8086 ARCHITECTURE

8086 Features

- 16-bit Arithmetic Logic Unit
- 16-bit data bus
- 20-bit address bus – 1,048,576 = 1 meg
- 16 I/O lines so it can access 64K I/O ports
- 16 bit flag
- It has 14 -16 bit registers
- Clock frequency range is 5-10 MHZ
- Designed by Intel
- Rich set of instructions
- 40 Pin DIP, Operates in two modes

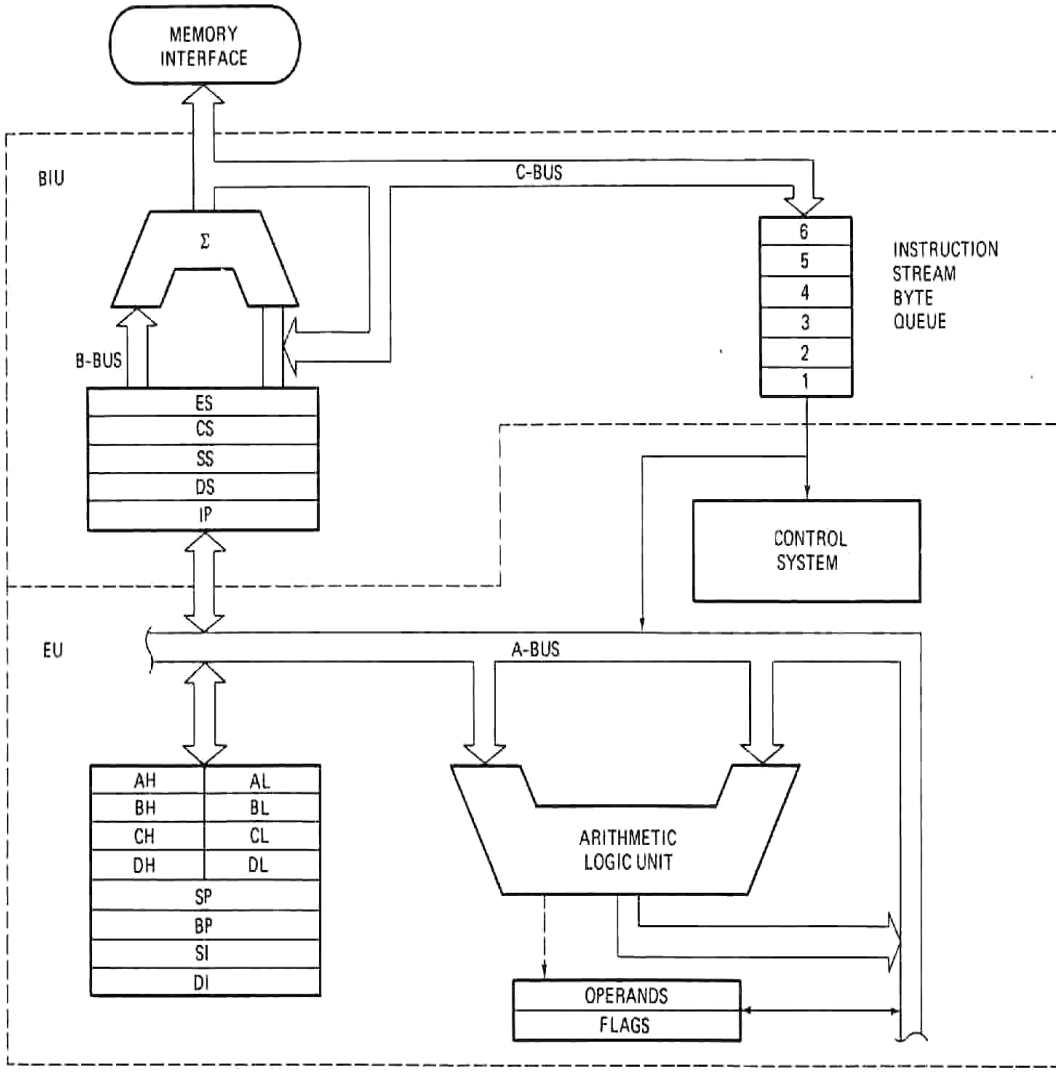


FIGURE 2-7 8086 internal block diagram. (*Intel Corp.*)

The address refers to a byte in memory. In the 8086, bytes at even addresses come in on the low half of the data bus (bits 0-7) and bytes at odd addresses come in on the upper half of the data bus (bits 8-15).The 8086 can read a 16-bit word at an even address in one operation and at an odd address in two operations. The least significant byte of a word on an 8086 family microprocessor is at the lower address.

The 8086 has two parts, the Bus Interface Unit (BIU) and the Execution Unit (EU).The BIU fetches instructions, reads and writes data, and computes the 20-bit address. The EU decodes and executes the instructions using the 16-bit ALU.
The BIU contains the following registers:

- IP - the Instruction Pointer
- CS - the Code Segment Register
- DS - the Data Segment Register
- SS - the Stack Segment Register
- ES - the Extra Segment Register

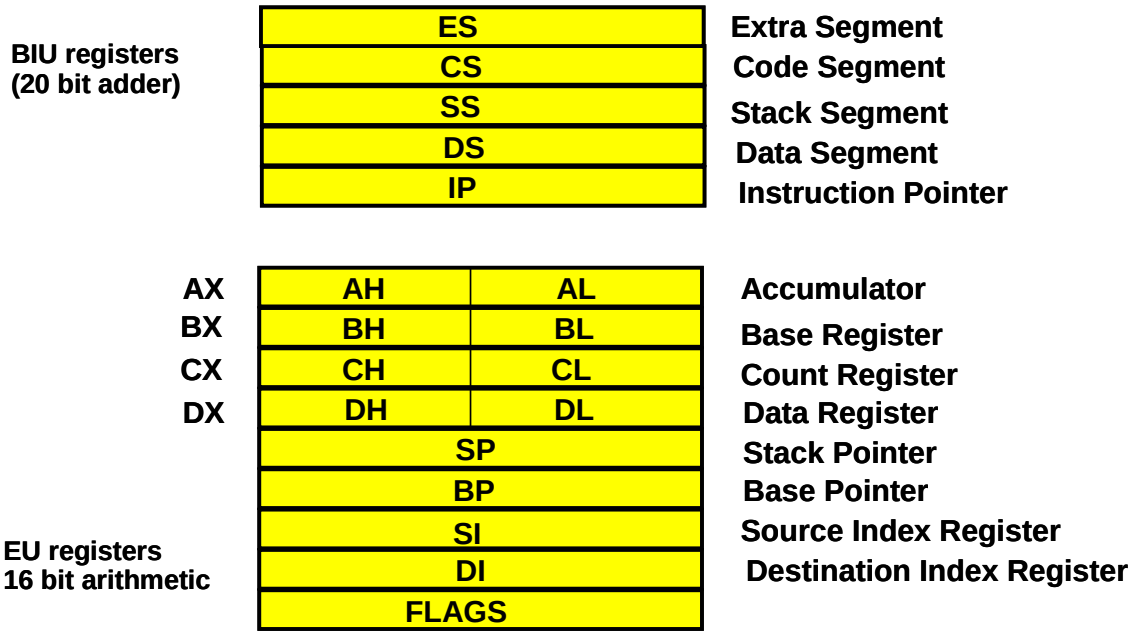
The BIU fetches instructions using the CS and IP, written CS: IP, to construct the 20-bit address. Data is fetched using a segment register (usually the DS) and an effective address (EA) computed by the EU depending on the addressing mode.
The EU contains the following 16-bit registers:

- AX - the Accumulator
- BX - the Base Register
- CX - the Count Register
- DX - the Data Register
- SP - the Stack Pointer
- BP - the Base Pointer
- SI - the Source Index Register
- DI - the Destination Register

These are referred to as general-purpose registers, although, as seen by their names, they often have a special-purpose use for some instructions. The AX, BX, CX, and DX registers can be considered as two 8-bit registers, a High byte and a Low byte. This allows byte operations and compatibility with the previous generation of 8-bit processors, the 8080 and 8085. The 8-bit registers are:

- AX --> AH,AL
- BX --> BH,BL
- CX --> CH,CL
- DX --> DH,DL

The ALU performs all basic computational operations: arithmetic, logical, and comparisons. The control unit orchestrates the operation of the other units. It fetches instructions from the on-chip cache, decodes them, and then executes them. Each instruction has the control unit direct the other function units through a sequence of steps that carry out the instruction's intent. The execution path taken by the control unit can depend upon status bits produced by the arithmetic logic unit or the floating-point unit (FPU) after the instruction sequence completes. This capability implements conditional execution control flow, which is a critical element for general-purpose computation.



Most of the registers contain data/instruction offsets within 64 KB memory segment. 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.

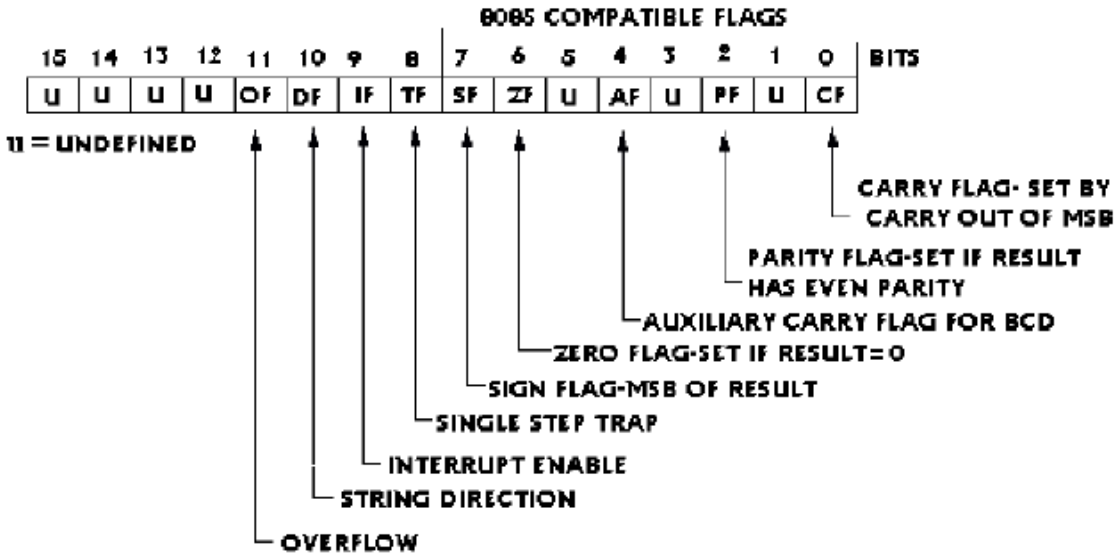
Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter 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 EU also contains the Flag Register which is a collection of condition bits and control bits. The condition bits are set or cleared by the execution of an instruction. The control bits are set by instructions to control some operation of the CPU.

