Lab 2 - x86 Registers

The main tools to write programs in x86 assembly are the processor registers. The registers are like memory locations built-in the processor. Using registers instead of main memory (RAM) makes the process faster and cleaner.

There are 8 general-purpose registers, 6 segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP).

2.1 Modes of Operation of x86 Processors

x86 processors have three primary modes of operation: protected mode, real-address mode, and system management mode. A sub-mode, named virtual-8086, is a special case of protected mode. Here are short descriptions of each:

2.1.1 Protected Mode

Protected mode is the native state of the processor, in which all instructions and features are available. Programs are given separate memory areas named segments, and the processor prevents programs from referencing memory outside their assigned segments. When the computer boots, it first enters real mode; the operating system is responsible for switching into protected mode.

2.1.1.1 Virtual-8086 Mode

While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe environment. In other words, if a program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time. A modern operating system can execute multiple separate virtual-8086 sessions at the same time.

2.1.2 Real-Address Mode

Real-address mode implements the programming environment of an early Intel processor with a few extra features, such as the ability to switch into other modes. This mode is useful if a program requires direct access to system memory and hardware devices. Programs running in real address mode can cause the operating system to crash (stop responding to commands).

2.1.3 System Management Mode

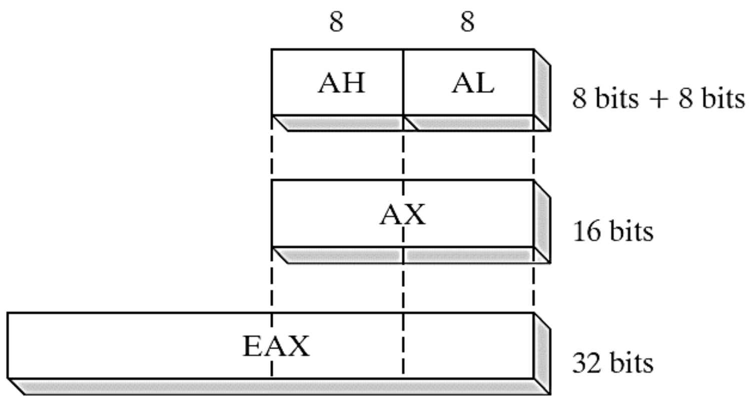
System management mode (SMM) provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

2.2 General Purpose Registers

The general-purpose registers are primarily used for arithmetic and data movement. General purpose registers are the one we use most of the time. Most of the instructions performed on these registers. They all can be broken down into 16-bit and 8-bit registers. As shown in figure below, the lower 16 bits of the EAX register can be referenced by the name AX. In protected mode, there are 8 32-bit general-purpose registers.

General-Purpose Registers divide for two groups:

- Data Registers
- Index and Pointer Registers



Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named **AH** and an 8-bit lower half named **AL**.

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

Register Name	32-Bit	16-Bit
Source Index	ESI	SI
Destination Index	EDI	DI
Base Pointer	EBP	BP
Stack Pointer	ESP	SP

2.2.1 Uses of General-Purpose Registers

Register	Complete Name	Uses
EAX	Accumulator Register	It is used for I/O port access, arithmetic operations, interrupt calls, etc.
EBX	Base Register	It is used as a base pointer for memory access.
ECX	Counter Register	It is used as a loop counter.
EDX	Data Register	It is also used with AX register for multiply and divide operations involving large values.

To access the lower 8 bits of AL register (AL), we can use this instruction:

```
mov AL,5            ; Moving 5 to AL 8 bit Register
```

Similarly, to access the higher 8 bits of AX register (AH), the instruction can be used in this way:

```
mov AH,10           ; Moving 10 to AH 8 bit Register
```

Write this program and find what is the output value?

```
; This Code adds two 32-bit integers.

Include Irvine32.inc

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword
.code
main proc
        mov al,5            ; Moving 5 to AL Register
        mov ah,6            ; Adding 6 to AH Register
        add ah,al           ; and store the result in the AH itself

        call WriteInt

        invoke ExitProcess,0
main endp
end main
```

The output of this Code is not **+11**. The reason is that we are working on 8-bit register parts of EAX register. So, after the addition we called the console print procedure **WriteInt** to display the results of whole EAX register not AX. That is why it produced the garbage value.

One solution is that, in the beginning we intentionally set the EAX to zero to erase garbage value.

```
mov EAX,0           ; Setting EAX register to zero
.
.
.
Call WriteInt
```

Then another approach is to move the sum value to EAX before calling **WriteInt** procedure:

```
mov al,5            ; Moving 5 to AL Register
mov ah,6            ; Moving 6 to AH Register
add ah,al           ; Adding result to AH itself

mov eax,ah          ; and store the result in the EAX
call WriteInt
```

The above code will produce error because we are trying to move AL to EAX register. The AH is 8-bit while EAX is 32-bit register that is why the system produced the error because it can not automatically convert the 8-bit value in AH to 32-bit value.

