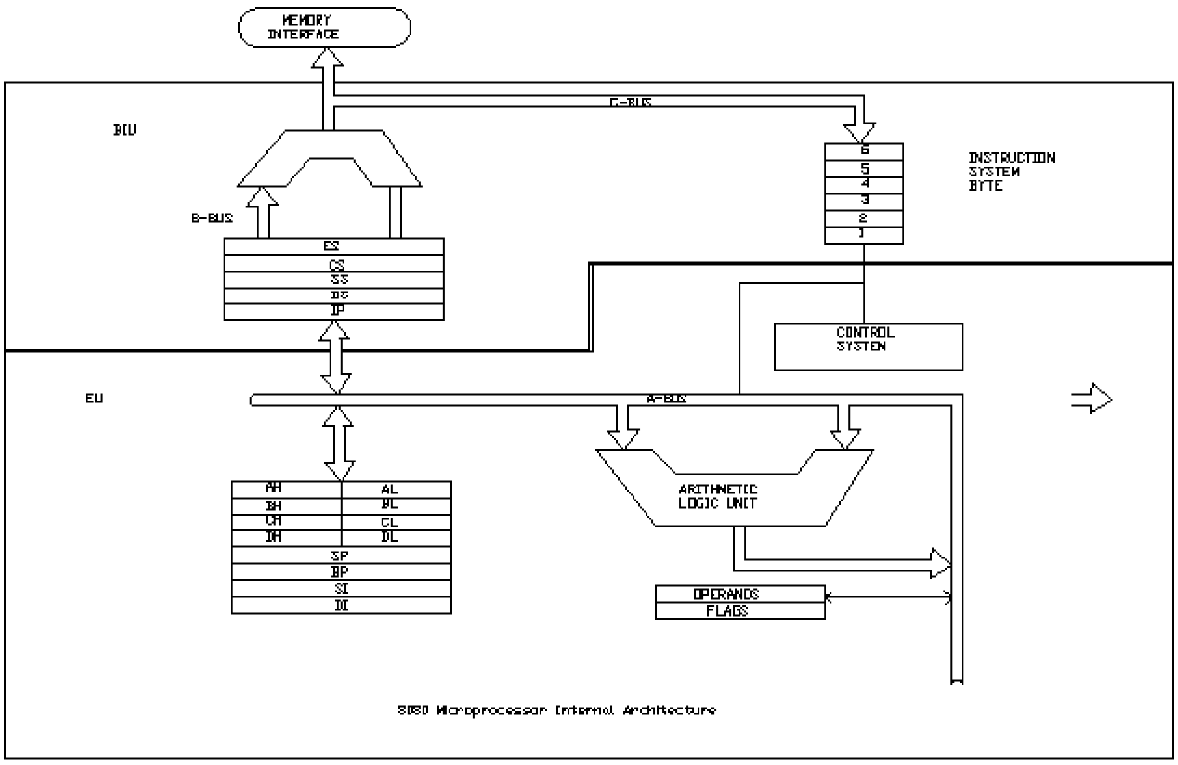
3. Assembly Language Programming (16 bit Microprocessor)

Introduction to 16 bit Microprocessor Architecture

The 8086 is a 16-bit microprocessor. The term 16 bit implies that its arithmetic logic unit, its internal registers, and most of its instructions are intended to work with 16 bit binary data. The 8086 has a 16 bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20 bit address bus, so it can address any one of 2^{20}, or 1,048,576 memory locations.

8086 CPU is divided into 2 independent functional parts to speed up the processing namely **BIU** (Bus interface unit) & **EU** (execution unit).



BIU: It handles all transfers of data and addresses on the buses for the execution unit.

- Sends out addresses
- Fetches instructions from memory
- Read / write data from/to ports and memory i.e handles all transfers of data and addresses on the busses

EU

- Tells BIU where to fetch instructions or data from
- Decodes instructions
- Executes instructions

Execution Unit

Instruction Decoder & ALU:

Decoder in the EU translates instructions fetched from the memory into a series of actions which the EU carries out.16-bit ALU in the EU performs actions such as AND, OR, XOR, increment, decrement etc.

FLAG Register:

It is a 16-bit register. 9-bit are used as different flags, remaining bits unused

U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Out of 9-flags, 6 are conditional flags and three are control flags

Conditional flags:

These are set or reset by the EU on the basis of the results of some arithmetic or logic operation. 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instructions.

