EE 213 Computer Organization and Assembly Language

Week # 3, Lecture # 8

2nd Muharram ul Haram, 1440 A.H

12th September 2018

Minds open...

... Laptops closed

This presentation helps in delivering the lecture.
Take notes, interact and read text book to learn and gain knowledge.

Today's Topics

- Register Recap
- Specialized uses of Registers
- Mode of operations
  - Protected mode
  - Virtual 8086 mode
  - Real Address mode
- Segment Registers
- Assembly Programming
  - Irvine template code
  - Declaring data in Assembly

Please read chapter # 2 (2.1 and 2.2) over the weekend.

FIGURE 2–3 Basic program execution registers.

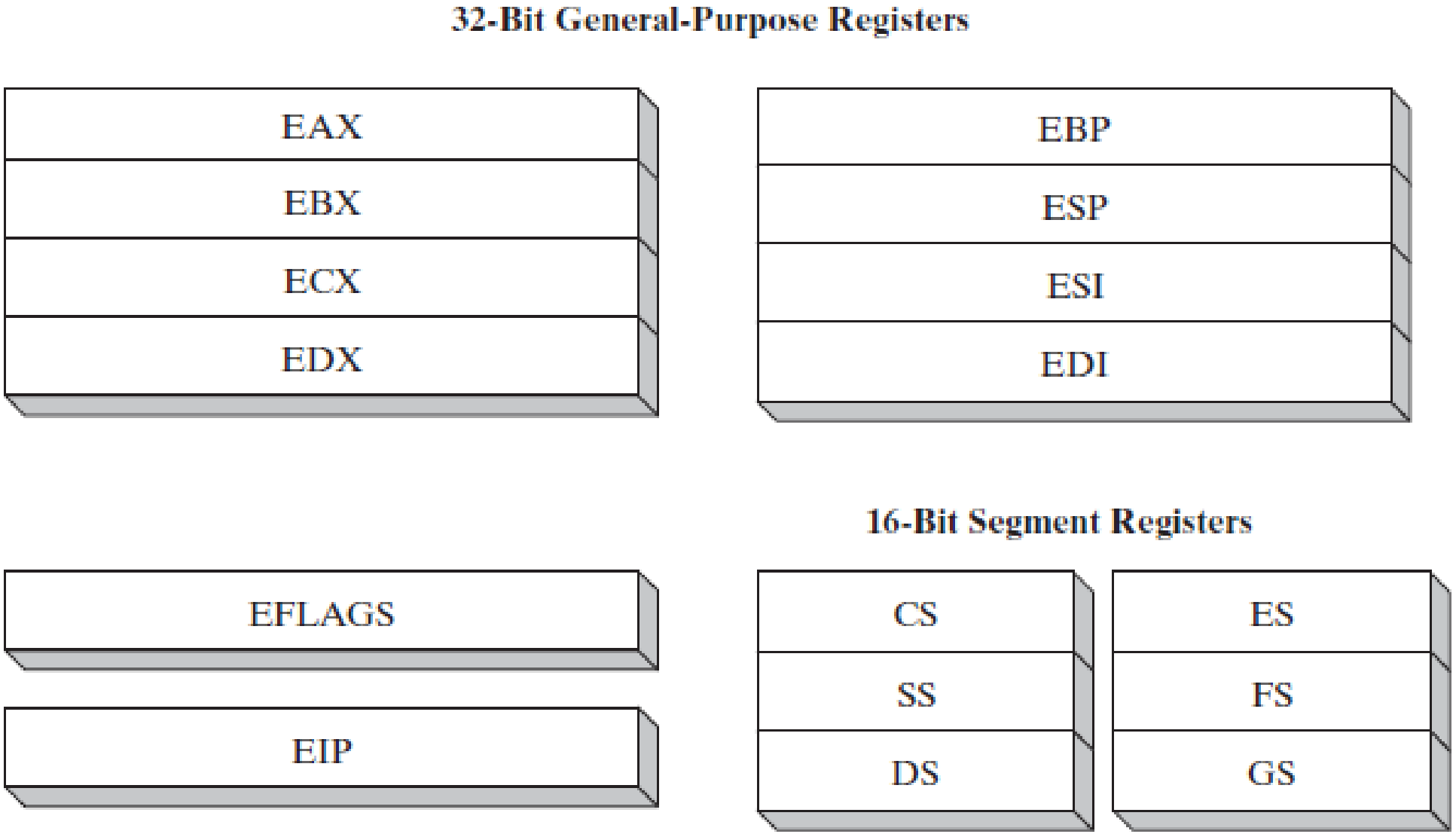


Table 2-1 Operand Sizes in 64-Bit Mode When REX Is Enabled.