We can solve this problem using **movzx** (move with zero-extend) instruction:

```
movzx eax,ah        ; Moving the 8-bit integer from AH to 32-bit register EAX
```

The above mechanism can also be used in to store the value from 16-bit AX register to 32-bit EAX register:

```
movzx eax,ax        ; Moving the 16-bit integer from AX to 32-bit register EAX
```

For the signed numbers we use **movsx** (move with sign-extend) instruction to perform identical operation:

```
mov al, 5           ; Moving 5 to AL Register
mov ah, -6          ; Moving signed -6 to AH Register
add ah, al          ; Adding result to AH itself

movsx eax, ah       ; and store the signed result in the EAX
call WriteInt
```

2.2.2 Specialized Uses

Some general-purpose registers have specialized uses:

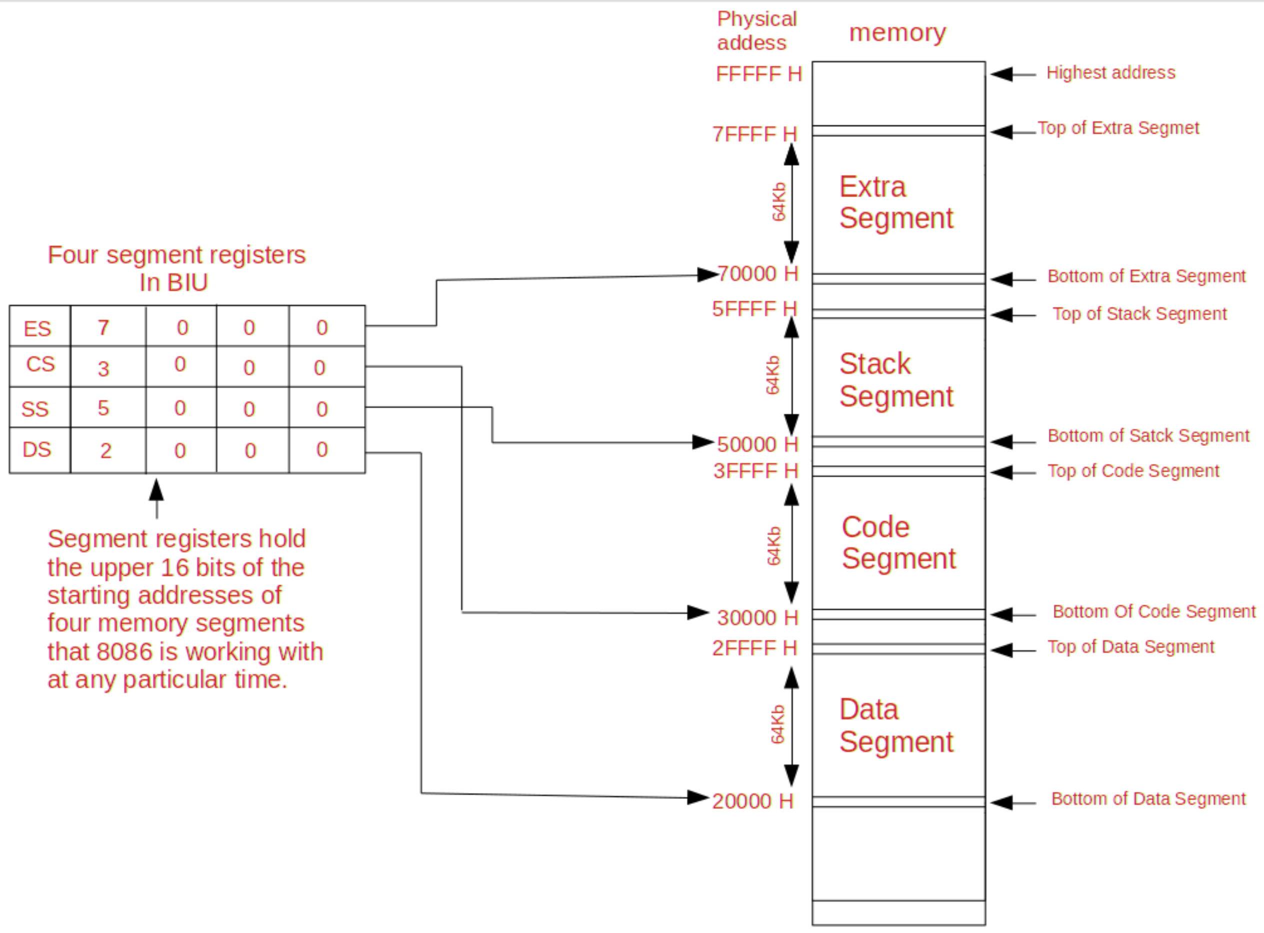
- **EAX** is automatically used by multiplication and division instructions. It is often called the *extended accumulator* register.
- The CPU automatically uses **ECX** as a loop counter.
- **ESP** addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the *extended stack pointer* register.
- **ESI** and **EDI** are used by high-speed memory transfer instructions. They are sometimes called the *extended source index* and *extended destination index* registers.
- **EBP** is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

2.3 Segment Registers

Segment registers hold the segment address of various items. They can only be set by a general-purpose register or special instructions. In real-address mode, 16-bit segment registers indicate base addresses of pre-assigned memory areas named segments. In protected mode, segment registers hold pointers to segment descriptor tables. Some segments hold program instructions (code), others hold variables (data), and another segment named the stack segment holds local function variables and function parameters.

Register	Complete Name	Uses
CS	Code Segment Register	It contains all the instructions to be executed. It stores the starting address of the code segment
DS	Data Segment Register	It contains data, constants and work areas. It stores the starting address of the data segment.
ES, FS, GS	Extra Segment Registers	These registers provide additional segments for storing data. The extra segment ES is the default destination for string operations (for example MOVS or CMPS). FS and GS have no hardware-assigned uses.

SS	Stack Segment Register	It contains data and return addresses of procedures or subroutines. It stores the starting address of the stack.