- OF (Overflow flag): is set if there is an arithmetic overflow, i.e. the size of the result exceeds the capacity of the destination location.
- SF (Sign flag): is set if the MSB of the result is 1
- ZF (Zero flag): if the result is zero
- AF (Aux carry flag): is set is there is carry from lower nibble to upper nibble or from lower byte to upper byte
- PF (Parity flag): is set if the result has even parity
- CF (Carry flag): is set if there is carry from addition or borrow from subtraction

Control flags:

They are set using certain instructions. They are used to control certain operations of the processor.

- TF (Trap flag): for single stepping through the program
- IF (Interrupt flag): to allow or prohibit the interruption of a program
- DF (Direction flag): Used with string instructions

General purpose Registers:

- 8 GPRs AH, AL (Accumulator), BH, BL, CH, CL, DH,DL are used to store 8 bit data.
- AL register is also called the accumulator
- Used individually for the temporary storage of data
- GPRs can be used together (as register pair) to store 16-bit data words. Acceptable register pairs are:

AH-AL pair AX register	BH-BL pair BX register
CH-CL pair CX register	DH-DL pair DX register

Pointer and Index registers: SP (Stack Pointer), BP (Base pointer), SI (Source Index), DI (Destination index)

The two pointer registers, SP and BP are used to access data in the stack segment. The SP is used as offset from current Stack Segment during execution of instruction that involve stack. SP is automatically updated. BP contains offset address and is utilized in based addressing mode.

Index Registers:

EU also contains a 16 bit source index (SI) register and 16 bit destination index (DI) register. These three registers can be used for temporary storage of data similarly as the general purpose registers. However they are specially to hold the 16-bit offset of the data word.

Bus Interface Unit

The QUEUE:

When EU is decoding or executing an instruction, bus will be free at that time. BIU prefetches up to 6-instructions bytes to be executed and places them in QUEUE. This improves the overall speed because in each time of execution of new instruction, instead of sending address of next instruction to be executed to the system memory and waiting fro the memory to send back the instruction byte, EU just picks up the fetched instruction byte from the QUEUE.

The BIU stores these pre-fetched bytes in a first-in-first-out (FIFO) register set called a queue. Fetching the next instruction while the current instruction executes is called pipelining.

Segment Registers:

The BIU contains a dedicated address, which is used to produce the 20 bit address. The bus control logic of the BIU generates all the bus control signals, such as the READ and WRITE signals, for memory and I/O. The BIU also has four 16 bit segments registers namely:

- Code segment: holds the upper 16-bits of the starting addresses of the segment from which BIU is currently fetching instruction code bytes.
- Stack segment: store addresses and data while subprogram executes
- Extra segment: store upper 16-bit s of starting addresses of two memory segments that are used for data.
- Data segment: store upper 16-bit s of starting addresses of two memory segments that are used for data.

Code Segment Register (CS) and Instruction Pointer (IP)

All program instructions located in memory are pointed using 16 bits of segment register Cs and 16 bits offset contained in the 16 bit instruction pointer (IP). The BIU computes the 20 bit physical address internally using the logical address that is the contents of CS and IP. 16 bit contents of CS will be shifted 4 bits to the left and then adding the 16 bit contents of IP. Thus all instructions of the program are relative contents of IP. Simply stated, CS contains the base or start of the current code segment, and IP contains the distance or offset from this address to the next instruction byte to be fetched.

Graphically,

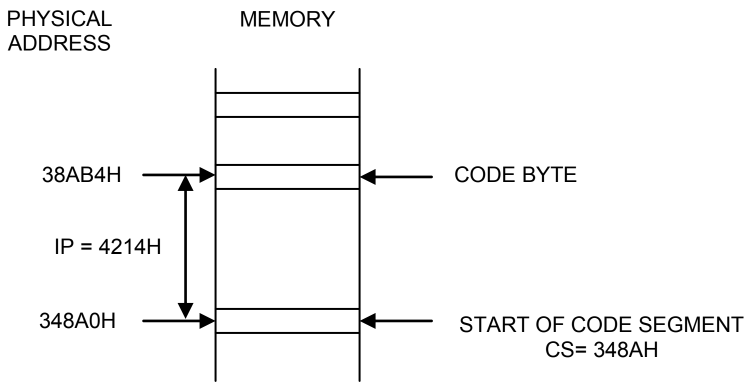


Fig: Diagram showing addition of IP to CS to produce the physical address of code byte

Stack Segment Register (SS) and Stack Pointer (SP)

A stack is a section of memory to store addresses and data while a subprogram is in progress. The stack segment register points to the current stack. The 20 bit physical stack address is calculated from the SS and SP. The programmer can also use Base Pointer (BP) instead of SP for addressing. In this case the 20 bit physical address is calculated using SS and BP.