- Bit 0 - CF Carry Flag - Set by carry out of MSB
- Bit 2 - PF Parity Flag - Set if result has even parity
- Bit 4 - AF Auxiliary Flag - for BCD arithmetic
- Bit 6 - ZF Zero Flag - Set if result is zero
- Bit 7 - SF Sign Flag = MSB of result
- Bit 8 - TF Single Step Trap Flag
- Bit 9 - IF Interrupt Enable Flag
- Bit 10 - DF String Instruction Direction Flag
- Bit 11 - OF Overflow Flag
- Bits 1, 3, 5, 12-15 are undefined



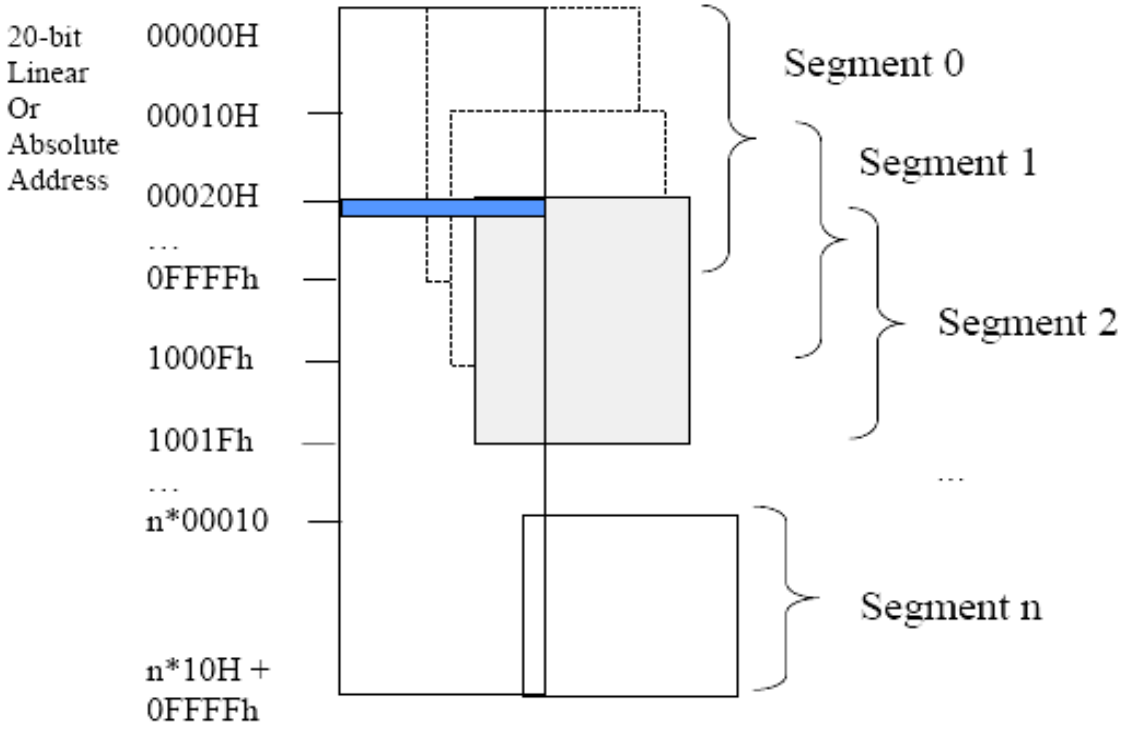
SEGMENT REGISTERS

It is used to store the memory addresses of instructions and data. Memory Organization. Each byte in memory has a 20 bit address starting with 0 to 220-1 or 1 Meg of addressable memory. Addresses are expressed as 5 hex digits from 00000 – FFFFF.

1. Problem: But 20 bit addresses are TOO BIG to fit in 16 bit registers?
Solution: Memory Segment.

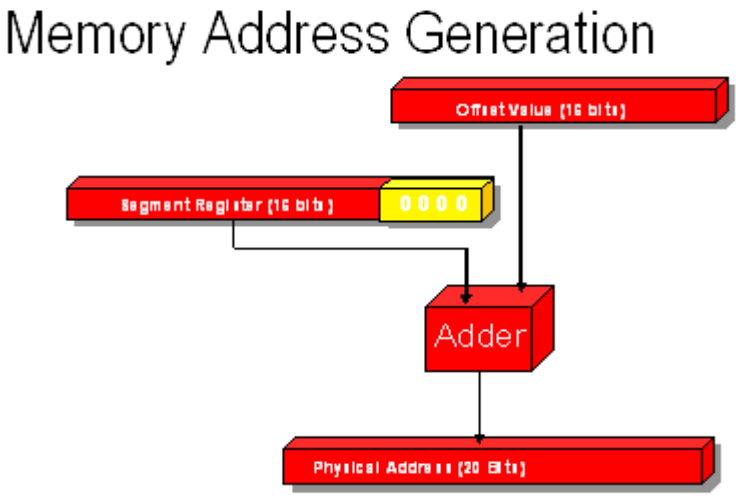
Block of 64K (65,536) consecutive memory bytes. A segment number is a 16 bit number. Segment numbers range from 0000 to FFFF. Within a segment, a particular memory location is specified with an offset. An offset also ranges from 0000 to FFFF

Memory Model for 20-bit Address Space



Question: how to generate memory address?
Ans: Physical address = segment address*10+offset address

Example: we have segment no 6020h and offset is 4267h then 60200+4267=64467h← physical address.



Memory:

Program, data and stack memories occupy the same memory space. As the most of the processor instructions use 16-bit pointers the processor can effectively address only 64 KB of memory. To access memory outside of 64 KB the CPU uses special segment registers to specify where the code, stack and data 64 KB segments are positioned within 1 MB of memory 16-bit pointers and data are stored as: address: low-order byte ,address+1: high-order byte.

1. Program memory - program can be located anywhere in memory. Jump and call instructions can be used for short jumps within currently selected 64 KB code segment, as well as for far jumps anywhere within 1 MB of memory.
All conditional jump instructions can be used to jump within approximately +127 to -127 bytes from current instruction.

2. Data memory - the processor can access data in any one out of 4 available segments, which limits the size of accessible memory to 256 KB (if all four segments point to different 64 KB blocks).

Accessing data from the Data, Code, Stack or Extra segments can be usually done by prefixing instructions with the DS:, CS:, SS: or ES: (some registers and instructions by default may use the ES or SS segments instead of DS segment).Word data can be located at odd or even byte boundaries. The processor uses two memory accesses to read 16-bit word located at odd byte boundaries. Reading word data from even byte boundaries requires only one memory access.

3. Stack memory can be placed anywhere in memory. The stack can be located at odd memory addresses, but it is not recommended for performance reasons (see "Data Memory" above).

Reserved locations

0000h - 03FFh are reserved for interrupt vectors. Each interrupt vector is a 32-bit pointer in format segment: offset.FFFF0h - FFFFFh - after RESET the processor always starts program execution at the FFFF0h address.

ADDRESSING MODES OF 8086

The 8086 processors let you access memory in many different ways. The 8086 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 8086 addressing modes is the first step towards mastering 8086 assembly language.

- 8086 Register Addressing Modes
- 8086 Memory Addressing Modes
    - The Displacement Only Addressing Mode
    - The Register Indirect Addressing Modes
    - Indexed Addressing Modes
    - Based Indexed Addressing Modes
    - Based Indexed Plus Displacement Addressing Mode

1. 8086 Register Addressing Modes

Most 8086 instructions can operate on the 8086's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 8086 mov (move) instruction:

```
mov     destination, source
```

This instruction copies the data from the source operand to the destination operand. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 8086 mov instructions:

```
mov     ax, bx  ;Copies the value from BX into AX
mov     dl, al  ;Copies the value from AL into DL
mov     si, dx  ;Copies the value from DX into SI
mov     sp, bp  ;Copies the value from BP into SP
mov     dh, cl  ;Copies the value from CL into DH
mov     ax, ax  ;Yes, this is legal!
```

In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify cs as the destination operand, second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of cs to ds, you'd have to use some sequence like:

```
mov     ax, cs
mov     ds, ax
```

You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses. But more on that, later. Throughout this text you'll see the abbreviated operand sreg used wherever segment register operands are allowed (or required).

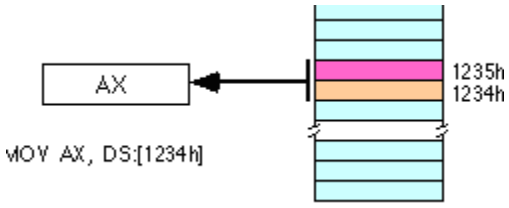
2. 8086 Memory Addressing Modes

- The Displacement Only Addressing Mode
- The Register Indirect Addressing Modes
- Indexed Addressing Modes
- Based Indexed Addressing Modes
- Based Indexed Plus Displacement Addressing Mode

2.1 The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the displacement-only (or direct) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory location 8088h. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h.

The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that. Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that- some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don't worry if this doesn't make a lot of sense right now. You'll get an opportunity to study segments to your heart's content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors:



By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a segment override prefix before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is not a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM's syntactical limitations.

2.2 The Register Indirect Addressing Modes

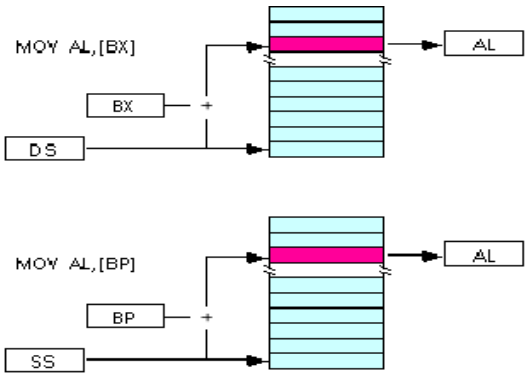
The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

```
mov   al, [bx]
mov   al, [bp]
mov   al, [si]
mov   al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment(ss)bydefault.You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov   al, cs:[bx]
mov   al, ds:[bp]
mov   al, ss:[si]
mov   al, es:[di]
```

Intel refers to [bx] and [bp] as base addressing modes and bx and bp as base registers (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as indexed addressing modes (si stands for source index, di stands for destination index). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.



