



**METHODIST COLLEGE OF
ENGINEERING & TECHNOLOGY**

Affiliated to Osmania University - College Code - 1607

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

EXTRA TOPICS



PROGRAMMER'S MODEL OF 8086

The programming model for a microprocessor shows the various internal registers that are accessible to the programmer.

The Following Figure is a model for the 8086. In general, each register has a special function.

In the programming model there are

1. 4 General Purpose registers(Data Registers)
2. 4 Segment registers
3. 2 Pointer registers
4. 2 Index registers
5. 1 Instruction Pointer register
6. 1 Flag register

General purpose registers:

AX Register (Accumulator): This is accumulator register. It gets used in arithmetic, logic and data transfer instructions. In manipulation and division, one of the numbers involved must be in AX or AL.

BX Register (Base Register): This is base register. BX register is an address register. It usually contain a data pointer used for based, based indexed or register indirect addressing.

CX Register (Counter register): This is Count register. This serves as a loop counter. Program loop constructions are facilitated by it. Count register can also be used as a counter in string manipulation and shift/rotate instruction.

DX Register (Data Register): This is data register. Data register can be used as a port number in I/O operations. It is also used in multiplication and division.

Segment Registers:

There are four segment registers in Intel 8086:

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

A segment register points to the starting address of a memory segment. Maximum capacity of a segment may be up to 64 KB.

Code segment Register(CS):- It is a 16-bit register containing the starting address of 64 KB segment. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register.

Stack segment Register (SS):- It 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 Register (DS):- It 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.

Extra segment Register (ES):- It is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS,DS or ES prefix.

Pointer Registers:

SP Register (Stack Pointer): This is stack pointer register pointing to program stack. It is used in conjunction with SS for accessing the stack segment.

BP Register (Base Pointer): This is base pointer register pointing to data in stack segment. Unlike



SP, we can use BP to access data in the other segments.

Index Registers:

SI Register (Source Index): This is used to point to memory locations in the data segment addressed by DS. By incrementing the contents of SI one can easily access consecutive memory locations.

DI Register (Destination Index): This register performs the same function as SI. There is a class of instructions called string operations, that use DI to access the memory locations addressed by ES.

Instruction Pointer: The Instruction Pointer (IP) points to the address of the next instruction to be executed. Its content is automatically incremented when the execution of a program proceeds further. The contents of the IP and Code Segment Register are used to compute the memory address of the instruction code to be fetched. This is done during the Fetch Cycle.

Flag Register: Status Flags determines the current state of the accumulator. They are modified automatically by CPU after mathematical operations. This allows to determine the type of the result. 8086 has 16-bit status register. It is also called Flag Register or Program Status Word (PSW). There are nine status flags and seven bit positions remain unused.

8086 has 16 flag registers among which 9 are active. The purpose of the FLAGS register is to indicate the status of the processor. It does this by setting the individual bits called flags. There are two kinds of FLAGS;

Status FLAGS and Control FLAGS. Status FLAGS reflect the result of an operation executed by the processor. The control FLAGS enable or disable certain operations of the processor.

8086 ASSEMBLER DIRECTIVES

SEGMENT

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement CODE SEGMENT indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to “bracket” a logical segment containing code or data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement CODE SEGMENT WORD tells the assembler that we want the content of this segment located on the next available word (even address) when segments are combined and given absolute addresses. Without this WORD addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the assembler that the segment may be put together with other segments named CODE from other assembly modules when the modules are linked together.

ENDS (END SEGMENT)

This directive is used with the name of a segment to indicate the end of that logical segment.

□ CODE SEGMENT Start of logical segment containing code instruction statements

CODE ENDS End of segment named CODE

END (END PROCEDURE)

The END directive is put 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 statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

ASSUME

The ASSUME directive is used to tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it



should use the logical segment called DATA.

DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

- PRICES DB 49H, 98H, 29H Declare array of 3 bytes named PRICE and initialize them

with specified values.