Addressing Modes

- Immediate Addressing mode: Operand (one of the operand say source) is immediate data in the instruction
  Example
  MOV CX,234AH
  ADI 23H

- Register Addressing mode: Transfers a copy of byte or word from source register or memory location to destination register or memory location.
  Example
  MOV CX, DX
  transfers the content of DX to CX register pair

- Direct Addressing mode: One operand should be memory location i.e moves byte or word between memory location and the register.
  Example
  MOV BX, [4371H]
  Copies 16-bit word into a memory pointed by the displacement address 4371H to the BX register. (In little endian format: lowest byre->lowest address)

- Register Indirect Addressing: Transfers a word or byte between register and memory addressed by index or base register. Index or base register are BP, BX, DI, SI.
  Example
  MOV AX,[BX]
  Copies the word-sized data from data segment offset address indexed by BX into AX

- Register Relative Addressing Mode: Transfers a byte or word between register and memory location addressed by an index or base register plus displacement.
  Example

  MOV AX, [BX+4]

- Base-Plus-Index Addressing Mode: Transfers a word or byte between register and memory location addressed by base register (BP or BX) plus index register (DI or SI).
  Example

  MOV [BX+DI], CL

  Used for locating array data type.

3.1 Assembler instructions format: Op-codes, memonics and operands

Machine Language:

There is only one programming language that any computer can actually understand and execute: its own native binary machine code. This is the lowest possible level of language in which it is possible to write a computer program. However, binary code is not native to humans and it is very easy for error to occur in the program. These bugs are not easy to determine in a pool of binary digits. Also for CPU like 8086 it is tedious and virtually impossible to memorize thousands of binary instructions codes.

Example

Program memory address	Content (binary)	Content (Hex)
00100H	0000 0100	04h

Assembly Language:

Programs in assembly language are represented by certain words representing the operation of instruction. Thus programming gets easier. These words (usually two-to-four letter) is used to represent each instruction are called Mnemonics. Assembly language statements are generally written in a standard form that has four fields.

- Label field
- Instruction, Mnemonic or Op-code field
- Operation field
- Comment field

Let us consider a simple example to add two numbers

Label	Mnemonic	Operand	Comment
Start:	MVI	A, 10h	Move 10h into accumulator
	MVI	B, 20h	Move 20h into register B
	ADD	B	Add the contents of register B with accumulator

Thus, we see the ease with the assembly language rather than machine language.

However, it must be remembered that for execution these codes are converted to machine codes. This is done by assembler. An assembler translates a program written in assembly language into machine language program (object code). Assembly language program are called ‘source codes’. Machine language programs are known as object codes. A translator converts source codes to object codes and then into executable formats. The process of converting source code into object code is called compilation and assembler does it. The process of converting object codes into executable formats is called linking and linker does it.

There are two ways of converting an assembly language program into machine language: manual assembly and using assembler. In manual assembly, the programmer is the assembler and he converts the entire mnemonic into its numerical machine language representation by looking up a table of microprocessor's instruction set. This method is only convenient for short programs.

An assembler reads each instruction of a program as ASCII characters and translates them to respective binary op-codes. An advantage of an assembler is address computation. Most programs used addresses within the program as data or as labels for jumps and calls. Assembler automatically takes care of addresses. Manual calculation of addresses for these instructions is a time consuming task.

3.2 Types of Assemblers

One Pass Assembler:

This assembler reads the assembly language program once and translates the assembly language program to machine code. This assembler has the program of defining forward references. This means that a branching instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.

Two Pass Assembler:

This assembler scans the assembly language program twice. In the first pass, the assembler generates the table of the symbols. A symbol table consists of labels with addresses assigned to them. In this way labels can be used for JUMP statements, and no address calculation has to be done by the user. On the second pass, the assembler translates the assembly language program into the machine code. The two pass assembler is thus easier to use.