2.3 Indexed Addressing Modes

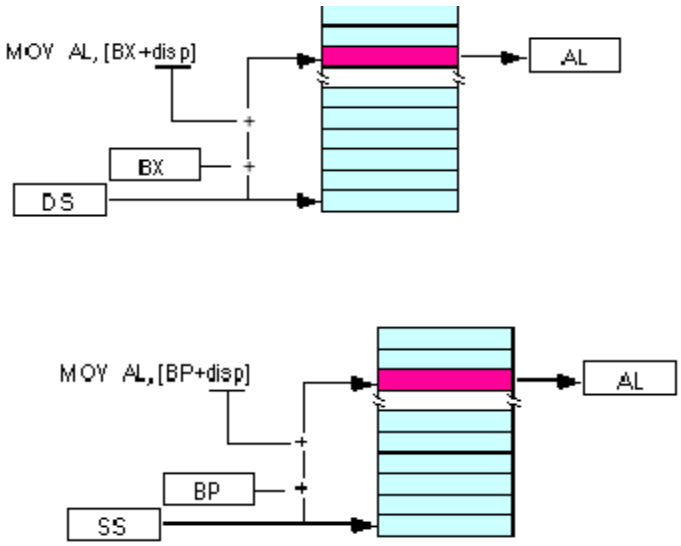
The indexed addressing modes use the following syntax:

```
mov     al, disp[bx]
mov     al, disp[bp]
mov     al, disp[si]
mov     al, disp[di]
```

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from locationss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov     al, ss:disp[bx]
mov     al, es:disp[bp]
mov     al, cs:disp[si]
mov     al, ss:disp[di]
```



You may substitute si or di in the figure above to obtain the [si+disp] and [di+disp] addressingmodes.

2.4 **Based Indexed Addressing Modes**

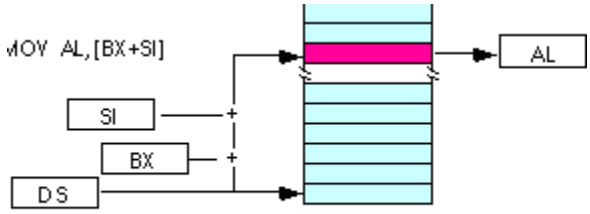
The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction

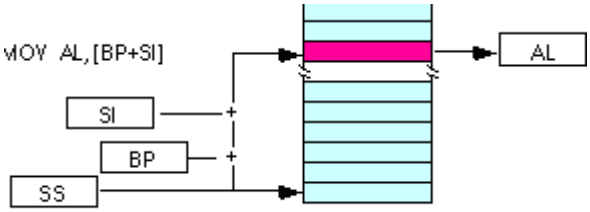
```
mov       al,[bx][si]
```

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.



.

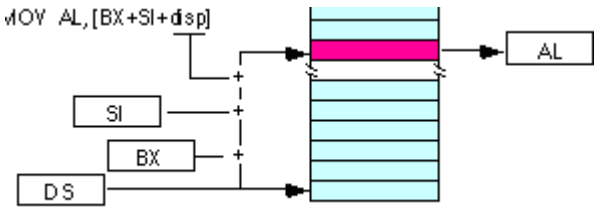
You substitute di in the figure above to obtain the [bx+di] addressing mode.



You substitute di in the figure above for the [bp+di] addressing mode.

2.5 Based Indexed Plus Displacement Addressing Mode

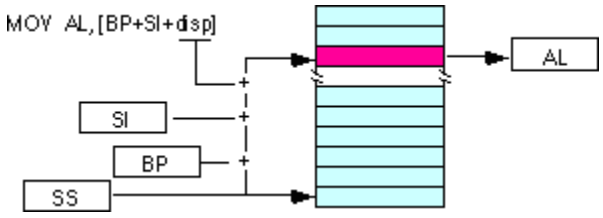
These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes:




```
mov     al, disp[bx][si]
mov     al, disp[bx+di]
```

```
mov     al, [bp+si+disp]
mov     al, [bp][di][disp]
```

You may substitute di in the figure above to produce the [bx+di+disp] addressing mode.



You may substitute di in the figure above to produce the [bp+di+disp] addressing mode.

8086 INSTRUCTION SET

DATA TRANSFER INSTRUCTIONS:

MOV Move byte or word to register or memory

IN, OUT Input byte or word from port, output word to port

LEA Load effective address

LDS, LES Load pointer using data segment, extra segment

PUSH, POP Push word onto stack, pop word off stack

XCHG Exchange byte or word

XLAT Translate byte using look-up table

LOGICAL INSTRUCTIONS:

NOT Logical NOT of byte or word (one's complement)

AND Logical AND of byte or word

OR Logical OR of byte or word

XOR Logical exclusive-OR of byte or word

TEST Test byte or word (AND without storing)

SHIFT AND ROTATE INSTRUCTIONS:

SHL, SHR Logical shift left, right byte or word? by 1 or CL

SAL, SAR Arithmetic shift left, right byte or word? by 1 or CL

ROL, ROR Rotate left, right byte or word? by 1 or CL

RCL, RCR Rotate left, right through carry byte or word? by 1 or CL

ARITHMETIC INSTRUCTIONS:

ADD, SUB Add, subtract byte or word

ADC, SBB Add, subtract byte or word and carry (borrow)

INC, DEC Increment, decrement byte or word

NEG Negate byte or word (two's complement)

CMP Compare byte or word (subtract without storing)

MUL, DIV Multiply, divide byte or word (unsigned)

IMUL, IDIV Integer multiply, divide byte or word (signed)

CBW, CWD Convert byte to word, word to double word (useful before multiply/divide)

1. Adjustments after arithmetic operations:

AAA, AAS, AAM,AAD

ASCII adjust for addition, subtraction, multiplication, division (ASCII codes 30-39)

DAA, DAS Decimal adjust for addition, subtraction (binary coded decimal numbers)

TRANSFER INSTRUCTIONS:

JMP Unconditional jump (*short* ?127/8, *near* ?32K, *far* between segments)

1. Conditional jumps:

JA (JNBE) Jump if above (not below or equal)? +127, -128 range only

JAE (JNB) Jump if above or equal(not below)? +127, -128 range only

JB (JNAE) Jump if below (not above or equal)? +127, -128 range only

JBE (JNA) Jump if below or equal (not above)? +127, -128 range only

JE (JZ) Jump if equal (zero)? +127, -128 range only

JG (JNLE) Jump if greater (not less or equal)? +127, -128 range only

JGE (JNL) Jump if greater or equal (not less)? +127, -128 range only

JL (JNGE) Jump if less (not greater nor equal)? +127, -128 range only

JLE (JNG) Jump if less or equal (not greater)? +127, -128 range only

JC, JNC Jump if carry set, carry not set? +127, -128 range only

JO, JNO Jump if overflow, no overflow? +127, -128 range only

JS, JNS Jump if sign, no sign? +127, -128 range only

JNP (JPO) Jump if no parity (parity odd)? +127, -128 range only

JP (JPE) Jump if parity (parity even)? +127, -128 range only

2. Loop control:

LOOP Loop unconditional, count in CX, short jump to target address

LOOPE (LOOPZ) Loop if equal (zero), count in CX, short jump to target address

LOOPNE (LOOPNZ) Loop if not equal (not zero), count in CX, short jump to target address

JCXZ Jump if CX equals zero (used to skip code in loop)

SUBROUTINE AND INTERRUPT INSTRUCTIONS:

CALL, RET Call, return from procedure (inside or outside current segment)

INT, INTO Software interrupt, interrupt if overflow

IRET Return from interrupt

STRING INSTRUCTIONS:

MOVS Move byte or word string

MOVSB, MOVSW Move byte, word string

CMPS Compare byte or word string

SCAS Scan byte or word string (comparing to A or AX)

LODS, STOS Load, store byte or word string to AL or AX

Repeat instructions *(placed in front of other string operations):*

REP Repeat

REPE, REPZ Repeat while equal, zero

REPNE, REPNZ Repeat while not equal (zero)

PROCESSOR CONTROL INSTRUCTIONS:

1. Flag manipulation:

STC, CLC, CMC Set, clear, complement carry flag

STD, CLD Set, clear direction flag

STI, CLI Set, clear interrupt enable flag

LAHF, SAHF Load AH from flags, store AH into flags

PUSHF, POPF Push flags onto stack, pop flags off stack

2.Coprocessor, multiprocessor interface:

ESC Escape to external processor interface

LOCK Lock bus during next instruction

Inactive states:

NOP No operation

WAIT Wait for TEST pin activity

HLT Halt processor

adc Add with carry flag

Syntax: adc dest, src

dest: memory or register

src: memory, register, or immediate

Action: dest = dest + src + CF

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

Notes: This instruction is used to perform 32-bit addition.

add Add two numbers

Syntax: add dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest + src

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

Notes: Works for both signed and unsigned numbers.

and Bitwise logical AND

Syntax: and dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest & src

Flags Affected: OF=0, SF, ZF, and AF=?, PF, CF=0

call Call procedure or function

Syntax: call addr

addr: register, memory, or immediate

Action: Push IP onto stack, set IP to addr.

Flags Affected: None

cbw Convert byte to word (signed)

Syntax: cbw

Action: Sign extend AL to create a word in AX.

Flags Affected: None

Notes: For unsigned numbers use "mov ah, 0".

cli Clear interrupt flag (disable interrupts)

Syntax: cli

Action: Clear IF

Flags Affected: IF=0

cmp Compare two operands

Syntax: cmp op1, op2

op1: register or memory

op2: register, memory, or immediate

Action: Perform op1-op2, discarding the result but setting the flags.

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

Notes: Usually used before a conditional jump instruction.

cwd Convert word to doubleword (signed)

Syntax: cwd

Action: Sign extend AX to fill DX, creating a dword contained in DX::AX.

Flags Affected: None

Notes: For unsigned numbers use "xor dx, dx" to clear DX.

dec Decrement by 1

Syntax: dec op

op: register or memory

Action: op = op - 1

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

div Unsigned divide

Syntax: div op8

div op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, unsigned AL = AX / op8 and AH = AX % op8

If operand is op16, unsigned AX = DX::AX / op16 and DX = DX::AX % op16

Flags Affected: OF=?, SF=?, ZF=?, AF=?, PF=?, CF=?

Notes: Performs both division and modulus operations in one instruction.

idiv Signed divide

Syntax: idiv op8

idiv op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, signed AL = AX / op8 and AH = AX % op8

If operand is op16, signed AX = DX::AX / op16 and DX = DX::AX % op16

Flags Affected: OF=?, SF=?, ZF=?, AF=?, PF=?, CF=?

Notes: Performs both division and modulus operations in one instruction.

imul Signed multiply

Syntax: imul op8

imul op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, signed AX = AL * op8

If operand is op16, signed DX::AX = AX * op16

Flags Affected: OF, SF=?, ZF=?, AF=?, PF=?, CF

in Input (read) from port

Syntax: in AL, op8

in AX, op8

op8: 8-bit immediate or DX

Action: If destination is AL, read byte from 8-bit port op8.

If destination is AX, read word from 16-bit port op8.

Flags Affected: None

inc Increment by 1

Syntax: inc op

op: register or memory

Action: op = op + 1

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

int Call to interrupt procedure

Syntax: int imm8

imm8: 8-bit unsigned immediate

Action: Push flags, CS, and IP; clear IF and TF (disabling interrupts); load word at address (imm8*4) into IP and word at (imm8*4 + 2) into CS.

Flags Affected: IF=0, TF=0

Notes: This instruction is usually used to call system routines.

iret Interrupt return

Syntax: iret

Action: Pop IP, CS, and flags (in that order).

Flags Affected: All

Notes: This instruction is used at the end of ISRs.

j?? Jump if ?? condition met

Syntax: j?? rel8

rel8: 8-bit signed immediate

Action: If condition ?? met, IP = IP + rel8 (sign extends rel8)
Flags Affected: None
Notes: Use the cmp instruction to compare two operands then j?? to jump conditionally. The ?? of the instruction name represents the jump condition, allowing for following instructions:

ja jump if above, unsigned >
jae jump if above or equal, unsigned >=
jb jump if below, unsigned <
jbe jump if below or equal, unsigned <=
je jump if equal, ==
jne jump if not equal, !=
jg jump if greater than, signed >
jge jump if greater than or equal, signed >=
jl jump if less than, signed <
jle jump if less than or equal, signed <=

All of the ?? suffixes can also be of the form n?? (e.g., jna for jump if not above). See 8086 documentation for many more ?? conditions.An assembler label should be used in place of the rel8 operand. The assembler will then calculate the relative distance to jump.

Note also that rel8 operand greatly limits conditional jump distance (-127 to +128 bytes from IP). Use the jmp instruction in combination with j?? to overcome this barrier.

jmp Unconditional jump

Syntax: jump rel
jump op16
jump seg:off
rel: 8 or 16-bit signed immediate
op16: 16-bit register or memory

seg:off: Immediate 16-bit segment and 16-bit offset

Action: If operand is rel, IP = IP + rel

If operand is op16, IP = op16

If operand is seg:off, CS = seg, IP = off

Flags Affected: None

Notes: An assembler label should be used in place of the rel8 operand. The assembler will then calculate the relative distance to jump.

lea Load effective address offset

Syntax: lea reg16, memref

reg16: 16-bit register

memref: An effective memory address (e.g., [bx+2])

Action: reg16 = address offset of memref

Flags Affected: None

Notes: This instruction is used to easily calculate the address of data in memory. It does not actually access memory.

mov Move data

Syntax: mov dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = src

Flags Affected: None

mul Unsigned multiply

Syntax: mul op8

mul op16

op8: 8-bit register or memory

op16: 16-bit register or memory

Action: If operand is op8, unsigned AX = AL * op8

If operand is op16, unsigned DX::AX = AX * op16

Flags Affected: OF, SF=?, ZF=?, AF=?, PF=?, CF

neg Two's complement negate

Syntax: neg op

op: register or memory

Action: op = 0 - op

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

nop No operation

Syntax: nop

Action: None

Flags Affected: None

not One's complement negate

Syntax: not op

op: register or memory

Action: op = ~op

Flags Affected: None

or Bitwise logical OR

Syntax: or dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest | src

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

out Output (write) to port

Syntax: out op, AL

out op, AX

op: 8-bit immediate or DX

Action: If source is AL, write byte in AL to 8-bit port op.

If source is AX, write word in AX to 16-bit port op.

Flags Affected: None

pop Pop word from stack

Syntax: pop op16

reg16: 16-bit register or memory

Action: Pop word off the stack and place it in op16 (i.e., op16 = [SS:SP]

then SP = SP + 2).

Flags Affected: None

Notes: Pushing and popping of SS and SP are allowed but strongly discouraged.

popf Pop flags from stack

Syntax: popf

Action: Pop word from stack and place it in flags register.

Flags Affected: All

push Push word onto stack

Syntax: push op16

op16: 16-bit register or memory

Action: Push op16 onto the stack (i.e., SP = SP - 2 then [SS:SP] = op16).

Flags Affected: None

Notes: Pushing and popping of SS and SP are allowed but strongly discouraged.

pushf Push flags onto stack

Syntax: pushf

Action: Push flags onto stack as a word.

Flags Affected: None

ret Return from procedure or function

Syntax: ret

Action: Pop word from stack and place it in IP.

Flags Affected: None

sal Bitwise arithmetic left shift (same as shl)

Syntax: sal op, 1

sal op, CL

op: register or memory

Action: If operand is 1, op = op << 1

If operand is CL, op = op << CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

sar Bitwise arithmetic right shift (signed)

Syntax: sar op, 1

sar op, CL

op: register or memory

Action: If operand is 1, signed op = op >> 1 (sign extends op)

If operand is CL, signed op = op >> CL (sign extends op)

Flags Affected: OF, SF, ZF, AF=?, PF, CF

sbb Subtract with borrow

Syntax: sbb dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest - (src + CF)

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

Notes: This instruction is used to perform 32-bit subtraction.

shl Bitwise left shift (same as sal)

Syntax: shl op, 1

shl op, CL

op: register or memory

Action: If operand is 1, op = op << 1

If operand is CL, op = op << CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

shr Bitwise right shift (unsigned)

Syntax: shr op, 1

shr op, CL

op: register or memory

Action: If operand is 1, op = (unsigned)op >> 1

If operand is CL, op = (unsigned)op >> CL

Flags Affected: OF, SF, ZF, AF=?, PF, CF

sti Set interrupt flag (enable interrupts)

Syntax: sti

Action: Set IF

Flags Affected: IF=1

sub Subtract two numbers

Syntax: sub dest, src

dest: regsiter or memory

src: register, memory, or immediate

Action: dest = dest - src

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

Notes: Works for both signed and unsigned numbers.

test Bitwise logical compare

Syntax: test op1, op2

op1: register, memory, or immediate

op2: register, memory, or immediate

Action: Perform op1 & op2, discarding the result but setting the flags.

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

Notes: This instruction is used to test if bits of a value are set.

xor Bitwise logical XOR

Syntax: xor dest, src

dest: register or memory

src: register, memory, or immediate

Action: dest = dest ^ src

Flags Affected: OF=0, SF, ZF, AF=?, PF, CF=0

SIMPLE PROGRAMS

1. Two 16 bitUnsigned addition