Operand Size	Available Registers
8 bits	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
16 bits	AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32 bits	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64 bits	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

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.

32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

2.2 32-Bit x86 Processors

In this section, we focus on the basic architectural features of all x86 processors. This includes members of the Intel IA-32 family as well as all 32-bit AMD processors.

2.2.1 Modes of Operation

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:

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.

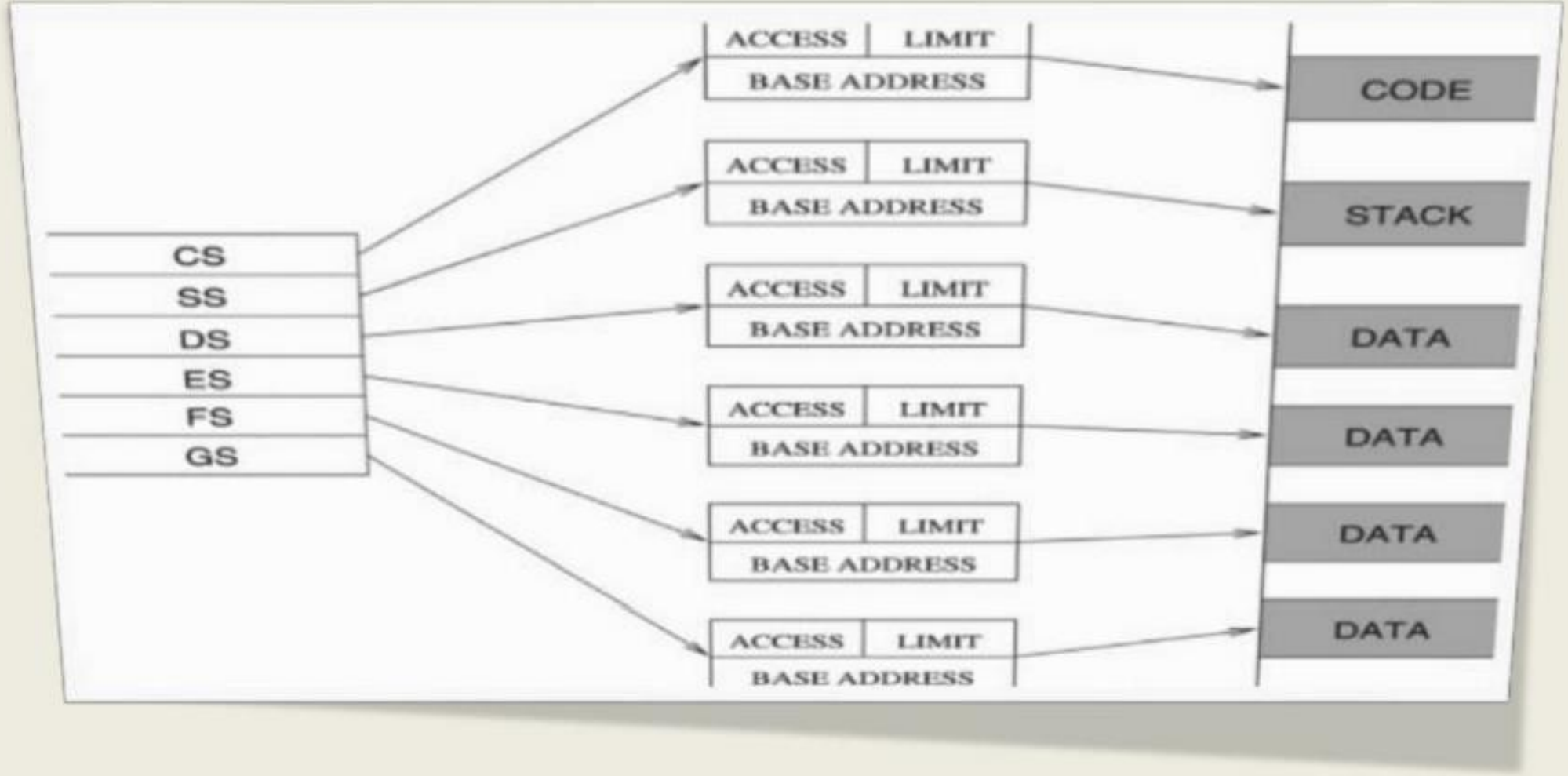
Virtual-8086 Mode While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe environment. In other words, if a program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time.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

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.

