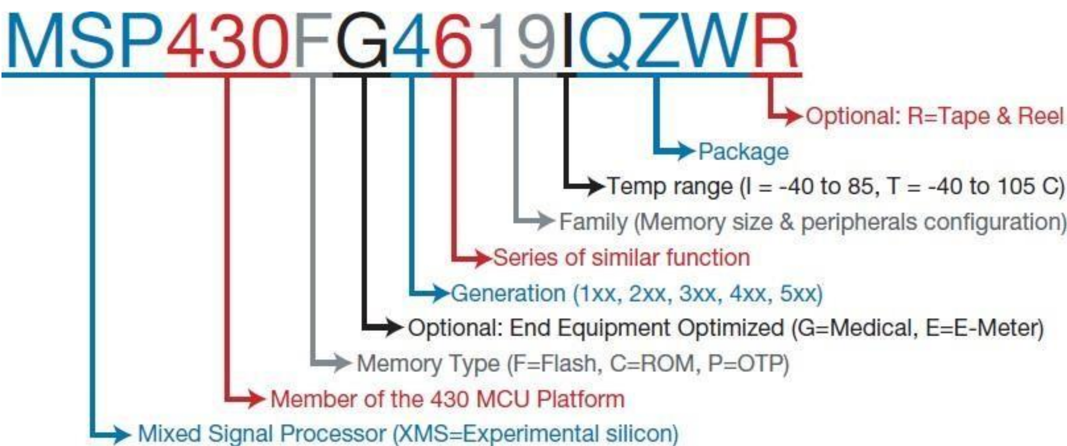
MSP430X :

- There is a new extended version of original MSP430 architecture, called MSP430X, which can address extra memory with other improvements as well.

- If CPU is MSP430, it has 16-bit address bus and it can address 64 KB of memory (0x0000 – 0xFFFF). If the CPU is MSP430X, it has 20-bit address bus and it can address 1 MB of memory. The bottom 64 KB of memory from 0x0000 to 0xFFFF is same way as in the original MSP430. The additional memory from 0x10000 to 0xFFFFF, is available for additional ROM. This allows larger programs and tables to be stored.

Variants of MSP430 family	MSP430x1xx	MSP430x2xx	MSP430x3xx	MSP430x4xx	MSP430x5xx
Clock	8 MHz	16 MHz	16 MHz	16 MHz	16 MHz
I_{active} /MIPS	200 µA	200 µA	160 µA	200 µA	165 µA
$I_{RTC\ mode}$	0.7 µA	0.7 µA	0.9 µA	0.7 µA	2.5 µA
I_{RAMret}	0.1 µA	0.1 µA	0.1 µA	0.1 µA	0.1 µA
Wake-up time	< 6 µs	< 1 µs	< 6 µs	< 6 µs	< 5 µs
Flash/ROM	1-60KB	1-60KB	2-32 KB	4-60 KB	up to 512KB
RAM	128 B -2KB	128 B -2KB	512 B -2KB	256 B -2KB	up to 66KB
GPIO	10-48	10-48	14-40	14-80	32-90

MSP430 PART NUMBERING



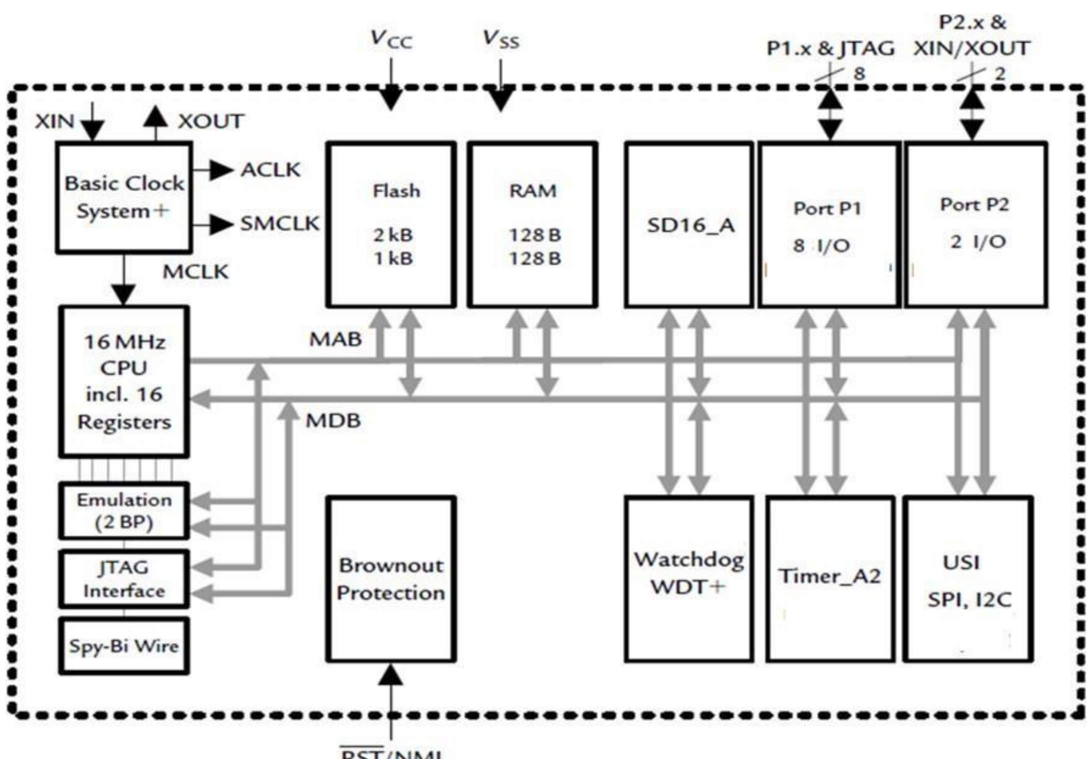
MSP 430	The letters MSP stand for *mixed signal processor* MSP430 indicates the member of the 430 MCU platforms.
F	The letter after MSP430 shows the type of memory. F : flash memory, C : masked ROM
G (optional)	There is a second letter for ASSPs to show the type of measurement for which they are intended. E : for electricity (Energy meters), W : for water (flow meters), and G : for signals that require gain stage provided by Op-Amps (Medical)
4	It indicates the generation of the device
6	Model within generation
19	Memory size and peripheral configuration
I	Temperature range I : -40 to 85 °C T : -40 to 105 °C S : 0 to 50 °C
QZW	Package : Ball Grid Array/ Pin Grid Array/ Dual In-line
R (optional)	Tape and Reel : T = Small Reel (7 –in), R = Large Reel (11 –in)

3.3. BLOCK DIAGRAM OF MSP430 F2013 / F2003

Introduction to MSP430 :

- MSP430 family microcontrollers from Texas Instruments (TI), are designed for low cost, low power and portable embedded applications
- MSP430 has **16-bit RISC based processor architecture**
- It supports different **Low power modes**
- It has 16 registers : R0-R15
- All registers are 16-bit wide
- It has 16-bit Address bus and 16-bit data bus
- Supports 27 core instructions, 24 emulated instructions and 7 addressing modes
- It is capable of wake-up time below 1 microsecond
- Extensive vectored-interrupt capability
- A wide range of on-chip peripherals are available

- The functional block diagram of the MSP430 F2003/F2013 is shown in figure. The main blocks are linked by the *memory address bus* (MAB) and *memory data bus* (MDB).



Block diagram of the MSP430F2003 and F2013

16- bit CPU :

- It consists of 16-bit ALU, set of 16-registers (R0 – R15) and Logic needed to decode and execute the instructions
- The CPU has RISC architecture
- Instructions processing on either bits, bytes or words
- Supports 27 instructions and 7 addressing modes.
- It can address the complete address range without paging
  - CPU clock : 16 MHz
  - Operating voltage : 1.8–3.6 V
  - Active operation : 200 µA/MIPS
  - RTC mode operation : 0.7 µA
  - RAM retention : 0.1 µA
  - Fast wake-up from standby mode in less than 1 µs

Basic Clock system :

The clock module provides the CLK for CPU and peripherals. Three clock signals are available from the basic clock module:

MCLK : Master clock is used by CPU and system
SMCLK : Subsystem Master clock is distributed to high speed peripherals
ACLK : Auxiliary clock is also distributed to low speed peripherals

The emulator and JTAG interface

The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.

Flash memory

Flash memory is used to store the programs and constant variables
The size of Flash memory in MSP430F2003 is 1 KB
The size of Flash memory in MSP430F2003 is 2 KB

RAM

RAM is used to store the temporary data (Read/Write memory)
The size of RAM in MSP430F2003 and F2013 is 128 Bytes

16- bit Sigma Delta ADC : SD16_A

The SD16_A is ahigh performance 16-bit analog to digital converter
used to interface analog signals with 200 KSPS

I/O ports : P1 & P2

The MSP430 has Two I/O ports: Port-P1 and Port-P2
The Port-P1 has 8- I/O pins and P2 has 2- I/O pins
Each I/O pin is individually configurable for input (or) output.
Each pin can be configurable for pull-up / pull-down resistors
Ports P1 and P2 have interrupt capability.

Watch Dog Timer

- A watchdog timer (WDT) is an electronic timer that is used to detect and recover from computer malfunctions. The WDT module restarts the system on occurrence of a software problem (or) if a selected time interval expires.
- During normal operation, the system regularly restarts the WDT to prevent it from elapsing, or "timing out". If the system fails to restart the WDT due to a hardware fault (or) program error, the timer will elapse and generate a timeout signal.
- The timeout signal is used to initiate corrective actions like placing the system in a safe state and restoring normal system operation.

Timer_A2

- Timers are essential to almost any embedded application
- Timers are used to
  - Generate fixed-period events
  - Periodic wakeup
  - Count edges
  - Generate delays
  - Measure time intervals
  - Replacing delay loops with timer calls allows CPU to sleep, consuming less power
- Timers can support multiple capture/compares, PWM outputs, interval timing and extensive interrupt capabilities

Universal Serial Interface (USI) :

The Universal Serial Interface (USI) module supports multiple serial communication modes

UART : Asynchronous, Full duplex
SPI : Serial Peripheral Interface - Synchronous, Full duplex
I2C : Serial Peripheral Interface -Synchronous, Half duplex

Brown out Reset

- The brownout protection comes into action if the supply voltage drops to a dangerous level
- The brownout reset circuit detects low supply voltages such as when a supply voltage is applied to (or) removed from the VCC terminal. The brownout reset circuit resets the device by triggering a POR (Power on Reset) signal when power is applied (or) removed. The **brownout** circuit is used to provide the proper internal reset signal to the device during power ON and power OFF.

- The Supply Voltage Supervisor (SVS) is used to monitor the supply voltage or an external voltage. The SVS can be configured to set a flag or generate a POR reset when the supply voltage or external voltage drops below a user selected threshold.

3.4. ADDRESS SPACE / MEMORY MAPPING OF MSP430

- The MSP430 has Von-Neumann architecture, in which the address space is shared with special function registers (SFRs), peripherals, RAM, and Flash/ROM memory.
- The following shows the memory map of the F2013. Most MSP430 devices have a similar memory map, differing only in the size of the regions for RAM and ROM.
- The data is stored in memory as **Little-endian ordering**, in which the low -byte of data is stored at lower memory address and high-byte of data is stored at the higher address.

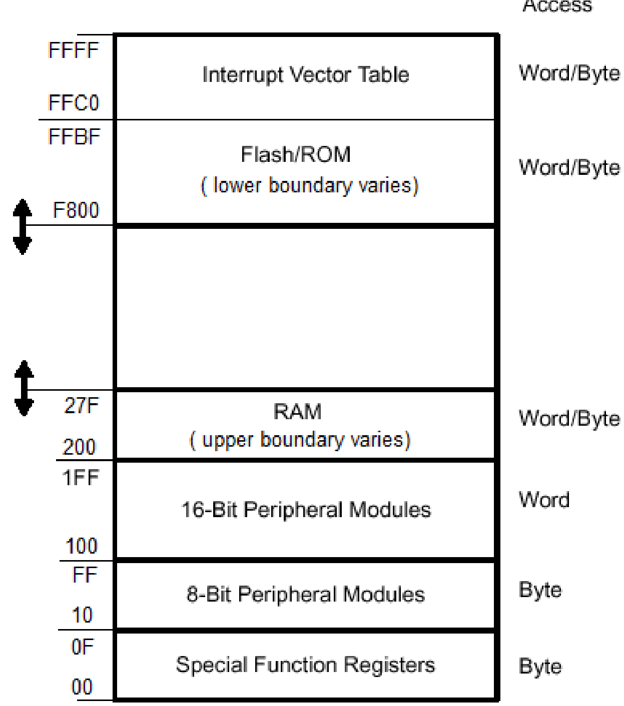


Figure : Memory map of the MSP430F2013

0000 – 000F	Special Function Registers (byte access)
0010 – 00FF	Peripheral registers with Byte access
0100 – 01FF	Peripheral registers with Word access
0200 – xxxx	RAM (upper boundary varies)
0C00 – 0FFF	Bootstrap loader (not in F20xx)
1000 – 10FF	Flash Information memory (available in Flash devices only)
xxxx - FFBF	Flash / ROM (lower boundary varies)
FFC0 - FFFF	Interrupt Vector Table

Special Function Registers :

- The SFRs are located in the lower 16 bytes of the address space.
- Some peripheral functions are configured in the SFRs for enabling and signaling interrupts from peripherals.

Peripheral registers

- 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.
- The address space from 0x0010 to 0x00FF is reserved for 8-bit peripheral modules. The address space from 0x0100 to 0x01FF is reserved for 16-bit peripheral modules.

Random Access Memory:

- RAM isused for storing the data variables.
- RAM always starts at address 0x0200 and the end address of RAM depends on the amount of RAM present on the device.
- The F2013 has 128 Bytes of RAM.

Bootstrap loader (*Flash devices only*) :

- The MSP430 flash devices contain an address space for boot memory, located between addresses 0xC00 through to 0x0FFF.
- The "bootstrap loader" can be used to program the flash memory in addition to the JTAG.
- This memory region is not accessible by other applications, so it cannot be overwritten accidentally.

Information memory (*Flash devices only*) :

- A 256 B 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, (or) variables that should be retained even when power is removed. For example, a printer might remember the settings from when it was last used and keep a count of the total number of pages printed.
- Flash memory may be written one byte or word at a time, but must be erased in segments.
- The information memory is divided into 2 segments (each 128-bytes) in 4xx devices, and 4 segments (each 64 bytes) in 2xx devices.

Flash/ROM /Code memory:

- It is used to store the program, including the executable code itself and any constant data.
- The start address of Flash/ROM depends on the amount of Flash/ROM present on the device. The end address for Flash/ROM is 0x0FFFF for devices with less that 60KB of Flash/ROM.
- The F2013 has 2 KB but the F2003 only 1KB flash memory

Interrupt and Reset Vectors :

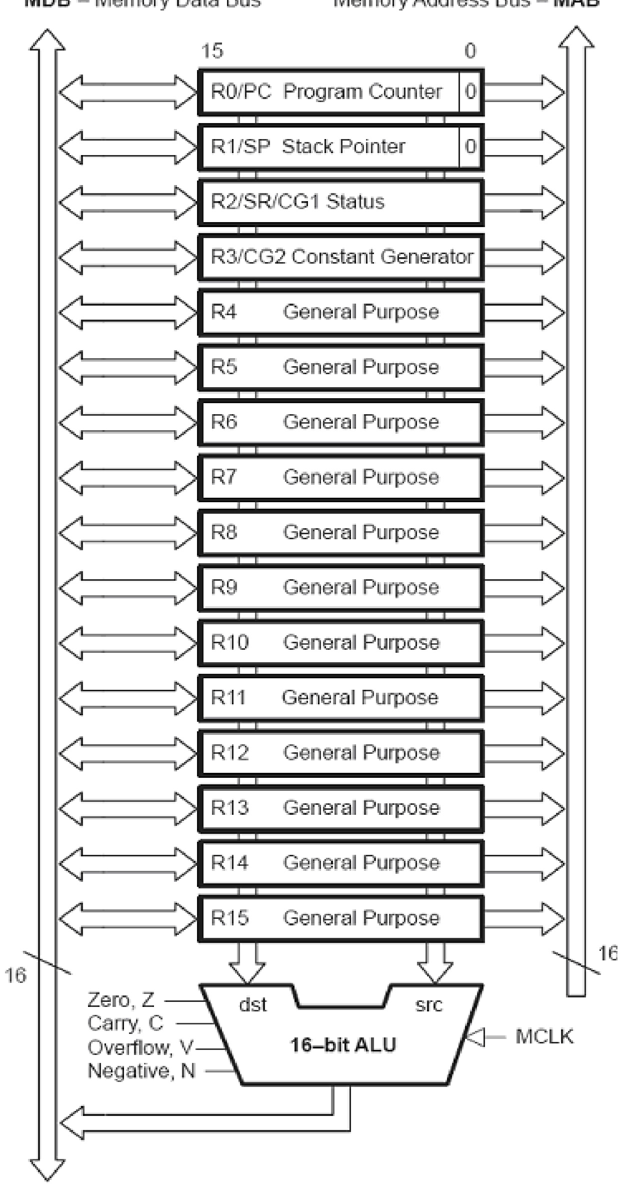
- Used to handle "exceptions," when normal operation of the processor is interrupted or when the device is reset.
- The interrupt vector table is mapped into the upper 16 words of Flash/ROM address space, with the highest priority interrupt vector at the highest Flash/ROM word address (0x0FFFE).

3.5. CPU ARCHITECTURE AND REGISTERS OF MSP430

The central processing unit (CPU) fetches the instructions from memory and executes the instructions. The CPU can run at a maximum clock frequency f_{MCLK} of 16MHz.

The features of MSP430 CPU are

- It has RISC architecture
- It has 16-bit ALU
- It has a set of 16-registers : R0 to R15
- It has 16-bit memory address bus (MAB) and 16-bit memory data bus (MDB)
- Instructions processing on either bits, bytes or words
- Supports 27 core instructions, 24 emulated instructions and 7 addressing modes
- Constant generator provides six most used immediate values and reduces code size.



MSP430 CPU registers

- ✓ The CPU incorporates sixteen 16-bit registers:
  - 4 registers have dedicated functions : R0, R1, R2 and R3
  - 12 working registers for general use : R4 to R15

- ✓ **The dedicated registers are**
  - R0 : Program Counter [PC]
  - R1 : Stack Pointer [SP]
  - R2 : Status Register [SR]
  - **R2/R3: Constant Generator Registers [CG1/CG2]**

- ✓ **The General–Purpose Registers (R4 to R15) :**
  - These are used to store data values, address pointers, or index values and can be accessed with byte or word instructions.

R0: Program Counter (PC) :

- The 16-bit Program Counter (PC/R0) points to the next instruction to be fetched from memory and executed by the CPU.
- It is important to remember that the PC is aligned at even addresses, because the instructions are composed of 1-3 words. Hence the LSB of PC is hard-wired to 0.
- The Program counter is incremented by the number of bytes used by the instruction (2, 4, or 6 bytes, always even).

R1: Stack Pointer (SP) :

- The stack memory is a memory block where the data is stored in LIFO manner.
- The Stack Pointer (SP/R1) holds the address of the stack-top.
- In the MSP430, the stack is allocated at the top of the RAM and grows down towards low addresses.
- The LSB of the stack pointer is hardwired to 0 in the MSP430, which guarantees that it always points to valid words.

The purpose of the stack :

- It used by subroutine calls - to store the Return address (PC value)
- It is used by interrupt – to store the Return address (PC value) and Status information (SR)
- It can be used by compiler for subroutine parameters
- It can be used by user to store data for later use by using PUSH and POP instructions.

Operation of the Stack :

The data can be stored and retrieved from the stack using ***'push'*** and ***'pop'*** instructions

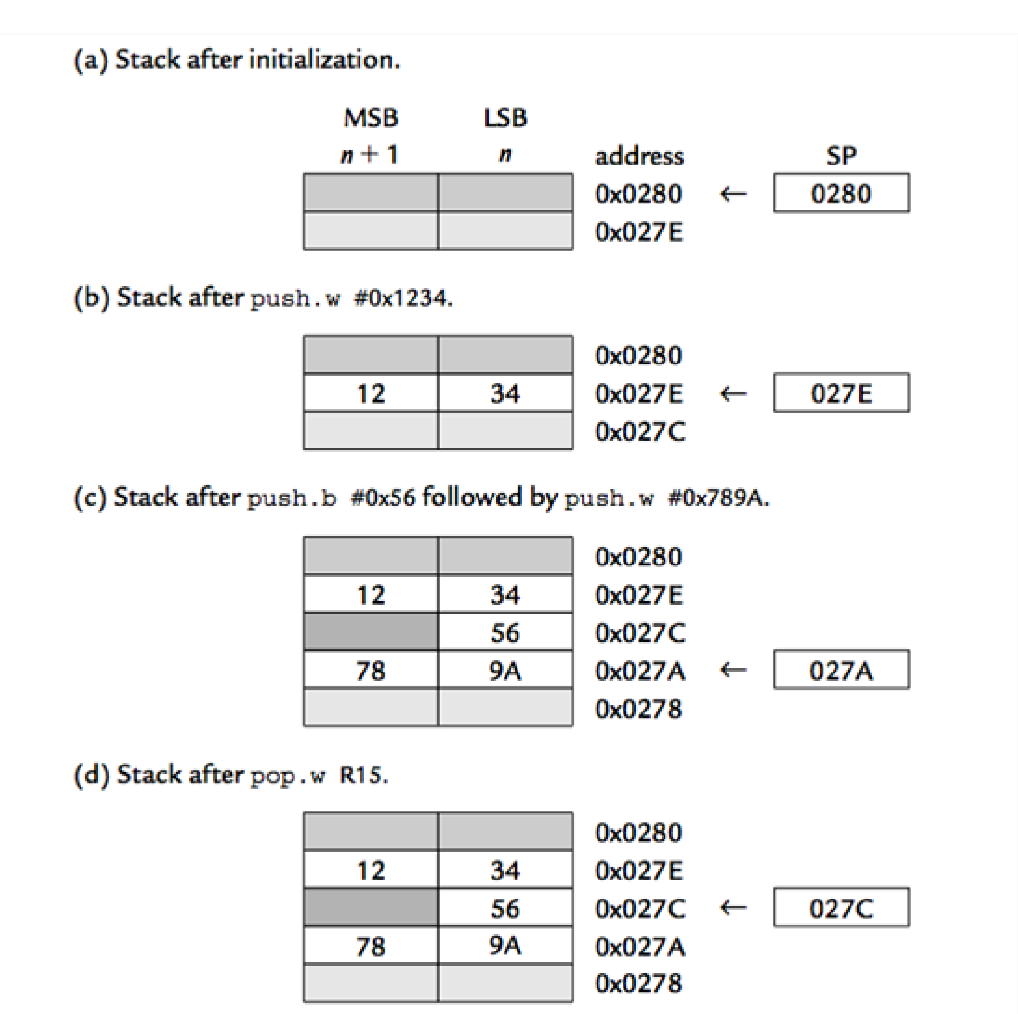
(a) **push src** : *This instruction is used to move the data from source operand to stack.*
During the execution of **'push'** instruction, first the SP is decremented by 2 and then the source content is stored at stacktop

SP ← SP-2
@ SP ← src

(b) **pop dst** : *This instruction is used to move the content of stack top to the destination*
During the execution of **'pop'** instruction, first the content of stack top is moved to the destination and SP is incremented by 2

dst ← @ SP
SP ← SP+2

The operation of the stack is illustrated in the following figure.
The specific addresses are for MSO430F2013 with 128 Bytes of RAM are from 0x0200 to 0x027F.
Hence, Before the execution, the initially the value of SP = 0x0280



R2: Status Register (SR)

The Status Register (SR/R2) stores the status bits and control bits.

15 … 9	8	7	6	5	4	3	2	1	0
reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

Status Flags: Indicate some condition produced by an instruction execution.
These are changed automatically by the CPU depending on the result of an operation.