```
data segment
num1 dw 3333h
num2 dw 2222h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ax,num1
mov bx,num2
div bx
mov result,ax
hlt
code ends
end start
```

2. Two 16 bitUnsigned subtraction

```
data segment
num1 dw 3333h
num2 dw 2222h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
```

```
mov ds,ax
mov ax,num1
mov bx,num2
sub ax,bx
mov result,ax
hlt
code ends
end start
```

3. Two 16 bitUnsigned multiplication

```
data segment
num1 dw 1234h
num2 dw 4321h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ax,num1
mov bx,num2
mul ax,bx
mov result,ax
hlt
code ends
end start
```

4. 32 / 16 bit unsigned division

```
data segment
num1 dw 0001h
num2 dw 0001h
```

```
num3 dw 0002h
result dw 2 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ax,num1
mov dx,num2
mov bx,num3
div bx
mov result,ax
mov result+2,dx
hlt
code ends
end start
```

8086 SIGNED OPERATIONS

1. Two 8 bit division

```
data segment
num1 db -06h
num2 db 03h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ah,00h
mov al,num1
mov bl,num2
idiv bl
mov result,ax
hlt
code ends
end start
```

2. Two 8 bit multiplication

```
data segment
num1 db -06h
num2 db 08h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov ah,00h
mov al,num1
mov bl,num2
imul bl
mov result,ax
hlt
code ends
end start
```

8086 ASCII OPERATIONS

1. Two 8 bit ASCII multiplication

```
data segment
n1 db 33h
n2 db 32h
result dw 1 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
        mov ax,data
        mov ds,ax
        mov al,n1
        mov bl,n2
        mov ah,00h
        and al,0fh
        and bl,0fh
        mul bl
        aam
        add ax,3030h
        mov result,ax
        hlt
        code ends
        end start
```

2. 16 / 8 bit ASCII division

```
data segment
n1 dw 3435h
n2 db 39h
result dw 1 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
      MOV ax,data
      MOV ds,ax
      mov ax,n1
      mov bx,n2
      and ax,0f0fh
      and bl,0fh
      aad
      div bl
      add ax,3030h
```

```
        mov result,ax
        hlt
        code ends
        end start
```

3. Two 8 bit ASCII addition.

```
data segment
n1 db 33h
n2 db 32h
result dw 1 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
        mov ax,data
        mov ds,ax
        mov ah,00h
        mov al,n1
        mov bl,n2
        add al,bl
        aaa
        add ax,3030h
        mov result,ax
        hlt
        code ends
        end start
```

4. Two 8 bit ASCII subtraction.

```
data segment
n1 db 33h
n2 db 32h
result dw 1 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
        mov ax,data
        mov ds,ax
        mov ah,00h
        mov al,n1
        mov bl,n2
        sub al,bl
        aas
        add ax,3030h
```

```
mov result,ax
hlt
code ends
end start
```

1. TO FIND OUT A MAXIMUM NUMBER GIVEN AN ARRY LIST

```
data segment
list db 06h,09h,39h,04h,20h
n1 db 04h
result db 1 dup(0)
data ends
code segment
assume cs:code,ds:data
start:
        mov ax,data
        mov ds,ax
        lea si,list
        mov cl,n1
        mov al,00h
        mov al,[si]
        inc si
back:
        cmp al,[si]
        jnc next
        mov al,[si]
next:
        inc si
        dec cl
        jnz back
        mov result,al
        hlt
        code ends
        end start
```

2. TO FIND OUT A MINIMUM NUMBER GIVEN AN ARRY LIST

```
data segment
list db 06h,09h,39h,04h,20h
n1 db 04h
result db 1 dup(0)
data ends
code segment
assume cs:code,ds:data
start:
        mov ax,data
        mov ds,ax
        lea si,list
        mov cl,n1
        mov al,00h
```

```
        mov al,[si]
        inc si
back:
        cmp al,[si]
        jc next
        mov al,[si]
 next:
        inc si
        dec cl
        jnz back
        mov result,al
        hlt
       code ends
       end start
```

3. TO ARRANGE GIVEN NUMBERS IN AN ASCENDING ORDER

```
data segment
list db 03h,05h,02h
data ends
code segment
assume ds:data,cs:code
start:
       mov ax,data
       mov ds,ax
       mov cl,03h
       dec cl
loop2:
        mov si,offset list
        mov dl,cl
loop1:
        mov al,[si]
        cmp al,[si+1]
        jl next
        xchg al,[si+1]
        xchg al,[si]
next:
        inc si
        dec dl
        jnz loop1
        dec cl
        jnz loop2
        hlt
        code ends
        end start
```

4. TO ARRANGE GIVEN NUMBERS IN DECENDING ORDER