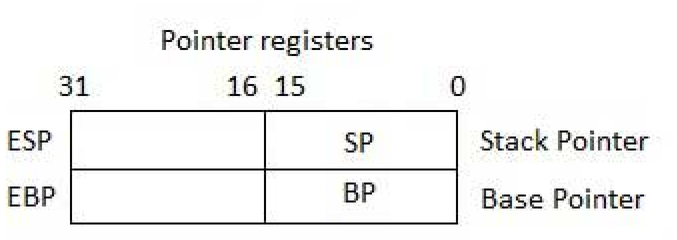
The main advantages of segmentation are as follows:

- It provides a powerful memory management mechanism.
- Data related or stack related operations can be performed in different segments.
- Code related operation can be done in separate code segments.
- It allows different processes to easily share data.
- It is possible to enhance the memory size of code, data or stack segments by allotting more than one segment for each area.

2.4 Pointer Registers

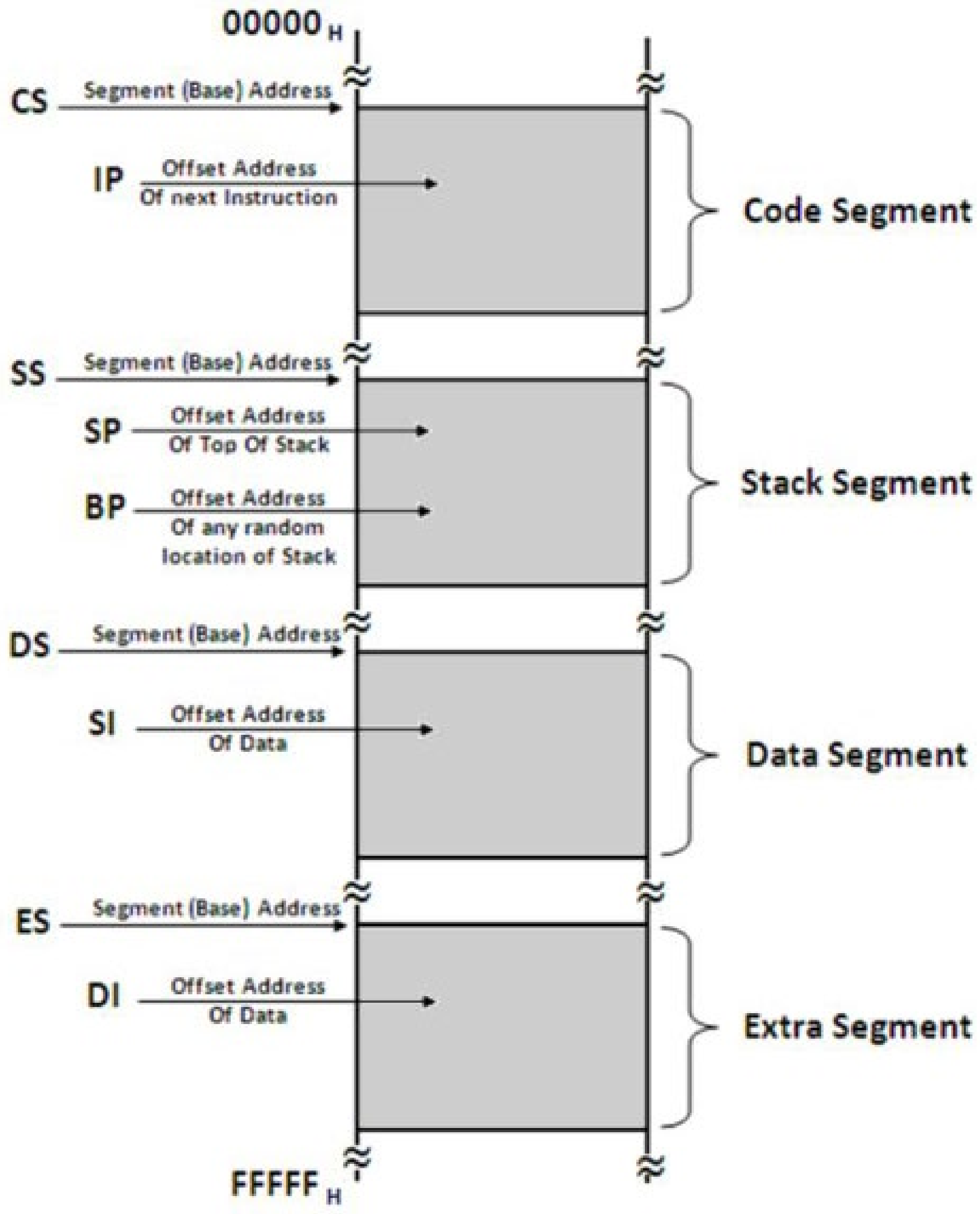
The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers. The register with an "E" prefix can only be used in protected mode.

Register	Complete Name	Uses
IP	Instruction Pointer	It stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
SP	Stack Pointer	It provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
BP	Base Pointer	It helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.



Each function has local memory associated with it to hold incoming parameters, local variables, and (in some cases) temporary variables. This region of memory is called a *stack frame* and is allocated on the process' stack. A frame pointer (the **EBP** register on intel x86 architectures, **RBP** on 64-bit architectures) contains the base address of the function's frame. The code to access local variables within a function is generated in terms of offsets to the frame pointer.