C : Carry Flag : It is set to 1 , if a carry is generated in an Addition (or) borrow in Subtraction.

Z : ***Zero Flag*** : It is set to 1, if the result of an operation is ZERO.
A common application of Zero flag is to check whether two values are equal or not

N : ***Negative Flag*** : It is used with signed numbers only.
N=1 for negative results and N =0 for positive results.

V: ***Signed overflow flag*** : It is used with signed numbers only
It is set to 1, when the result of a signed operation has overflowed.

Enable Interrupts :
Setting the *general interrupt enable* (GIE) bit enables maskable interrupts.
Clearing the bit disables all maskable interrupts.

Control of Low-Power Modes using Status Register :

The CPUOFF, OSCOFF, SCG0, and SCG1 bits control the mode of operation of the MSP430MCU. All peripherals are fully operational when all bits are clear. Setting combinations of these bits puts the device into one of its low-power modes.

CPUOFF : When set, turns off the MCLK, which stops the CPU.

OSCOFF (Oscillator OFF) : When set, turns off the VLO and LF XT1 crystal oscillator
if LFXT1-CLK is not used for MCLK or SMCLK.

SCG0 (System clock generator 0) : When set, turns off the DCO-DC generator
if DCO-CLK is not used for MCLK or SMCLK.

SCG1 (System clock generator 1) : When set, turns off the SMCLK.

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

- The registers R2 and R3 are the constant generator registers, which can be used to generate **6- commonly used constants**, depends on source addressing mode (As).

Register	Addressing mode for Source operand (As)	Constant
R_3	Register	0
R_3	Indexed	+1
R_3	Register Indirect	+2
R_3	Indirect with Auto increment	-1 (FFFF)
R_2	Register Indirect	**+4**
R_2	Indirect with Auto increment	**+8**

- The advantages of constant generators are:
  - **Reduces the code size**
  - **Emulated instructions can be implemented**
  - No additional code for the six constants
  - No code memory access required to retrieve the constant

- The assembler uses the constant generator automatically if one of the 6- constants is used as an immediate source operand.

Constant Generator - Expanded Instruction Set

- The RISC instruction set of the MSP430 has only 27 core instructions.
- However, the constant generator allows the MSP430 assembler to support 24 additional, 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.

  For example, the single-operand instruction

  clr.w dst is replaced by : **mov.w R3, dst**
  ; where R3=0 (Register addressing As =00)

  inc.w dst is replaced by : **add.w 0(R3), dst**
  ; where 0(R3)=1 (Indexed addressing As =01)

3.6. BLOCK DIAGRAM OF MSP430x5xx

- The MSP430x5xx Series are able to run up to 25 MHz, have up to 512 KB flash memory and up to 66 KB RAM. It includes an innovative power management module for optimal power consumption and integrated USB.
- The MSP4430x5xx microcontroller has 20-bit address bus and 16-bit data bus.
- The functional block diagram of the MSP430 F5529 / F5519 / F5528 consists of the following blocks. The main blocks are linked by the *memory address bus* and *memory data bus.*

1. 16-bit CPU with RISC architecture
2. Unified Clock System (UCS)
3. Embedded Emulation Module (EEM) and JTAG interface
4. Flash memory : 128 KB
5. RAM : 8 KB + 2 KB if USB is disabled
6. Power Management Module (PMM)
7. Watchdog Timer
8. I/O PORTS : P1 to P8
9. Full speed USB (Universal Serial Bus)
10. 32-Bit Hardware Multiplier (MPY32)
11. Timer modules : TA0, TA1, TA2, TB0
12. Real Time Clock (RTC_A)
13. Cyclic Redundancy Check (CRC)
14. Universal Serial Communication Interface (Two channels: USCI-A & USCI-B)
15. ADC12_A
16. Reference Module (REF)
17. Comparator (Comp_B)
18. Direct Memory Access (DMA) Controller Module

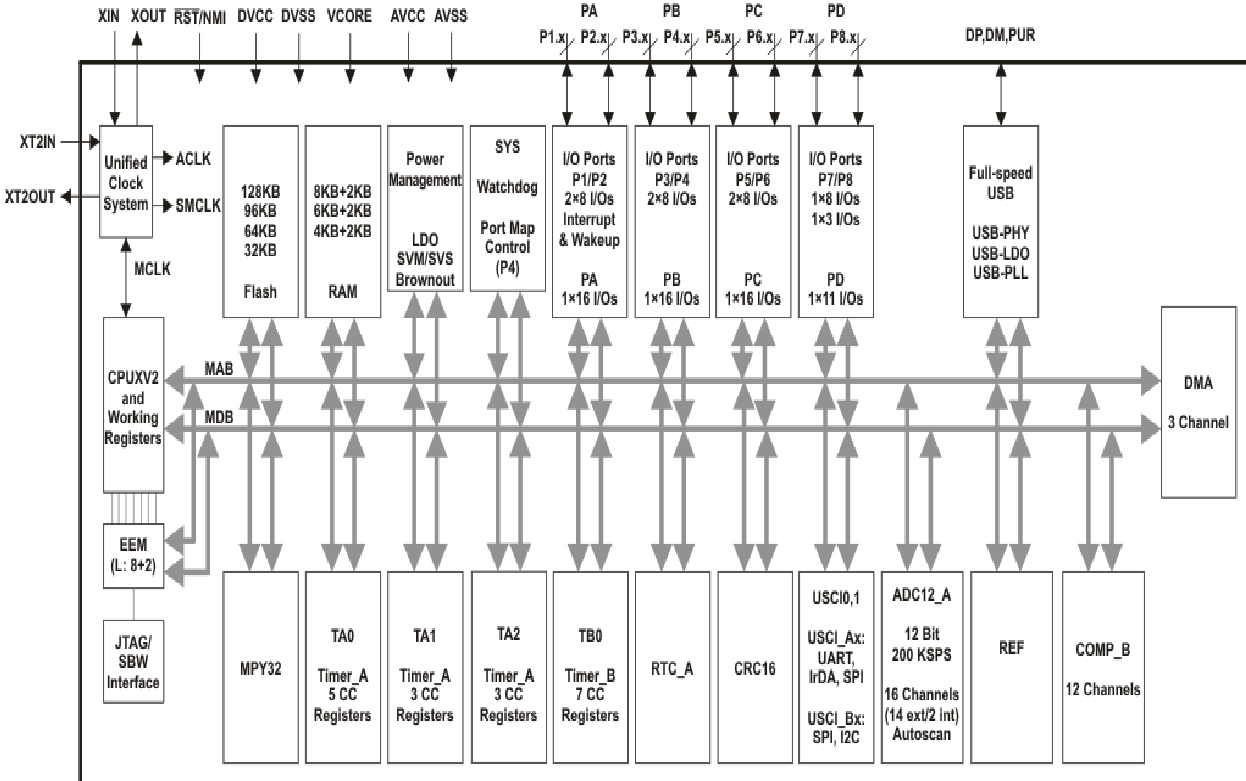


*Fig : Block diagram of **MSP430 F5529**/ F5528 / F5519*

(1) 16-bit CPUX

- It consists of 16-bit ALU, set of 16-registers (R0 – R15) and Logic needed to decode and execute the instructions. Note that all registers are 20-bit wide, except Status register.
- The CPU has 16-bit RISC architecture
- Instructions processing on either bits, bytes or words
- Supports 27 instructions and 7 addressing modes.
- It can address the complete address range without paging
    - CPU clock : 25 MHz
    - Operating voltage : 1.8–3.6 V
    - Active operation : 165 µA/MIPS
    - RTC mode operation : 2.5 µA
    - RAM retention : 0.1 µA
    - Fast wake-up from standby mode in less than 5 µs

(2) Unified Clock System (UCS) :

- The UCS module provides the various clocks for CPU and Peripherals
- The UCS module supports low system cost and ultra-low power consumption.
- It provides 3- internal clock signals : MCLK, SMCLK and ACLK. Using three internal clock signals, the user can select the best balance of performance and low power consumption.

MCLK : Master clock is used by CPU and system
SMCLK : Subsystem Master clock is distributed to high speed peripherals
ACLK : Auxiliary clock is also distributed to low speed peripherals

(3) Embedded Emulation Module (EEM) and JTAG interface

- EEM is accessed and controlled through either 4-wire JTAG mode or Spy-Bi-Wire mode. The emulation, JTAG interface and Spy-Bi-Wire interface are used to communicate with a desktop computer when downloading a program and for debugging.

(4) Flash memory : 128 KB

Flash memory is used to store the programs and constant variables
The size of the Flash memory varies from one device to another device

(5) RAM : 8 KB + 2 KB if USB is disabled

RAM is used to store the temporary data (Read/Write memory)
The size of the Flash memory varies from one device to another device

(6) Power Management Module (PMM):

- The PMM manages all functions related to the power supply and its supervision for the device.
- The PMM uses an integrated low-drop-out voltage regulator (LDO)
- The Supply Voltage Supervisor (SVS) is used to monitor the supply voltage or an external voltage. The SVS can be configured to set a flag or generate a Power on Reset (POR) when the supply voltage drops below a user selected threshold.
- The brownout reset circuit detects low supply voltages such as when a supply voltage is applied to (or) removed from the VCC terminal. The brownout reset circuit resets the device by triggering a POR signal when power is applied (or) removed.

(7) Watch Dog Timer

- A watchdog timer (WDT) is an electronic timer that is used to detect and recover from computer malfunctions. The WDT module restarts the system on occurrence of a software problem (or) if a selected time interval expires.
- During normal operation, the system regularly restarts the WDT to prevent it from elapsing, or "timing out". If the system fails to restart the WDT due to a hardware fault (or) program error, the timer will elapse and generate a timeout signal.
- The timeout signal is used to initiate corrective actions like placing the system in a safe state and restoring normal system operation.

(8) I/O PORTS : P1, P2, P3, P4, P5, P6, P7 & P8

- MSP430F5529 has 8-digital I/O ports : P1 to P8.
- The I/O Ports P1 to P7 have **8- I/O pins** and P8 has **3- I/O pins**.
- Individual ports can be accessed as byte wide ports or can be combined into word wide ports as PA(P1&P2), PB(P3&P4), PC(P5&P6) and PD(P7&P8)
- Each I/O pin is individually configurable for input (or) output direction, and each I/O line can be individually read or written to.
- All ports have individually configurable pull-up (or) pull-down resistors
- Ports P1 and P2 pins have interrupt capability.

(9) Full speed USB (Universal Serial Bus)

- Full-speed integrated USB transceiver (PHY)- Physical layer Interface
- Supports control, interrupt, and bulk transfers
- Integrated 1.8 V and 3.3 V low drop-out (LDO) linear regulator
- Integrated programmable PLL
- Highly flexible clock frequencies
- 2 KB of dedicated USB buffer.
- If USB is disabled, 2 KB buffer space is mapped into RAM and USB pins can be used as GPIO pins

(10) 32-Bit Hardware Multiplier (MPY32)

- The MPY32 is a peripheral and is not part of the CPU. This means its activities do not interfere with the CPU activities.
- It performs Signed and Unsigned multiplications
- 8-bit/16-bit/24-bit/32-bit operations

(11) Timers

- The timers are used to generate fixed-period events, periodic wakeup, count edges, generate delays and measure time intervals. Replacing delay loops with timer calls allows CPU to sleep, consuming much less power.
- Timers can support multiple capture/compares, PWM outputs, interval timing and extensive interrupt capabilities.
- The MSP430F5529 has 4- timer modules

  16-bit Timer TA0 : with 5 - CC registers
  16-bit Timer TA1 : with 3 - CC registers
  16-bit Timer TA2 : with 3 - CC registers
  16-bit Timer TB0 : with 7 - CC registers

(12) Real Time Clock (RTC_A)

- Configurable for real-time clock with calendar function or general-purpose counter.
- RTC module provides seconds, minutes, hours, day of week, day of month, month, and year in real-time clock with calendar function.
- Interrupt capability

(13) Cyclic Redundancy Check (CRC) :

- The cyclic redundancy check (CRC) module provides a signature for a given data sequence.
- The signature is generated through a feedback path from data bits 0, 4, 11 & 15

(14) Universal Serial Communication Interface :

- Two channels : USCI-A & USCI-B
- Each channel consists of Two ports : 0 & 1

USCI-A0 & USCI-A1 – supports UART, IrDA and SPI

USCI-B0 & USCI-B1 – supports I2C and SPI

(15) ADC12_A

- The ADC12_A is a high-performance 12-bit analog-to-digital converter (ADC).
- The module implements a 12-bit SAR core, sample select control and a 16-word conversion-and-control buffer.
- 16-channels : 14- external & 2- internal
- 200 KSPS
- Internal Reference
- S/H and Auto scan feature

(16) Reference Module (REF)

- The REF module is a general purpose reference system that is used to generate voltage references required for other peripheral modules such as digital-to-analog converters, analog-to digital ,converters, or comparators.
- It generates reference voltages : 1.5 V, 2.0V, 2.5V … selectable references

(17) Comparator (Comp_B)

- Comp_B is an analog voltage comparator.
- Comp_B covers general comparator functionality for up to 16 channels.
- The Comp_B module supports precision slope analog-to-digital conversions, supply voltage supervision, and monitoring of external analog signals.

(18) Direct Memory Access (DMA) Controller Module

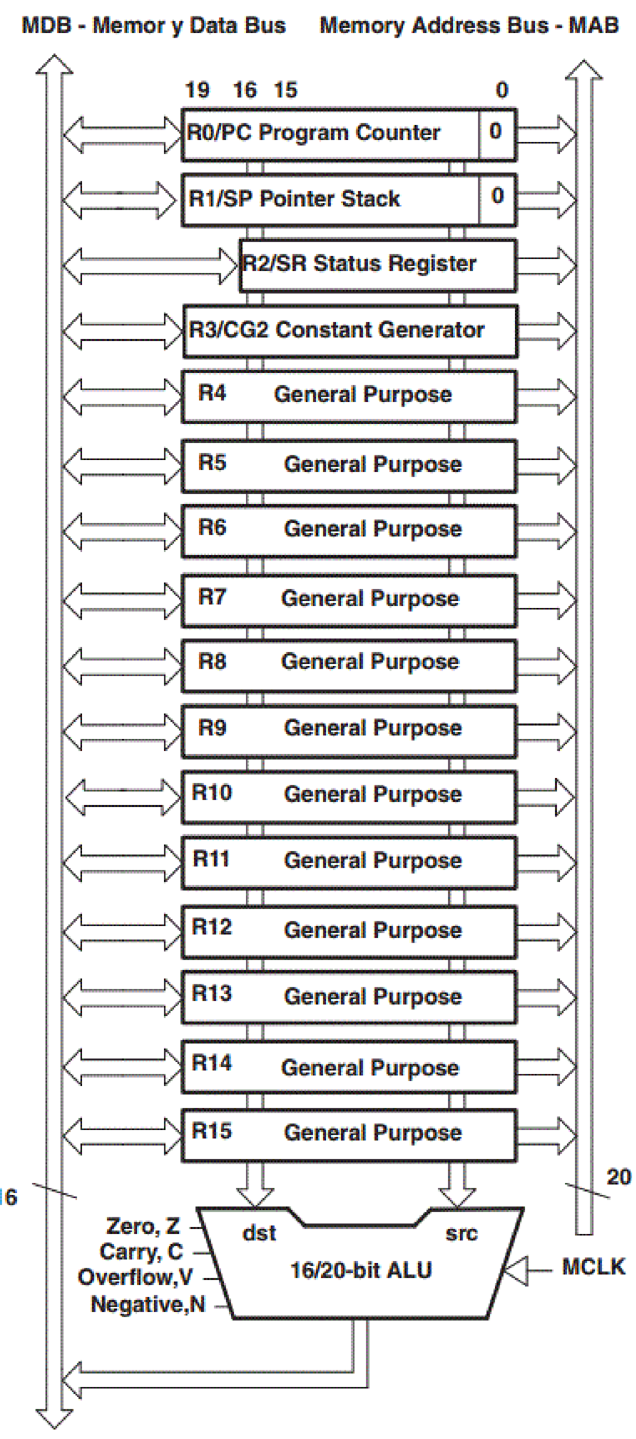
- The direct memory access (DMA) controller module transfers data from one address to another without CPU intervention.
- Devices that contain a DMA controller may have up to eight DMA channels available. (The MSP430F5529 has 3-channels)
- It can also reduce system power consumption by allowing the CPU to remain in a low-power mode, without having to awaken to move data to or from a peripheral.

ADDRESS SPACE / MEMORY MAPPING OF MSP430x5xx

00000 h - 00FFF h	Peripherals Addresses	4 KB	4 KB
01000 h - 017FF h	Bootstrap Loader (BSL) memory (Flash)	2KB (512B *4 segments)	BSL0 (512 B)
			BSL1 (512 B)
			BSL2 (512 B)
			BSL3 (512 B)
01800 h - 019FF h	Flash Information memory	512 B (128B *4 segments)	Info A (128 B)
			Info B (128 B)
			Info C (128 B)
			Info D (128 B)
01C00 h to 023FF h	USB RAM	2 KB	Sector 7 (2KB)
02400 h to 043FF h	RAM	8 KB (2KB * 4 sectors)	Sector 0 (2KB)
			Sector 1 (2KB)
			Sector 2 (2KB)
			Sector 3 (2KB)
04400 h to 243FF h	Code memory (Flash)	128 KB (32 KB *4 banks)	Bank A (32 KB)
			Bank B (32 KB)
			Bank C (32 KB)
			Bank D (32 KB)
0FF80 h – 0FFFF h	Interrupt Vector Table	128 KB	128 KB

Note : For details, Refer Page No. 9

CPU ARCHITECTURE AND REGISTERS OF MSP430x5xx



Note : For details, Refer Page No. 11

3.9. ADDRESSINGMODES OF MSP430

Instruction is a command given to the processor to perform a specific operation on specified data.

Every instruction has 2- parts

Op-code → The operation to be done
Operand → The data to be operated

Instruction

OPCODE	OPERAND

The method of specifying data to be operated by an instruction is called as addressing mode.
The MSP430 supports the following addressing

1. Register mode :

- In this addressing mode, the data is available in any one of the registers of CPU.
- This is available for both source and destination

Ex: **mov.w R5, R6** ; copies word from R5 to R6

2. Register Indirect mode :

- In this mode, the address of the data is available in the register.
- Indirect register addressing is shown by the symbol @ in front of a register such as @Rn. In other words, Rn is used as a pointer. *This mode is available for only source operand.*

Ex: **mov.w @R5, R6** ; load word from address (R5) into R6
If R5 = 0004 then, the content of memory address 0004 is moved to R6

3. Register Indirect with Auto-increment mode :

- In this addressing mode, the address of the data is available in the register and it is automatically incremented by 1 for BYTE operation and by 2 for WORD operation.
- *This mode is available for only source operand.*

Ex: **mov.w @R5+, R6** ; load word from address (R5) into R6 and increment R5 by 2
If R5 = 0004 then, the content of memory address 0004 is moved to R6 and the pointer R5 is incremented by 2.

4. Indexed mode :

- In this addressing mode, the address of data is the sum of Register and Displacement
- This mode is available for both source and destination

Ex: **mov.b 3(R5), R6** ; load byte from address (3+R5) into R6
If R5 = 0004 then, the content of memory address 0007 is moved to R6

5. Absolute mode :

- In this addressing mode, the absolute address of the data is available in the instruction.
- Absolute addressing is shown by the prefix " & "
- This mode is similar to indexed mode, in which Status Register is used as Index register.

Ex: **mov.b &0245H, R6** ; *copy data from memory address **0245 H** into R6*

The assembler replaces the above instruction by the indexed form, where SR=0 (Constant generator)

mov.b 0245(SR), R6 ; *copy data from address (SR+ **0245**) to register R6*

6. Symbolic mode / PC Relative :

- This mode is similar to Indexed mode, where PC is used as index register. Hence the address of data is the sum of PC and Displacement.
- The assembler calculates the Displacement (relative offset address) of the data

Ex: **mov.w LABLE, R6** ; load word pointed by LABLE into R6

The assembler replaces the above instruction by the indexed form

mov.w X(PC), R6 ; load word pointed by (PC+X) into R6
where X = Displacement = LABLE – PC

7. Immediate mode :

- In this addressing mode, the data is available immediately after the instruction.
- The PC is automatically incremented after the instruction is fetched and therefore points to the following word. Hence it is a special case of Indirect auto-increment mode.
- *This mode is available for only source operand.*

Ex: **mov.w #1234, R6** ; move the immediate constant #1234 into R6.

The assembler replaces the above instruction as

mov.b @PC+, R6 ;

3.10. INSTRUCTION FORMATS OF MSP430

Instruction is a command given to the processor to perform a specific operation on specified data.

Every instruction has 2- parts

Op-code → The operation to be done
Operand → The data to be operated

Instruction

OPCODE	OPERAND

There are 3- core instruction formats of MSP430

Format-I : Instructions with Two-operands

Ex: add.w R5, R6
add.w @R5, R6
add.w 3(R5), R6

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				S-reg				Ad	b/w	As		D-reg			

Format-II : Instructions with Single Operand

Ex: push.w R5
pop.w R6
rrc.w R5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode (4 + 5 bits)									b/w	As		D/S-reg			

Format-III : Jump Instructions

Ex: jmp LABEL
jc UP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode (4 + 2 bits)						10-bit, 2's complement PC offset									

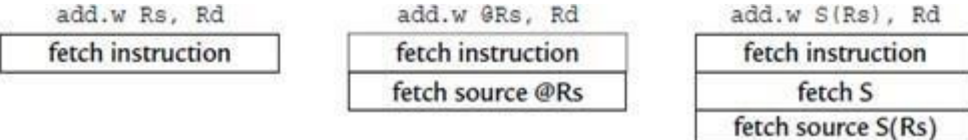
Opcode : These bits represents the type of operation to be performed
S-reg : Source Operand Register [0000 for R0, …… 1111 for R15]
D-reg : Destination Operand Register [0000 for R0, …… 1111 for R15]

As : Addressing mode for Destination Operand
Ad : Addressing mode for Destination Operand
b/w: Byte / Word operation [0- Word , 1- Byte]

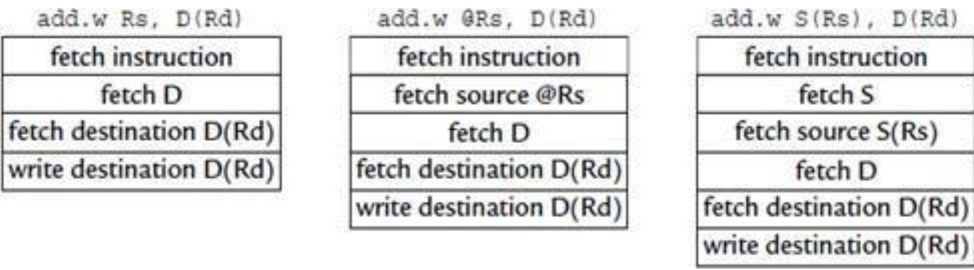
INSTRUCTION TIMINGS OF MSP430

- The number of CPU clock cycles required for an instruction depends on the instruction format and addressing modes. In MSP430, the number of clock cycles refers to the MCLK.
- The instruction timings for some instructions are given below.

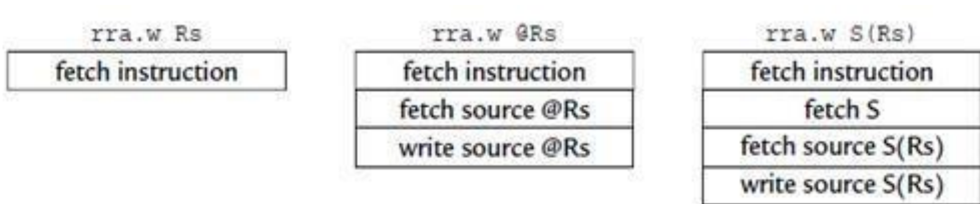
(a) Instructions with Two operands (Format-I) and destination is Register



(b) Instructions with Two operands (Format-I) and destination is Indexed



(c) Instructions with One operand (Format-II)



add.w Rs, Rd : It takes only ONE cycle

- It takes only One cycle to fetch the instruction word itself, because both Source and Destination are in CPU registers

add.w @Rs, Rd : It takes TWO cycles.