```
data segment
list db 03h,05h,02h
data ends
code segment
assume ds:data,cs:code
start:
      mov ax,data
      mov ds,ax
      mov cl,03h
      dec cl
loop2:mov si,offset list
       mov dl,cl
loop1:mov al,[si]
       cmp al,[si+1]
       jg next
       xchg al,[si+1]
       xchg al,[si]
next:  inc si
       dec dl
       jnz loop1
       dec cl
       jnz loop2
       hlt
       code ends
       end start
```

ASSEMBLER DIRECTIVES

The assembly language program consists of some statements which are not the commands to the processor. They are not translated to machine code. They only guide the assembler, linker and loader. They are pseudo operations and called assembler directives.

SOME ASSEMBLER DIRECTIVES:

1. ASSUME: Tells the assembler what segments to use.

    Syntax: Assume segment register: segment name, segment register: segment name

    Ex: Assume Cs: Code, Ds:Data

2. SEGMENT: Defines the segment name and specifies that the code that follows is in that segment.

    Syntax: Segment name SEGMENT

3. ENDS: End of segment

4. ORG: Originate or Origin: sets the location counter.

    Syntax: a. ORG numeric value

        b. ORG $+ numeric value

5. END: End of source code. This directive is used to inform the assembler the end of a program.

    Syntax: a. END

        b. END label

6. NAME: Give source module a name.

7. EQU: Equate or equivalence

    Syntax: a. variable name EQU expression

        b. string name EQU ‘string’

8. LABEL: Assign current location count to a symbol.

Syntax : label name LABEL label type

9. $: Current location count

10. ALIGN: aligns the next segment at even address

11. STACK

Stack 100d (or) stack 64h; default 1024 bytes

12. DATA

- DB – Define Byte- Variable Name DB value1, value2…..
- DW – Define Word - Variable Name DW value1, value2…..
- DD – Define Double word - Variable Name DW value1, value2…..
- DQ – Define Quad word - Variable Name DQ value1, value2…..
- DT – Define Ten bytes - Variable Name DT value1, value2…..
- DUP – used to declare an array of bytes

  Syntax: Variable name data type Num DUP (Value)

13. CODE – indicates the beginning of code segment

Ex: code [name].

14. EXIT: marks the end of CS, most of the time we use 4CH function of DOS interrupt 21H.

15. OFFSET: this directive is an operation and informs the assembler to determine the displacement of the specified variable from the start of the segment.

Syntax: OFFSET variable name

16. PTR: Pointer – This directive is used to specify the type of memory access.

Syntax: Data Type PTR

17. PROC, ENDP & MACRO, ENDM directives support modularity.

A. MACRO – It indicates the start of a macro. Arguments are dummy variables and are optional.

B. ENDM – End of macro – it indicates the assembler the end of macro.

C. PROC – Procedure

Syntax: Procedure name PROC NEAR/FAR

D. ENDP – End of procedure – this directive informs the end of procedure whose name is specified before ENDP.

Syntax: Procedure name ENDP

PROCEDURES AND MACROS

1. PROCEDURES

A procedure is a collection of instructions to which we can direct the flow of our program, and once the execution of these instructions is over control is given back to the next line to process of the code which called on the procedure.

Procedures help us to create legible and easy to modify programs.

At the time of invoking a procedure the address of the next instruction of the program is kept on the stack so that, once the flow of the program has been transferred and the procedure is done, one can return to the next line of the original program, the one which called the procedure.

SYNTAX OF A PROCEDURE

There are two types of procedures, the intrasegments, which are found on the same segment of instructions, and the inter-segments which can be stored on different memory segments.

When the intrasegment procedures are used, the value of IP is stored on the stack and when the intrasegments are used the value of CS:IP is stored.

To divert the flow of a procedure (calling it), the following directive is used:

CALL Name Of The Procedure

The part which make a procedure are:

Declaration of the procedure
Code of the procedure
Return directive
Termination of the procedure

For example, if we want a routine which adds two bytes stored in AH and AL each one, and keep the addition in the BX registers:

```
Adding Proc Near ; Declaration of the procedure
Mov Bx, 0 ; Content of the procedure
Mov B1, Ah
Mov Ah, 00
Add Bx, Ax
Ret ; Return directive
Add Endp ; End of procedure declaration
```

On the declaration the first word, Adding, corresponds to the name of out procedure, Proc declares it as such and the word Near indicates to the MASM that the procedure is intra segment.

The Ret directive loads the IP address stored on the stack to return to the original program, lastly, the Add Endp directive indicates the end of the procedure.

To declare an inter segment procedure we substitute the word Near for the word FAR.

The calling of this procedure is done the following way:

Call Adding Macros offer a greater flexibility in programming compared to the procedures, nonetheless, these last ones will still be used.

2. MACROS

Definition of the macro

A macro is a group of repetitive instructions in a program which are codified only once and can be used as many times as necessary.

The main difference between a macro and a procedure is that in the macro the passage of parameters is possible and in the procedure it is not, this is only applicable for the TASM - there are other programming languages which do allow it. At the moment the macro is executed each parameter is substituted by the name or value specified at the time of the call.

We can say then that a procedure is an extension of a determined program, while the macro is a module with specific functions which can be used by different programs.

Another difference between a macro and a procedure is the way of calling each one, to call a procedure the use of a directive is required, on the other hand the call of macros is done as if it were an assembler instruction.

Syntax of a Macro

The parts which make a macro are:

Declaration of the macro
Code of the macro
Macro termination directive

The declaration of the macro is done the following way:

Name Macro MACRO [parameter1, parameter2...]

Even though we have the functionality of the parameters it is possible to create a macro which does not need them.

The directive for the termination of the macro is: ENDM

An example of a macro, to place the cursor on a determined position on the screen is:

```
Position MACRO Row, Column
PUSH AX
PUSH BX
PUSH DX
MOV AH, 02H
MOV DH, Row
MOV DL, Column
MOV BH, 0
INT 10H
POP DX
POP BX
POP AX
ENDM
```

To use a macro it is only necessary to call it by its name, as if it were another assembler instruction, since directives are no longer necessary as in the case of the procedures. Example:

Macro Libraries

One of the facilities that the use of macros offers is the creation of libraries, which are groups of macros which can be included in a program from a different file.

The creation of these libraries is very simple, we only have to write a file with all the macros which will be needed and save it as a text file.

To call these macros it is only necessary to use the following instruction Include NameOfTheFile, on the part of our program where we would normally write the macros, this is, at the beginning of our program, before the declaration of the memory model.

The macros file was saved with the name of MACROS.TXT, the instruction Include would be used the following way:

```
;Beginning of the program
Include MACROS.TXT
.MODEL SMALL
.DATA
;The data goes here
.CODE
Beginning:
;The code of the program is inserted here
.STACK
;The stack is defined
End beginning
;Our program ends
```

PROGRAMS ON STRING MANIPULATIONS

a) Block Transfer:

```
data segment
num db 32h,30h,29h,25h
num2 db 04 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov es,ax
mov si,offset num
mov di,offset num2
mov cl,04h
rep movsb
hlt
code ends
end start
```

b) Block Transfer:

```
data segment
num db 32h,30h,29h,25h
num2 db 04 dup(0)
data ends
code segment
assume ds:data,cs:code
start:mov ax,data
mov ds,ax
mov es,ax
mov si,offset num
mov di,offset num2
mov bx,0004h
step:
dec bx
lodsb
mov [bx][di],al
jnz step
hlt
code ends
end start
```

c) Insertion of a string:

```
data segment
num1 db 32h,30h,29h,25h
num2 db 26h,31h
place dw 0001h
length1 db 04h
length2 db 02h
result db 07 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
mov ax,data
mov ds,ax
mov es,ax
mov si,offset num1
mov bx,offset num2
mov di,offset result
mov cl,length1
mov ch,length2
back:
cmp si,place
je li
here:
lodsb
stosb
dec cl
jnz back
hlt
li:
mov al,[bx]
mov [di],al
inc di
inc bx
dec ch
jnz li
jmp here
code ends
end start
```

d) Deletion of a string:

```
data segment
giv db 32h,30h,29h,25h
```

```
place dw 0000h
lgiv dw 0004h
ldel dw 0002h
result db 06 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
mov ax,data
mov ds,ax
mov es,ax
mov si,offset giv
mov di,offset result
mov cx,0000h
back:
cmp si,place
je li
lodsb
stosb
inc cx
here:
cmp cx,lgiv
jb back
hlt
li:
add cx,ldel
add si,ldel
jmp here
code ends
end start
```

PROGRAMS ON CONVERSIONS

a) BCD to Hexa conversion

```
data segment
num1 db 32h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
mov ax,data
mov ds,ax
mov al,num1
mov bl,num1
mov cl,04h
and al,0f0h
and bl,0fh
shr al,cl
mov ch,0ah
mul ch
add al,bl
mov result,al
hlt
code ends
end start
```

b) Hexa to Ascii coded BCD

```
data segment
num1 db 22h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
mov ax,data
mov ds,ax
mov al,num1
mov ah,00h
mov bl,0ah
div bl
xchg al,ah
add ax,3030h
mov result,ax
hlt
```

