Introduction to IA-32 (Intel Architecture 32-Bit)

CSU34021 - Computer Architecture II

Suggested Readings

- "**Computer Architecture – A quantitative Approach**", *John Hennessey* and *David Patterson.* [One of the appendix sections depending on the edition you have access to.]

- "**Overview of IA-32 assembly programming**", *Lars Ailo Bongo.* Accessible at: https://www.cs.umd.edu/~meesh/cmsc311/links/handouts/ia32.pdf.

- If you want to read more: "**Introduction to assembly Language programming**" *Dandamudi*. [1st edition for MASM, 2nd edition for NASM!].

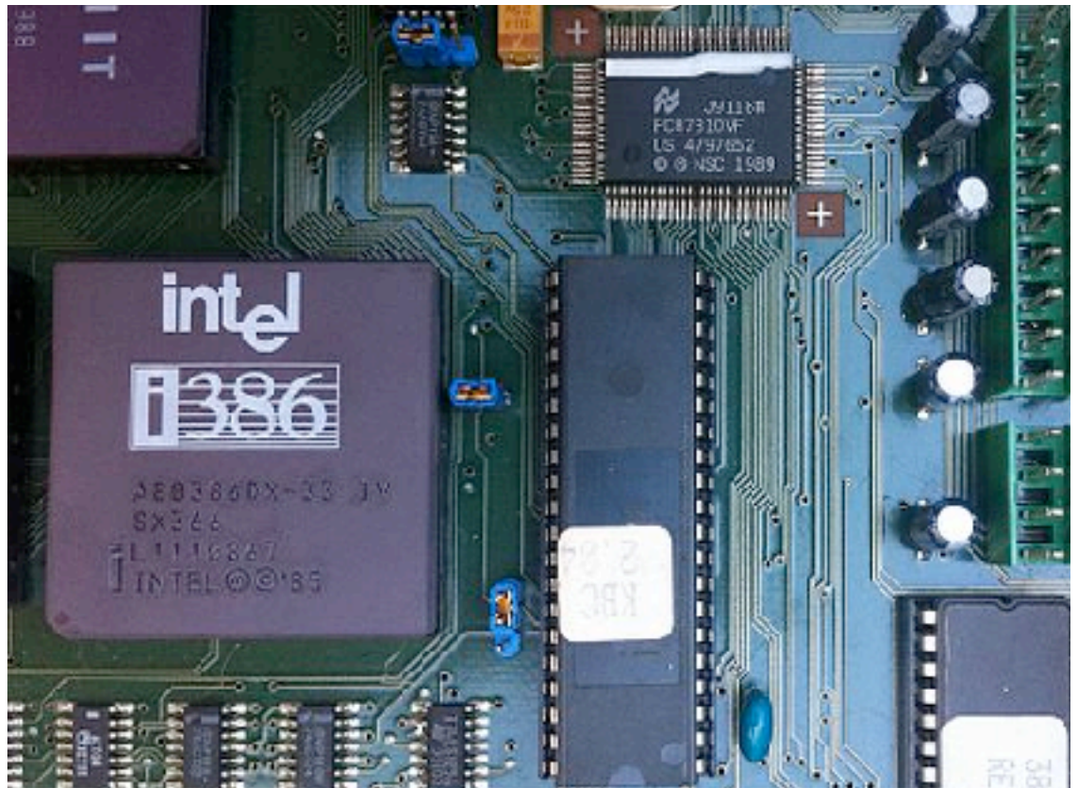
Some Background/History

- First Intel micro-processor (4-bit, 4004 in 1971), later development (8080, 8085).

- Development of IA (8086, 20bit address bus and 16bit data bus) - later: 80186, 80286.

- 80386 (x86 and Pentium family): 4GB address space, paging, on-chip cache and L2 cache.

- x86 one of the most used family of processor.