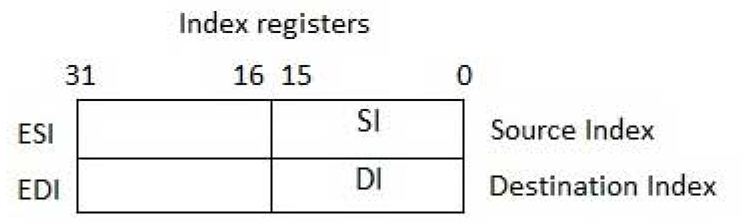
The stack pointer (the **ESP** register on intel x86 architectures or **RSP** on 64-bit architectures) may change during the execution of a function as values are pushed or popped off the stack (such as pushing parameters in preparation to calling another function). The frame pointer doesn't change throughout the function.



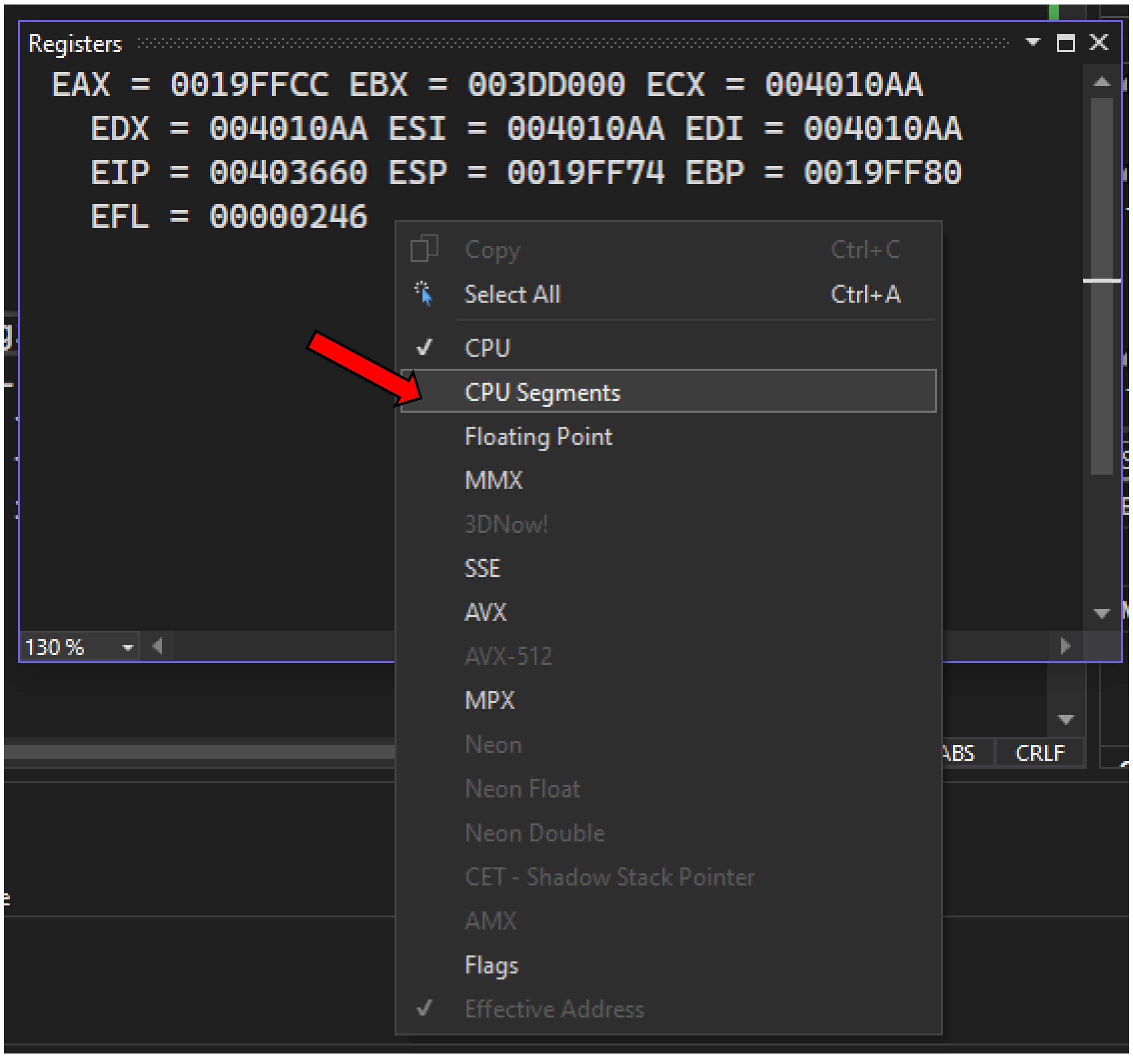
2.5 Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers.