```
code ends
end start
```

c) Packed to unpacked BCD

```
data segment
num1 db 32h
result dw 01 dup(0)
data ends
code segment
assume ds:data,cs:code
start:
mov ax,data
mov ds,ax
mov al,num1
mov bl,num1
and al,0fh
and bl,0f0h
mov cl,04h
shr bl,cl
mov ah,bl
add ax,3030h
mov result,ax
hlt
code ends
end start
```

ASSEMBLY LANGUAGE PROGRAMS

A. About ALP:

Assembly languages are a family of low-level languages for programming computers, microprocessors, microcontrollers, and other (usually) integrated circuits. They implement a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called mnemonics) that help the programmer remember individual instructions, registers, etc. An assembly language is thus specific to certain physical or virtual computer architecture (as opposed to most high-level languages, which are usually portable).

A utility program called an assembler is used to translate assembly language statements into the target computer's machine code. The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data. This is in contrast with high-level languages, in which a single statement generally results in many machine instructions.

Many sophisticated assemblers offer additional mechanisms to facilitate program development, control the assembly process, and aid debugging. In particular, most modern assemblers include a macro facility (described below), and are called macro assemblers.

Assemblers are generally simpler to write than compilers for high-level languages, and have been available since the 1950s. Modern assemblers, especially for RISC based architectures, such as MIPS, Sun SPARC, HP PA-RISC and x86(-64), optimize instruction scheduling to exploit the CPU pipeline efficiently.

There are two types of assemblers based on how many passes through the source are needed to produce the executable program. One pass assemblers go through the source code once and assumes that all symbols will be defined before any instruction that

references them. Two pass assemblers (and multi-pass assemblers) create a table with all unresolved symbols in the first pass, then use the 2nd pass to resolve these addresses. The advantage in one pass assemblers is speed - which is not as important as it once was with advances in computer speed and capabilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in the program source. As a result, the program can be defined in a more logical and meaningful way. This makes two-pass assembler programs easier to read and maintain.

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing
- Object-Oriented features such as encapsulation, polymorphism, inheritance, interfaces.

B. BASIC ELEMENTS

Any Assembly language consists of 3 types of instruction statements which are used to define the program operations:

1. Opcode mnemonics
2. Data sections
3. Assembly directives

1. Opcode mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level languages. Generally, an opcode is a symbolic name for a single executable machine language instruction, and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or

opcode plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be either immediate (typically one byte values, coded in the instruction itself) or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works.

2. Data sections

There are instructions used to define data elements to hold data and variables. They define what type of data, length and alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined.

3. Assembly directives / pseudo-ops

Assembly Directives are instructions that are executed by the Assembler at assembly time, not by the CPU at run time. They can make the assembly of the program dependent on parameters input by the programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of the program to make it easier for the programmer to read and maintain. The names of pseudo-ops often start with a dot to distinguish them from machine instructions.

Some assemblers also support *pseudo-instructions*, which generate two or more machine instructions. Most assemblers provide flexible symbol management, allowing programmers to manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with replaceable addresses.

Assembly languages, like most other computer languages, allow comments to be added to assembly source code that are ignored by the assembler. Good use of comments is even

more important with assembly code than with higher-level languages, as the meaning and purpose of a sequence of instructions is harder to decipher from the code itself.

Wise use of these facilities can greatly simplify the problems of coding and maintaining low-level code. *Raw* assembly source code as generated by compilers or dis assemblers — code without any comments, meaningful symbols, or data definitions — is quite difficult to read when changes must be made.

PROGRAM CONTROL INSTRUCTIONS

Program control instructions change or modify the flow of a program. The most basic kind of program control is the **unconditional branch** or **unconditional jump**. **Branch** is usually an indication of a short change relative to the current program counter. **Jump** is usually an indication of a change in program counter that is not directly related to the current program counter (such as a jump to an absolute memory location or a jump using a dynamic or static table), and is often free of distance limits from the current program counter.

The penultimate kind of program control is the **conditional branch** or **conditional jump**. This gives computers their ability to make decisions and implement both loops and algorithms beyond simple formulas.

Most computers have some kind of instructions for **subroutine call** and **return** from subroutines.

There are often instructions for **saving** and **restoring** part or all of the processor state before and after subroutine calls. Some kinds of subroutine or return instructions will include some kinds of save and restore of the processor state.

Even if there are no explicit hardware instructions for subroutine calls and returns, subroutines can be implemented using jumps (saving the return address in a register or memory location for the return jump). Even if there is no hardware support for saving the processor state as a group, most (if not all) of the processor state can be saved and restored one item at a time.