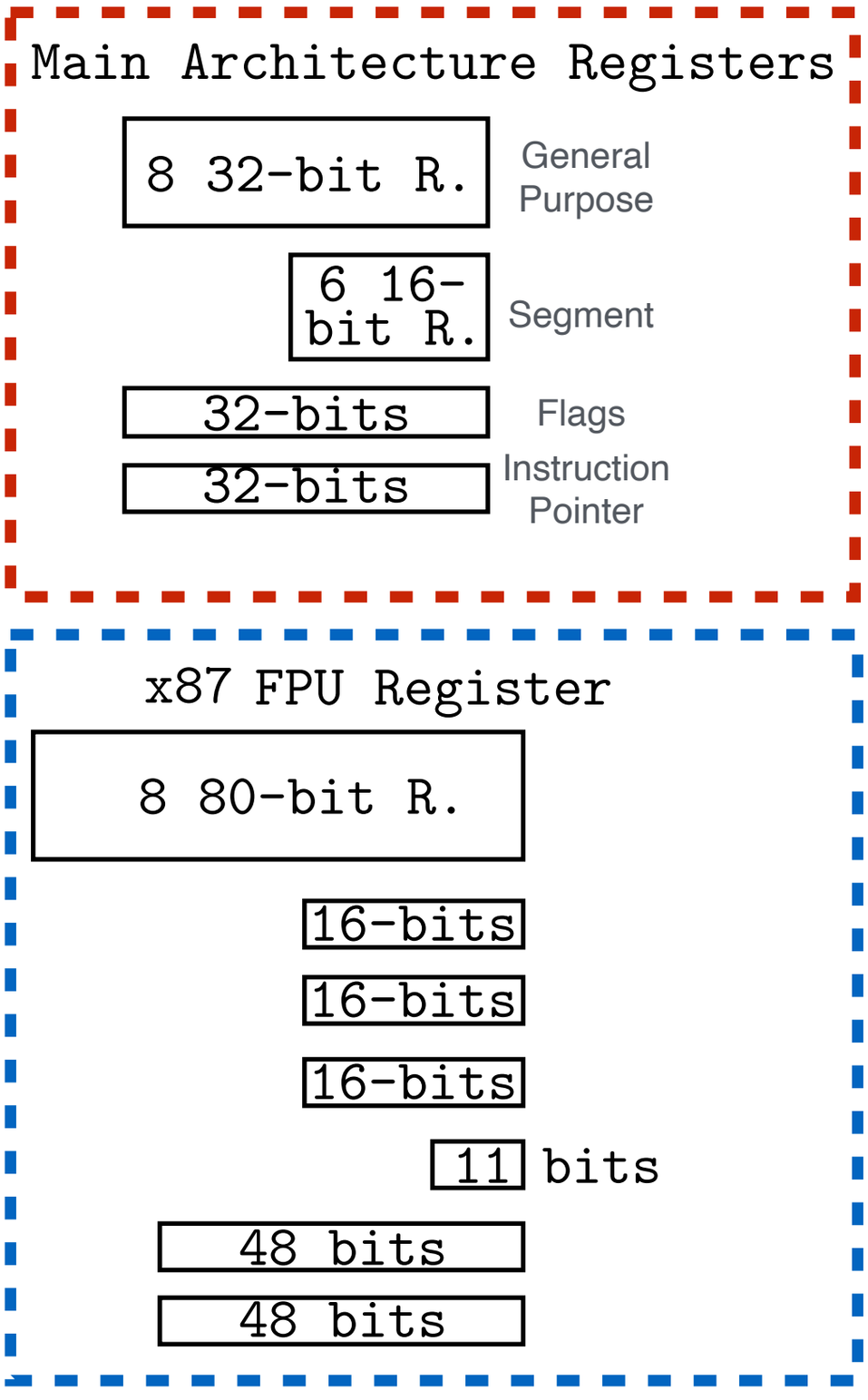
Instruction Set Architecture

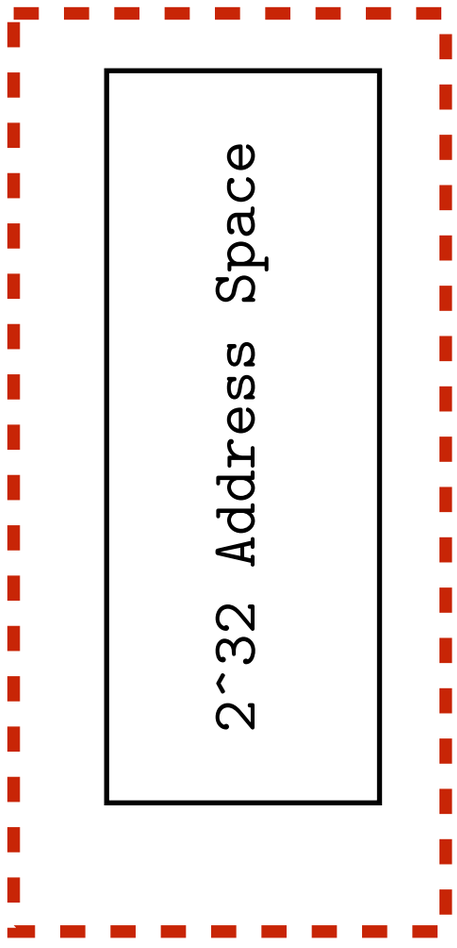
- Instruction Set Architecture: interface between software and hardware.