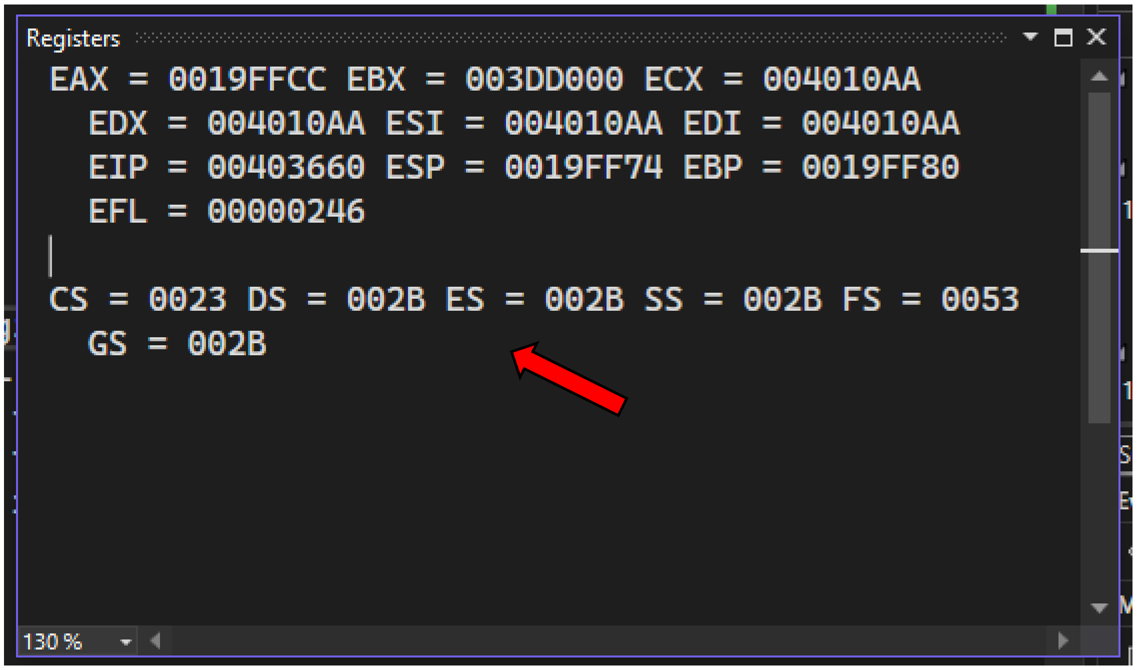
Register	Complete Name	Uses
SI	Source Index	It is used as source index for string operations.
DI	Destination Index	It is used as destination index for string operations.



White debugging, to show the values of the Segment registers, first open the Registers window and then right click in the window. A menu will appear. Click the "CPU Segment" option. It will start showing the segment registers in that Registers window.



```
EAX = 0019FFCC EBX = 003DD000 ECX = 004010AA
  EDX = 004010AA ESI = 004010AA EDI = 004010AA
  EIP = 00403660 ESP = 0019FF74 EBP = 0019FF80
  EFL = 00000246
```



```
EAX = 0019FFCC EBX = 003DD000 ECX = 004010AA
  EDX = 004010AA ESI = 004010AA EDI = 004010AA
  EIP = 00403660 ESP = 0019FF74 EBP = 0019FF80
  EFL = 00000246

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053
  GS = 002B
```

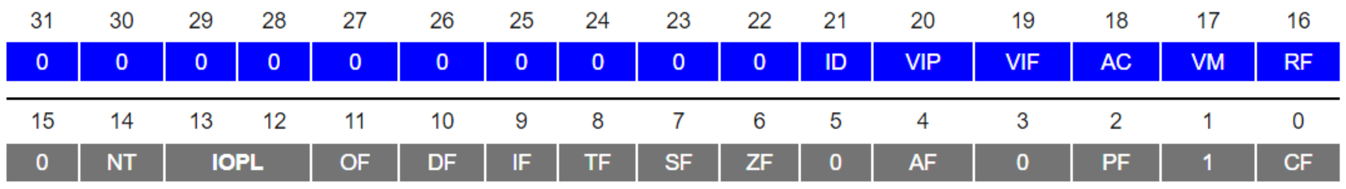
Now write this code in Visual Studio and start debugging and trace all the registers:

```
; This Code adds two 8-bit integers.

Include Irvine32.inc

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.data
sum BYTE 0

.code
main proc
      mov EAX,0           ; Setting AL register to zero
      mov al, 5           ; Moving 5 to AL Register
      mov ah, -6          ; Moving signed -6 to AH Register
      add ah, al          ; Adding result to AH itself
      mov sum, ah         ; and store the result in the sum variable
      movsx eax,sum
      call WriteInt

      invoke ExitProcess,0
main endp
end main
```

2.6 EFLAGS Register

The EFLAGS (or just Flags) register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation. Some instructions test and manipulate individual processor flags. A flag is set when it equals 1; it is clear (or reset) when it equals 0.

The EFLAGS is a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor. The names of these bits are:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Note: The bits named 0 and 1 are reserved bits and shouldn't be modified.

The different use of these flags are as follows:

Bit Position	Flag	Flag Name	Use
0	CF	Carry Flag	It is set when the result of an unsigned arithmetic operation is too large to fit into the destination. Value of this flag equal to 1 if we have a carry value from or to the Most Significant Bit (MSB), and usually this carry happens in the add operation. Carry flag also serves as a borrow flag for subtraction. In case of subtraction it is set when borrow is needed
2	PF	Parity Flag	It is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error checking when there is a possibility that data might be altered or corrupted.
4	AF	Auxiliary Carry Flag/Adjust Flag	Contains Carry bit of Binary Code Decimal (BCD) numbers arithmetic operations. Value of this flag equal to 1 if we have a carry value from or to the fourth bit (bit-3), this flag used in the Binary Coded Decimal (BCD).
6	ZF	Zero Flag	It is set (1) when the result of an arithmetic or logical operation generates a result of zero.
7	SF	Sign Flag	It is set when the result of an arithmetic or logical operation generates a negative result.