1st cycle → to fetch the instruction
2nd cycle → to read the data from memory location pointed by Rs

add.w S(Rs), Rd : It takes THREE cycles.

1st cycle → to fetch the instruction
2nd cycle → to fetch the Source displacement 'S'
3rd cycle → to read the data from memory location pointed by Rs+S

add.w Rs, D(Rd) : It takes FOUR cycles

1st cycle → to fetch the instruction
2nd cycle → to fetch the Destination displacement 'D'
3rd cycle → to read the data from memory location pointed by Rd+D
4th cycle → to store the result at destination memory location pointed by Rd+D

add.w @Rs, D(Rd) : It takes FIVE cycles

1st cycle → to fetch the instruction
2nd cycle → to read the data from memory location pointed by Rs
3rd cycle → to fetch the Destination displacement 'D'
4th cycle → to read the data from memory location pointed by Rd+D
5th cycle → to store the result at destination memory location pointed by Rd+D

add.w S(Rs), D(Rd) : It takes SIX cycles

1st cycle → to fetch the instruction
2nd cycle → to fetch the Source displacement 'S'
3rd cycle → to read the data from memory location pointed by Rs+S
4th cycle → to fetch the Destination displacement 'D'
5th cycle → to read the data from memory location pointed by Rd+D
6th cycle → to store the result at destination memory location pointed by Rd+D

No. of MCLK cycles required for Typical Instructions

Format	Source		
	R	@R	S(R)
Format –I with destination is Register	1	2	3
Format –I with destination is Indexed	4	5	6
Format-II	1	3	4

3.11. INSTRUCTION SET OF MSP430

- 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.

1. Movement Instructions (Data Transfer)
2. Arithmetic Instructions
3. Logic Instructions
4. Shift and Rotate Instructions
5. Control Transfer instructions (Branch/Subroutine/Interrupt)

(1) Data transfer / Movement instructions

	Instruction	*Operation*	*Example*
1	**mov.w src, dst**	Copies data from source to destination *dst* ← *src*	*mov.w R5, R6* R6 ← R5
2	**push.w src**	Push data onto stack *(first the SP is decremented by 2 and the source content is stored at stacktop)* *SP ← SP-2* *@ SP ← src*	*push.w R5* SP ← SP-2 @ SP ← R5
3	**pop.w dst**	Pop data from stack *(first the content of stacktop is moved to destination and SP is incremented by 2)* *dst ← @ SP* *SP ← SP+2*	*pop.w R6* R6 ← @ SP SP ← SP+2

(2) Arithmetic instructions

(a) Binary Arithmetic Instructions with Two operands

	Instruction	*Operation*	*Example*
1	**add.w src, dst**	Add the content of source to destination *dst ← dst + src*	*add.w R5, R6*
2	**addc.w src, dst**	Add with carry *dst ← dst + (src + C)*	*addc.w R5, R6*
3	**adc.w dst**	Add carry bit to the destination *dst ← dst + C*	*adc.w R6*
4	**sub.w src, dst**	Subtract the content of source from destination *dst ← dst - src*	*sub.w R5, R6*
5	**subc.w src, dst**	Subtract with borrow *dst ← dst –(src + C)*	*subc.w R5, R6*
6	**sbc.w dst**	Subtract borrow bit from the destination *dst ← dst – C*	*sbc.w R6*
7	**cmp.w src,dst**	Compares source and destination. Performs (dst – src), but Only flags are changed If dst > src : C=0, Z=0 If dst < src : C=1, Z=0 If dst = src : C=0, Z=1	*cmp.w R5,R6*

(b) Arithmetic Instructions with One operand :

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

	Instruction	*Operation*	*Example*
1	**clr.w dst**	Clear destination *dst* ← *0*	*clr.w R6* R6 ← 0
2	**dec.w dst**	The content of destination is decremented by 1 *dst* ← *dst – 1*	*dec.w R6* R6 ← R6 - 1
3	**decd.w dst**	Double decrement The content of destination is decremented by 2 *dst* ← *dst – 2*	*decd.w R6* R6 ← R6 – 2
4	**inc.w dst**	The content of destination is inccremented by 1 *dst* ← *dst + 1*	*inc.w R6* R6 ← R6 + 1
5	**incd.w dst**	Double increment The content of destination is inccremented by 2 *dst* ← *dst + 2*	*incd.w R6* R6 ← R6 + 2
6	**tst.w dst**	Test (compare with zero) Performs (dst – 0), but Only flags are changed If dst > 0 : C=0, Z=0 If dst < 0 : C=1, Z=0 If dst = 0 : C=0, Z=1	*tsd.w R6*

(c) Decimal Arithmetic Instructions :

These instructions are used to perform BCD addition

	Instruction	*Operation*	*Example*
1	**dadd.w src, dst**	Perfroms the decimal addtion of destination and source with carry *dst* ← *dst + src + C*	*dadd.w R5, R6* R6 ← R6 + R5+C
2	**dadc.w dst**	Performs the decimal addition of destination and carry. *dst* ← *dst + C*	*dadc.w R6* R6 ← R6 + C

(3) Logic instructions

(a) Logic Instructions with Two operands

	Instruction	*Operation*	*Example*
1	**and.w src, dst**	Performs bit-wise logic AND operation *dst ← dst AND src*	*and.w R5, R6*
2	**xor.w src, dst**	Performs bit-wise logic Ex-OR operation *dst ← dst XOR src*	*xor.w R5, R6*
3	**bit.w src, dst**	Performs **bit-wise Test operation** It performs Logic AND operation, But Only flags are affected	*bit.w R5, R6*
4	**bis.w src, dst**	***Set bits in destination*** The source operand and the destination operand are logically ORed. The result is placed into the destination. The source operand is not affected. *dst ← dst OR src*	*bis.w R5, R6*
5	**bic.w src, dst**	***Clear bits in destination*** The inverted source operand and the destination operand are logically ANDed. The result is placed into the destination. *dst ← dst AND ~src*	*bic.w R5, R6*

(b) Logic Instructions with ONE operand

	Instruction	Operation	Example
1	**inv.w dst**	*Invert destination* Performs bit-wise NOT operation (1's complement) *dst ← ~dst*	*inv.w R6*

(c) Byte manipulation

	Instruction	*Operation*	*Example*
1	**swpb dst**	*Swap upper and lower bytes* The high and the low byte of the operand are exchanged dst.15:8 ↔ dst.7:0	*swpb R6*
2	**sxt dst**	*Extend sign of lower byte* The sign of the low byte of the operand is extended into the high byte *dst. 15:8 ← dst.7* If dst.7 = 0: high byte = 00 H afterwards If dst.7 = 1: high byte = FF H afterwards	*sxt R6*

(d) Operations on Bits in Status Register

These instructions are used to set orclear the falgs in Status Register.
All these instructions are emulated instructions.

	Instruction	*Operation*	*Example*
1	**clrc**	Clear Carry bit	*C = 0*
2	**clrn**	Clear Negative bit	*N = 0*
3	**clrz**	Clear Zero bit	*Z = 0*
4	**setc**	Set Carry bit	*C = 1*
5	**setn**	Set Negative bit	*N = 1*
6	**setz**	Set Zero bit	*Z = 1*
7	**dint**	Disable General Interrupts	*GIE = 0*
8	**eint**	Enable General Interrupts	*GIE = 1*

(4) Shift and Rotate instructions

Instruction	Description	Operation
rla dst	Arithmetic shift Left	MSB LSB C 0
rra dst	Arithmetic shift Right	MSB LSB C
rlc dst	Rotate Left through Carry	MSB LSB C
rrc dst	Rotate Right through Carry	MSB LSB C

(5) Control Trasfer instructions

	Instruction	*Operation*	*Example*
1	**br src**	Branch (go to)	*PC ← src*
2	**call src**	Call Subroutine	*SP ← SP-2* *@ SP ← PC*
3	**ret**	Return from Subroutine	*PC ← @ SP* *SP ← SP+2*
4	**reti**	Return from Interrupt	*SR ← @ SP* *SP ← SP+2* *PC ← @ SP* *SP ← SP+2*
5	**nop**	No operation (consumes single cycle)	
	JUMP Instructions		**Condition**
1	**jmp label**	Unconditional Jump	*Jump to specicifed location*
2	**jc / jlo label**	Jump if carry / Jump if lower	*Jump if C =1*
3	**jnc / jhs label**	Jump if not carry / Jump higher or same	*Jump if C =0*
4	**jz / jeq label**	Jump if zero / Jump if equal	*Jump if Z =1*
5	**jnz / jne label**	Jump if not zero / Jump if not equal	*Jump if Z =0*
6	**jn label**	Jumpif negative	*Jump if N =1*
7	**jge label**	Jump if greater or equal (signed values)	*Jump if (N xor V) =0*
8	**jl label**	Jump if less than (signed values)	*Jump if (N xor V) =1*

3.12. SAMPLE EMBEDDED SYSTEM USING MSP430

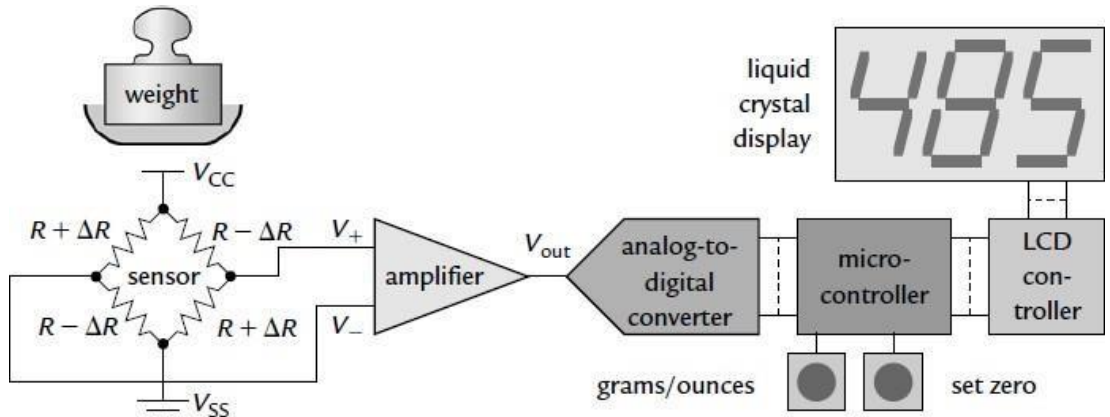


Figure : Weighing machine with a liquid crystal display, broken down into individual functions.

The letters MSP stand for *mixed signal processor*, which is a reminder that many practical applications require analog inputs. There is a selection of analog-to-digital converters with a resolution of up to 16 bits. An example of a system where this choice is important is the weighing machine shown in above Figure. It includes the following functional blocks:

- The sensor has four resistive elements arranged as a Wheatstone bridge. Ideally, this is balanced when there is no load, giving $V^{+} = V^{-}$. Two of the resistances increase and two decrease when a weight is placed on the scale pan, driving the bridge out of balance.
- A differential amplifier magnifies the difference in voltage between its input terminals, giving $V\text{out} = A(V^{+} - V^{-})$, where A is the gain.
- The analog output of the amplifier is converted to a binary value by A/D converter.
- The microcontroller multiplies the input by an appropriate factor so that the display gives the weight in grams or ounces and subtracts an offset so that the display reads zero when no weight is present. It also reads the buttons and supervises the complete system.
- There is a serial interface between the microcontroller and the LCD contoller.

This system clearly needs a lot of components, including several integrated circuits. In contrast, the whole system can be constructed from a sensor, an MSP430F42x, a simple LCD without a controller, and a couple of decoupling capacitors.

The MSP430x4xx family drives segmented LCDs directly, which eliminates the need for a controller. Several devices contain ADCs with high-resolution, differential inputs, which would work directly from the sensor without the need for an amplifier. The microcontroller can also manage the power drawn by the circuit so that the processor would be switched off when it was not needed and the whole system shut down after a period of inactivity.

UNIT- 4 : PERIPHERALS OF MSP430

1. I/O Ports (GPIO pins)
2. Pull-up and Pull-down Resistors concepts
3. Interrupts and Interrupt processing
4. Clock system
5. Low power modes and Active Vs Stand by currents
6. Watchdog Timer (WDT)
7. Real Time Clock (RTC)
8. Basic Timer_!
9. Timer Modules : Timer_A/B
10. PWM output
11. Measurement of time and frequency using Timers
12. Analog interfacing – ADC
13. Comparator
14. Data transfer using DMA

I/O PORTS (GPIO PINS)

- MSP430x5xx devices have up to 12 digital I/O ports : P1 to P12
- Most ports have 8- I/O pins, however some ports may contain less.
- Each I/O pin is individually configurable for input (or) output direction, and each I/O line can be individually read or written to.
- Ports P1 and P2 always have interrupt capability.
- All ports have individually configurable pull-up (or) pull-down resistors
- Individual ports can be accessed as Byte wide ports (or) can be combined into Word wide ports i.e PA= P1&P2, PB = P3&P4 …etc

Each port is assigned several 8-bit registers that control the function of the pins

Port Control Register	Description
PxSEL (Port Selection)	Selects either digital I/O (or) an alternate function 0 - digital I/O 1- Altenate function
PxDIR (Port Direction)	Configures the pin for input mode (or) output mode 0 - input mode 1- ouput mode
PxIN (Port Input)	Read data from **input port** pins, if they are configured as GPIO
PxOUT (Port Output)	Send data to **output port** pins, if it is configured as GPIO
PxREN (Port Resistor Enable)	Enables pull-up / pull-down resistors on input pins. 1 → Enables pull-up / pull-down resistors 0 → Disables pull-up / pull-down resistors

If the pin is configured in input mode and PxREN is enabled then,
PxOUT register selects whether the resistors are pull-up (1) or pull-down (0)

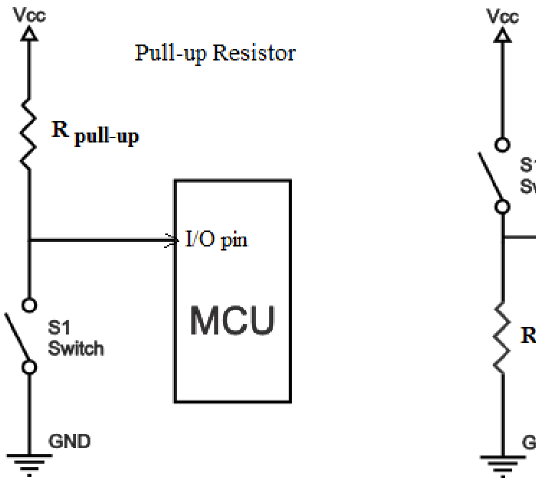
PxDIR	PxREN	PxOUT	I/O Configuration
0	0	x	**Input**
0	1	0	**Input** with Pull-down
0	1	1	**Input** with Pull-UP
1	x	x	Output

Some GPIO pins of MSP430 have interrupt capability, typically on Ports 1 and Port 2 pins.
The registers controlling these options are as follows

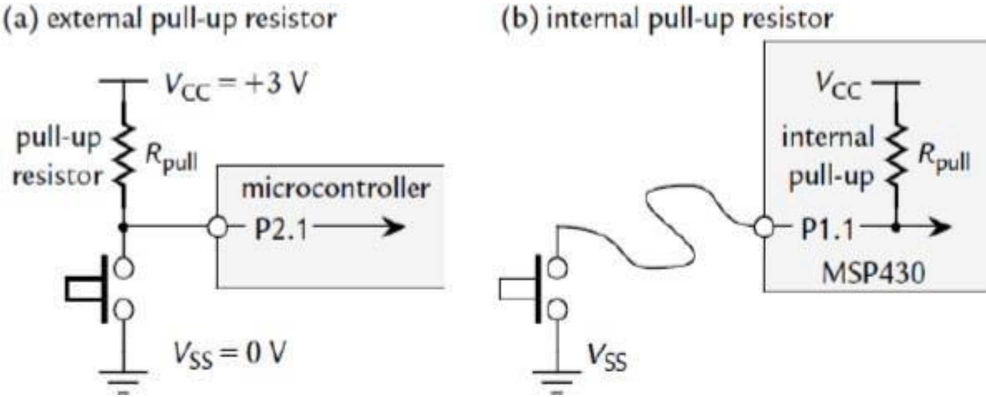
Port Control Register	Description
PxIE (Interrupt Enable)	Enable or Disables the interrupt for that particular pin 1 – Enable 0- Disable the Interrupt
PxIES (Interrupt Edge Select)	Selects either falling-edge (or) rising-edge interrupt 1 – falling edge 0- rising edge
PxIFG (Interrupt Flag)	Set whenever the interrupt is detected on a particular pin

PULL-UP & PULL-DOWN RESISTORS CONCEPTS

- The Pull-up and Pull-down resistors are used to maintain proper logic levels at input pins under all conditions.
- When no external devices are connected to an input pin, it doesn't mean that it is a logical 'ZERO'. Hence, Pull-up and Pull-down resistors ensure that the inputs to the micro-controller settle at expected logic levels, if external devices are disconnected (or) high-impedance.



- ✓ Pull-up resistors are used to pull-up the logic level at the input pin to logic 'HIGH' state, as shown in the Fig.(a). If there was no pull-up resistor, the MCU's input would be floating when the switch is open.
- ✓ Pull-down resistors are used to pull-down the logic level at input pin to logic 'LOW' state, as shown in the Fig.(b). If there was no pull-down resistor, the MCU's input would be floating when the switch is open.
- ✓ Pull-up and pull-down resistors are often used when interfacing a switch or some other input with a microcontroller. Hence, most microcontrollers have internal pull up/down resistors to reduce the no. of external components.



Pull-up or pull-down resistors can be activated by setting bits in the PxREN registers, provided that the pin is configured as an input.

Application of Pull up/down Resistors:

(1) Interfacing of switches / other input devices with microcontrollers
(2) Interfacing of A/D converters
(3) I2C protocol bus

4.3. MSP430 INTERRUPTS AND PROCESSING OF INTERRUPTS

Interrupt :

- Interrupt is an event that causes the CPU to stop the normal program execution.
- Interrupts are generated by external devices connected to the microcontroller (or) Interrupts can be requested by internal peripheral modules in the core of the MCU, such as the clock generator, Timer, ADC, Serial UART.. etc.

Why Interrupts?

- ✓ To execute Urgent tasks
- ✓ To provide the service for the peripherals
- ✓ To wake-up the CPU from sleep
- ✓ To process the Infrequent tasks

Maskable & Non-maskable interrupts

- Most interrupts are maskable, which means that they are effective only if GIE =1 in the status register (SR). They are ignored if GIE is clear.
- However, the non-maskable interrupts cannot be suppressed by clearing GIE.
- There are 3- non maskable interrupts:
  - ✓ Oscillator fault (OF IFG)
  - ✓ Access violation to flash memory (ACCV IFG)
  - ✓ An active edge on the external RST/NMI pin

Interrupt Service Routine [ISR]

The program to handle an interrupt is called an *interrupt handler or Interrupt Service Routine (ISR).*

Interrupt Vector Table [IVT]

- Interrupt Vector Table is a memory block which contains the addresses of ISR.
- The vector table is at a fixed location, but the ISR programs can be located anywhere in memory.
- The MSP430 uses *vectored interrupts.* Each ISR has its own vector, which is stored at a predefined address in a *vector table* at the end of the program memory (0xFFC0–0xFFFF)

Processing of Interrupts:

When the INTR request is received, the CPU performs the following actions

1. The CPU completes the execution of current instruction.
2. The PC is pushed onto the stack.
3. The SR is pushed onto the stack.
4. The interrupt with the highest priority is selected, if multiple interrupts are enabled
5. The SR is cleared, which has two effects. First, further maskable interrupts are disabled because the GIE bit is cleared; Second, it terminates any low-power mode.
6. The PC is loaded with the address of ISR, which is available in IVT, and the CPU starts to execute the interrupt service routine.

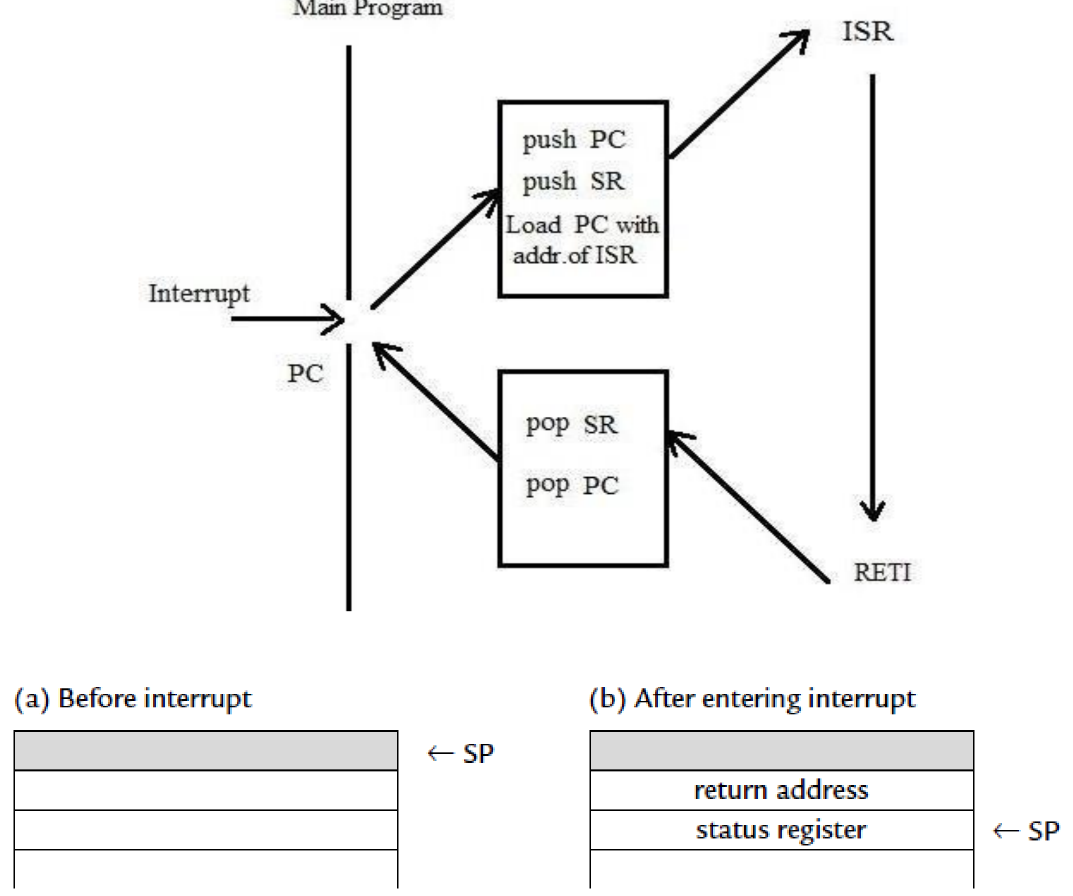


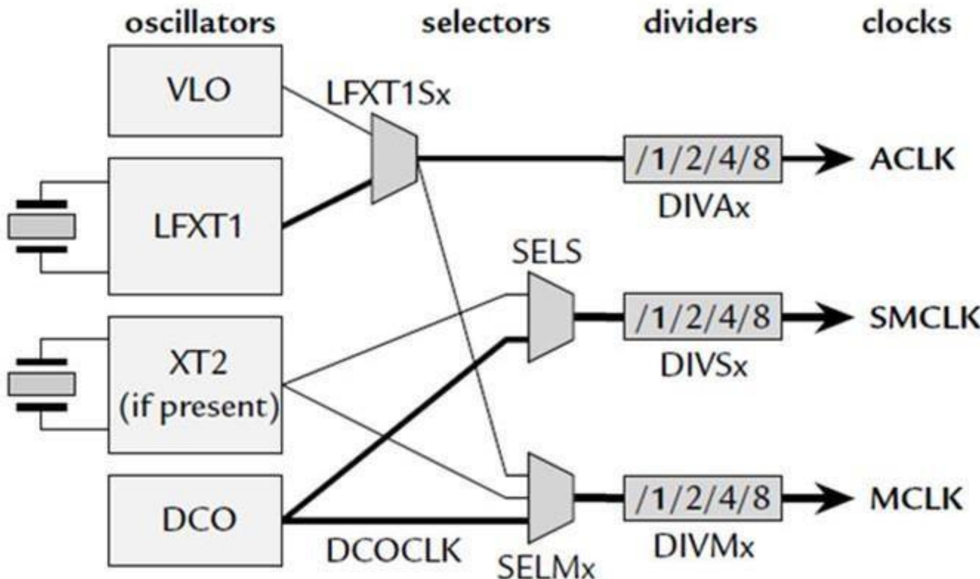
Figure: Stack before and after entering an ISR.
The return address (PC) and Status register (SR) have been saved

An ISR must always finish with the special instruction **RETI** (return from interrupt), which has the following actions.