In *protected* mode, the processor can run multiple programs at the same time. It assigns each process (running program) a total of 4 GByte of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidentally accessing each other's code and data. MS-Windows and Linux run in protected mode.

In *virtual-8086* mode, the computer runs in protected mode and creates a virtual-8086 machine with its own 1-MByte address space that simulates an 80x86 computer running in real-address mode. Windows NT and 2000, for example, create a virtual-8086 machine when you open a *Command* window. You can run many such windows at the same time, and each is protected from the actions of the others. Some MS-DOS programs that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP.

In *real-address* mode, only 1 MByte of memory can be addressed, from hexadecimal 00000 to FFFFF. The processor can run only one program at a time, but it can momentarily interrupt that program to process requests (called *interrupts*) from peripherals. Application programs are permitted to access any memory location, including addresses that are linked directly to system hardware. The MS-DOS operating system runs in real-address mode, and Windows 95 and 98 can be booted into this mode.



Segment Registers

"Segmentation provides a mechanism for dividing the processor's addressable memory space (called the **linear address space**) into smaller protected address spaces called **segments**

Segment Registers In real-address mode, 16-bit segment registers indicate base addresses of preassigned 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.

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.

```
.486                                    ; create 32 bit code
.model flat, stdcall                    ; 32 bit memory model
option casemap:none                     ; case sensitive

; -----------------------------------------------
; Irvine library
; -----------------------------------------------

include     \Irvine\Irvine32.inc
includelib  \Irvine\Irvine32.lib
includelib  \Irvine\kernel32.lib
includelib  \Irvine\user32.lib

.data
   val1     dword  10000h
   val2     dword  40000h
   val3     dword  20000h
   finalVal dword  ?

.code
main PROC
      mov       eax,val1              ; start with 10000h
      add       eax,val2              ; add 40000h
      sub       eax,val3              ; subtract 20000h
      mov       finalVal,eax          ; store the result (30000h)
      call      DumpRegs              ;
      call      WaitMsg               ; wait for a keypress
      exit
main ENDP
END main
```