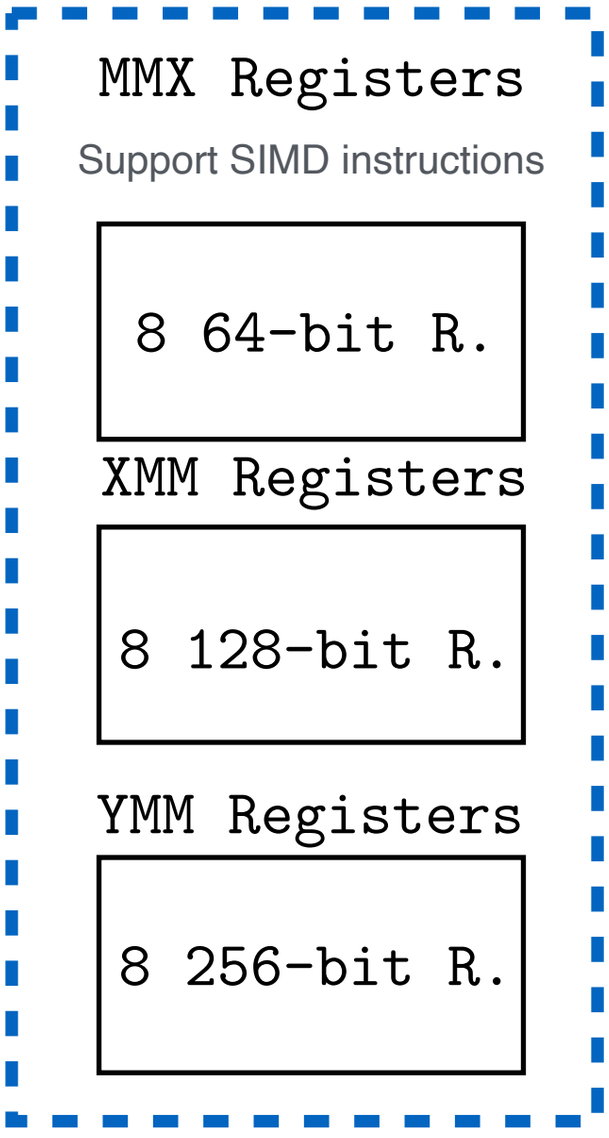
- An ISA is broadly defined by a set of instructions, registers, the data ‘types’, and the methods for accessing main memory.

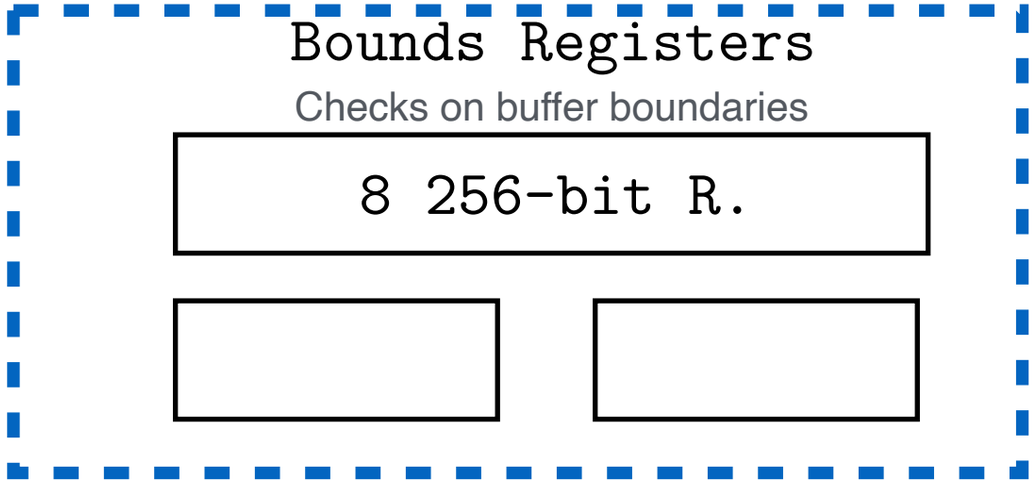
- IA-32 is the ISA, x86 the family of processors that implemented IA-32.