3.3 Macro assemblers, linking assemblers and assembler directives

Macro assembler is an assembly language that allows macros to be defined and used. The **Microsoft Macro Assembler** (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows.

Assembler can be linked with two or more than two assembler called linking assemblers.

Assembly directives, also called pseudo opcodes, pseudo-operations or pseudo-ops, are instructions that are executed by an assembler at assembly time, not by a CPU at run time. They can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of a program to make it easier to read and maintain.

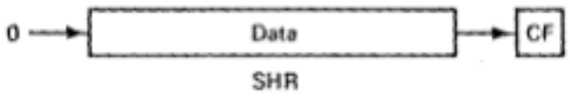
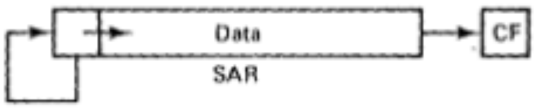
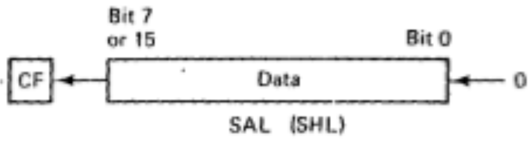
Some Common instruction of 8086

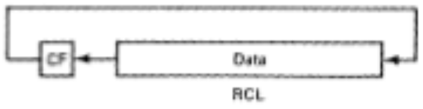
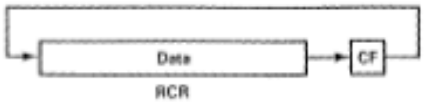
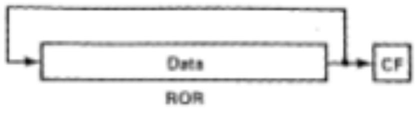
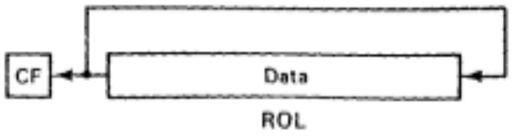
a) Data transfer instruction	
Mnemonics	**Description**
Mov R/M,RM/Immediate eg: MOV AX,BX MOV BX,500H MOV BH,50H	Copy specified byte or word from specified source to specified destination
Push R/M POP R/M	Copy specified word to top of stack (TOS). Copy word from top of stack to specified location
XCHG R/M,R/M	Exchange bytes or exchange word
IN AL/AX,port no/DX	Copy a byte or word from specified port to accumulator
OUT Port no/DX,AL/AX	Copy a byte or word from accumulator to specified port
LEA R,M	Load effective address of operand into specified register
SAHF	Store (copy) A H register to low byte of flag register
LAHF	Load(copy to) AH with the low byte of the flag register
Push F	Copy flag register to TOS
POP F	copy word at TDS to flag register
b) Arithmetic instruction	
Mnemonics	**Description**
ADD R/M, R/M/Immediate	Add specified byte to byte or specified word to word [Flag affected AF, CF, OF, PF, SF & ZF]
ADC R/M,/IMMEDIATE	Adds content of the CF flag to operand 1,and then adds operand 2 to operand
INCR/M	Increment specified byte or word by 1
SUB R/M,R/M/IMMEDIATE	Subtract specified byte from byte or word from word
SBB R/M,R/M/IMMEDIATE	Subtracts the contents of the CF from operand 1 and then subtracts operand 2 from operand 1

DEC R/M	Decrement 1 from a byte or word
NEG R/M	Negate- invert each bit of a specified byte or word and add 1
CMPR/M,R/M/IMMEDIATE	Compare two specified bytes or two specified words

	CF	SF	ZF
OP1>OP2	1	1	0
OP1=OP2	0	0	1
OP1<OP2	0	0	0

MUL R/M	Multiplies an unsigned multiplicand by an unsigned multiplier AL OR AX are assumed as multiplicand
IMUL R/M	Multiplies on signed multiplicand by a signed multiplier
DNR/M	Divides an unsigned dividend in operand 1 by an unsigned division
c) Bit manipulation instructions	
Mnemonics	**Description**
<u>Logical instructions</u>	
NOT R/M	Invert each bit of a byte or word
AND/OR/XOR R/M, R/M/IMMEDIATE	Perform respective logical operation on each bit of a byte or word
<u>Shift instruction</u>	
SHL/SHR eg: SHL/SHR R/M,CL/IMMEDIATE	Shift logical left /shift logical right [Flag affected: CF, OF, PF, SF&ZF] Puts zero (s) in MSB(s)