			Value of this flag equal to 1 if the value of the Most Significant Bit (MSB) equal 1, which means the value is negative. If the value of the Most Significant Bit (MSB) equal 0 the value of flag is equal to 0.
8	TF	Trap Flag	It allows setting the operation of the processor in single-step mode
9	IF	Interruption Flag	It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
10	DF	Direction Flag	It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
11	OF	Overflow Flag	It is set when the result of a signed arithmetic operation is too large or too small to fit into the destination register. The Overflow flag is set by value 1 when the signed result of an operation is invalid or out of range. This flag is set if the result is out of range. For addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice-versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice-versa. When adding two integers, remember that the Overflow flag is only set when: • Two positive operands are added, and their sum is negative. • Two negative operands are added, and their sum is positive. See example 1 and example 2 below.
12-13	IOPL	I/O Privilege Level Flag	It determines I/O Privilege Level of the current process.
14	NT	Nested Task Flag	It controls chaining of interrupts. Set if the current process is linked to the next process.
16	RF	Resume Flag	Response to debug exceptions.
17	VM	Virtual-8086 Mode Flag	Set if in 8086 compatibility mode.
18	AC	Alignment Check Flag	Set if alignment checking of memory references is done.
19	VIF	Virtual Interrupt Flag	Virtual image of IF (Interruption Flag).

20	VIP	Virtual Interrupt Pending Flag	Set if an interrupt is pending.
21	ID	Identification Flag	Support for CPUID instruction if can be set.

These above flags can be divided into 2 categories:

- Control Flags
- Status Flags

2.6.1 Control Flags

Control flags control the CPU's operation. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode.

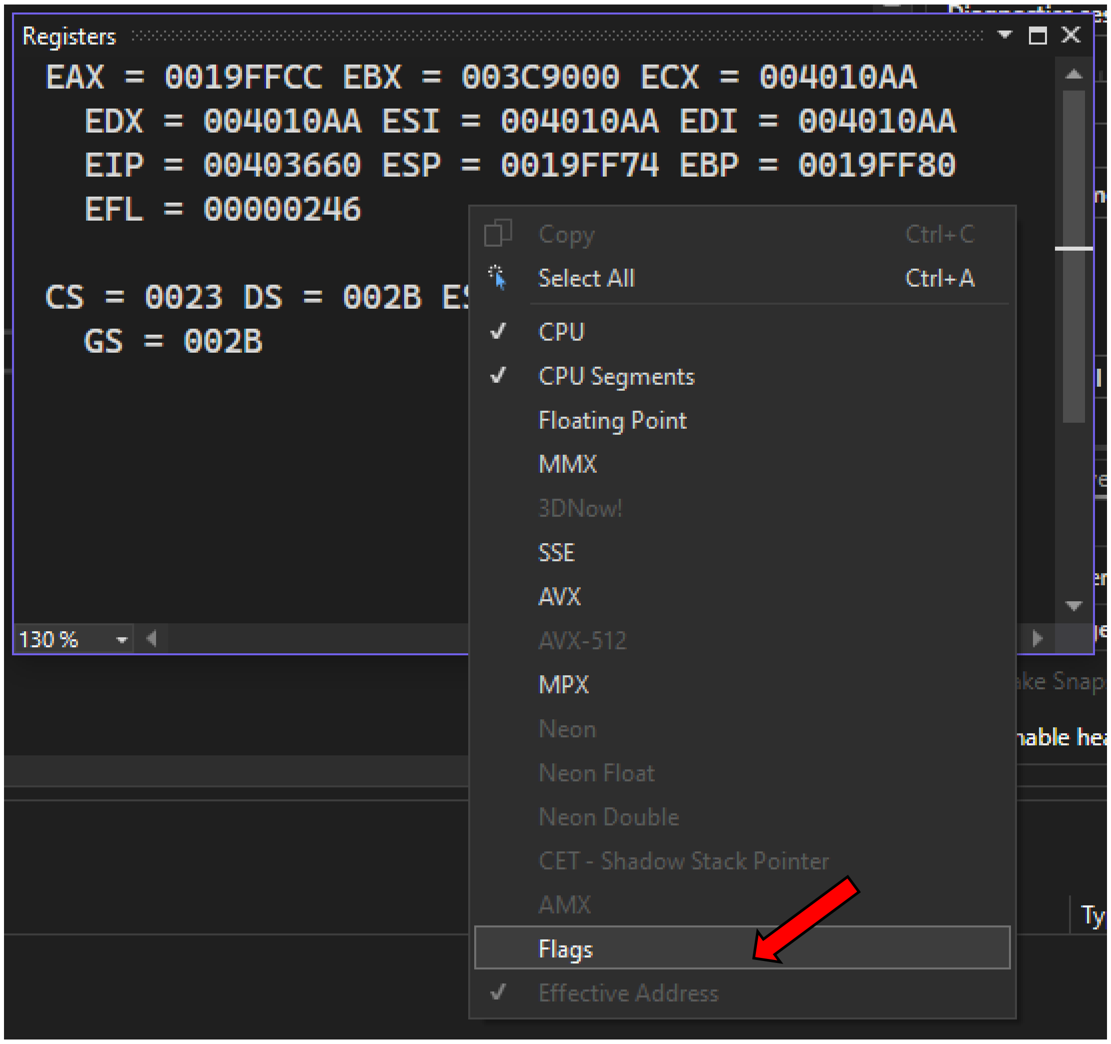
Programs can set individual bits in the EFLAGS register to control the CPU's operation. Examples are the *Direction* and *Interrupt* flags.