- NAMES DB "THOMAS" Declare array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.
- TEMP DB 100 DUP (?) Set aside 100 bytes of storage in memory and give it the name TEMP. But leave the 100 bytes un-initialized.
- PRESSURE DB 20H DUP (0) Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

DD (DEFINE DOUBLE WORD)

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. The statement ARRAY DD 25629261H, for example, will define a double word named ARRAY and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, 9261H, will be put in memory at a lower address than the high word.

DQ (DEFINE QUADWORD)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement BIG_NUMBER DQ 243598740192A92BH, for example, will declare a variable named BIG_NUMBER and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

DT (DEFINE TEN BYTES)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement PACKED_BCD DT 11223344556677889900 will declare an array named PACKED_BCD, which is 10 bytes in length. It will initialize the 10 bytes with the values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run. The statement RESULT DT 20H DUP (0) will declare an array of 20H blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

DW (DEFINE WORD)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement MULTIPLIER DW 437AH, for example, declares a variable of type word named MULTIPLIER, and initialized with the value 437AH when the program is loaded into memory to be run.

- WORDS DW 1234H, 3456H Declare an array of 2 words and initialize them with the specified values.
- STORAGE DW 100 DUP (0) Reserve an array of 100 words of memory and initialize all 100 words with 0000. Array is named as STORAGE.
- STORAGE DW 100 DUP (?) Reserve 100 word of storage in memory and give it the name

STORAGE, but leave the words un-initialized.

EQU (EQUATE)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name. Suppose, for example, you write the statement FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement ADD AL, FACTOR. When the assembler codes this



instruction statement, it will code it as if you had written the instruction `ADD AL, 03H`.

`CONTROL EQU 11000110 B` Replacement

`MOV AL, CONTROL` Assignment

`DECIMAL_ADJUST EQU DAA` Create clearer mnemonic for DAA

`ADD AL, BL` Add BCD numbers

`DECIMAL_ADJUST` Keep result in BCD format Page 28

LENGTH

`LENGTH` is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement `MOV CX, LENGTH STRING1`, for example, will determine the number of elements in `STRING1` and load it into `CX`. If the string was declared as a string of bytes, `LENGTH` will produce the number of bytes in the string. If the string was declared as a word string, `LENGTH` will produce the number of words in the string.

OFFSET

`OFFSET` is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement `MOV BX, OFFSET PRICES`, for example, it will determine the offset of the variable `PRICES` from the start of the segment in which `PRICES` is defined and will load this value into `BX`.

PTR (POINTER)

The `PTR` operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction `INC [BX]`, for example, it will not know whether to increment the byte pointed to by `BX`. We use the `PTR` operator to clarify how we want the assembler to code the instruction. The statement `INC BYTE PTR [BX]` tells the assembler that we want to increment the byte pointed to by `BX`. The statement `INC WORD PTR [BX]` tells the assembler that we want to increment the word pointed to by `BX`. The `PTR` operator assigns the type specified before `PTR` to the variable specified after `PTR`.

We can also use the `PTR` operator to clarify our intentions when we use indirect Jump instructions. The statement `JMP [BX]`, for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as `JMP WORD PTR [BX]`. If we want to do a far jump, we write the instruction as `JMP DWORD PTR [BX]`.

EVEN (ALIGN ON EVEN MEMORY ADDRESS)

As an assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The `EVEN` directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A `NOP` instruction will be inserted in the location incremented over.

`DATA SEGMENT`

`SALES DB 9 DUP (?)` Location counter will point to 0009 after this instruction.

`EVEN` Increment location counter to 000AH

`INVENTORY DW 100 DUP (0)` Array of 100 words starting on even address for quicker read

`DATA ENDS`

PROC (PROCEDURE)

The `PROC` directive is used to identify the start of a procedure. The `PROC` directive follows a name you give the procedure. After the `PROC` directive, the term *near* or the term *far* is used to specify the type of the procedure. The statement `DIVIDE PROC FAR`, for example, identifies the start of a procedure named `DIVIDE` and tells the assembler that the procedure is far (in a segment with different name from the one that contains the instructions which calls the procedure). The `PROC` directive is used with the `ENDP` directive to “bracket” a procedure. Page 29



ENDP (END PROCEDURE)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive, together with the procedure directive, PROC, is used to “bracket” a procedure.