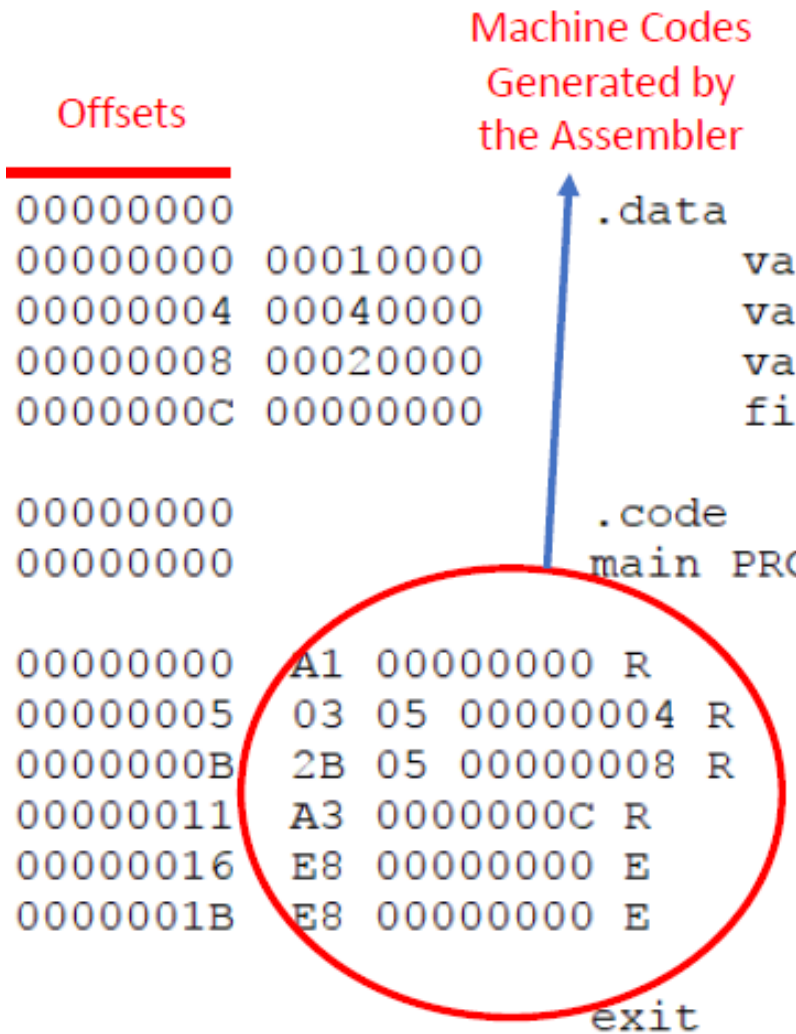
Include file

Library files

Functions from Irvine book library

Specific the name of startup function

Listing File (.lst)

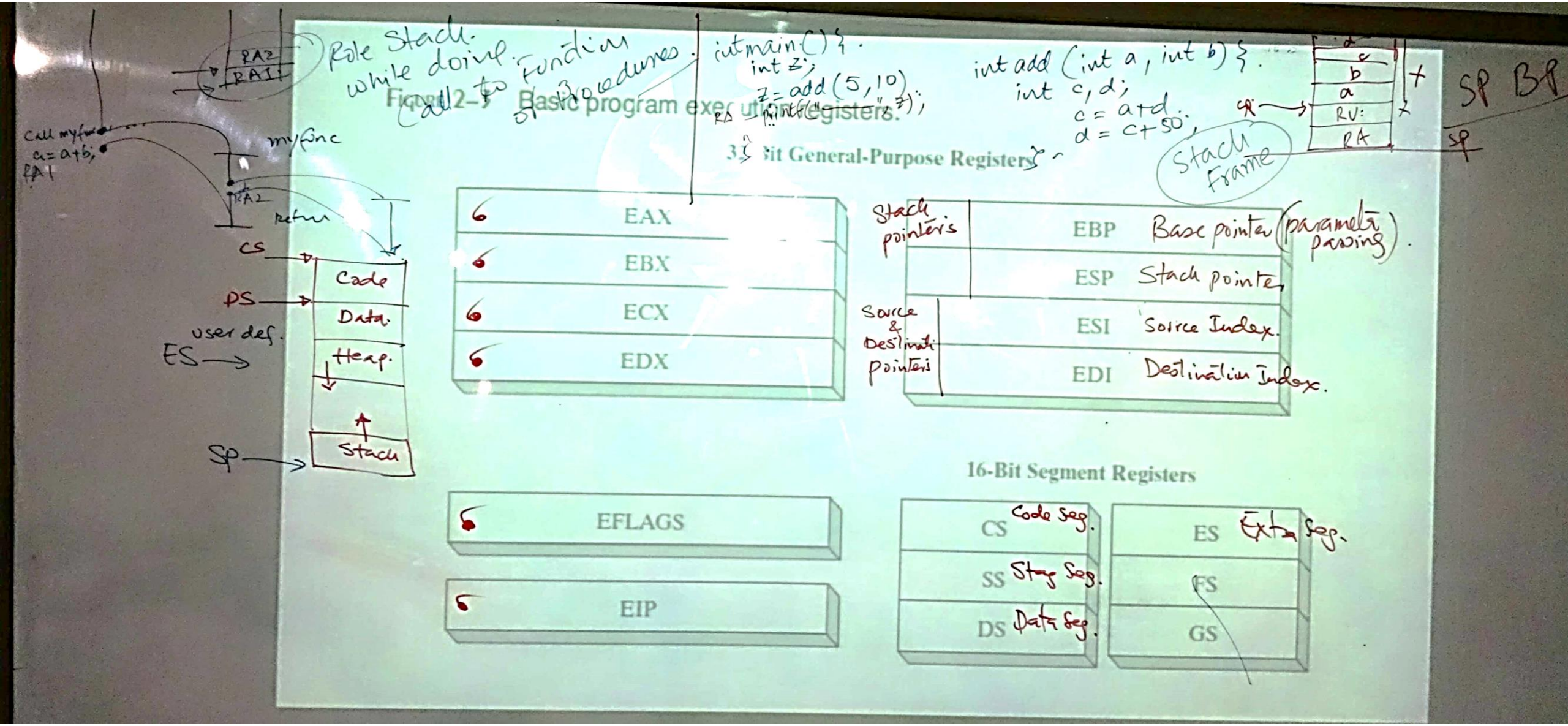



```
00000000                .data
00000000 00010000       val1      dword  10000h
00000004 00040000       val2      dword  40000h
00000008 00020000       val3      dword  20000h
0000000C 00000000       finalVal  dword  ?

00000000                .code
00000000                main PROC

00000000  A1 00000000 R        mov   eax,val1          ; start with 10000h
00000005  03 05 00000004 R     add   eax,val2          ; add 40000h
0000000B  2B 05 00000008 R     sub   eax,val3          ; subtract 20000h
00000011  A3 0000000C R        mov   finalVal,eax      ; store the result (30000h)
00000016  E8 00000000 E         call DumpRegs             ; ldsjfldsjfs
0000001B  E8 00000000 E         call  WaitMsg             ; wait for a keypress

                        exit
00000027                main ENDP
                        END main
```