2.6.2 Status Flags

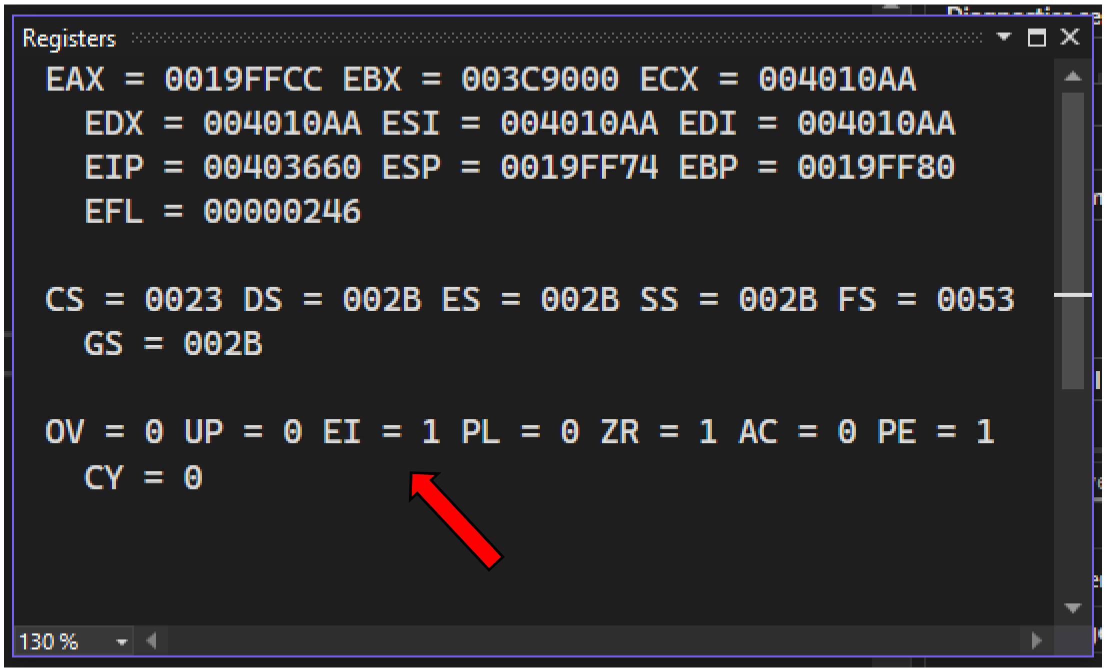
The status flags reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags.

2.7 Examples of EFLAGS Register

First, we enable the Flags register in the Registers window in Visual Studio. While debugging right click in the Registers window and click on the "Flags" option. It will start showing the EFLAGS Register values in the same Registers window.



```
EAX = 0019FFCC EBX = 003C9000 ECX = 004010AA
  EDX = 004010AA ESI = 004010AA EDI = 004010AA
  EIP = 00403660 ESP = 0019FF74 EBP = 0019FF80
  EFL = 00000246

CS = 0023 DS = 002B ES
  GS = 002B
```



```
EAX = 0019FFCC EBX = 003C9000 ECX = 004010AA
  EDX = 004010AA ESI = 004010AA EDI = 004010AA
  EIP = 00403660 ESP = 0019FF74 EBP = 0019FF80
  EFL = 00000246

CS = 0023 DS = 002B ES = 002B SS = 002B FS = 0053
  GS = 002B

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1
  CY = 0
```

The names of the Flags registers showing in Visual Studio are somewhat different from we read earlier:

Flag	Names in Book	Name Showing in Visual Studio
Overflow	OF	OV
Direction	DF	UP
Interrupt	IF	EI
Sign	SF	PL
Zero	ZF	ZR
Auxiliary Carry	AF	AC
Parity	PF	PE
Carry	CF	CY

Example 1

```
Include Irvine32.inc
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.code
main proc
      mov AL,-128
      add AL,-10                ; Overflow occurs

      call ReadInt              ; wait
      invoke ExitProcess,0
main endp
end main
```

Example 2

```
mov AL,0FFh                 ; adding 255 to AL
add AL,1                    ; adding another 1 will cause overflow
```

Example 3

If AX contain the value ffffh (65535) and BX contain the value fffeh (65534) and execute the instruction:

```
mov AX,0FFFFh       ; MSB 0 added to process it as +ive number
mov BX,0FFFEh
add AX,BX
```

$$FFFFh + FFFEh = 1FFFCh$$

$$AX = FFFFh = 1111\ 1111\ 1111\ 1111$$

$$BX = FFFEh = 1111\ 1111\ 1111\ 1101$$

$$AX = FFFCh = 1111\ 1111\ 1111\ 1100$$

Flags value for this code are:

- CF = 1 because there is carry from the MSB.
- PF = 1 because the low byte contain 14th ones (even number of ones).
- ZF = 0 because result is not equal zero.
- SF = 1 because the MSB equal 1.
- OF = 0 because result has the same sign.

Example 4:

If AL contain the value 80h (128) and BL also contain the value 80h (128) and execute the instruction:

```
mov AL,080h         ; MSB 0 added to process it as +ive number
mov AL,080h
add AL,BL
```

$$80h\ +\ 80h = 100h$$

$$AL = 00h = 0000\ 0000$$

Flags value for this code are:

- CF = 1 because there is carry from the MSB.
- PF = 1 because the low byte contain 0 ones (even number of ones).
- ZF = 1 because result is equal zero.
- SF = 0 because the MSB equal 0.
- OF = 1 because result has the different sign.