1. The SR is popped from the stack. All previous settings of control bits are now in effect.
2. The PC (return address) is popped from the stack and execution continues at the point where it was interrupted (or) reverts to its low-power mode, before the interrupt

4.4. CLOCK SYSTEM OF MSP430

- ✓ All microcontrollers contain a clock module to drive the CPU and peripherals.
- ✓ Three clock signals are available from the clock module:
  - (1) **MCLK** : Master clock is used by CPU and system
  - (2) **SMCLK** : Subsystem Master clock is distributed to high speed peripherals
  - (3) **ACLK** : Auxiliary clock is also distributed to low speed peripherals
- ✓ Using three internal clock signals, the user can select the best balance of performance and low power consumption. All Clock signals are software selectable for individual peripheral modules and divided by 1, 2, 4, or 8



The above 3-CLK signals can be generated by following 4 -clock sources:

VLO : Very low-power, low-frequency oscillator

- The VLO is an internal RC oscillator that runs at around 12 KHz
- It saves the cost and space required for a crystal and reduces the current drawn
- It is an alternative to LFXT1 when the accuracy of a crystal is not needed

LFXT1 : Low-frequency crystal oscillator

- It is usually used with a low-frequency watch crystal (32 KHz) but can also run with a high-frequency crystal (typically a few MHz) in most devices.
- The LFXT1 oscillator supports ultra-low current consumption using a 32 KHz watch crystal in LF mode

XT2 : High-frequency crystal oscillator

- Some devices have a second crystal oscillator XT2.
- It is Similar to LFXT1 except that it is restricted to high frequencies.
- It can be used with standard crystals, resonators, or external clock sources in the range of 400-kHz to 16-MHz

DCO : Digitally controlled oscillator

- The DCO is an integrated digitally controlled oscillator
- It is basically a highly controllable RC oscillator that starts in less than 1 sec
- Both MCLK and SMCLK are supplied by an internal digitally controlled oscillator (DCO), which is controlled by a frequency-locked loop.

Control of the Clock Module through the Status Register :

The clock module is controlled by 4 bits in the STATUS REGISTER as well as its own peripheral registers. All bits are clear in the full-power, active mode.

15 … 9	8	7	6	5	4	3	2	1	0
reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

CPUOFF : When set, turns off the MCLK, which stops the CPU.
OSCOFF (Oscillator OFF) : When set, turns off the VLO and LF XT1 crystal oscillator if LFXT1-CLK is not used for MCLK or SMCLK.
SCG0 (System clock generator 0) : When set, turns off the DCO-DC generator if DCO-CLK is not used for MCLK or SMCLK.
SCG1 (System clock generator 1) : When set, turns off the SMCLK.

4.5. LOW POWER MODES & ACTIVE Vs STAND-BY CURRENTS

- The msp430 architecture supports different low-power modes, and it is optimized to achieve extended battery life in portable measurement applications.
- The digitally controlled oscillator (DCO) allows wake-up from low-power modes to active mode in less than 1 micro second.
- 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. The MSP430 is awakened by an interrupt and returns automatically to its low-power mode after handling the interrupt

Why low power?

- To increase the life time of the battery. The portable and mobile devices are getting popular, which have power limitations.
- Power generates heat, which affects the performance of the system
- Power optimization becomes a new dimension in system design, besides the performance and cost

Principles for Low-Power Applications

- Put the system in low-power modes. Use interrupts to wake up the CPU and return to sleep after the completion of the given task.
- Provide clocks of different frequencies → frequency scaling
- Turn off clocks when no work to do → clock gating
- Lower supplied voltage → voltage scaling
- Peripherals should be switched on only when needed

Entering and Exiting Low-Power Modes

By setting the bits SCG1, SCG0, OSCOFF and CPUOFF Status register, the processor can enter into low power mode. The MSP430 is generally operated in any one of the low-power modes. An enabled interrupt event wakes the MSP430 from low-power mode to active mode.

When interrupt occurs:

- The PC and SR are stored on the stack
- The SR is cleared to ZERO which implies that, the bits SCG1, SCG0, OSCOFF and CPUOFF are automatically reset. Hence the CPU enters into active mode
- MCLK must be started so CPU can handle interrupt

Options for returning from the interrupt service routine:

- When the RETI instruction is executed, the original SR is popped from the stack, restoring the previous operating mode.
- Hence the processor enters into its original low-power mode.

Control of Low-Power Modes :

- The low-power modes are controlled by CPUOFF, OSCOFF, SCG0, and SCG1 bits in the status register. All systems are fully operational when all bits are cleared to ZERO. Setting combinations of these bits puts the device into one of its low-power modes LPM0 to LMP4

15 … 9	8	7	6	5	4	3	2	1	0
reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

CPUOFF : When set, turns off the MCLK, which stops the CPU.

OSCOFF (Oscillator OFF) : When set, turns off the VLO and LF XT1 crystal oscillator if LFXT1-CLK is not used for MCLK or SMCLK.

SCG0 (System clock generator 0) : When set, turns off the DCO-DC generator if DCO-CLK is not used for MCLK or SMCLK.

SCG1 (System clock generator 1) : When set, turns off the SMCLK.

Operating Modes for Basic Clock System

SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active and all Clocks are active
0	0	0	1	LPM0	CPU & MCLK are disabled. SMCLK & ACLK are active
0	1	0	1	LPM1	CPU & MCLK are disabled. ACLK is active. DCO and DC generator are disabled if it is not used for SMCLK
1	0	0	1	LPM2	CPU, MCLK, SMCLK are disabled. ACLK is active. DCO & DC-generators are enabled
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO & DC generators are disabled. ACLK is active.
1	1	1	1	LPM4	CPU and all clocks disabled

The MSP430 typical current consumption in various modes is shown in Figure

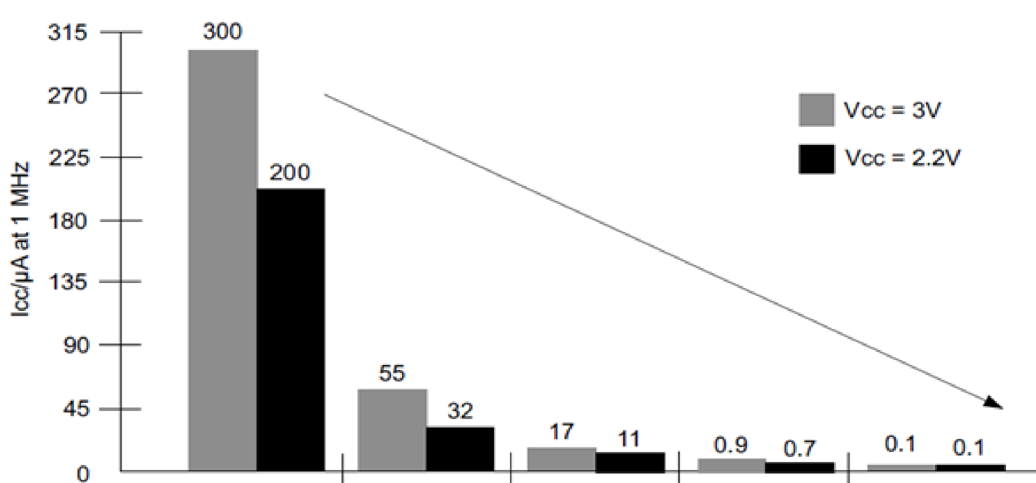


Figure: Typical Current consumption for Various Operating Modes

Active mode:

- CPU, all modules and all clocks are active
- *I ≈ 200μA to 300 μA*
- The MSP430 enters into active mode, when the CPU is required to run the code.
- An interrupt automatically switches the device to active mode.

LPM-0 :

- CPU and MCLK are disabled, SMCLK and ACLK remain active
- *I ≈ 30 to 50 μA*
- This is used when the CPU is not required but some modules require a fast clock from SMCLK and the DCO.

LPM-3 :

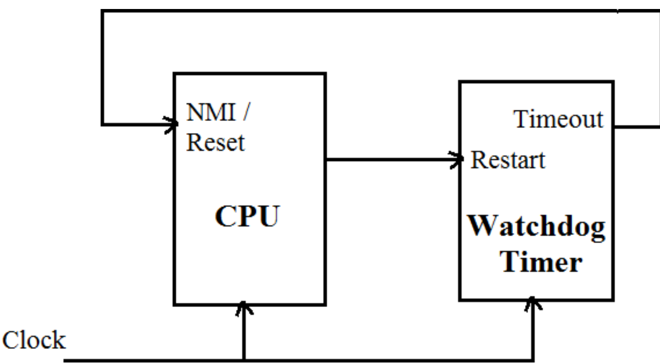
- CPU, MCLK, SMCLK, and DCO are disabled; only ACLK remains active
- I ≈ 1 μA.
- This is the standard low-power mode when the device must wake itself at regular intervals and therefore needs a (slow) clock.
- It is also required if the MSP430 must maintain a real-time clock.

LPM-4:

- CPU and all clocks are disabled
- I ≈ 0.1 μA.
- The device can be wakened only by an external signal.
- This is also called RAM retention mode.

4.6. WATCHDOG TIMER

- A watchdog timer (WDT) is an electronic timer that is used to detect and recover from computer malfunctions. The WDT module restarts the system on occurrence of a software problem (or) if a selected time interval expires.
- During normal operation, the system regularly restarts the WDT to prevent it from elapsing, or "timing out". If the system fails to restart the WDT due to a hardware fault (or) program error, the timer will elapse and generate a timeout signal.
- The timeout signal is used to initiate corrective actions like placing the system in a safe state and restoring normal system operation.



- **Corrective actions :** A watchdog timer may initiate any of several types of corrective actions, including processor reset, non-maskable interrupt, power cycling, fail-safe state activation, or combinations of these, depending on its architecture. In embedded systems and control systems, watchdog timers are often used to activate **fail-safe** circuitry, which forces all control outputs to safe states (e.g., turns off motors, heaters, and high-voltages) to prevent injuries and equipment damage while the fault persists.
- Microcontrollers often include an integrated, on-chip watchdog. The watchdog and CPU may share a common clock signal, or they may have independent clock signals.

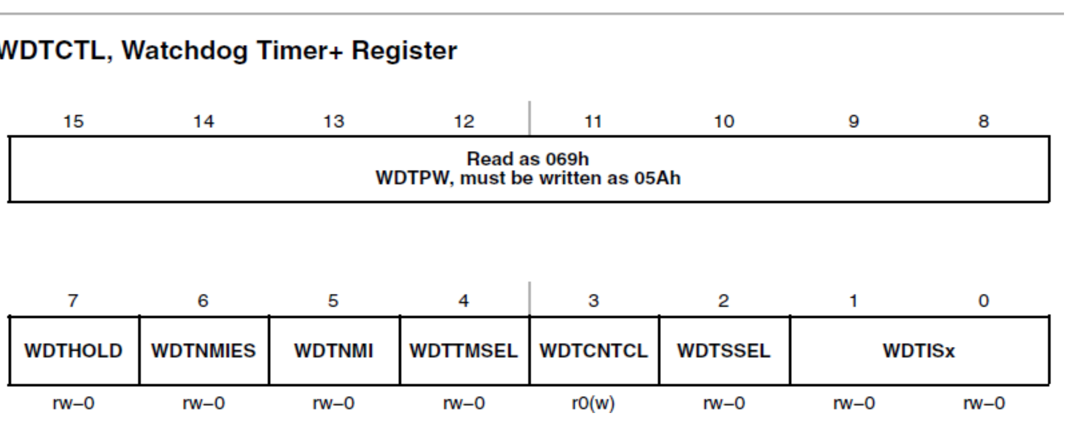
- **Features of the watchdog timer module include:**
  - ✓ Four software-selectable time intervals
  - ✓ Watchdog mode
  - ✓ Interval mode
  - ✓ Access to WDT control register is password protected
  - ✓ Control of RST/NMI pin function
  - ✓ Selectable clock source
  - ✓ Can be stopped to conserve power
  - ✓ Clock fail-safe feature

Watchdog Timer Operation

- 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. Left to itself, the 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.

- **Time intervals:** Watchdog timers may have either fixed or programmable time intervals. Typically, watchdog time intervals range from ten milliseconds to a minute or more.
- The watchdog is always active by default, after the MSP430 has been reset. The WDT is clocked from either SMCLK (default) or ACLK, according to the WDTSSEL bit. If the clock is SMCLK, which is 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.

- The operation of the watchdog is controlled by the 16-bit password-protected register WDTCTL. Any read (or) write access must use password in the upper byte. **0x5A** is the password for write access and **0x69** is the password for read access. The lower byte of WDTCTL contains the bits that control the operation of the WDT as shown in Figure.



Bits 15-8	**WDTPW**	Watchdog timer Password	Always read as 0x69. Must be written as 0x5A. (or) Reset is generated
Bit 7	**WDTHOLD**	Watchdog timer Hold	1 → Stops the watchdog timer 0 → Watchdog timer is not stopped
Bit 6	**WDTNMIES**	Watchdog timer NMI edge select	Selects the interrupt edge for the NMI 1 → NMI on falling edge 0 → NMI on rising edge
Bit 5	**WDTNMI**	Watchdog timer NMI select	1 → NMI function 0 → Reset function
Bit 4	**WDTTMSEL**	Watchdog timer Mode select	1 → Interval timer mode 0 → Watchdog mode
Bit 3	**WDTCNTCL**	Watchdog timer Counter clear	1 → clears the count value to 0000H 0 → No action
Bit 2	**WDTSSEL**	Watchdog timer clock source select	1 → ACLK 0 → SMCLK
Bit 1 & Bit 0	**$WDTIS_1$ & $WDTIS_0$**	Watchdog Timer Interval Select	00 → Watchdog clock source /32,768 01 → Watchdog clock source /8,192 10 → Watchdog clock source /512 11 → Watchdog clock source /64

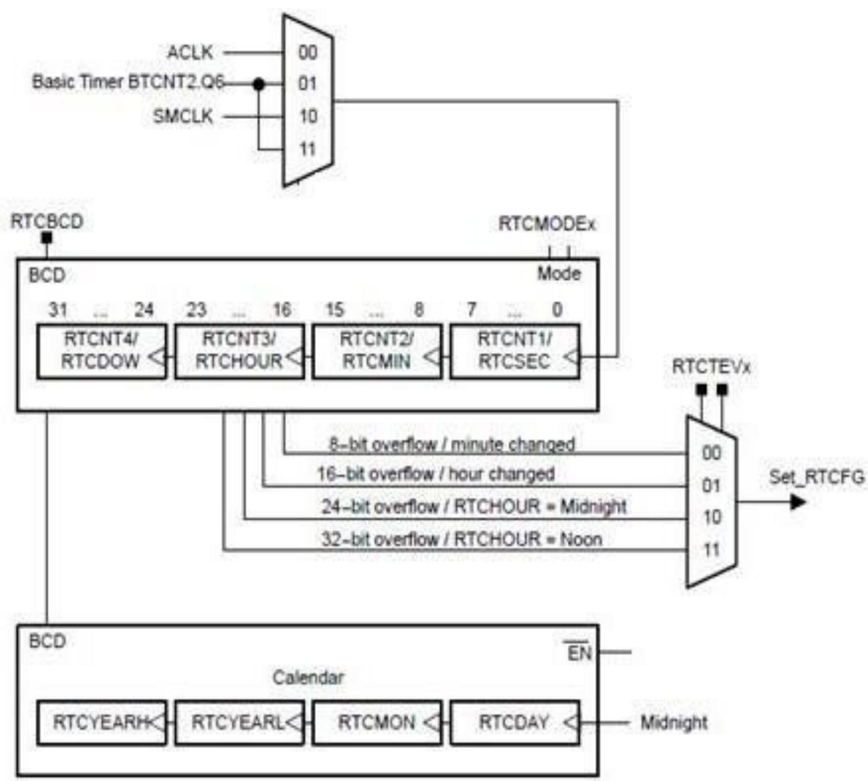
4.7. REAL TIME CLOCK

The Real-Time Clock (RTC) is a timer module that provides a clock with calendar. It provides seconds, minutes, hours, day, month, and year.

Real-Time Clock features include:

- Configurable for Real-Time Clock mode (or) general purpose counter.
- 32-bit counter mode with selectable clock sources
- Provides seconds, minutes, hours, day of week, day of month, month and year.
- Interrupt capability
- Selectable BCD or binary format in Real-Time Clock mode.
- Programmable alarms in Real-Time Clock mode.
- Calibration logic for time offset correction in Real-Time clock mode.
- The current time and date are held in a set of registers that contain the following bytes:

RTCSEC	: Seconds
RTCMIN	: Minute
RTCHOUR	: Hour which runs from 0–23 (24-hour format).
RTCDOW	: Day of week which runs from 0–6.
RTCDAY	: Day of month (1-31)
RTCMON	: Month (1-12)
RTCYEARL	: Year, asuming BCD format.
RTCYEARH	: Century, assuming BCD format.



Real-Time Clock

The Real-Time Clock module can be configured as a real-time clock with calendar function (or) as a 32-bit general purpose counter with the RTCMODEx bits

(a) Calendar Mode :

- Calendar mode is selected when RTCMODEx = 11.
- In calendar mode, the RTC provides seconds, minutes, hours, day of week, day of month, month, and year in selectable BCD or hexadecimal format.
- Switching from counter to calendar mode clears the seconds, minutes, hours, day-of-week, and year counts and sets day-of-month and month counts to 1.
- When RTCBCD = 1, BCD format is selected for the calendar registers.
- The calendar includes a leap year algorithm

(b) Counter Mode :

- Counter mode is selected when RTCMODEx < 11.
- In this mode, a 32-bit counter is provided that is directly accessible by software.
- Switching from calendar mode to counter mode resets the count value.
- The clock source can be selected from ACLK, SMCLK, or from BTCNT2 clock.
- The counter can be stopped by setting the RTCHOLD bit.
- The following counters are available : RTCNT1, RTCNT2, RTCNT3, RTCNT4
- These Four individual 8-bit counters are cascaded to provide the 32-bit counter.
- This provides 8-bit, 16-bit, 24-bit, or 32-bit overflow intervals of the counter clock.
- Each counter is individually accessible and may be read or written to.

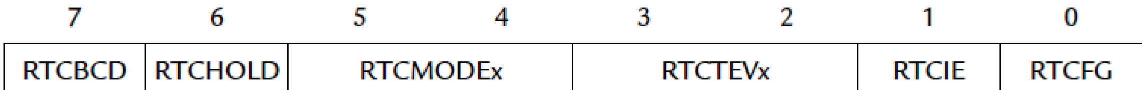


Figure 8.4: The Real-Time Clock control register RTCCTL.

Bit	Name	Description	Values
Bit 7	**RTCBCD**	BCD format select	0 → Hexadecimal format 1 → BCD format
Bit 6	**RTCHOLD**	Real-Time Clock hold	0 → Real-Time Clock is operational 1 → The RTC module is stopped
Bit 5-4	**RTCMODEx**	Real-Time Clock mode and clock source select	00 → 32-bit counter with ACLK input 01 → 32-bit counter with BTCNT2.Q6 10 → 32-bit counter with SMCLK 11 → Calendar mode with BTCNT2.Q6
Bits 3-2	**RTCEVx**	Real-Time Clock interrupt event	Counter Mode : 00: 8-bit overflow 01 : 16-bit overflow 10 : 24-bit overflow 11 : 32-bit overflow Calendar Mode : 00 : Minute changed 01 : Hour changed 10 : Every day at midnight 11 : Every day at noon
Bit 1	**RTCIE**	Real-Time Clock interrupt enable	0 Interrupt not enabled 1 Interrupt enabled
Bit 0	**RTCFG**	Real-Time Clock interrupt flag	0 No time event occurred 1 Time event occurred

Note : TIMERS OF MSP430

- ✓ Timers are essential to almost any embedded application
  - Generate fixed-period events
  - Periodic wakeup
  - Count edges
  - Generate delays
  - Measures time intervals
  - Replacing delay loops with timer calls allows CPU to sleep, consuming less power

- ✓ Five types of MSP430 timer modules are available. All devices contain two types of timer and some have five. Each type of timer module works in essentially the same way in all devices.

  Basic timer1 : Present in the MSP430x4xx family only.
  It provides the clock for the LCD and acts as an interval timer.

  Timer_A : Present in all devices. It typically has 3 channels.

  Timer_B : Included in larger devices of all families.
  It is similar to Timer_A with some extensions that make it more suitable for driving outputs such as PWM.

  Watchdog timer : Included in all devices
  Its main function is to protect the system against malfunctions.

  Real-time clock : In which the basic timer has been extended to provide a real-time clock in the most recent MSP430 devices.

4.8. BASIC TIMER_1

The Basic Timer1 supplies LCD timing and low frequency time intervals. The Basic Timer1 is two independent 8-bit timers that can also be cascaded to form one 16-bit timer function.

Some of the features of Basic Timer1 include:

- Real-time clock (RTC) function
- Software time increments
- Basic Timer1 features include:
- Selectable clock source
- Two independent, cascadable 8-bit timers
- Interrupt capability
- LCD control signal generation

Basic Timer1 Operation

- ✓ The Basic Timer1 module can be configured as two 8-bit timers with the BTCTL register.
- ✓ The BTCTL is an 8-bit read/write Basic Timer Control Register. Any read or write access must use byte instructions.
- ✓ The Basic Timer1 controls the LCD frame frequency with BTCNT1.

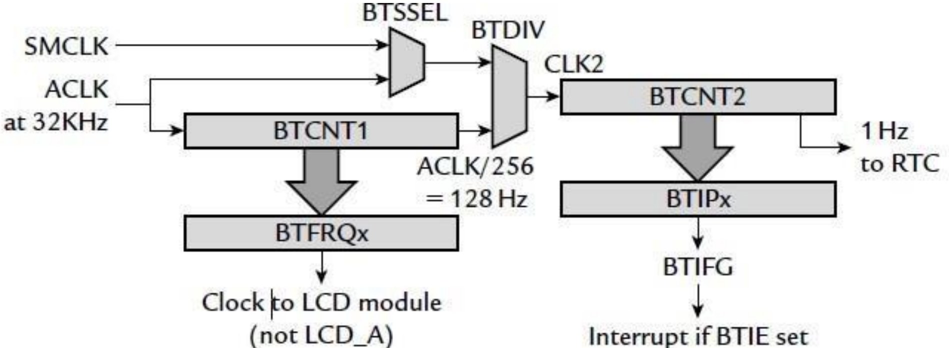


Figure 8.2: Simplified block diagram of Basic Timer1.

Figure 8.3: The Basic Timer1 control register BTCTL.

BTCNT1 (Basic Timer Counter-1) :

- ✓ BTCNT1 is an 8-bit timer/counter directly accessible by software.
- ✓ BTCNT1 is incremented with ACLK and provides the frame frequency for the LCD controller.
- ✓ BTCNT1 can be stopped by setting the BTHOLD and BTDIV bits.