SHR Shift Right 	
SAL/SAR eg: SAL/SAR R/M,CL/IMMEDIATE	Shift algebraic left /shift algebraic right [Flag affected: CF,OF,PF,SF&ZF] Puts zero(s) to vacant places [MSB does not changes]
SAR Shift Arithmetic Right Shift bits right with sign preservation  SAL Shift Arithmetic Left Shift left, sign bit to CF (OF = 1 if sign bit changes) 	
Rotate instruction	
RCL/RCR eg: RCL/RCR,CL/IMMEDIATE	Rotate left through carry /Rotate right through carry [Flag affected CF &OF]

RCL — rotate carry left  RCR — rotate carry right 	
ROL/ROR eg: ROL R/M,CL/Immediate	Rotate left/Rotate right [Flag affected CF&OF]
ROR — rotate right  ROL — Rotate Left 	
d) Program Execution Transfer instruction	
<u>Unconditional transfer instruction</u>	
Call R/M eg:- CALL MULTI CALL BX	Call a procedure (sub program) save return address on stack [BX contains the offset of the first instruction of the procedure replaces contents of IP with contents of register BX]
RET	Returns from a procedure previously entered by a call
JMP R/M eg: JMP continue	Jumps to a designed address under any conditions [BX contains the offset of the destination instruction

JMP BX	in CS Replaces contents of IP with the contents of BX]
Conditional transfer instruction	
Flag Checked	**Description**
1. JA/JNBE CF=0,z=0	1. Jump if above /Jump if not below or equal
2. JAE/JNB CF=0 (unsigned data)	2. Jump if above or equal/Jump if not below
3. JB/JNAE CF=1 (unsigned data)	3. Jump if below/Jump if not above or equal
4. JBE/JNA CF=1 or AF=1 (unsigned data)	4. Jump if below or equal/Jump if not above
5. JC CF=1	5. Jump if carry flag set
6. JE/JZ ZF=1 (signed/unsigned data)	6. Jump if equal /Jump if zero flag set
7. JG/JNLE ZF=0, SF=OF (signed data)	7. Jump if greater /Jump if not less than or equal
8. JGE/JNL SF=OF (signed data)	8. Jump if greater than or equal/Jump if not less than
9. JL/JNGE ZF=1 or SF not=OF	9. Jump if less than or equal/Jump if not greater than
10. JNC CF=0	10. Jump if no carry
11. JNE/JNZ ZF=0 (signed/unsigned)	11. Jump if not equal/Jump if not zero
12. JNO OF=0	12. Jump if no over flow
13. JNP/JPO PF=0	13. Jump if not parity/Jump it parity odd
14. JNC SF=0	14. Jump if not sign
15. JO OF=1	15. Jump if over flow flag set
16. JP/JPE PF=1	16. Jump if parity/Jump if parity even
17. JS SF=1	**17.** Jump if sign set

Sample 80x86 Assembly Language Programs

Now we need to tie the concepts of assembly language to the specifics of the 80x86 assembly language. The best way to do this is to create a simple program. We begin with the general framework used to support the program. Figure 17-20 presents the basic skeleton code of an

80x86 assembly language program.

```
.MODEL SMALL
       .STACK 100H
       .DATA
       .CODE
MAIN PROC FAR

MAIN ENDP END MAIN
```

Figure 17-20 Skeleton Code for a Simple Assembly Program

Let's examine this code line-by-line.

- The first line contains the string ".MODEL SMALL". We see from Table 17-1 that this tells the compiler to use one code segment less than or equal to 64K and one data segment less than or equal to 64K. The program we are writing here is quite small and will easily fit in this memory model.
- The next line, ".STACK 100H", tells the instructor to reserve 256 bytes (hexadecimal 100) for the stack.
- The next line, ".DATA", denotes the beginning of the data segment. All of the data for the application will be defined between the .DATA and .CODE directives.
- The next line, ".CODE", denotes the beginning of the code segment. All of the code will be defined after this directive.
- "MAIN PROC FAR" identifies a block of code named main that will use both the segment and offset for addressing.
- "MAIN ENDP" identifies the end of the block of code named MAIN.
- "END MAIN" tells the assembler when it has reached the end of all of the code.