NOP, or no operation, takes up the space of the smallest possible instruction and causes no change in the processor state other than an advancement of the program counter and any time related changes. It can be used to synchronize timing (at least crudely). It is often used during development cycles to temporarily or permanently wipe out a series of instructions without having to reassemble the surrounding code.

Stop or **halt** instructions bring the processor to an orderly halt, remaining in an idle state until restarted by interrupt, trace, reset, or external action.

Reset instructions reset the processor. This may include any or all of: setting registers to an initial value, setting the program counter to a standard starting location (restarting the computer), clearing or setting interrupts, and sending a reset signal to external devices.

- **JMP** Jump; Intel 80x86; unconditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed])
- **JMP** Jump; MIX; unconditional jump to location M; J-register loaded with the address of the instruction which would have been next if the jump had not been taken
- **Jcc** Jump Conditionally; Intel 80x86; conditional jump (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register, memory location, or indexed]) based on a tested condition: JA/JNBE, JAE/JNB, JB/JNAE, JBE/JNA, JC, JE/JZ, JNC, JNE/JNZ, JNP/JPO, JP/JPE, JG/JNLE, JGE/JNL, JL/JNGE, JLE/JNG, JNO, JNS, JO, JS
- **Jcc** Jump on Condition; MIX; conditional jump to location M based on comparison indicator; if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken; JL (less), JE (equal), JG (greater), JGE (greater-or-equal), JNE (unequal), JLE (less-or-equal)
- **LOOP** Loop While ECX Not Zero; Intel 80x86; used to implement DO loops, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero then the program continues to the next instruction (exiting the loop), otherwise the program makes a byte branch to continue the loop; does not modify flags
- **LOOPE** Loop While Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit

the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPZ; does not modify flags

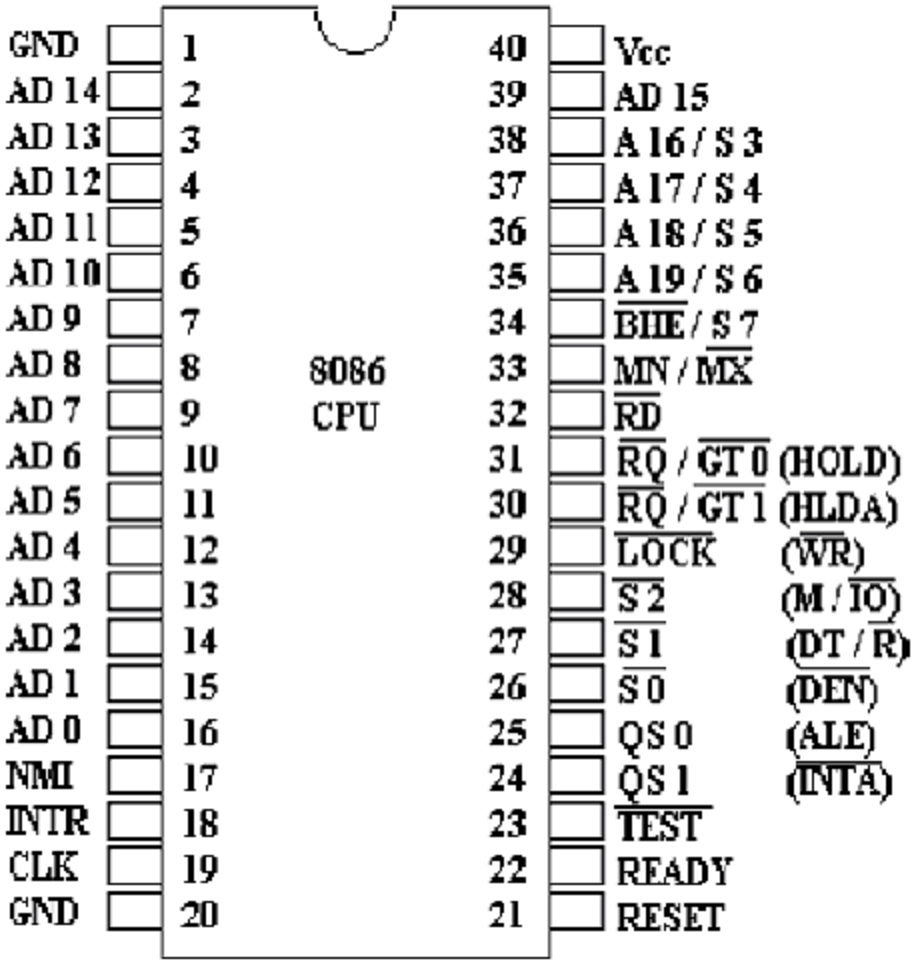
- **LOOPNE** Loop While Not Equal; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNZ; does not modify flags
- **LOOPNZ** Loop While Not Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is set (one) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPNE; does not modify flags
- **LOOPZ** Loop While Zero; Intel 80x86; used to implement DO loops, WHILE loops, UNTIL loops, and similar constructs, decrements the ECX or CX (count) register and then tests to see if it is zero, if the ECX or CX register is zero or the Zero Flag is clear (zero) then the program continues to the next instruction (to exit the loop), otherwise the program makes a byte branch (to continue the loop); equivalent to LOOPE; does not modify flags
- **JCXZ** Jump if Count Register Zero; Intel 80x86; conditional jump if CX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **JECXZ** Jump if Extended Count Register Zero; Intel 80x86; conditional jump if ECX (count register) is zero; used to prevent entering loop if the count register starts at zero; does not modify flags
- **CALL** Call Procedure; Intel 80x86; pushes the address of the next instruction following the subroutine call onto the system stack, decrements the system stack pointer, and changes program flow to the address specified (near [relative displacement from PC] or far; direct or indirect [based on contents of general purpose register or memory location])

- **RET** Return From Procedure; Intel 80x86; fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **IRET** Return From Interrupt; Intel 80x86; transfers the value at the top of the system stack into the flags register, increments the system stack pointer, fetches the return address from the top of the system stack, increments the system stack pointer, and changes program flow to the return address; optional immediate operand added to the new top-of-stack pointer, effectively removing any arguments that the calling program pushed on the stack before the execution of the corresponding CALL instruction; possible change to lesser privilege
- **PUSHA** Push All Registers; Intel 80x86; move contents all 16-bit general purpose registers to memory pointed to by stack pointer (in the order AX, CX, DX, BX, original SP, BP, SI, and DI); does not affect flags
- **POPA** Pop All Registers; Intel 80x86; move memory pointed to by stack pointer to all 16-bit general purpose registers (except for SP); does not affect flags
- **NOP** No Operation; no change in processor state other than an advance of the program counter
- **HLT** Halt; stop machine, computer restarts on next instruction

8086 PIN CONFIGURATION

The pins and signals of 8086 can be classified into six groups, they are as follows.

- Address/status bus
- Address/data bus
- Control and status signals
- Interrupts and external initiated signals
- Power supply and clock frequency signals



The 8086 works in two modes, minimum and maximum. Accordingly pin configuration changes. They are 32 signals common to both modes. Remaining 8 pins are different for each mode. Let us describe each signal/pin.

Vcc (pin 40): Power

Gnd (pin 1 and 20): Ground

AD0...AD7, A8...A15, A19/S6, A18/S5, A17/S4, and A16/S3: 20 -bit Address Bus

MN/MX' (input): Indicates Operating mode

READY (input, Active High): take µP to wait state

CLK (input): Provides basic timing for the processor

RESET (input, Active High): At least 4 clock cycles Causes the µP immediately terminate its present activity.

TEST' (input, Active Low): Connect this to HIGH

HOLD (input, Active High): Connect this to LOW (BR)

HLDA (output, Active High): Hold Ack (BG)

INTR (input, Active High): Interrupt request

INTA' (output, Active Low): Interrupt Acknowledge

NMI (input, Active High): Non-mask able interrupt

DEN' (output): Data Enable. It is LOW when processor wants to receive data or processor is giving out data (to74245)

DT/R' (output): Data Transmit/Receive. When high, data from µP to memory When Low, data is from memory to µP (to74245 dir)