BTCNT2 (Basic Timer Counter-2) :

- ✓ BTCNT2 is an 8-bit timer/counter directly accessible by software.
- ✓ BTCNT2 can be sourced from ACLK or SMCLK, or from ACLK/256 when cascaded with BTCNT1.
- ✓ The BTCNT2 clock source is selected with the BTSSEL and BTDIV bits.
- ✓ BTCNT2 can be stopped to reduce power consumption by setting the HOLD bit.
- ✓ BTCNT2 sources the Basic Timer1 interrupt, BTIFG.
- ✓ The interrupt interval is selected with the BTIPx bits

16- Bit Counter Mode :

- ✓ The 16-bit timer/counter mode is selected when control the BTDIV bit is set.
- ✓ In this mode, BTCNT1 is cascaded with BTCNT2.
- ✓ The clock source of BTCNT1 is ACLK, and the clock source of BTCNT2 is ACLK/256.

Basic Timer1 Control Register (BTCTL) :

<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>BTSSEL</td><td>BTHOLD</td><td>BTDIV</td><td colspan="2">BTFREQx</td><td colspan="3">BTIPx</td></tr>
</table>

<table>
<tr><td>Bit 7</td><td>BTSSEL</td><td>BTCNT2 clock select</td><td>This bit, together with the BTDIV bit, selects the clock source for BTCNT2.</td></tr>
<tr><td>Bit 6</td><td>BTHOLD</td><td>Basic Timer1 hold</td><td>Stops BTCNT1 & BTCNT2</td></tr>
<tr><td>Bit 5</td><td>BTDIV</td><td>Basic Timer1 clock divide</td><td>This bit together with the BTSSEL bit, selects the clock source for BTCNT2.
<table>
<tr><td>BTSEL</td><td>BTDIV</td><td>Clock source</td></tr>
<tr><td>0</td><td>0</td><td>ACLK</td></tr>
<tr><td>0</td><td>1</td><td>ACLK/128</td></tr>
<tr><td>1</td><td>0</td><td>SMCLK</td></tr>
<tr><td>1</td><td>1</td><td>ACLK/256</td></tr>
</table></td></tr>
<tr><td>Bits 4-3</td><td>BTFRFQx</td><td>f_{LCD} frequency</td><td>These bits control the LCD update frequency
00 → $f_{ACLK}/32$
01 → $f_{ACLK}/64$
10 → $f_{ACLK}/128$
11 → $f_{ACLK}/256$</td></tr>
<tr><td>Bits 2-0</td><td>BTIPx</td><td>Basic Timer1 interrupt interval</td><td>00 → $f_{CLK2}/2$
01 → $f_{CLK2}/4$
010 → $f_{CLK2}/8$
011 → $f_{CLK2}/16$
100 → $f_{CLK2}/32$
101 → $f_{CLK2}/64$
110 → $f_{CLK2}/128$
111 → $f_{CLK2}/256$</td></tr>
</table>

4.9. TIMER_A/B

1. Timer_A is a 16-bit timer/counter with 4- operating modes
2. Supports multiple capture/compare registers (up to 7 CC registers)
3. Timer_A also has extensive interrupt capabilities. Selectable and configurable clock source
4. Configurable outputs with 8 – output modes
5. PWM capability
6. Interrupt vector register for fast decoding of all Timer_A interrupts

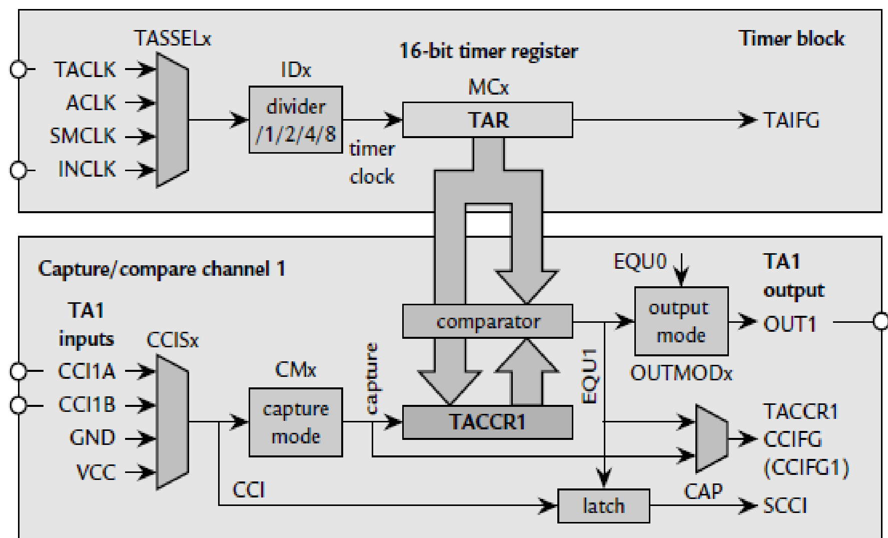


Figure 8.5: Simplified block diagram of Timer_A showing the timer block and capture/compare channel 1. The circles show external signals that may be brought out to pins of the device.

Timer block:

- Timer block consists of 16-bit counter named as Time_A register (TAR) which increments or decrements (depending on mode of operation) with each rising edge of the clock signal.
- Selectable clock sources - TACLK, ACLK, SMCLK, INCLK
- The timer can generate an interrupt TAIFG when it overflows.
- TAR may be cleared by setting the TACLR bit.
- It is controlled by the Timer_A control register TACTL.

Timer Modes :

The timer has 4- modes of operation, can be selected with the MCx bits of TACTL.

MCx	Mode	Description
00	Stop	The timer is halted
01	Up	Counts from zero to TACCR0.
10	Continuous	Counts from zero to 0FFFFh
11	Up/down	Counts from zero to TACCR0 and back down to zero

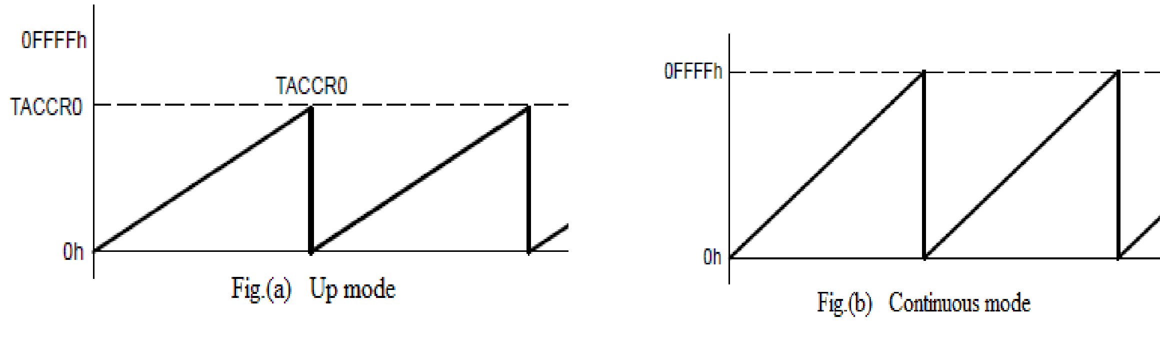


Fig.(a) Up mode

Fig.(b) Continuous mode

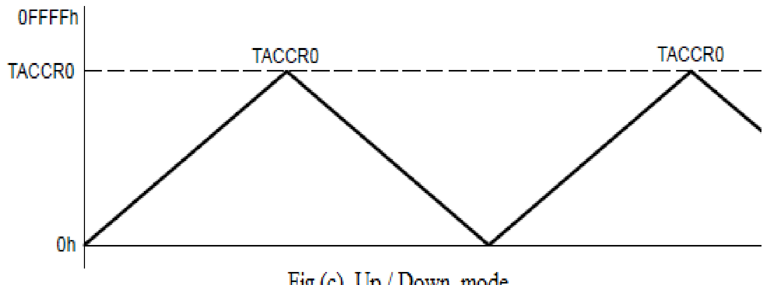


Fig.(c) Up / Down mode

Stop mode (MCx = 00):

- The timer is halted.
- All registers, including TAR, retain their values so that the timer can be restarted later where it left off.

Up Mode (MCx = 01):

- The timer repeatedly counts from zero to the value of TACCR0.
- The period is (TACCR0+1) counts.
- For ex., if TACCR0 = 4, the sequence of counts is 0, 1, 2,3, 4, 0, 1,2,3,4,0,1,2....
- The TACCRx CCIFG interrupt flag is set when the timer *counts* to the TACCRx value.
- The TAIFG interrupt flag is set when the timer *counts* from TACCRx to zero.

Continuous Mode (MCx = 10):

- The timer repeatedly counts from zero to 0FFFFh.
- The period is 65,536 counts
- The TAIFG interrupt flag is set when the timer *counts* from 0FFFFh to zero.

Up/Down Mode (MCx = 11):

- The timer repeatedly counts from zero up to the value of TACCR0 and back down to zero.
- The period is 2*TACCR0 counts.
- For example, if TACCR0=4, the sequence of counts is 0, 1, 2, 3, 4, 3, 2, 1, 0,1, 2,. .. .
- The TACCR0 CCIFG flag is set when the timer *counts* from TACCR0-1 to TACCR0, and TAIFG is set when the timer completes *counting* down from 0001h to 0000h.
- The up/down mode is used, if symmetrical pulse generation is needed.

Capture/compare channels :

The times have capture /compare channels, which can be used

1. **Capture** an input, which means recording the "time" (the value in TAR) at which the input changes in TACCRn; the input can be either external or internal from another peripheral or software.
2. **Compare** the current value of TAR with the value stored in TACCRx and update an output when they match; the output can again be either external or internal.
3. **Request an interrupt** by setting its flag CCIFG on either of these events; this can be done even if no output signal is produced.
4. **Sample** an input at a compare event; this special feature is useful if Timer_A is used for serial communication in a device that lacks a dedicated interface.

Capture Mode

- The capture mode is selected when CAP = 1.
- Capture mode is used to record time events.
- It can be used for speed computations or time measurements.
- The CMx bits select the capture edge of the input signal as rising, falling, or both.
- The capture inputs CCIxA and CCIxB are connected to external pins or internal signals and are selected with the CCISx bits.
- If a capture occurs:
    - ✓ The timer value is copied into the TACCRx register
    - ✓ The interrupt flag CCIFG is set
- Overflow logic is provided in each capture/compare register to indicate if a second capture was performed before the value from the first capture was read. Bit COV is set when this occurs.

- The capture signal can be asynchronous to the timer clock and cause a race condition. Setting the SCS bit will synchronize the capture with the next timer clock. Setting the SCS bit to synchronize the capture signal with the timer clock is recommended. This is illustrated in the following figure.

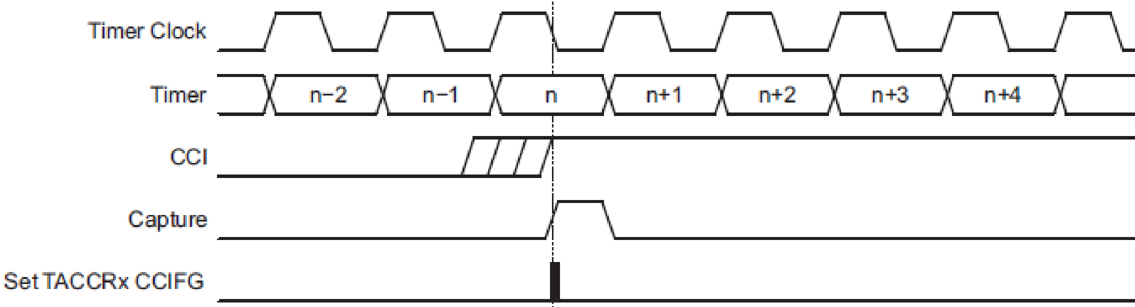


Figure 12-10. Capture Signal (SCS = 1)

Compare Mode

- The compare mode is selected when CAP = 0.
- The compare mode is used to generate PWM output signals or interrupts at specific time intervals.
- When TAR *counts* to the value in a TACCRx:
  - ✓ Interrupt flag CCIFG is set
  - ✓ EQUx affects the output according to the output mode
  - ✓ The input signal CCI is latched into SCCI

Output Unit

- Each capture/compare block contains an output unit.
- The output unit is used to generate output signals such as PWM signals.
- Each output unit has 8- operating modes, defined by the OUTMODx bits.

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value

Output Example — Timer in Up Mode

The OUTx signal is changed when the timer *counts* up to the TACCRx value, and rolls from TACCR0 to
zero, depending on the output mode. An example is shown in Figure 12-12 using TACCR0 and TACCR1.

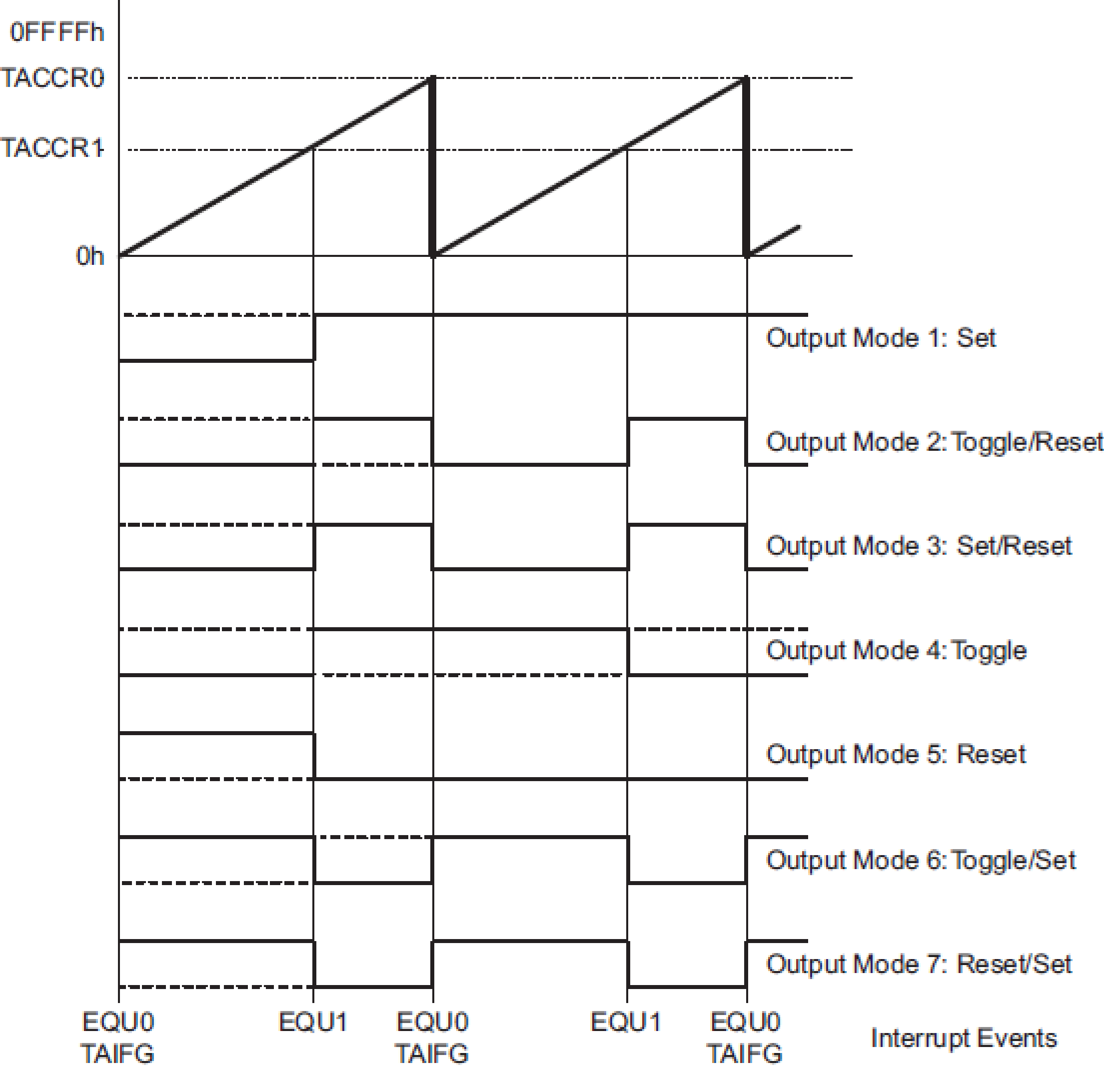


Figure 12-12. Output Example—Timer in Up Mode

Output Example — Timer in Continuous Mode

The OUTx signal is changed when the timer reaches the TACCRx and TACCR0 values, depending on the output mode. An example is shown in Figure 12-13 using TACCR0 and TACCR1.

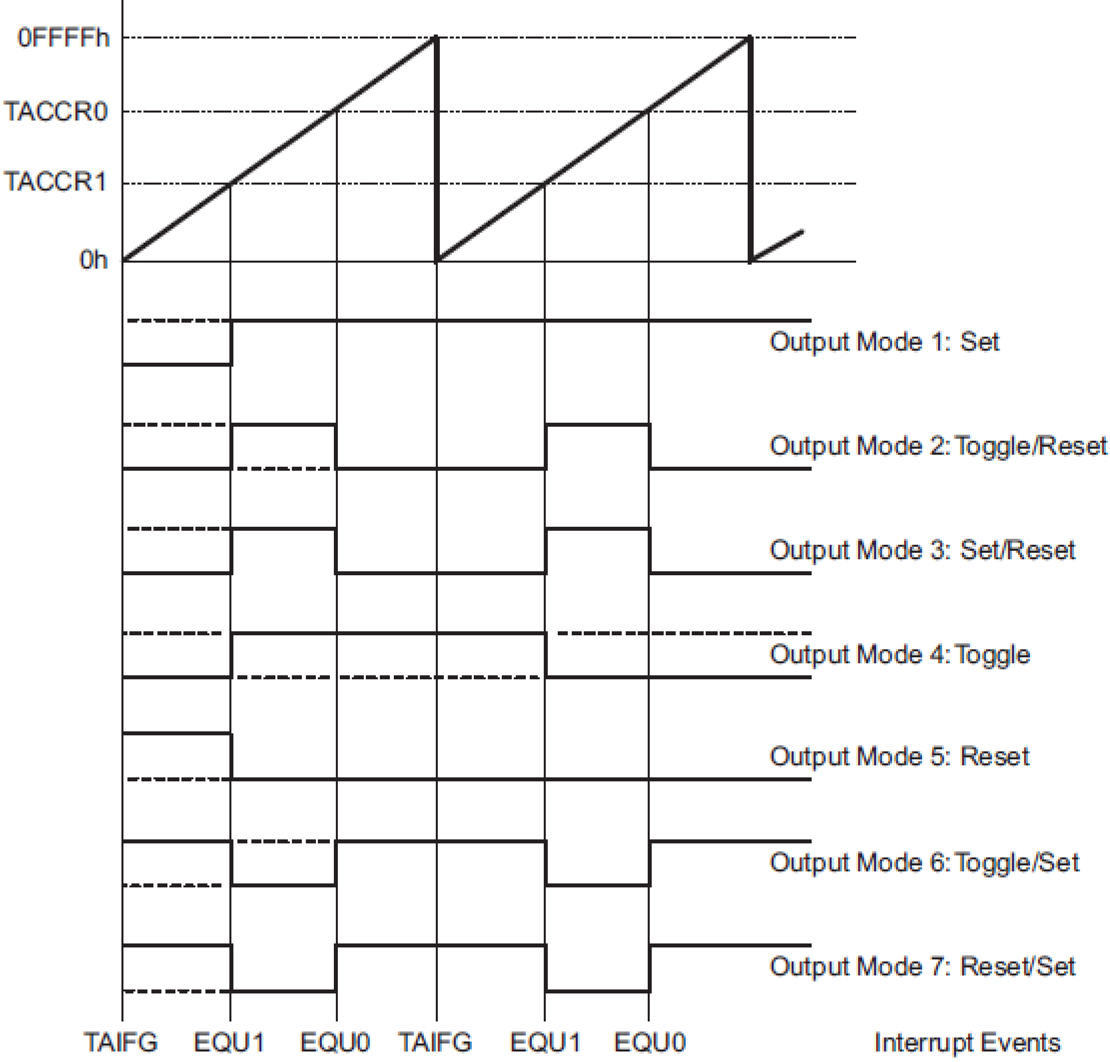


Figure 12-13. Output Example—Timer in Continuous Mode

Output Example — Timer in Up/Down Mode

The OUTx signal changes when the timer equals TACCRx in either count direction and when the timer
equals TACCR0, depending on the output mode. An example is shown in Figure 12-14 using TACCR0
and TACCR2.

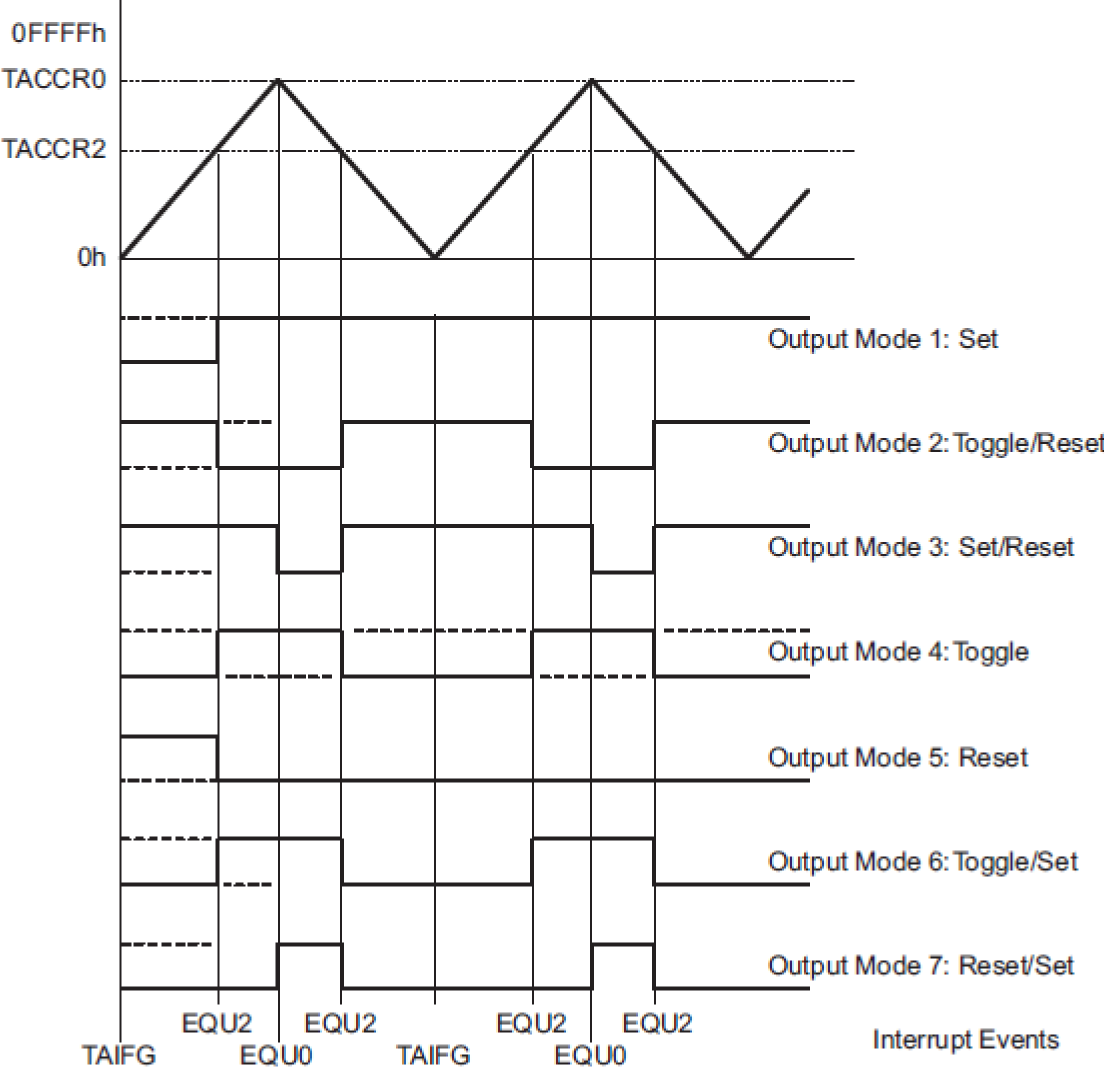


Figure 12-14. Output Example—Timer in Up/Down Mode

Timer_A Registers :

Register	Short Form	Register Type
Timer_A control	TACTL	Read/write
Timer_A counter	TAR	Read/write
Timer_A capture/compare control x (x= 0,1,2…)	TACCTLx	Read/write
Timer_A capture/compare x (x= 0,1,2…)	TACCRx	Read/write
Timer_A interrupt vector	TAIV	Read only

TACTL (Timer_A control register)

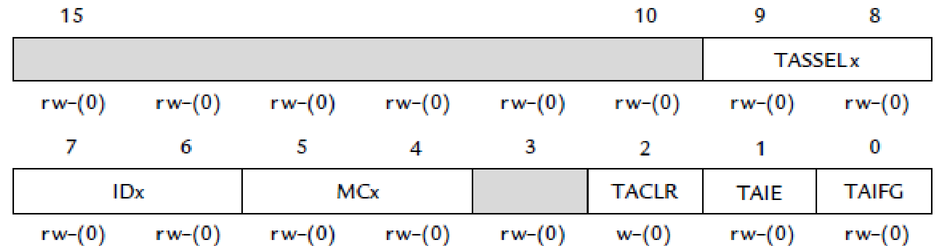
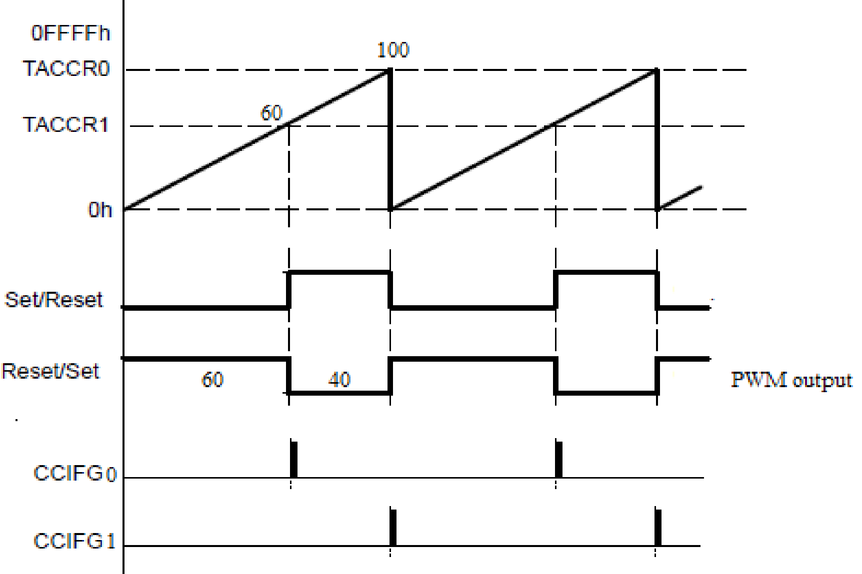


Figure 4.8: The Timer_A Control Register TACTL. Gray bits are unused.