SQUARE_ROOT PROC Start of procedure.

SQUARE_ROOT ENDP End of procedure.

ORG (ORIGIN)

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The location counter is automatically set to 0000 when assembler starts reading a segment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A “\$” it often used to symbolically represent the current value of the location counter, the \$ actually represents the next available byte location where the assembler can put a data or code byte. The \$ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG \$ + 100 tells the assembler increment the value of the location counter by 100 from its current value.

NAME

The NAME directive is used to give a specific name to each assembly module when programs consisting of several modules are written.

LABEL

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to be keep track of how many bytes it is from the start of a segment at any time. The LABEL directive is used to give a name to the current value in the location counter. The LABEL directive must be followed by a term that specifies the type you want to associate with that name. If the label is going to be used as the destination for a jump or a call, then the label must be specified as type *near* or type *far*. If the label is going to be used to reference a data item, then the label must be specified as type *byte*, type *word*, or type *double word*. Here’s how we use the LABEL directive for a jump address.

ENTRY_POINT LABEL FAR Can jump to here from another segment

NEXT: MOV AL, BL Can not do a far jump directly to a label with a colon

The following example shows how we use the label directive for a data reference.

STACK_SEG SEGMENT STACK

DW 100 DUP (0) Set aside 100 words for stack

STACK_TOP LABEL WORD Give name to next location after last word in stack

STACK_SEG ENDS

To initialize stack pointer, use MOV SP, OFFSET STACK_TOP.

EXTRN

The EXTRN directive is used to tell the assembler that the name or labels following the directive are in some other assembly module. For example, if you want to call a procedure, which in a program module assembled at a different time from that which contains the CALL instruction, you must tell the assembler

that the procedure is external. The assembler will then put this information in the object code file so that the linker can connect the two modules together. For a reference to externally named variable, you must specify the type of the variable, as in the statement EXTRN DIVISOR: WORD. The statement EXTRN DIVIDE: FAR tells the assembler that DIVIDE is a label of type FAR in another assembler module. Name or labels referred to as external in one module must be declared public with the PUBLIC directive in the module in which they are defined.



PROCEDURE SEGMENT

EXTRN DIVIDE: FAR Found in segment PROCEDURES

PROCEDURE ENDS

PUBLIC

Large program are usually written as several separate modules. Each module is individually assembled, tested, and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared PUBLIC in the module in which it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules. An example is the statement PUBLIC DIVISOR, DIVIDEND, which makes the two variables DIVISOR and DIVIDEND available to other assembly modules.

SHORT

The SHORT operator is used to tell the assembler that only a 1 byte displacement is needed to code a jump instruction in the program. The destination must in the range of -128 bytes to +127 bytes from the address of the instruction after the jump. The statement JMP SHORT NEARBY_LABEL is an example of the use of SHORT.

TYPE

The TYPE operator tells the assembler to determine the type of a specified variable. The assembler actually determines the number of bytes in the type of the variable. For a byte-type variable, the assembler will give a value of 1, for a word-type variable, the assembler will give a value of 2, and for a double word-type variable, it will give a value of 4. It can be used in instruction such as ADD BX, TYPE-WORD-ARRAY, where we want to increment BX to point to the next word in an array of words.

GLOBAL (DECLARE SYMBOLS AS PUBLIC OR EXTRN)

The GLOBAL directive can be used in place of a PUBLIC directive or in place of an EXTRN directive. For a name or symbol defined in the current assembly module, the GLOBAL directive is used to make the symbol available to other modules. The statement GLOBAL DIVISOR, for example, makes the variable DIVISOR public so that it can be accessed from other assembly modules.

INCLUDE (INCLUDE SOURCE CODE FROM FILE)

This directive is used to tell the assembler to insert a block of source code from the named file into the current source module.



**METHODIST COLLEGE OF
ENGINEERING & TECHNOLOGY**

Affiliated to Osmania University - College Code - 1607

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**Prepare well
All The Best**