Role Stack.
while doing function call to Procedures.
Figure 2–5 Basic program execution registers.
32-Bit General-Purpose Registers
EAX
EBX
ECX
EDX
EFLAGS
EIP
16-Bit Segment Registers
CS
SS
DS
ES
FS
GS
Stack pointers
EBP Base pointer (parameter passing)
ESP Stack pointer
Source & Destination pointers
ESI Source Index.
EDI Destination Index.
Code seg.
Stack Seg.
Data Seg.
Extra Seg.
int main(){
int z;
z = add(5,10);
printf("%d", z);
int add (int a, int b){
int c, d;
c = a+d;
d = c+50;
Stack Frame
SP BP
sp
RA2
RA1
call myfunc
c = a+b;
myfunc
return
CS
DS
user def.
ES
Code
Data
Heap
Stack
Sp

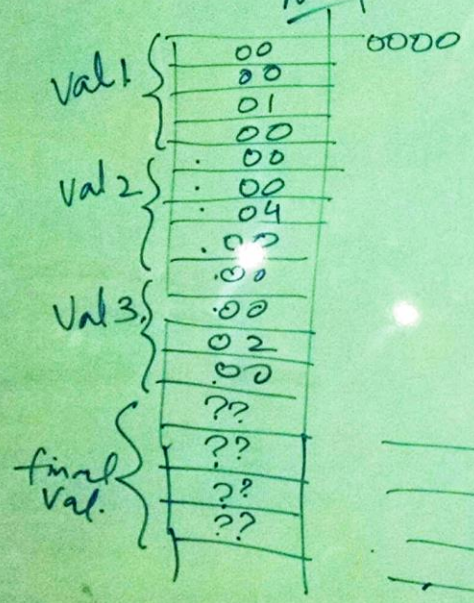
Assembler Directives.

```
.486   .686
.model flat, stdcall
option casemap:none
```

```
; create 32 bit code
; 32 bit memory model
; case sensitive
```

0000:10000h.

Memory Map.




```
; ----------------------------------------------
; Irvine library
; ----------------------------------------------

include      \Irvine\Irvine32.inc
includelib   \Irvine\Irvine32.lib
includelib   \Irvine\...2.lib
includelib   \Irvine\...32.lib
```

Include file

Library files

byte
word
double word

```
.data
val1      .dword  10000h
val2      dword   40000h
val3      dword   20000h
finalVal  dword   ?
```

Specific the name of startup function

```
.code
main PROC
        mov     eax,val1
        add     eax,val2
        sub     eax,val3
        mov     finalVal,eax
        call    DumpRegs
        call    WaitMsg
        exit
main ENDP
END main
```

Chapter #5
Functions from Irvine book library

```
; start with 10000h
; add 40000h
; subtract 20000h
; store the result (30000h)
;
; wait for a keypress
```