Bit 15-10	UNUSED		
Bits 9-8	**TASSELx**	Timer_A clock source select	00 TACLK 01 ACLK 10 SMCLK 11 INCLK
Bits 10-9	**IDx**	Input divider	00→ /1 01→ /2 10 → /4 11 → /8
Bits 5-4	**MCx**	Mode control	00 → Stop mode (halt) 01 → Up mode (0 to TACCR0) 10 → Continuous mode (0 to 0FFFF) 11 → Up/down mode (the timer counts up to TACCR0 then down to 0000)
Bit 2	**TACLR**	Timer_A clear.	Setting this bit resets TAR, the clock divider, and the count direction
Bit 1	**TAIE**	Timer_A interrupt enable	0 → Interrupt disabled 1 → Interrupt enabled
Bit 0	**TAIFG**	Timer_A interrupt flag	0 → No interrupt pending 1 → Interrupt pending

4.10. PWM OUTPUT

- The idea behind PWM is very simple: The load is switched on and off periodically so that the *average* voltage has the desired value.
- The fraction of the time while the load is active is called the *duty cycle D*.
- The duty cycle is almost always varied by keeping the period constant and changing the width of the pulses, hence the name of PWM.



- The period of Timer_A is set by TACCR0 in the Up mode.
- TAR counts from 0 up to the value in TACCR0, and returns to 0 to for the next clock
- The period is therefore TACCR0 + 1

- The flag **CCIFG0** is set when TAR counts to **TACCR0** and the TAIFG flag is set when TAR returns to 0, one cycle later.
- The flag **CCIFG1** is set when TAR counts to **TACCR1**, which is 60 here.

- There are two main parameters be chosen for PWM output.

(a) The time period of the output PWM waveform $T_0 = [TACCR0 + 1]\, T_{CLK}$

(b) Duty cycle of PWM output $D = \frac{\text{Pulse width}}{\text{Time Period}} = \frac{TACCR1}{TACCR0+1}$

- Now, the average voltage across the output is given by

$$V_{Avg} = D\, V_{CC} = \frac{TACCR1}{TACCR0+1} V_{CC}$$

This means that, by changing the value in TACCR1, we can change the duty cycle. The above Figure shows that, the Reset/Set output mode (7) is used for active high loads, called as *positive PWM* and the Set/Reset mode (3) is used for active low loads, called as negative PWM.

4.11. MEASUREMENT OF TIME & FREQUENCY IN CAPTURE MODE

The Capture mode is used to take a time stamp of an event, and to note the time at which it occurred. The timer usually runs in the Continuous mode for captures because this makes it easy to calculate differences of times when TAR has rolled over between them

***(a) Measurement of duration and Time period* :**

- In most cases the timer clock is either ACLK or SMCLK, whose frequency is known, and the unknown signal is applied to the capture input.
- To measure the duration of the pulse, we should capture both rising and falling edges and subtract the captured times.
- To measure the time period of the signal, we might capture only the rising edges (or falling if preferred) and the difference gives the period directly.

***(b) Measurement of frequency* :**

- The signal is used as the timer clock (TACLK) and the edges of ACLK are captured whose frequency is known.
- The difference between the captured value gives the number of cycles of the signal in one cycle of ACLK. This gives the frequency rather than the period.

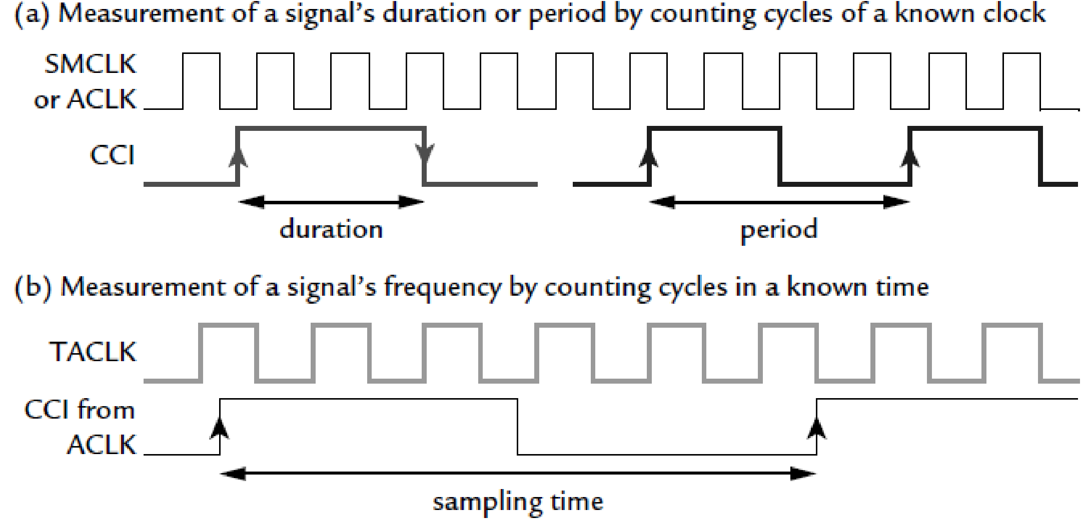


Figure : Two ways in which the Capture mode is used to time a signal.

Here are a few examples of the use of the Capture mode:

- Many speed sensors produce no. of pulses per revolution. The Capture mode is used to measure the period between pulses to determine the speed.
- Some sensors encode their outputs as a frequency, length of a pulse, or the duty cycle of a square wave, the fraction of the time during which the signal is high.
- The delay between transmission and reception of an ultrasonic pulse is measured in the range finder application -*Ultrasonic Distance Measurement with the MSP430*

4.12. COMPARATOR

- An analog comparator compares the voltages on its two input terminals, $V+$ and $V-$.
- The comparator output CAOUT is high if $V+ > V-$ and low if $V+ < V-$.
- It provides a basic bridge between analog & digital domains and acts as a 1-bit ADC
- The comparator can be switched ON or OFF using control bit CAON.
- The comparator should be switched off when not in use to reduce current consumption.
- When the comparator is switched off, the CAOUT is always low.

- Features of Comparator_A+ include:
  - ✓ Inverting and non-inverting terminal input multiplexer
  - ✓ Software selectable RC-filter for the comparator output
  - ✓ Output provided to Timer_A capture input
  - ✓ Software control of the port input buffer
  - ✓ Interrupt capability
  - ✓ Selectable reference voltage generator
  - ✓ Comparator and reference generator can be powered down in LPMs

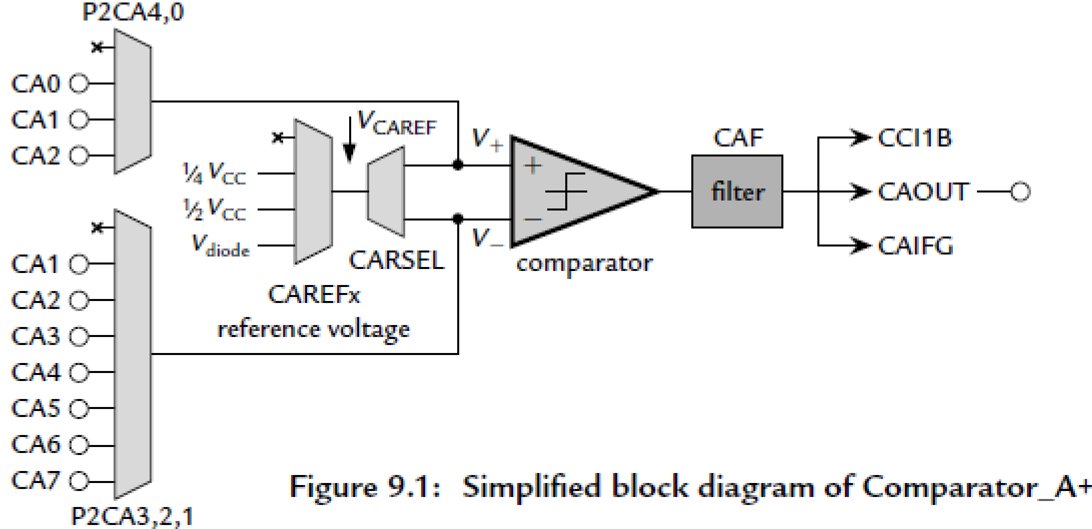


Figure 9.1: Simplified block diagram of Comparator_A+

- The comparator is used to compare a variable input voltage with a reference.

- The internal reference voltage V_{CAREF} can be chosen from ¼ VCC, ½ VCC or a nominally fixed voltage from a transistor, V_{diode}. This is selected with the CAREFx bits.

- The voltage reference V_{CAREF} can be applied to either comparator input terminal according to the CARSEL bit.
- If external signals are applied to both comparator input terminals, the internal reference generator should be turned OFF to reduce current consumption.
- The non-inverting input $V+$ can be connected to external signals CA0–CA2 or left without an external connection. This is selected using bits P2CA4 and P2CA0.

- Similarly, the inverting input *V*− can be connected to external signals CA1–CA7 or left unconnected, according to bits P2CA3 –P2CA1.
- The output of the comparator can be used with or without internal filtering. When control bit CAF is set, the output is filtered with an on-chip RC-filter to reduce oscillations in the signal.
- The output is brought to an external pin CAOUT. It is also connected internally to capture input CCI1B of Timer_A, which allows precise timing without delays.
- The flag CAIFG is raised on either a rising or falling edge of the comparator output, selected with the CAIES bit. This can in turn request an interrupt if CAIE is set. Comparator_A has its own interrupt vector and the flag is cleared automatically when the interrupt is serviced.

COMPARATOR_A CONTROL REGISTER (CACTL)

7	6	5	4	3	2	1	0
CAEX	CARSEL	CAREFx		CAON	CAIES	CAIE	CAIFG
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

Bit 7	**CAEX**	Comparator_A exchange	This bit exchanges the comparator inputs and inverts the comparator output.
Bit 6	**CARSEL**	Comparator_A reference select	This bit selects terminal for Reference voltage 0 → V_{CAREF} is applied to the + terminal 1 → V_{CAREF} is applied to the – terminal
Bits 5-4	**CAREF**	Comparator_A reference	These bits select the reference voltage V_{CAREF}. 00 → Internal reference off. An external reference can be applied. 01 → 0.25*V_{CC} 10 → 0.5 *V_{CC} 11 → Diode reference is selected
Bit 3	**CAON**	Comparator_A on.	0 → Comparator OFF 1 → Comparator ON
Bit 2	**CAIES**	Comparator_A interrupt edge select	0 → Rising edge 1 → Falling edge
Bit 1	**CAIE**	Comparator_A interrupt enable	0 → Disabled 1 → Enabled
Bit 0	**CAIFG**	The Comparator_A interrupt flag	0 → No interrupt pending 1 → Interrupt pending

4.13 THE ADC10 : SUCCESSIVE APPROXIMATION ADC

- The ADC10 module supports fast, 10-bit analog-to-digital conversions.
- The module implements a 10-bit SAR core, sample select control, reference generator, and data transfer controller (DTC).
- The DTC allows ADC10 samples to be converted and stored anywhere in memory without CPU intervention.

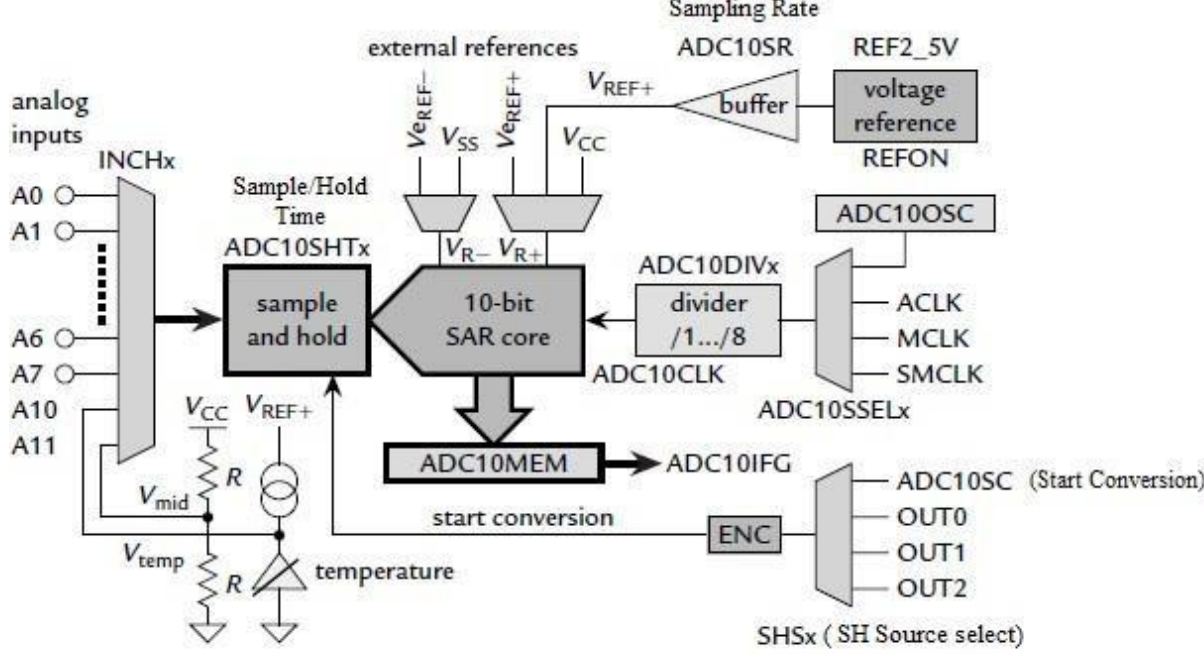


Figure 9.16: Simplified block diagram of the ADC10

ADC10 features include:

- ✓ Greater than 200 Ksps maximum conversion rate
- ✓ Sample-and-hold with programmable sample periods
- ✓ Conversion initiation by software or Timer_A
- ✓ Software selectable internal or external reference
- ✓ Software selectable on-chip reference voltage generation (1.5 V or 2.5 V)
- ✓ Up to 12- external input channels
- ✓ Conversion channels for internal temperature sensor, VCC, and external references
- ✓ Selectable conversion clock source
- ✓ Single-channel, repeated single-channel, sequence, & repeated sequence conversion modes.
- ✓ ADC core and reference voltage can be powered down separately
- ✓ Data transfer controller (DTC) for automatic storage of conversion results

10- Bit ADC Core :

The ADC core converts an analog input to its 10-bit digital representation and stores the result in the ADC10MEM register. Conversion results may be in straight binary format or 2's complement format. The conversion formula for the ADC result when using straight binary format is:

$$N_{ADC} = \left(1023 \cdot \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}\right)$$

Conversion Clock Selection

The ADC10CLK is used both as the conversion clock and to generate the sampling period. The ADC10 source clock is selected using the ADC10SSELx bits and can be divided from 1-8 using the ADC10DIVx bits.

Possible ADC10CLK sources → SMCLK, MCLK, ACLK , internal oscillator ADC10OSC.

ADC10 Inputs and Multiplexer

The 8-external and 4-internal analog signals are selected for the conversion by the analog input multiplexer (INCHx). The input multiplexer is a break-before-make type to reduce input-to-input noise injection resulting from channel switching

Voltage Reference Generator

The ADC10 module contains a built-in voltage reference with two selectable voltage levels. Setting REFON = 1 enables the internal reference.

When REFOUT =0, externally on pin $V_{REF}+$.

Sample and Conversion Timing

An analog-to-digital conversion is initiated with a rising edge of sample input signal, selected with the Sample and Hold source (SHSx) bits - ADC10SC, Timer_A outputs OUT1, OUT2, OUT3.

Conversion Sequence Modes :

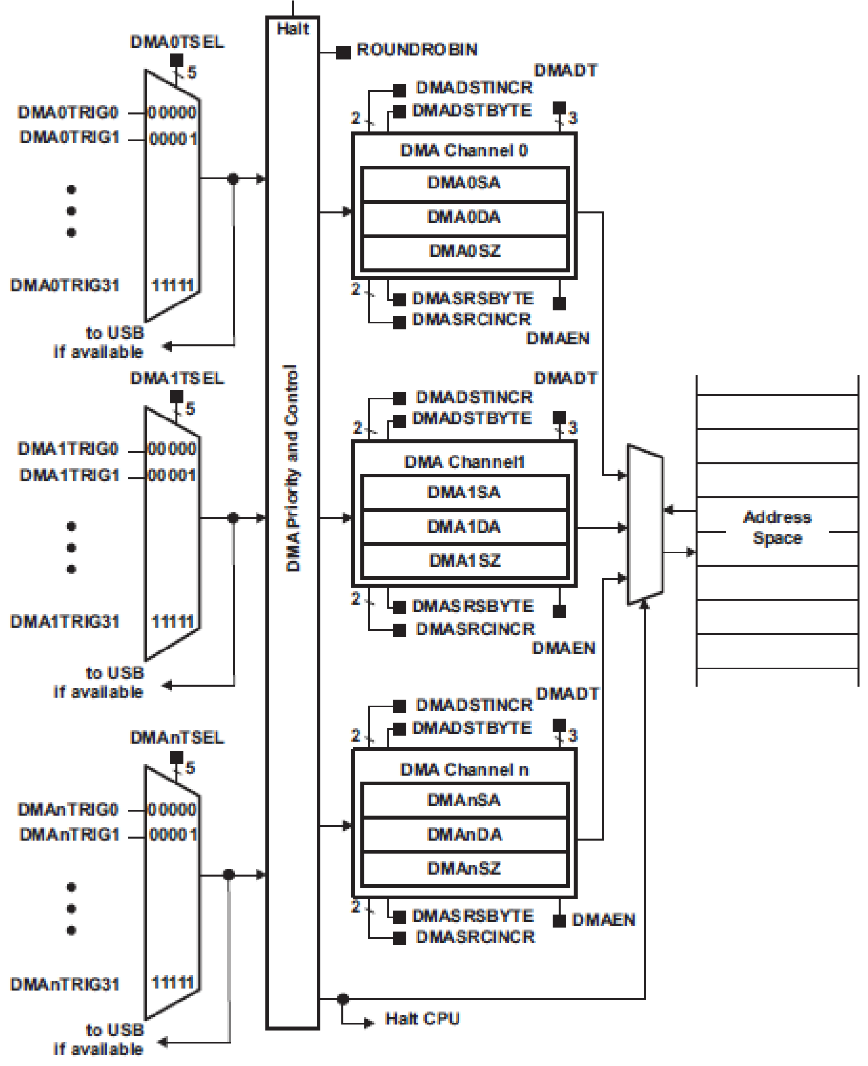
CONSEQx	Mode	Operation
00	Single channel single-conversion	Single channel is converted once.
01	Sequence of channels	Sequence of channels is converted once
10	Repeat single channel	Single channel is converted repeatedly
11	Repeat sequence of channels	Sequence of channels is converted repeatedly

ADC10 Data Transfer Controller

The ADC10 includes a data transfer controller (DTC) to automatically transfer conversion results from ADC10MEM to other on-chip memory locations.

4.14. DATA TRANSFER USING DMA CONTROLLER

- The direct memory access (DMA) controller transfers data from one address to another, without CPU intervention, across the entire address range. For example, the DMA controller can move data from the ADC10 conversion memory to RAM.
- Devices that contain a DMA controller may have up to eight DMA channels available.
- DMA controller can increase the throughput of peripheral modules. It can also reduce system power consumption by allowing the CPU to remain in a low-power mode without having to awaken to move data to or from a peripheral.



DMA Controller Block Diagram

DMA Addressing Modes :

- Fixed address to fixed address
- Fixed address to block of addresses
- Block of addresses to fixed address
- Block of addresses to block of addresses

The addressing mode for each DMA channel is independently configurable with the control bits DMASRCINCRx and DMADSTINCRx. These bits are also used to select the increment / decrement mode for source and destination addresses, after each transfer.

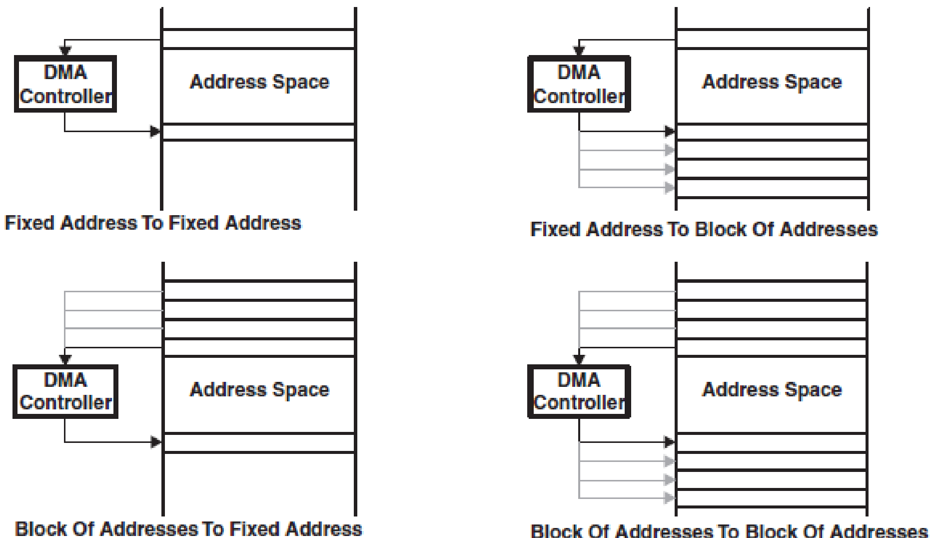


Figure . DMA Addressing Modes

Transfers may be byte-to-byte, word-to-word, byte-to-word, or word-to-byte. When transferring word-to-byte, only the lower byte of the source-word transfers. When transferring byte-to-word, the upper byte of the destination-word is cleared when the transfer occurs.