- X86 extended many times. We'll look only at the general purpose features.
- Later on we'll look at the rest (high level).









IA-32 and Assembly

- It's a **CISC** architecture with **32-Bit CPU**, supports **4GB of physical address space**. Supports **8, 16, 32 bit integer** operations and **32 and 64 floating point** operations.

- Introduction to the **architecture** and **assembly language**.

MASM (Microsoft Assembler)

IA-32 ISA

Main Architecture Registers

8 32-bit R.

General Purpose Registers

6 16-bit R.

Segment Registers

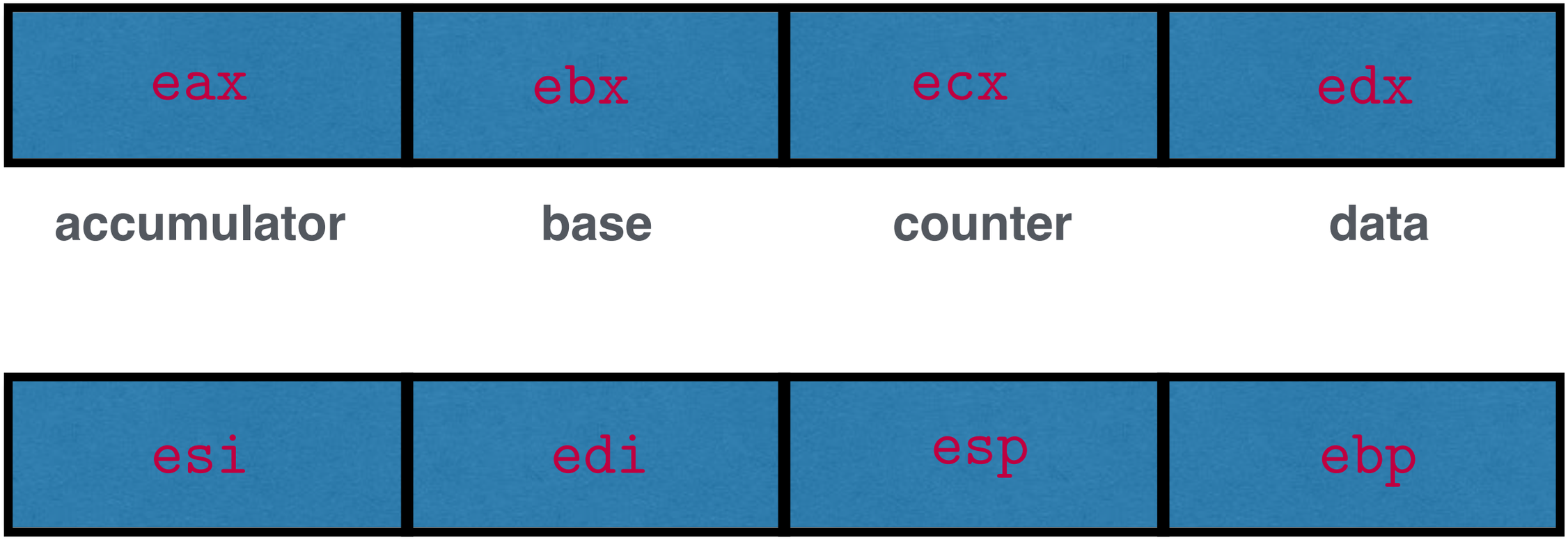
32-bits

Flags Register

32-bits

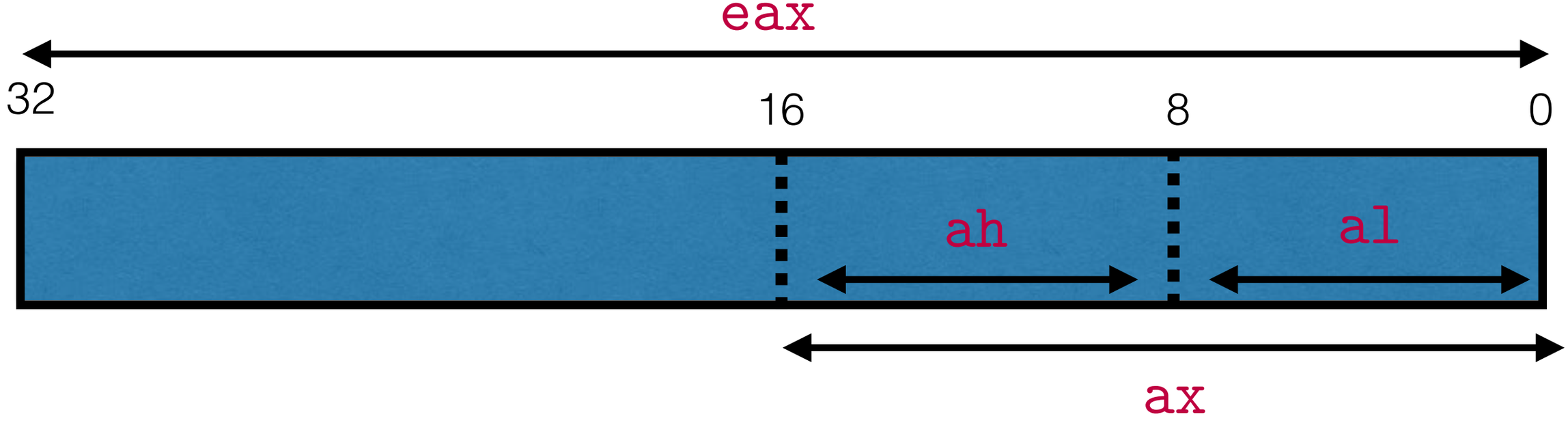
Instruction Pointer Register

8 32-Bit General Purpose Registers



Partial access to the registers

It is possible to access the 8-bit, or 16-bit part of EAX, EBX, ECX and EDX.



Register naming (e = extended, h = high, l = low, x = pair)

Other Registers

- **Six 16-Bit segments – Define segments in memory.**

`CS`	→	Code Segment
`DS`, `ES`, `FS`, `GS`	→	Data Segment(s)
`SS`	→	Stack Segment

- **Two 32-Bit Control Registers**

`(E)IP`	→	(Extended) Instruction Pointer
`(E)FLAGS`	→	(Extended) Flags

6 status flags that carry information about latest arithmetic operation + 1 Control flag for string operations + 10 System flags (e.g., legacy environment + memory alignment + debugging etc.)

Statement Format

```
[label:]    mnemonic    [operand(s)]    [;comment]
```

- Two-address machine, in MASM the destination is the first operand [NB: it's different in NASM!]

```
add     eax, ebx     ; eax = eax + ebx