Example Program

The program shown below implements various arithmetic expressions using the ADD, SUB, INC, DEC, and NEG instructions, and shows how certain status flags are affected:

```
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.data
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 40

.code
main PROC

      ; INC and DEC
      mov ax,1000h
      inc ax                    ;1001h
      dec ax                    ;1000h

      ; Expression: Rval = -Xval + (Yval - Zval)
      mov eax,Xval
      neg eax                         ;-26
```

```
        mov ebx,Yval
        sub ebx,Zval        ;-10
        add eax,ebx
        mov Rval,eax        ;-36

        ; Zero flag example:
        mov cx,1
        sub cx,1                  ; ZF = 1
        mov ax,0FFFFh
        inc ax                    ; ZF = 1

        ; Sign flag example:
        mov cx,0
        sub cx,1                  ; SF = 1
        mov ax,7FFFh
        add ax,2                  ; SF = 1

        ; Carry flag example:
        mov al,0FFh
        add al,1                  ; CF = 1, AL = 00

        ; Overflow flag example:
        mov al,+127
        add al,1                  ; OF = 1
        mov al,-128
        sub al,1                  ; OF = 1

        INVOKE ExitProcess,0
main ENDP
END main
```

Note: For more details and examples regarding flags register, see book's page numbers 105 to 110.

2.8 MMX Registers

MMX is officially a meaningless trademarked by Intel. Unofficially, the initials have been variously explained as standing for

- MultiMedia eXtension
- Multiple Math eXtension
- Matrix Math eXtension

MX defines eight processor registers, named MM0 through MM7, and operations that operate on them. Each register is 64 bits wide and can be used to hold 64-bit integers.

MMX technology improves the performance of Intel processors when implementing advanced multimedia and communications applications. The eight 64-bit MMX registers support special instructions called SIMD (Single-Instruction, Multiple-Data). As the name implies, MMX instructions operate in parallel on the data values contained in MMX registers. Although they appear to be separate registers, the MMX register names are in fact aliases to the same registers used by the floating-point unit.

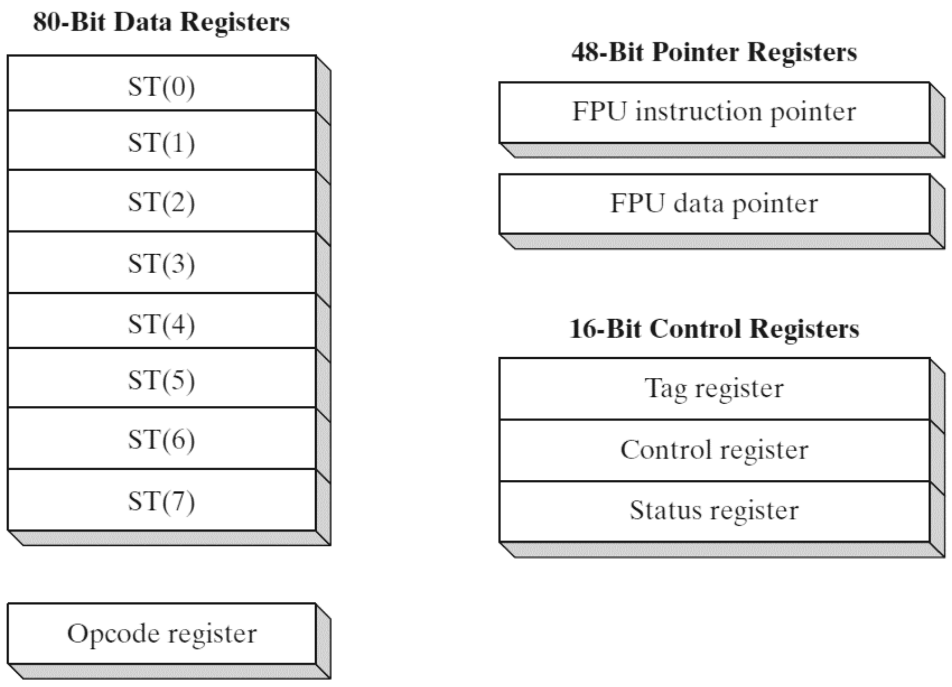
2.9 XMM Registers

The 128-bit XMM registers are part of the SSE extension (where SSE is short for Streaming SIMD Extension, and SIMD, in turn, stands for *single instruction multiple data*). There are eight XMM registers available in non 64-bit modes and 16 XMM registers in *long mode*, which allow simultaneous operations on 16 bytes.

In the x86-64 computer architecture, long mode is the mode where a 64-bit operating system can access 64-bit instructions and registers. 64-bit programs are run in a sub-mode called 64-bit mode, while 32-bit programs and 16-bit protected mode programs are executed in a sub-mode called compatibility mode. Real mode or virtual 8086 mode programs cannot be natively run in long mode.

2.9.1 Floating-Point Unit

The floating-point unit (FPU) performs high-speed floating-point arithmetic. At one time a separate coprocessor chip was required for this. From the Intel486 onward, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, named ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7). The remaining control and pointer registers are shown in the figure below.


80-Bit Data Registers
ST(0)
ST(1)
ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)
Opcode register
48-Bit Pointer Registers
FPU instruction pointer
FPU data pointer
16-Bit Control Registers
Tag register
Control register
Status register