DMA Transfer modes :

The DMA controller has 6- transfer modes selected by the DMADTx bits as listed in the Table. Each channel is individually configurable for its transfer mode. For example, channel 0 may be configured in single transfer mode, while channel 1 is configured for burst-block transfer mode, and channel 2 operates in repeated block mode. The transfer mode is configured independently from the addressing mode. Any addressing mode can be used with any transfer mode.

Table : DMA transfer modes

DMADTx	Transfer Mode
000	Single transfer
001	Block transfer
010,011	Burst-block transfer
100	Repeated single transfer
101	Repeated block transfer
110,111	Repeated burst-block transfer

UNIT-5 : COMMUNICATION INTERFACES OF MSP430

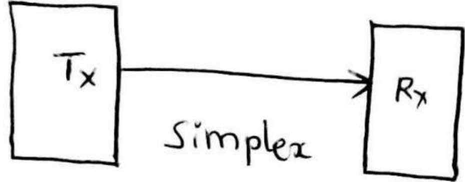
1. Basics of Serial communication
2. UART
3. SPI
4. I2C
5. Network processor CC3100
6. Adding Wi-Fi to MCU

5.1. INTRODUCTION TO SERIAL COMMUNICATION

Within a micro-computer system, the data transfer is in parallel because it is the fastest method. But transferring the data over long distances, the parallel data transmission requires too many wires and it is complicated and expensive. Therefore, the data to be sent for long distances is converted into serial form so that it can be sent on a single wire. At the destination, the received serial data is converted into parallel form so that it can be easily transferred on the micro-computer buses.

Methods of serial data transmission:

(i) Simplex :



In this mode, the data is transmitted only in one direction over a single communication channel.

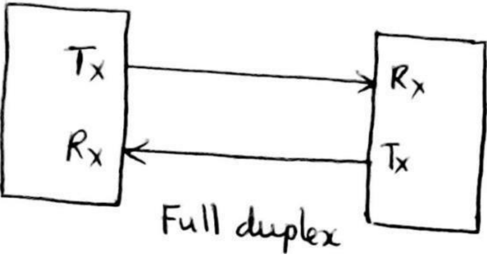
Ex: CPU to CRT display, Key board to CPU, Radio signal

(ii) Half-dupplex :

In this mode, the data is transmitted in both directions, but only one direction at a time.
i.e., simultaneous data transfer is not possible

Ex: Walkie Talkie

(iii) Full-dupplex :



In this mode, the data transmission takes place in both directions simultaneously.
It requires two channels.

Ex: Telephone communication

Synchronous Vs Asynchronous data transfer modes:

Synchronous data transmission	Asynchronous data transmission
Transmitter and Receiver are operated with same CLK frequency.	Transmitter and Receiver can operated with different CLK frequency.
SYNC pulses are required	START and STOP bits are required
A group of characters can be transmitted after sending the SYNC pulses	For each character, the START & STOP bits are required.
It is used in high speed data transmission	It is used in low speed data transmission
Generally used between CPU and other devices on the same PCB, as the same power supply and CLK are used.	It is used to exchange data with other equipment such as PC.
Ex: SPI, I2C	Ex: UART

Serial Communication Protocols :

The universal serial communication interface (**USCI**) module of MSP430 supports multiple serial communication modes. The THREE common types of serial communication protocols are

- Asynchronous serial communication (UART)
- Serial peripheral interface (SPI).
- Inter-integrated circuit (I^2C) bus.

Universal Asynchronous Receiver / Transmitter (UART)

- It is an **Asynchronous - Full duplex** Serial communication protocol
- No CLK signal is transmitted along with data
- UART requires 2- signal lines for communication

  TxD - Transmit data

  RxD – Receive data

Serial Peripheral Interface (SPI) :

- It is a **Synchronous - Full duplex** Serial communication protocol
- CLK signal is transmitted along with data
- It is a 4-wire communication (3-pin SPI mode and 4-pin SPI modes)
- SPI requires 4- signal lines for communication:

  MOSI - Master out slave in

  MISO - Master in slave out

  SCLK – Serial Clock

  SS - Slave select

Inter Integrated Circuit (I2C) :

- It is a **Synchronous - Half duplex** Serial communication protocol
- CLK signal is transmitted along with data
- It is often called as 2-wire interface
- The I^2C bus uses only two bidirectional lines for communication

  SDA - Serial data

  SCL – Serial Clock

✓ SPI and I^2C have similar applications. SPI and I^2C are often used to communicate with

- Port expanders to increase the effective no. of pins for digital I/O.
- ADCs and DACs.
- Sensors with digital outputs, such as thermometers.
- External memory (data flash, EEPROM).
- Real-time clocks.
- Interfacing of other processors.

5.2. UART (Universal Asynchrnous Receiver Trasmitter)

Universal Asynchronous Receiver / Transmitter (UART)

- It is an **Asynchronous - Full duplex** Serial communication protocol
- No CLK signal is transmitted along with data
- UART requires 2- signal lines for communication

  TxD - Transmit data

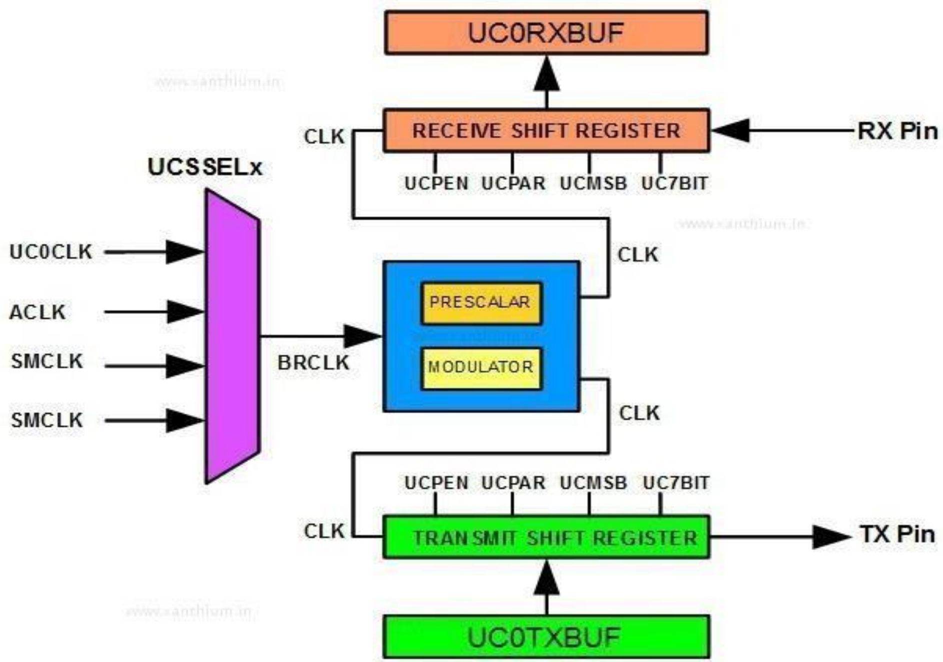
  RxD – Receive data

- The UART takes bytes of data and transmits the individual bits in a sequential fashion.
- At the destination, a second UART reassembles the bits into complete bytes.
- Each UART contains a shift register, which is the fundamental method of conversion between serial and parallel forms.

- The UART module consists of

  TX section - Transmit Buffer [TXBUF] and Transmit Shift Register

  RX section - Receive Buffer [RXBUF] and Receive Shift Register



UART mode features include:

1. 7 or 8-bit data with odd, even, or non-parity
2. Independent transmit and receive shift registers
3. Separate transmit and receive buffer registers
4. LSB-first or MSB-first data transmit and receive
5. Built-in idle-line and address-bit communication protocols for multiprocessor systems
6. Programmable baud rate
7. Status flags for error detection and suppression
8. Status flags for address detection
9. Independent interrupt capability for receive and transmit

Format of Data for Asynchronous Transmission

Data are sent in short *frames*, and each frame contains

- One low start bit (ST).
- Eight data bits, usually LSB first
- Address bit (used in address bit mode)
- Parity bit (PA)
- One or Two high stop bits (SP)

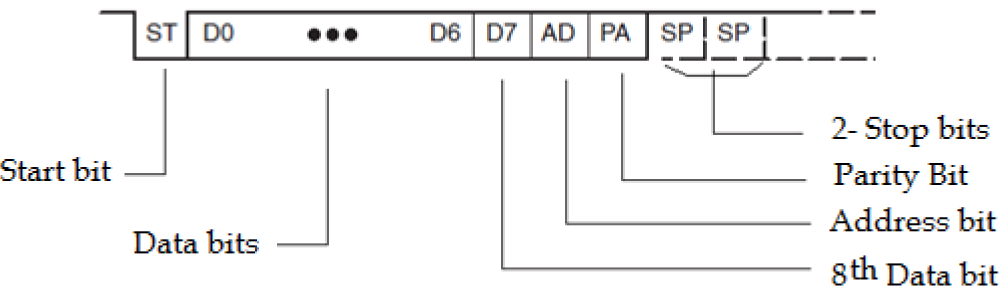


Figure : UART character format

The start bit signals the receiver that a new character is coming. The next bits represent the character. If a parity bit is used, it would be placed after all of the data bits. The next one or two bits are always in the **mark** (logic high, i.e., '1') condition and called the stop bit(s). They signal the receiver that the character is completed. Since the start bit is logic low (0) and the stop bit is logic high (1) there are always at least two guaranteed signal changes between characters.

All operations of the UART hardware are controlled by a clock signal which runs at a multiple of the data rate, typically 8 times the bit rate. The receiver tests the state of the incoming signal on each clock pulse, looking for the beginning of the start bit.

Setting a Baud Rate

- The *baud rate* gives the frequency at which bits are transmitted on the line. It is the inverse of the bit period and the name is used to distinguish it from the rate at which useful data are communicated. If the baud rate is ‘x’ bits per second, the time slot available for one bit is 1/x seconds.

- Each 8 bits of data are accompanied by a start and stop bit so the maximum data rate is only 8/10 of the baud rate.
- No clock is transmitted in asynchronous communication so the transmitter and receiver must run independently at nearly the same baud rates. For a given BRCLK clock source, the baud rate is used determines the required division factor N :

  $N = f_{BRCLK}/Baudrate$

- The division factor N is often a non-integer value, thus, at least one divider and one modulator stage is used to meet the factor as closely as possible.

Automatic Error Detection

The USCI module automatically detects framing errors, parity errors, overrun errors, and break conditions when receiving characters. The bits UCFE, UCPE, UCOE, and UCBRK are set when their respective condition is detected. When the error flags UCFE, UCPE, or UCOE are set, UCRXERR is also set.

Error	Error flag	Description
Framing error	UCFE	Framing error occurs when a low stop bit is detected
Parity error	UCPE	Parity error is mismatch between the no.of 1's in a character and the value of parity bit.
Over run error	UCOE	Overrun error occurs when a character is loaded into UCARxBUF before the prior character has been read.
Break condition	UCBRK	Break is detected when all data, parity and stop bits are LOW.

UCAxCTL0 (USCI_Ax Control register0) in UART mode :

This register controls the settings for Parity selection, direction of data transmission (LSB or MSB first),character length, no of stop bits, modes of serial transmission.

7	6	5	4	3	2	1	0
UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC=0

Bit 7	UCPEN	Parity enable	0 → Disables Parity, 1 → Enables Parity
Bit 6	UCPAR	Parity select	0 → Odd parity, 1 → Even parity
Bit 5	UCMSB	MSB first select.	0 → LSB first, 1 → MSB first
Bit 4	UC7BIT	Character length	0 → 8-bit data, 1 → 7-bit data
Bit 3	UCSPB	No. of Stop bits select.	0 → 1-Stop bit, 1 → 2-Stop bits
Bit 2-1	UCMODEx	USCI mode.	If UCSYNC = 0 then 00 → UART mode 01 → Idle-line multiprocessor mode 10 → Address-bit multiprocessor mode 11 → UART mode with Auto baud-rate detection
Bit 0	UCSYNC	Synchronous mode enable	0 → Asynchronous mode 1 → Synchronous mode

5.3. SERIAL PERIPHERAL INTERFACE (SPI)

Serial Peripheral Interface (SPI) :

- It is a **Synchronous - Full duplex** Serial communication protocol
- CLK signal is transmitted along with data
- It is a 4-wire communication (3-pin SPI mode and 4-pin SPI modes)
- SPI requires 4- signal lines for communication:

  MOSI : Master out slave in
  MISO : Master in slave out
  SCLK : Serial Clock
  SS : Slave select

The concept of SPI is shown in figure

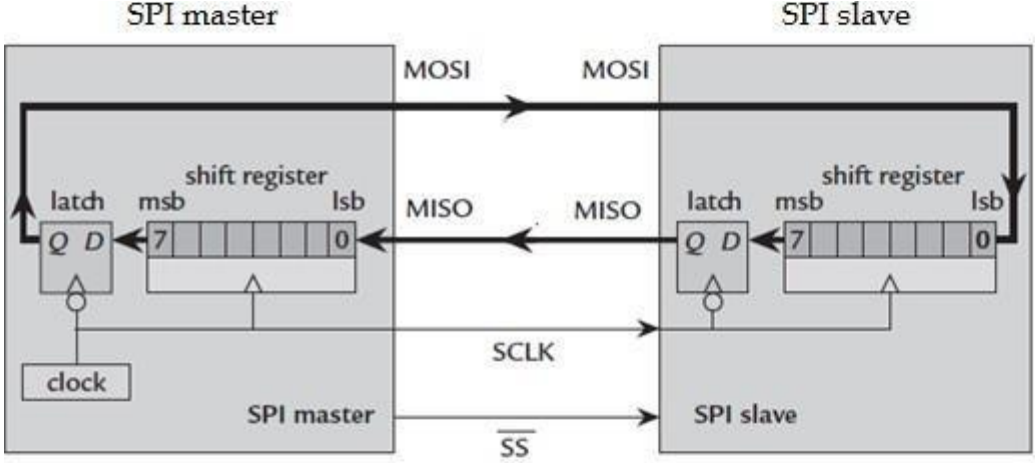


Figure : Serial peripheral interface between a master and a single slave

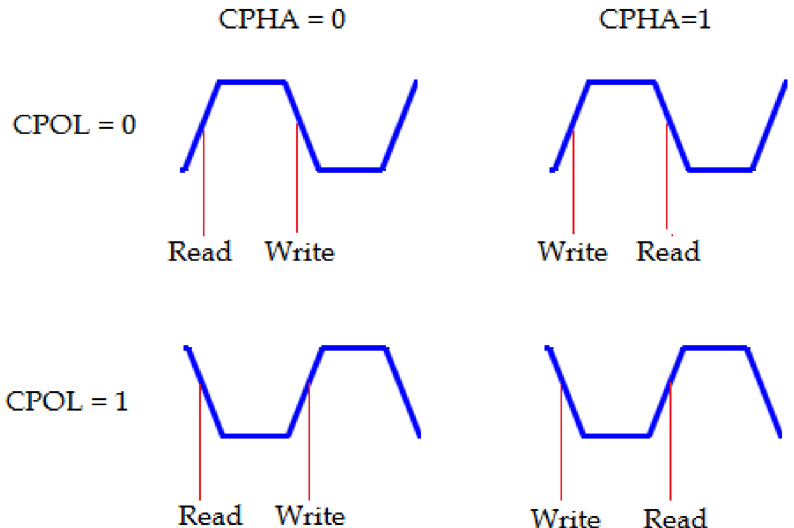
- The SPI transmits data simultaneously in both directions (full duplex) between two devices. One device is the master and the other the slave.
- The master provides the clock for both devices and a signal to select the slave (SS)
- SPI works on the principle of 'Shift Registers'. The master and slave devices contain a special shift register for the data transmit and receive.
- For every clock pulse,

  Master will send a bit to the Slave through MOSI pin and at the same time
  Slave will send a bit to the Master through MISO pin

Thus a bit is transferred in each direction during each clock cycle. After eight cycles, the contents of the shift registers have been exchanged and the transfer is complete. Transmission and reception are clearly inseparable: Thus a byte must be transmitted in order to receive a byte.

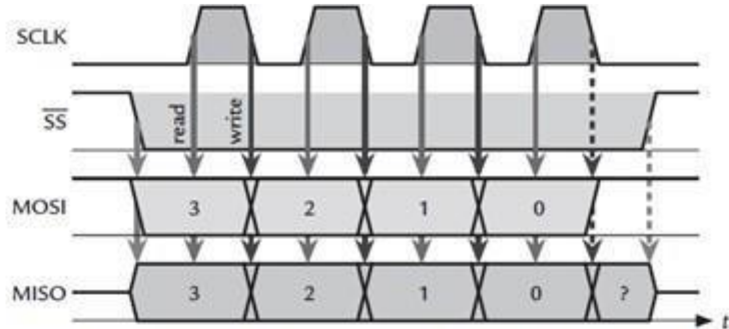
Four standard modes of SPI :

- For every clock pulse, the SPI performs two operations – Read data bit and Send data bit.
- These two operations are done at two edges of the clock signal. Hence, there are 4-standard modes of SPI depends on Clock Polarity (CPOL) and Clock Phase (CPHA)
- The CPOL selects the active state
  CPOL = **0:** Active state is HIGH (Positive Clock)
  CPOL = **1:** Active state is LOW (Negative Clock)
- The CPHA selects Read/Write operation at Rising/Falling edges of the clock
  CPHA = **0 :** Read on Rising edge and Write on Falling edge of the clock pulse
  CPHA = 1 **:** Write on Rising edge and Read on Falling edge of the clock pulse.

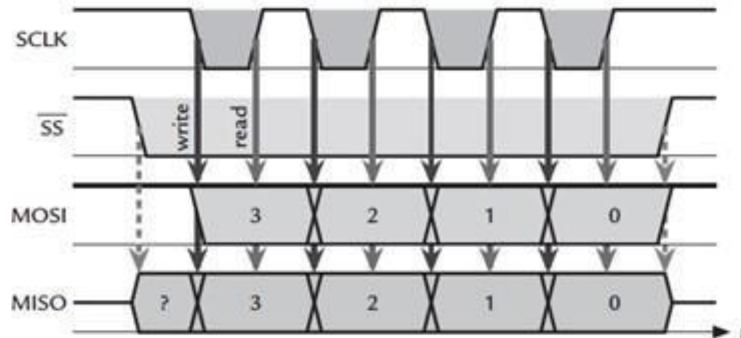


CPOL CPHA	SPI Mode	Data Read & Write time
0 0	Mode-0	Read on Rising edge, Write on Falling edge
0 1	Mode-1	Write on Rising edge, Read on Falling edge,
1 0	Mode-2	Read on Falling edge, Write on Rising edge
1 1	Mode-3	Write on Falling edge, Read on Rising edge

The following figures illustrate complete waveforms for a 4-bit transfer in modes 0 and 3



Fig(a) : SPI operation in Mode-0 [CPOL=0, CPHA=0]



Fig(b) : SPI operation in Mode-3 [CPOL=1, CPHA=1]

USCI - SPI Mode

- SPI mode is selected when the UCSYNC bit = 1 and SPI mode (3-pin or 4-pin) is selected with the UCMODEx bits of USCI control register.

- SPI mode features include:
  1. 7- or 8-bit data length
  2. LSB-first or MSB-first data transmit and receive
  3. 3-pin and 4-pin SPI operation
  4. Master or slave modes
  5. Independent transmit and receive shift registers
  6. Separate transmit and receive buffer registers
  7. Continuous transmit and receive operation
  8. Selectable clock polarity and phase control
  9. Programmable clock frequency in master mode
  10. Independent interrupt capability for receive and transmit

- In SPI mode of USCI, the MSP430 is connected to an external system via 3 (or) 4 pins: UCxSIMO, UCxSOMI, UCxCLK, and UCxSTE.
  UCxSIMO : Slave In and Master Out
  UCxSOMI : Slave Out and Master In
  UCxCLK : USCI Clock
  UCxSTE : Slave Transmit Enable

In master mode, UCxSIMO, UCxCLK, and UCxSTE are outputs and UCxSOMI is the input. In slave mode, UCxSIMO, UCxCLK, and UCxSTE are inputs and UCxSOMI is the output.

USCI - SPI master mode :

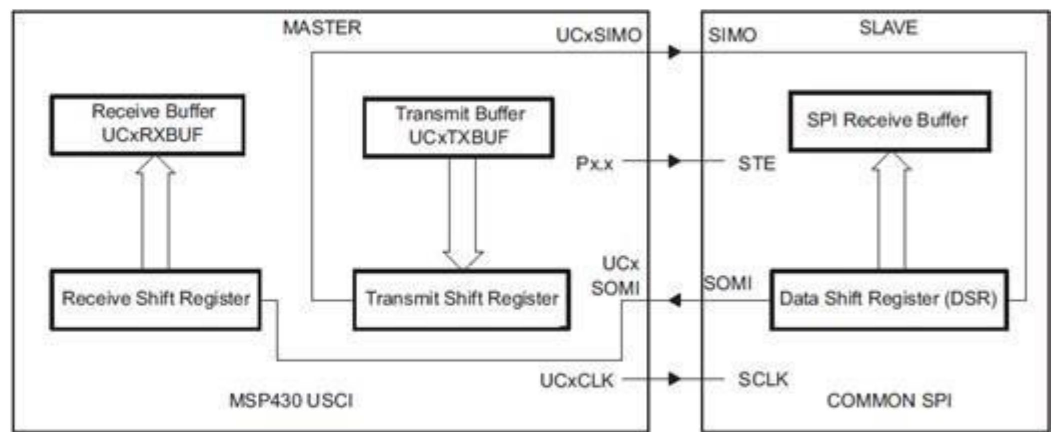


Figure 16-2. USCI Master and External Slave

USCI – SPI slave mode :

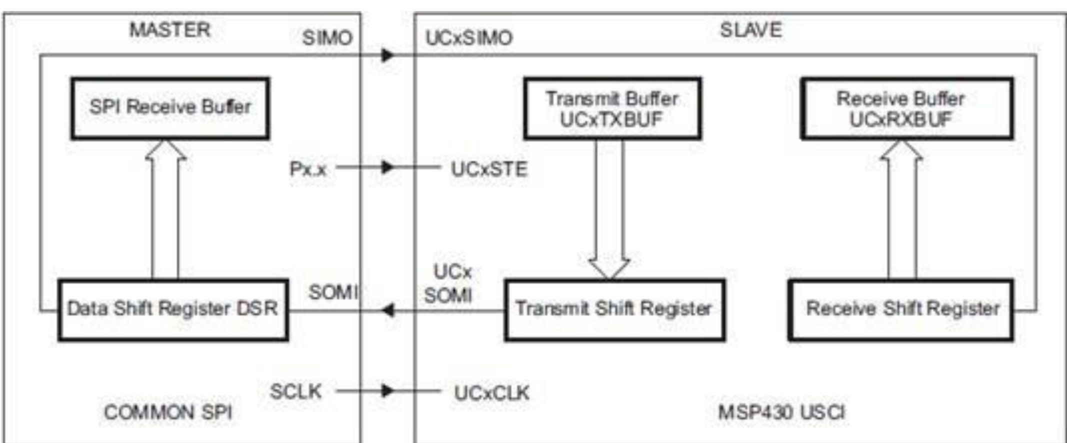


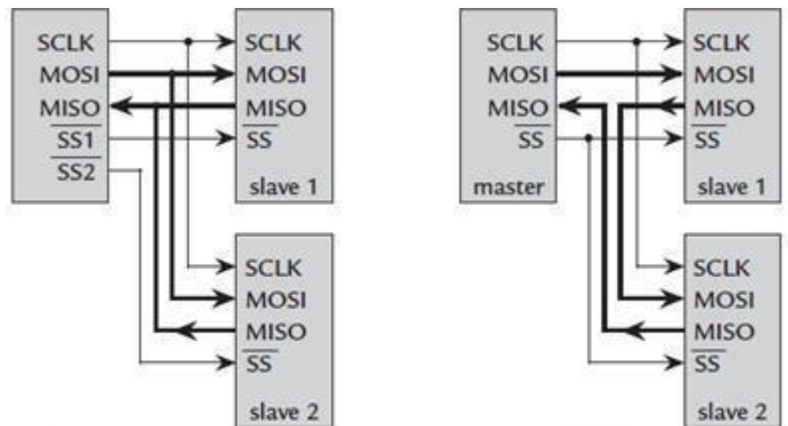
Figure 16-3. USCI Slave and External Master

- The above figure shows the USCI module in SPI mode with 3-pin and 4-pin configurations.
- The master module provides the UCxCLK to the slave device.
- The USCI initiates data transfer when data is moved to the transmit data buffer TXBUF.
- The data in TXBUF is moved to the TX shift register.