sub     eax, ebx     ; eax = eax - ebx
dec     ecx          ; ecx = ecx - 1
```

Data Allocation

```
[variable-name] define-directive initial-value [, initial-value1] [, ...]
```

- Notice there is no column after the variable name.

- The define directive specifies the amount of space required (**Remember: No type!**):

```
        DB    [Define Byte]             ; 1 Byte
        DW    [Define Word]             ; 2 Bytes
        DD    [Define Doubleword]       ; 4 Bytes
        DQ    [Define Quadword]         ; 8 Bytes
```

- For example:

```
an_int DD     4                ; associate offset in the SS to an_int
       sub    eax, eax         ; initialise eax to 0
       add    eax, an_int      ; eax = an_int
```

Operands

- Operands need to be:

```
register, register
register, immediate
register, memory
memory, register
```

NB: No such a thing as Memory-Memory or Memory-Immediate!!!

```
add eax, ebx
add eax, 13
add an_int, eax
```

~~`add an_int, 13`~~

~~`add an_int, another_int`~~

IA-32 Addressing Modes

- Addressing modes of an ISA refers to how instructions operands can be identified by the instruction.
- IA-32 supports several advanced addressing modes

```
mov    eax, ebx      <-- Register Addressing Mode
mov    eax, 123      <-- Immediate Addressing Mode
mov    eax, an_int   <-- Direct Addressing Mode
```

Addressing Continued…

```
array   DD   20      DUP (0)          ; array of 20 int set to 0
```

Addressing Continued…

```
array   DD   20      DUP (0)          ; array of 20 int set to 0
...
        mov ebx,    OFFSET array     ; move start address of array to ebx
```