The next step is to insert the data definitions and code that go after the .DATA and .CODE directives respectively. The first piece of code we need to write will handle some operating system house keeping. First, we need to start the program by retrieving the address that the operating system has assigned to the data segment. This value needs to be copied to the DS register. We do this with the two lines of code presented in Figure 17-21.

These lines need to be placed immediately after the MAIN PROC FAR line. **MOV AX,@DATA ;Get assigned data segment ; address from O/S MOV DS,AX ;Copy it to the**

DS register **Figure 17-21** Code to Assign Data Segment Address to DS Register When the program ends, we need to transfer control back to the operating system. This is done using a software interrupt. At this point it is not necessary to understand this process other than to say that when the O/S receives this interrupt, it knows that the application is finished and can be removed from memory. Placing the lines from Figure 17-22 immediately before the line MAIN ENDP in the code will do this.

MOV AX,4C00H ;Load code indicating normal ; program termination

INT 21H ;Call interrupt to end program

Figure 17-22 Code to Inform O/S that Program is Terminated

At this point, our skeleton code should look like that shown inFigure 17-23.

```
      .MODEL SMALL
      .STACK 100H
      .DATA
      .CODE
MAIN PROC FAR MOV AX,@DATA

MOV DS,AX

MOV AX,4C00H

INT 21H MAIN ENDP

END MAIN

;Load DS with assigned
; data segment address
;Use software interrupt
; to terminate program
```

Figure 17-23 Skeleton Code with Code Added for O/S Support

Now all we need is a program to write. The program presented here is a simple mathematical calculation using data from the data segment. Specifically, we will be calculating the following algebraic expression where A, B, C, and RESULT are defined to be 16-bit words in the data segment.

RESULT = (A÷8) + B – C

Let's begin by defining what the data segment is going to look like. Each of the variables, A, B, C, and RESULT, need to have a word- sized location reserved in memory for them. Since the first three will be used as inputs to the expression, they will also need to be initialized.

For the sake of this example, let's initialize them to 104_{10}, 100_{10}, and 52_{10} respectively. Since

RESULT is where the calculated result will be stored, we may leave that location undefined. Figure 17-24 presents the four lines of directives used to define this memory.

```
A DW104 B DW100 C DW52 RESULT DW ?
```

Figure 17-24 Data Defining Directives for Example Code

This code will be inserted between the .DATA and .CODE directives of the code in Figure 17-23.

The next step is to write the code to compute the expression. Begin by assuming the computation will occur in the accumulator register, AX. The process will go something like this.

- Load AX with value stored at the memory location identified by A.
- Divide AX by eight using the arithmetic right shift instruction.
- After dividing AX, add the value stored at the memory location identified by B.
- After adding B to AX, subtract the value stored at the memory location identified by C.
- Lastly, store the result contained in AX to the memory location RESULT.

Converting this step-by-step sequence into assembly language results in the code presented in Figure 17-25.

```
MOV  AX,A
SAR  AX,3
ADD  AX,B
SUB  AX,C
MOV  RESULT,AX
;Load A from memory
;Divide A by 8
;Add B to (A/8)
;Subtract C from (A/8)+B
;Store (A/8)+B-C to RESULT
```

Figure 17-25 Step-by-Step Example Operation Converted to Code

The last step is to insert this code after the two lines of code that load the data segment register but before the two lines of code that perform the program termination in Figure 17-23. Figure 17-26 presents the final program.

```
.MODEL SMALL
      .STACK 100H
      .DATA
.  A  DW 104
. B  DW 100

. C  DW 52

RESULT DW ?

.CODE MAIN PROC

FAR
AX,@DATA   ;Load DS with assigned
DS,AX      ; data segment address
MOV MOV MOV AX,A SAR AX,3 ADD AX,B SUB AX,C MOV RESULT,AX MOV
AX,4C00H INT 21H

;Load A from memory
;Divide A by 8
;Add B to (A/8)
;Subtract C from (A/8)+B
;Store A/8+B-C to RESULT
;Use software interrupt
; to terminate program
MAIN ENDP END MAIN
```

Figure 17-26 Final Code for Example Assembly Language Program