- The data in TX shift register is shifted out to RX receive buffer of the receiver, serially starting with either the MSB bit or LSB bit.
- When a character is received, the data in RX shift register is moved to RX buffer UCxRXBUF and the receive interrupt flag (RXIFG) is set, indicating the RX/TX operation is completed.
- A set transmit interrupt flag (TXIFG) indicates that data has moved from UCxTXBUF to the TX shift register and TXBUF is ready for new data transmission.
- The overrun error bit is set when the previously received data is not read from RXBUF before new data is moved to RXBUF.

Connecting Multiple Slaves to a Single Master in SPI :

There are two ways of connecting multiple slaves to a single master. In both cases the master provides the clock to all the slaves. Slaves can be addressed individually.



Two ways of connecting two slaves to a single master using SPI.

(a) Bus with slaves individually selected (b) Daisy chain

In the first configuration,

- The MOSI and MISO pins of Master are connected to respective pins of all Slaves
- The Master uses separate SS signals to select the one of the slaves.
- In this configuration, the data transmission takes place between Master and any one of the selected Slave device.

In the second configuration,

- All the devices are connected in a “daisy chain” as shown in the figure.
- The master will send a bit to the first slave, the first slave will send a bit to the next slave and the final slave will send a bit to the master.
- The MISO pin of first slave is connected to MOSI pin of next slave in the chain. The MISO pin of final slave is connected to MISO of Master. Effectively all the shift registers inside each device are connected into a single, long loop.
- The SS pin of master is connected to all the slaves.

UCAxCTL0 (USCI_Ax Control Register 0) in SPI mode

7	6	5	4	3	2 1	0
UCCKPH	UCCKPL	UCMSB	UC7BIT	UCMST	UCMODEx	UCSYNC=1

Bit 7	UCCKPH	Clock Phase select	0 - Read on rising edge & Write on falling edge 1 - Write on rising edge & Read on falling edge
Bit 6	UCCKPL	Clock Polarity select	0 - Active state is HIGH (Positive Clock) 1 - Active state is LOW (Negative Clock)
Bit 5	UCMSB	MSB first select	0 → LSB first, 1→ MSB first
Bit 4	UC7BIT	Character length	0 → 8 bit data, 1 → 7 bit data
Bit 3	UCMST	Master mode select	0 → Slave mode, 1 → Master mode
Bit 2-1	UCMODEx	USCI mode.	If UNSYNC = 1 then 00 : 3-pin SPI 01 : 4-pin SPI with UCSTE active high 10 : 4-pin SPI with UCSTE active high 11 : I2C
Bit 0	UCSYNC	Synchronous mode enable	0 → Asynchronous mode 1 → Synchronous mode

5.4. INTER INTEGRATED CIRCUIT (I2C) BUS

Inter Integrated Circuit (I2C) :

- It is a **Synchronous - Half duplex** Serial communication protocol
- CLK signal is transmitted along with data
- It is often called as 2-wire interface
- The I²C bus uses only two bidirectional lines for communication

  SDA - Serial data

  SCL – Serial Clock

Hardware for I²C

- The electronic interface to the I²C bus is shown in figure for a master and two slaves.
- The transfers on the bus take place between a master and selected slave.
- Each slave has a unique address, which is usually 7 bits long.
- The master starts the transfer, provides the clock, addresses a particular slave, manages the transfer, and finally terminates it.
- There may be more than one master on the bus, but only one can be in control at a time.

- The pull-up resistor R_P holds the line at V_{CC} when there is no activity.
- Hence, both the clock and data lines are at HIGH. If a single n-MOSFET is turned on, its line is pulled down to logic 0.

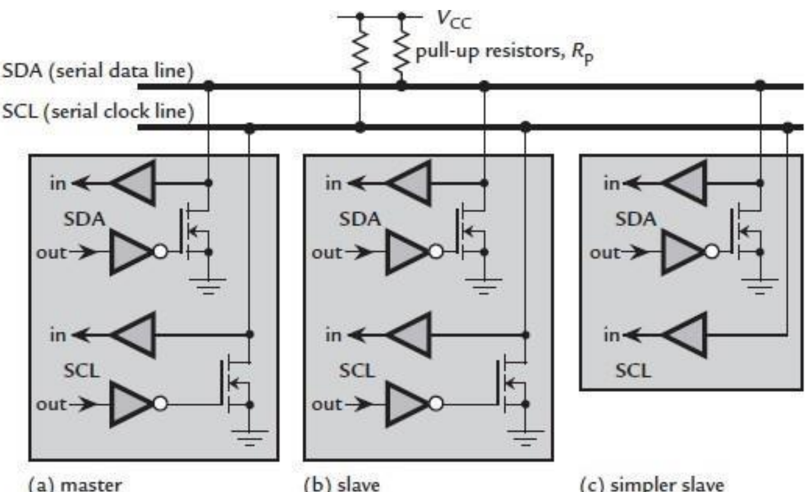


Figure 10.11: Electronic interface to the I²C bus. The two lines of the bus, SCL and SDA, are bidirectional and pulled up to V_{CC} with resistors R_p. (a) A master device can read and write both SCL and SDA independently. (b) A slave may have an identical interface but (c) most slaves cannot drive SCL.

USCI - I2C Mode

- In I2C mode, the USCI module provides an interface between the MSP430 and I2C-compatible devices connected by way of the two-wire I2C serial bus.
- External components attached to the I2C bus serially transmit and/or receive serial data to/from the USCI module through the 2-wire I2C interface.
- The I2C mode features include:
  1. 7-bit and 10-bit device addressing modes
  2. Multi-master transmitter/receiver mode
  3. Slave receiver/transmitter mode
  4. Standard mode up to 100 kbps and fast mode up to 400 kbps support
  5. Programmable UCxCLK frequency in master mode
  6. Designed for low power
  7. Slave receiver START detection for auto-wake up from LPMx modes
  8. Slave operation in LPM4

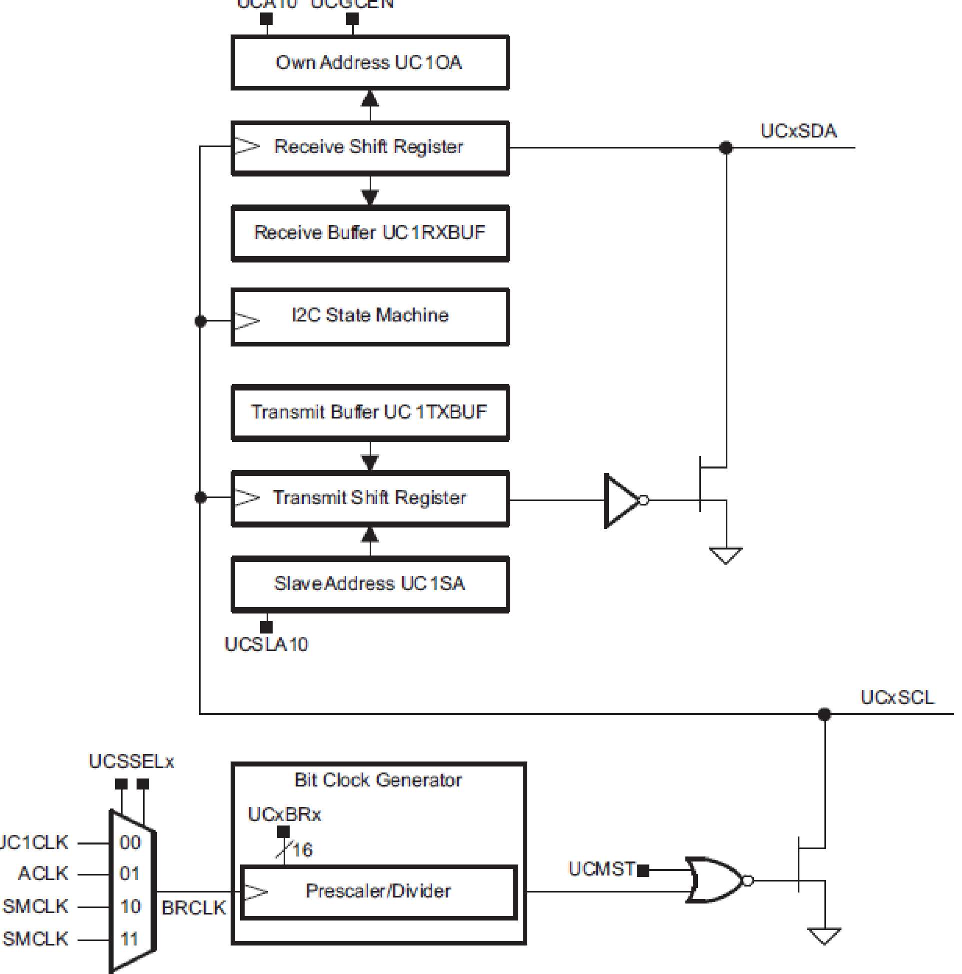


Figure 17-1. USCI Block Diagram: I²C Mode

USCI Operation: I2C Mode

- The I2C mode supports any slave or master I2C-compatible device.
- Each I2C device is recognized by a unique address and can operate as either a transmitter or a receiver.
- The master initiates a data transfer and generates the clock signal SCL. Any device addressed by a master is considered a slave.
- I2C data is communicated using the serial data pin (SDA) and the serial clock pin (SCL).
- Both SDA and SCL are bidirectional, and must be connected to a positive supply voltage using a pull-up resistor.
- The clock source can be selected from ACLK, SMCLK, and UC1CLK

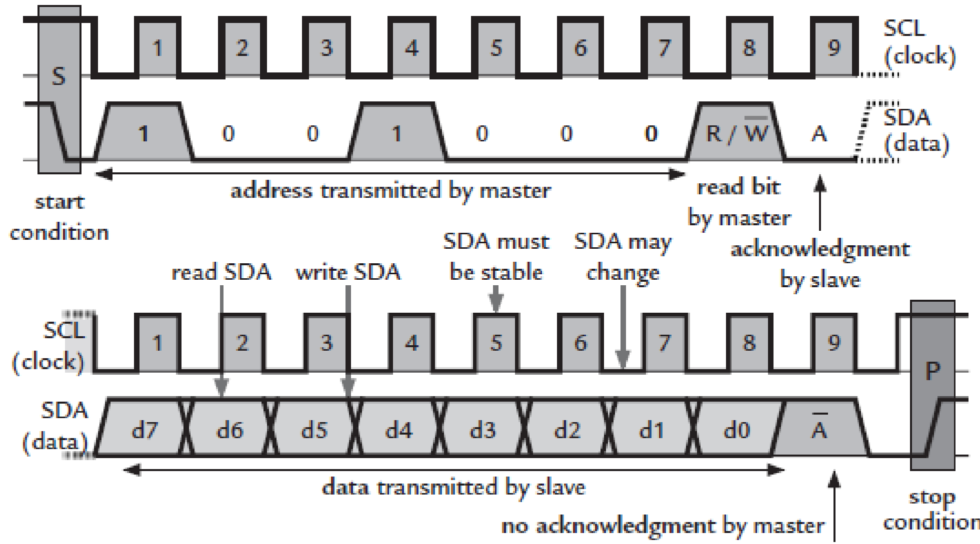


Figure 10.12: Simple transfer over I²C. The master writes an address, which is acknowledged by the slave, and reads a single byte from the slave.

A simple example of a transfer on I²C is shown in above, where the master reads a single byte from the slave. Transfers consist of a sequence of 8-bit bytes, which are sent with the MSB first and must be acknowledged to confirm successful reception.

The steps involved in a simple transfer over I2C bus

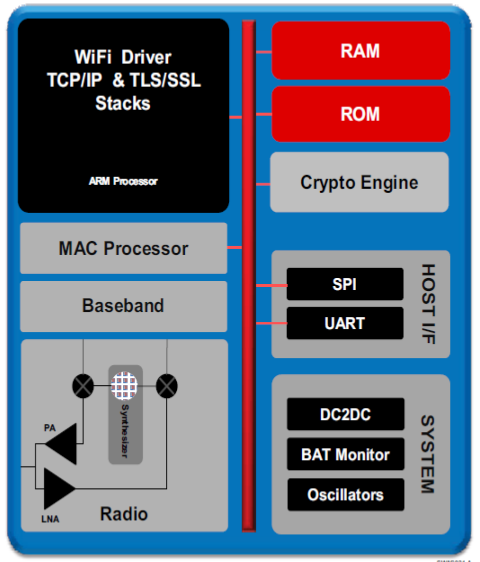
1. The master sends a start condition (S) by pulling SDA low while SCL is high.
2. The master sends the 7-bit slave address to select a slave
3. The master sends the control bit to select the Read(1) / Write(0) operation.
4. The selected slave will send the acknowledgment (A)
5. After receiving ACK from selected slave, the data transfer takes place for next 8-clock pulses.
6. After the transfer of 8-bits, the receiver sends the ACK
7. Finally, the master sends the stop condition (P) by pulling SDA high while SCL is high.

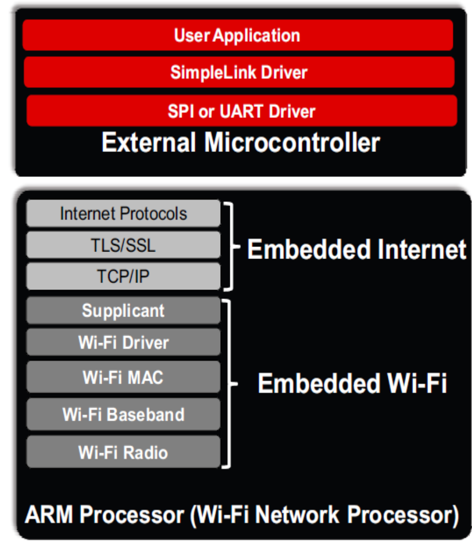
UCAxCTL0 (USCI_Ax Control Register 0) in I2C mode :

7	6	5	4	3	2	1	0
UCA10	UCSLA10	UCMM	--	UCMST	UCMODEx = 11		UCSYNC=1

Bit 7	UCA10	Own Address mode select	0 → Own address is a 7-bit address 1 → Own address is a 10-bit address
Bit 6	UCSLA10	Slave addressing mode select	0 → Address slave with 7-bit address 1 → Address slave with 10-bit address
Bit 5	UCMM	Multi-master environment select	0 Single master environment. 1 Multi-master environment
Bit 4	NOT USED		
Bit 3	UCMST	Master mode select	0 → Slave mode 1 → Master mode
Bit 2-1	UCMODEx	USCI mode.	If UNSYNC = 1 then 00 → 3-pin SPI 01→ 4-pin SPI with UCSTE active high 10 → 4-pin SPI with UCSTE active high 11→ I2C
Bit 0	UCSYNC	Synchronous mode enable	0 → Asynchronous mode 1 → Synchronous mode

5.5. SIMPLE LINK WI-FI NETWORK PROCESSOR : CC3100





The CC3100 is a **Simple Link Wi-Fi Network Processor** which integrates all protocols for Wi-Fi and Internet with built-in security protocols.

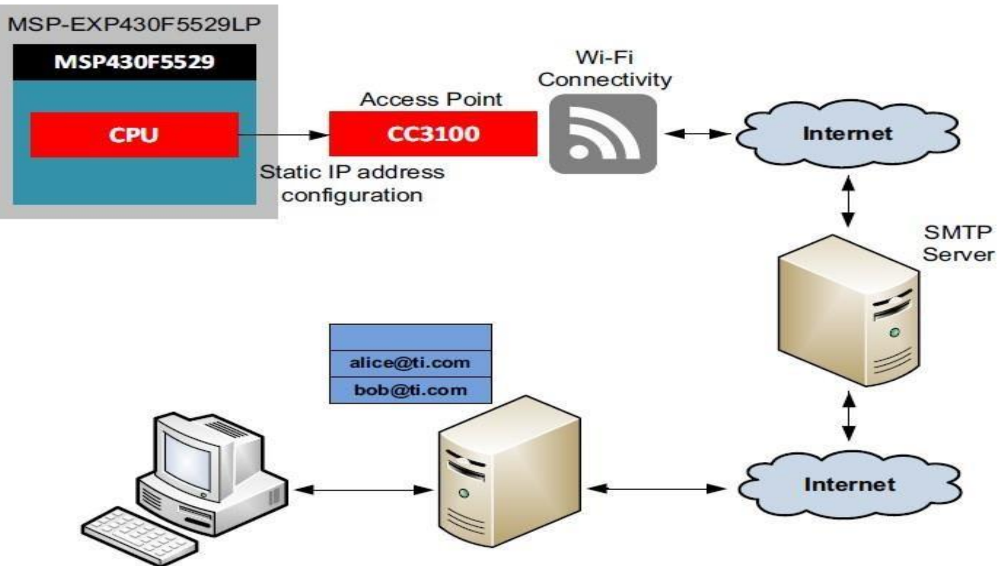
The features of CC3100 :

- It has dedicated ARM MCU that completely offloads the host MCU.
- Embedded TCP/IP & TLS/SSL stacks, HTTP server, Wi-Fi Driver and Multiple Internet Protocols in ROM
- 802.11 b/g/n Radio, Baseband, and MAC with a Powerful Crypto Engine for fast, secure Wi-Fi and Internet connections with 256-Bit AES.
- Industry-Standard Application Programming Interfaces (APIs)
- WPA2 Personal and Enterprise Security
- Host interface : Interfaces with 8/16/32-bit MCU over SPI (or) UART Interface
- 7KB ROM (Code Memory) and 700 Bytes RAM (Data memory)
- Power Management System
- Advanced Low-Power Modes
  - *Hibernate with RTC* : *4 μA*
  - *Low-Power Deep Sleep (LPDS)* : *115 μA*
  - *Idle Connected* : *690 μA*
  - *RX Traffic (MCU Active)* : *53 mA*
  - *TX Traffic (MCU Active)* : *223 mA*
- Clock Source
  - *40 MHz Crystal with Internal Oscillator*
  - *32.768 kHz Crystal (or) External RTC Clock*

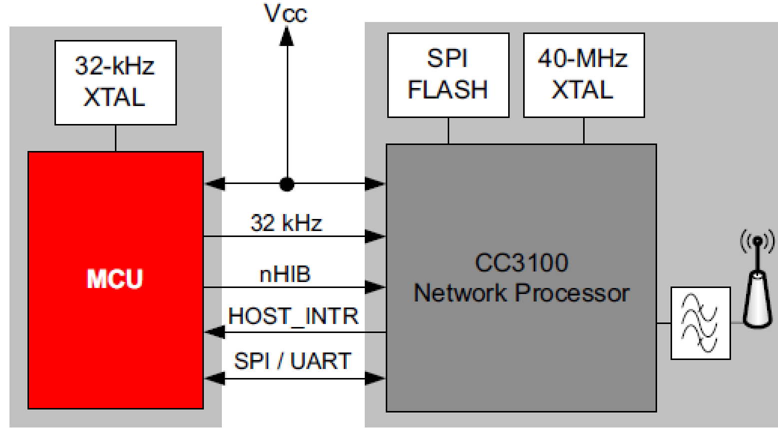
Applications of CC3100 :

(1) Connected appliances
(2) IoT applications
(3) Smart energy
(4) M2M communication
(5) Cloud connectivity
(6) Home automation
(7) Security systems
(8) Industrial control
(9) IP network sensor nodes

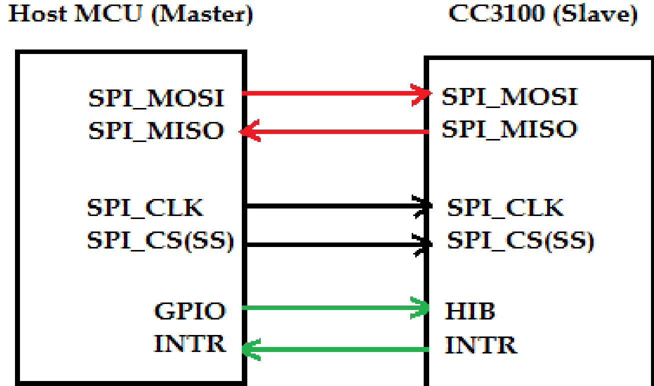
5.6. ADDING WI-FI CAPABILITY TO MCU



- Microcontroller based design can be easily connected to the internet with Wi-Fi using the TI Simple Link CC3100 network processor module.
- The CC3100 is a **Simple Link Wi-Fi Network Processor** which integrates all protocols for Wi-Fi and Internet with built-in security protocols.
- The CC3100 minimizes the software requirements of the host microcontroller (MCU) and is thus the ideal solution for embedded networking applications using any low-cost and low-power MCU.
- The CC3100 has Embedded TCP/IP & TLS/SSL stacks, HTTP server, Wi-Fi Driver and Multiple Internet Protocols in ROM
- The CC3100 technology can transfer the information between your design and networked PCs, smart phones, and tablets with just SPI communication connectivity and basic Wi-Fi connectivity knowledge.
- The CC3100 device can connect to any 8/16/32-bit MCU over the SPI or UART Interface



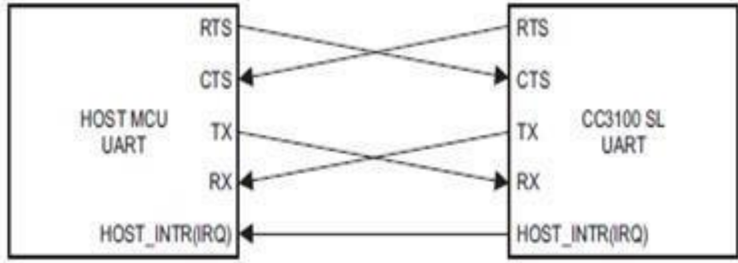
(a) SPI Host Interface



The MSP430F5529 and CC3100BOOST are connected via the SPI interface as shown in aboce. The CC3100 device can interrupt the host using the INTR line to initiate the data transfer over the interface. The SPI host interface can work up to a speed of 20 MHz.

Pin Name	Description
SPI_MOSI	Data from Host MCU to CC3100 device
SPI_MISO	Data from CC3100 device to Host MCU
SPI_CLK	Clock from Host MCU to CC3100 device (up to 20MHz)
SPI_nCS	CS (active low) signal from Host MCU to CC3100 device
nHIB	Active-low signal that commands the CC3100 device to enter hibernate mode (lowest power state)
INTR	Interrupt from CC3100 device to Host MCU

(b) 5-wire UART Interface



The above figure shows a typical 5-wire UART topology comprised of 4 standard UART lines and one INTR Request line from CC3100 to Host MCU to allow efficient low power mode.

Pin Name	Description
RTS (Request to Send)	Signal from Host MCU to CC3100 device
CTS (Clear to Send)	Signal from CC3100 device to Host MCU
TxD (Transmit data)	Data transmit pin
RxD (Receive data)	Data receive pin
INTR	Interrupt Request from CC3100 device to MCU host