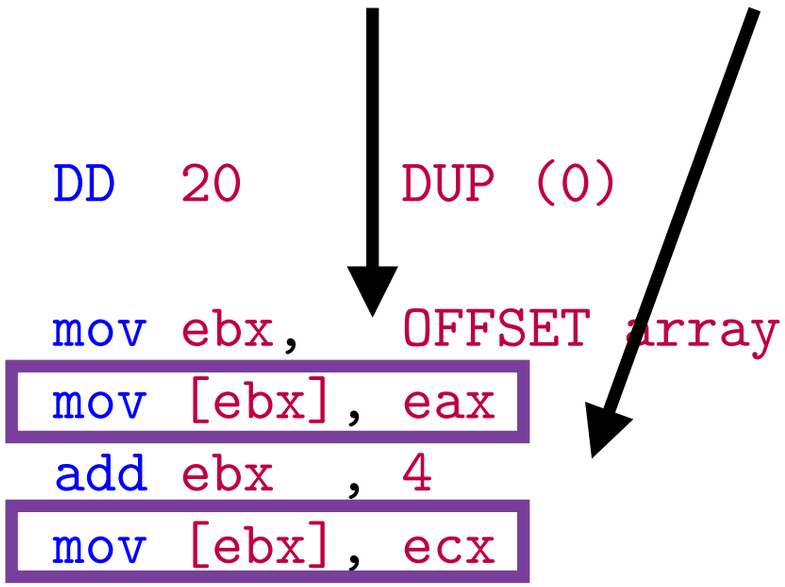
Addressing Continued…

Memory offset of variable “array”

```
array   DD   20      DUP (0)          ; array of 20 int set to 0
  ...
  ...
        mov ebx,    OFFSET array   ; move start address of array to ebx
```

Addressing Continued…

Register Indirect Addressing Mode.



```
array   DD   20     DUP (0)          ; array of 20 int set to 0
 ...
 mov ebx,    OFFSET array   ; move start address of array to ebx
 mov [ebx], eax             ; array[0] = eax
 add ebx   , 4              ; ebx points to array[1]
 mov [ebx], ecx             ; array[1] = ecx
```

Addressing Continued…

```
array   DD   20      DUP (0)          ; array of 20 int set to 0
...
        mov ebx,    OFFSET array     ; move start address of array to ebx
        mov [ebx], eax               ; array[0] = eax
        add ebx   , 4                ; ebx points to array[1]
        mov [ebx], ecx               ; array[1] = ecx
        mov eax, array[esi*4]        ; moves to eax the memory content of array[esi]
```

Scaled Indexed Addressing Mode.

4 Is the size of array element in byte

Addressing Continued…

```
array   DD   20      DUP (0)          ; array of 20 int set to 0
...
        mov ebx,    OFFSET array     ; move start address of array to ebx
        mov [ebx], eax               ; array[0] = eax
        add ebx   , 4                ; ebx points to array[1]
        mov [ebx], ecx               ; array[1] = ecx
        mov eax, array[esi*4]        ; moves to eax the memory content of array[esi]
        mov eax, [ebx+esi*4+2]
```

Scaled Indexed Addressing Mode.

4 Is the size of array element in byte

Based-Indexed Addressing Mode.

IA32 Basic Instructions

```
mov     ; move source into destination
xchg    ; exchange the two arguments
and     ; logical 'and'
or      ; logical 'or'
xor     ; logical 'xor'
neg     ; logical negation
test    ; non-destructive logical 'and' (useful for conditional jumps)
jmp     ; unconditional jump
je, jne, jl, jle, jg, jge, jz, jnz, jc, jnc  ;conditional jumps
push    ; push to stack
pop     ; pop from stack
call    ; call procedure
ret     ; return from procedure
```

Arithmetic Instructions

```
add      ; addition
sub      ; subtraction
inc      ; increment by 1
dec      ; decrement by 1
cmp      ; non-dsetructive subtraction (useful for conditional jumps)
mul      ; multiplication instruction (destination: eax)
imul     ; signed multiplication instruction (destination: eax)
div      ; unsigned integer division (destination: to follow...)
idiv     ; signed integer division (destination: to follow...)
cbw, cwd, cdw ; auxiliary operands: convert B, W and D to W, D and Q
```