IO/M' (output): If High µP access I/O Device. If Low µP access memory

RD' (output): When Low, µP is performing a read operation

WR' (output): When Low, µP is performing a write operation

ALE (output): Address Latch Enable, Active High Provided by µP to latch address When HIGH, µP is using AD0...AD7, A19/S6, A18/S5, A17/S4, A16/S3 as address lines

S4	S3	Function
0	0	Extra segment access
0	1	Stack segment access
1	0	Code segment access
1	1	Data segment access

/S2, /S1, /S0 status bits (output): pin 26, 27 and 28

These signals indicate the status of current bus cycle i.e. type of machine cycle. The following table gives the type of machine cycles.

S2	S1	S0	Characteristics
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 State

QS0, QS1 queue status (output) pin 24 and 25:

These signals indicate the status of instruction queue during the previous clock cycle. These are accessed by the co-processor 8087. Following table gives the queue status.

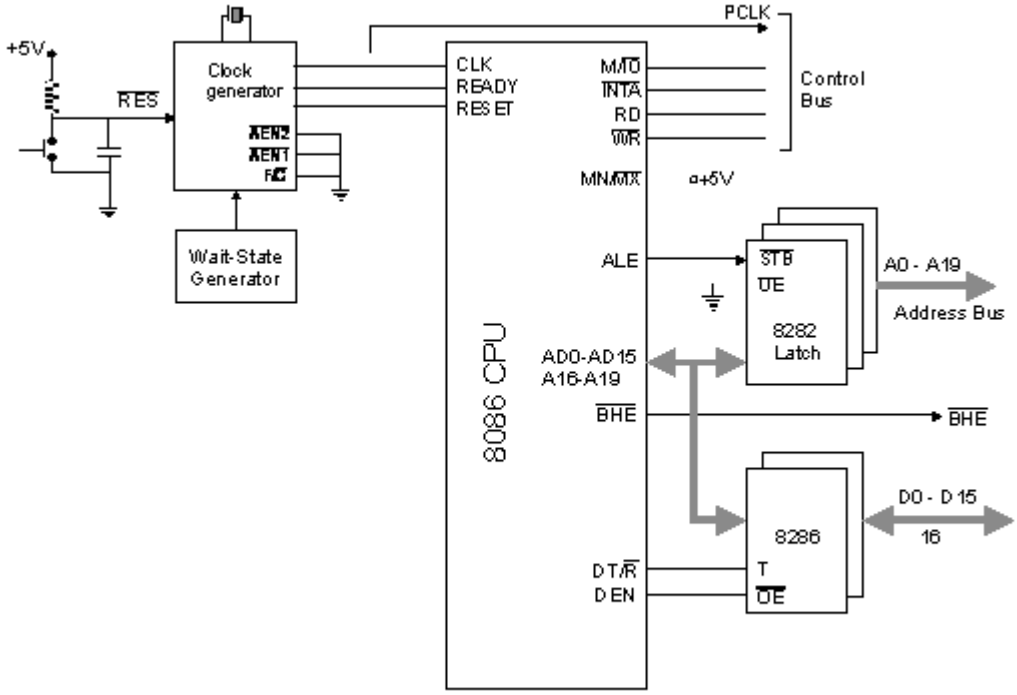
QS1	QS0	Characteristics
0	0	No operation
0	1	First byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

/LOCK: This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the /LOCK signal is low. The /LOCK signal is activated by the lock prefix instruction and remains active until the completion of the next instruction.

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, Tran receivers, 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 some times 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.

- Usually, EPROM is used for monitor storage, while RAM for user's program storage. A system may contain I/O devices.

- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.

- 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 T_1 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 A_0 signals address low, high or both bytes. From T_1 to T_4, the M/IO signal indicates a memory or I/O operation.

- At T_2, 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 T_2.

- 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.

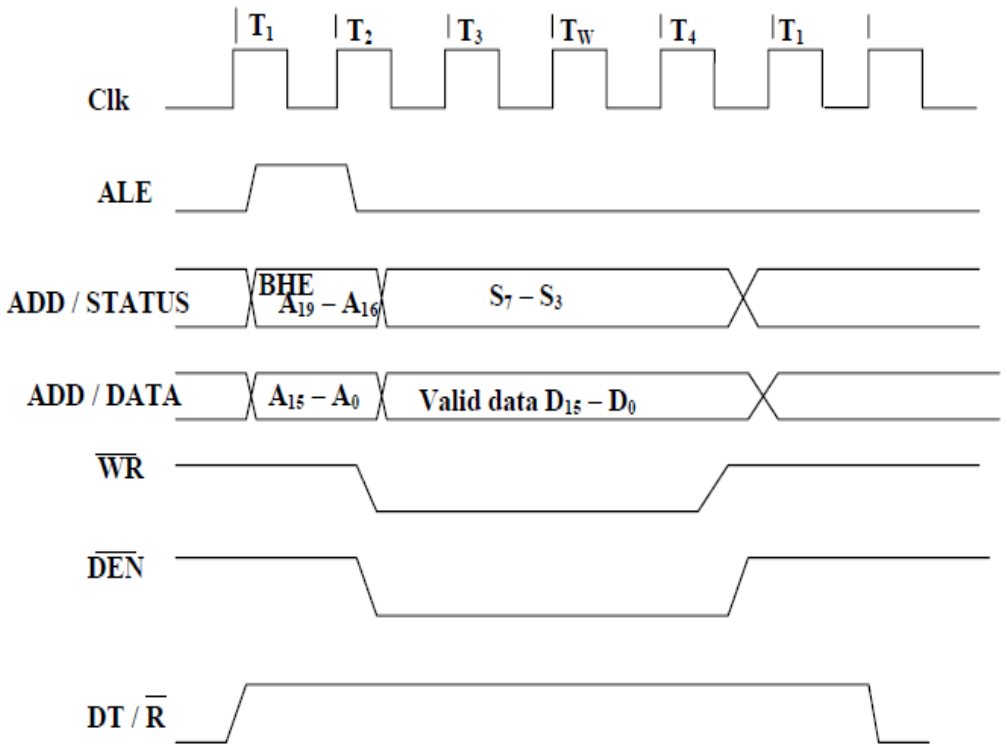
- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tri state its bus drivers.

- 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 T_2, after sending the address in T_1, the processor sends the data to be written to the addressed location.

- The data remains on the bus until middle of T_4 state. The WR becomes active at the beginning of T_2 (unlike RD is somewhat delayed in T_2 to provide time for floating).

- The BHE and A_0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.

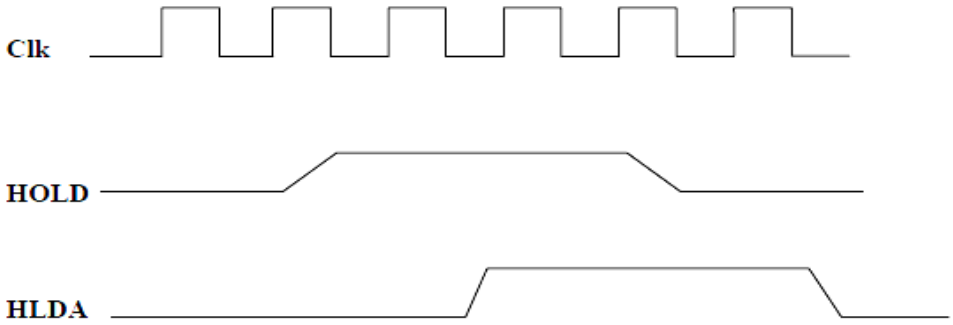
- The M/IO, RD and WR signals indicate the type of data transfer as specified in table below.



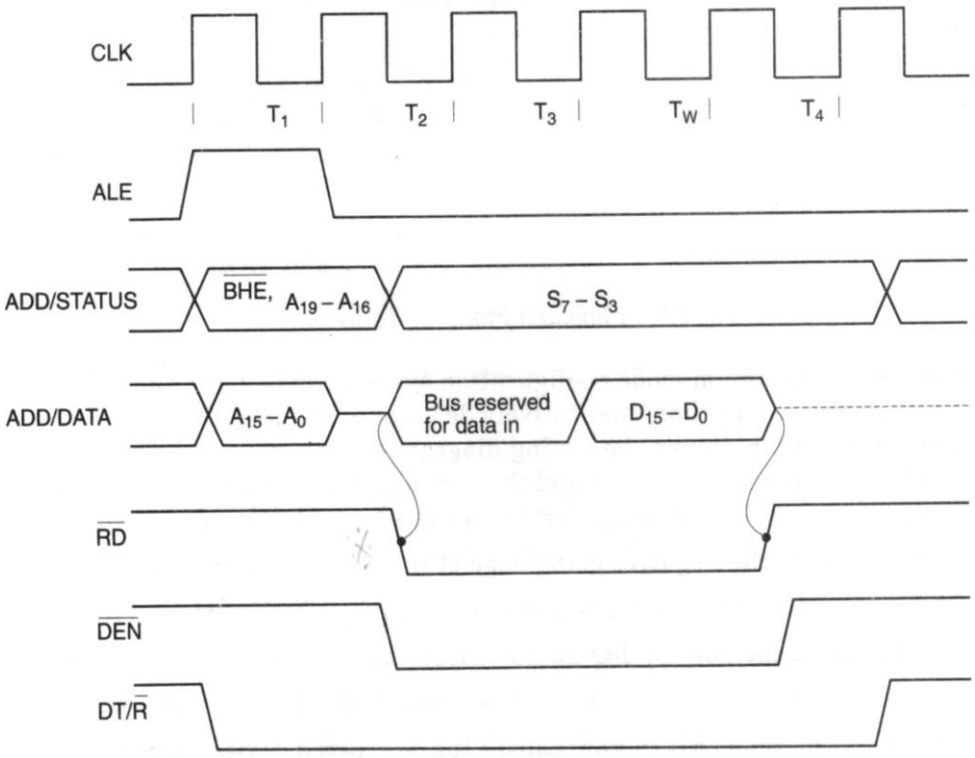
Write Cycle Timing Diagram for Minimum Mode

Hold Response sequence: The HOLD pin is checked at leading edge of each clock pulse. If it is received active by the processor before T_4 of the previous cycle or during T_1 state of the current cycle, the CPU activates HLDA in the next clock cycle and for succeeding bus cycles, the bus will be given to another requesting master.

The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock.



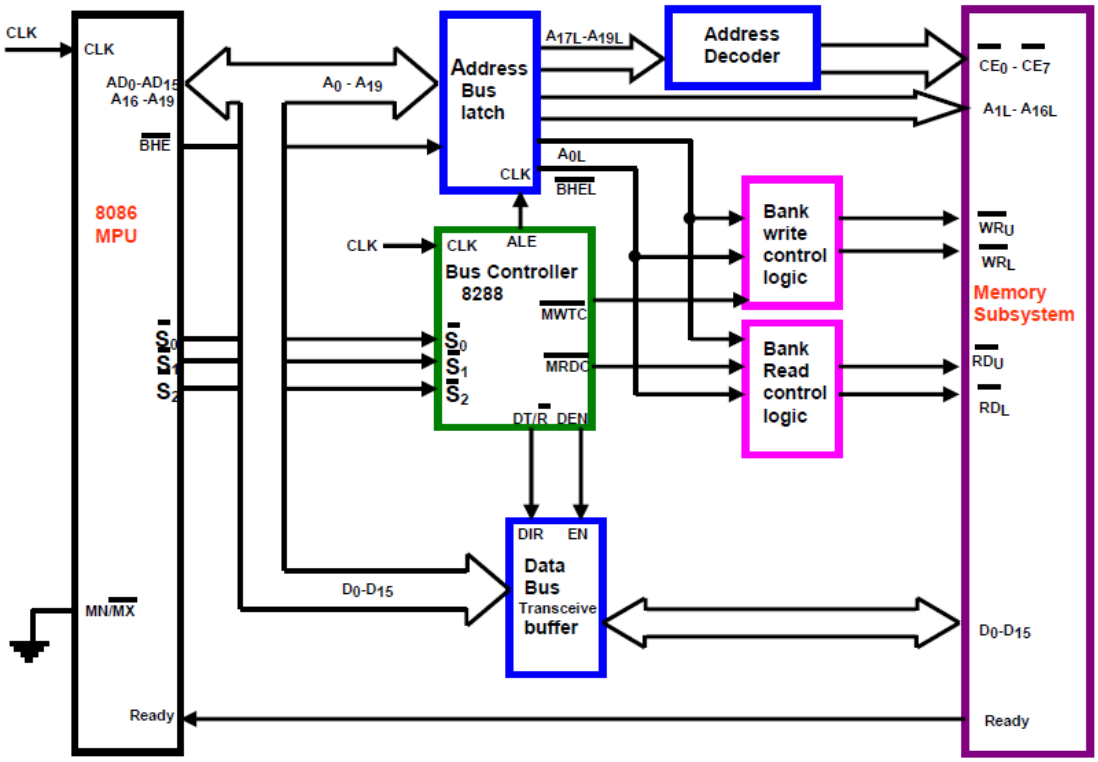
Bus Request and Bus Grant Timings in Minimum Mode System



Read Cycle timing Diagram for Minimum Mode

8086 MAXIMUM MODE

Maximum mode is one of the two hardware modes available to the Intel 8086 and 8088 processors (CPU).Maximum mode is for large applications such as multiprocessing. The mode is hard-wired into the circuit and cannot be changed by software.



The Bus controller is introduced here due to the support of multiprocessor environment of Maximum mode. The decoder is used to select desired memory chips. The remaining components of this circuit are similar to 8088 minimum mode circuit, as shown in figure. Note that the bank high enable signal is used to control the access of even or odd memory banks of 8086 system.

The status codes (S0, S1, S2) of the CPU is used by the bus controller to activate maximum mode memory control signals: These codes are important for multiprocessor environment, supported by Maximum mode.

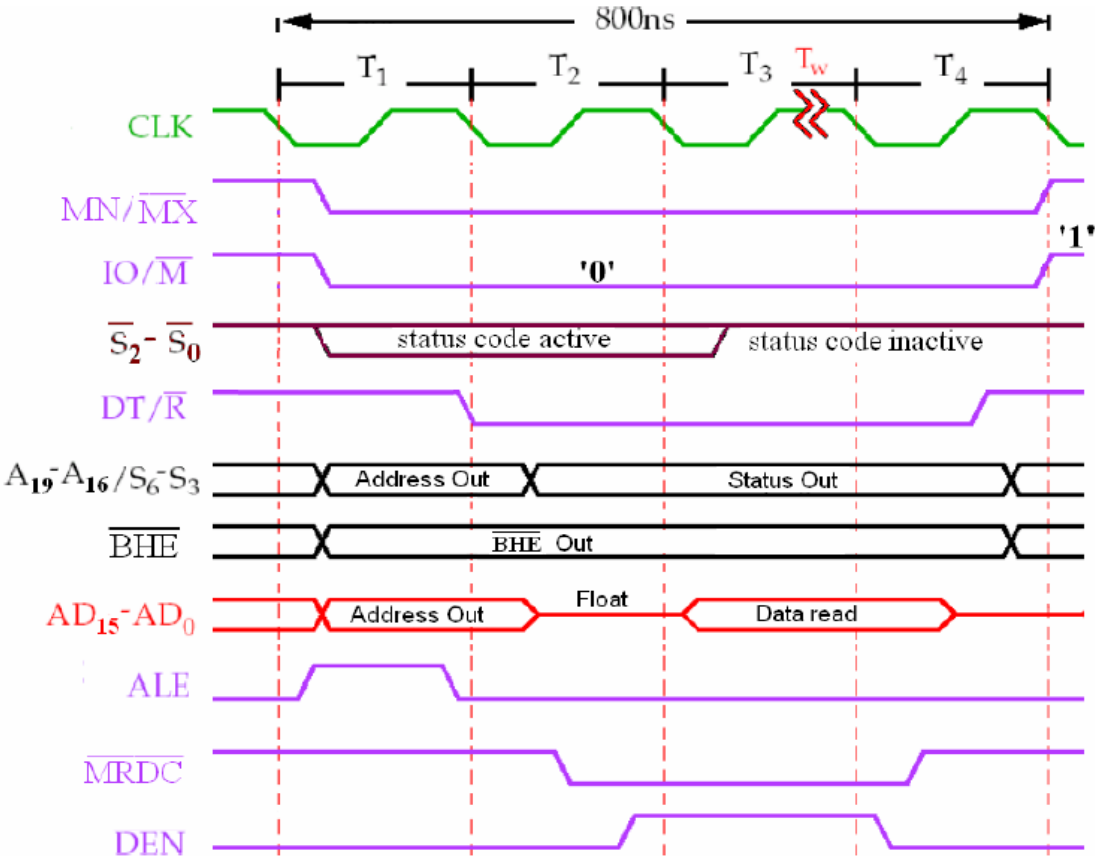
Status Inputs			CPU Cycle	8288 Command
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

Maximum-mode Memory-Read bus-cycle of 8086 system

To complete the minimum-mode memory-read bus-cycle, the required control signals with appropriate active logic levels are:

- IO/M = ‘logic 0’, to select memory interface
- MN/MX = ‘logic 0’, to select maximum-mode of operation
- DT/R = ‘logic 0’, to activate the data-receive mode of ‘Data-bus-buffer’
- Valid Physical-address (A0 to A19) and BHE signal is generated by CPU
- ALE-pulse, to latch the valid Physical-address. ( )
- Proper status code S0 to S2 (as shown in table of slide 8) is generated by
- CPU to initiate data reading (MRDC) from the desired memory bank
- DEN = ‘1’, enables the ‘Data-Bus-transceiver-buffer’ to let data pass
- Reset MRDC and DEN signals to END the read-bus-cycle.

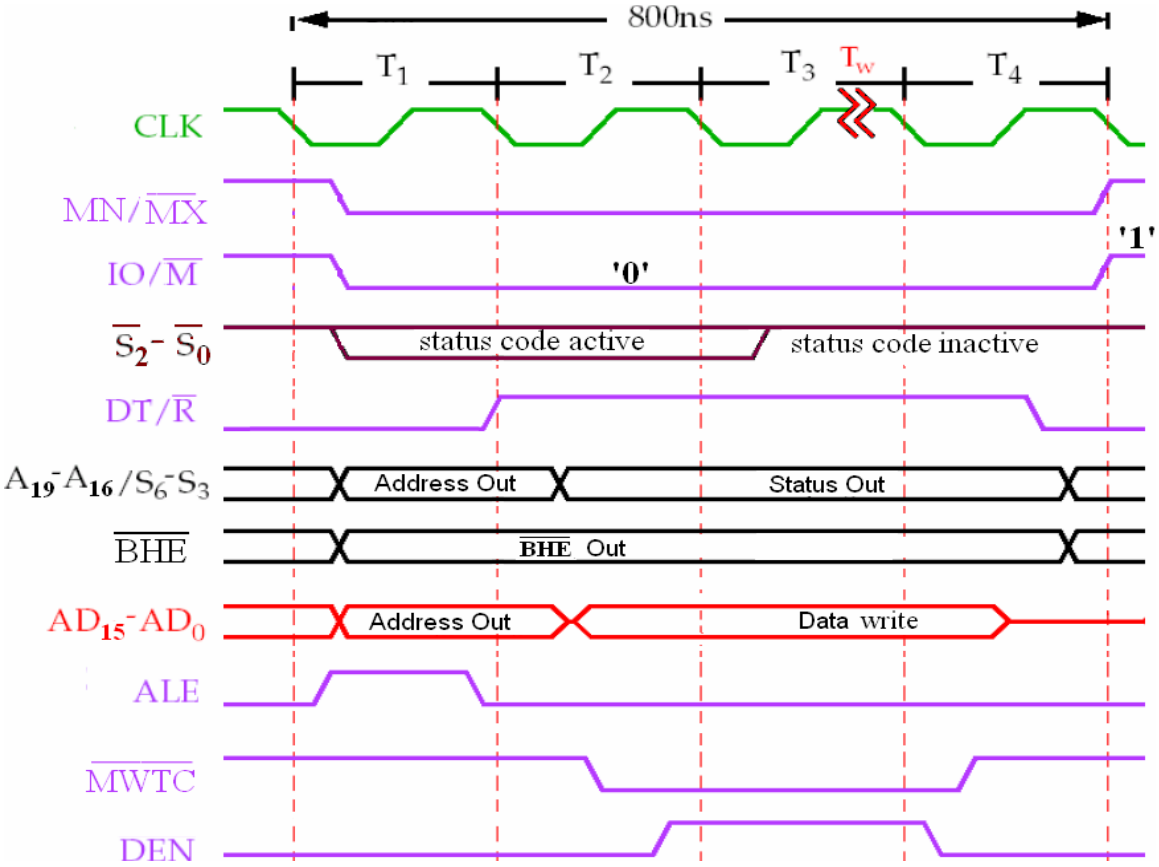
The timing diagram for 8086 maximum mode memory read operation is shown below using logic ‘0’ and ‘1’ waveforms.



Maximum-mode Memory-Read cycle of 8086

The timing diagram for 8086 maximum mode memory read operation is shown below using logic '0' and '1' wave forms.To complete the maximum-mode memory-write bus-cycle, the required control signals with appropriate active logic levels are:

- IO/M = 'logic 0', to select memory interface
- MN/MX = 'logic 0', to select maximum-mode of operation
- DT/R = 'logic 1', to activate the data-transmit mode of 'Data-bus-buffer'
- Valid Physical-address (A0 to A19) and BHE signal is generated by CPU
- ALE-pulse, to latch the valid Physical-address.
- Proper status code S0 to S2 (as shown in table of slide 8) is generated by
- CPU to initiate data writing (MRTC) from the desired memory bank
- DEN = '1', enables the 'Data-Bus-transceiver-buffer' to let data pass
- Reset MRTC and DEN signals to END the read-bus-cycle


800ns
T1
T2
T3
Tw
T4
CLK
MN/MX
'1'
IO/M
'0'
S2- S0
status code active
status code inactive
DT/R
A19-A16/S6-S3
Address Out
Status Out
BHE
BHE Out
AD15-AD0
Address Out
Data write
ALE
MWTC
DEN