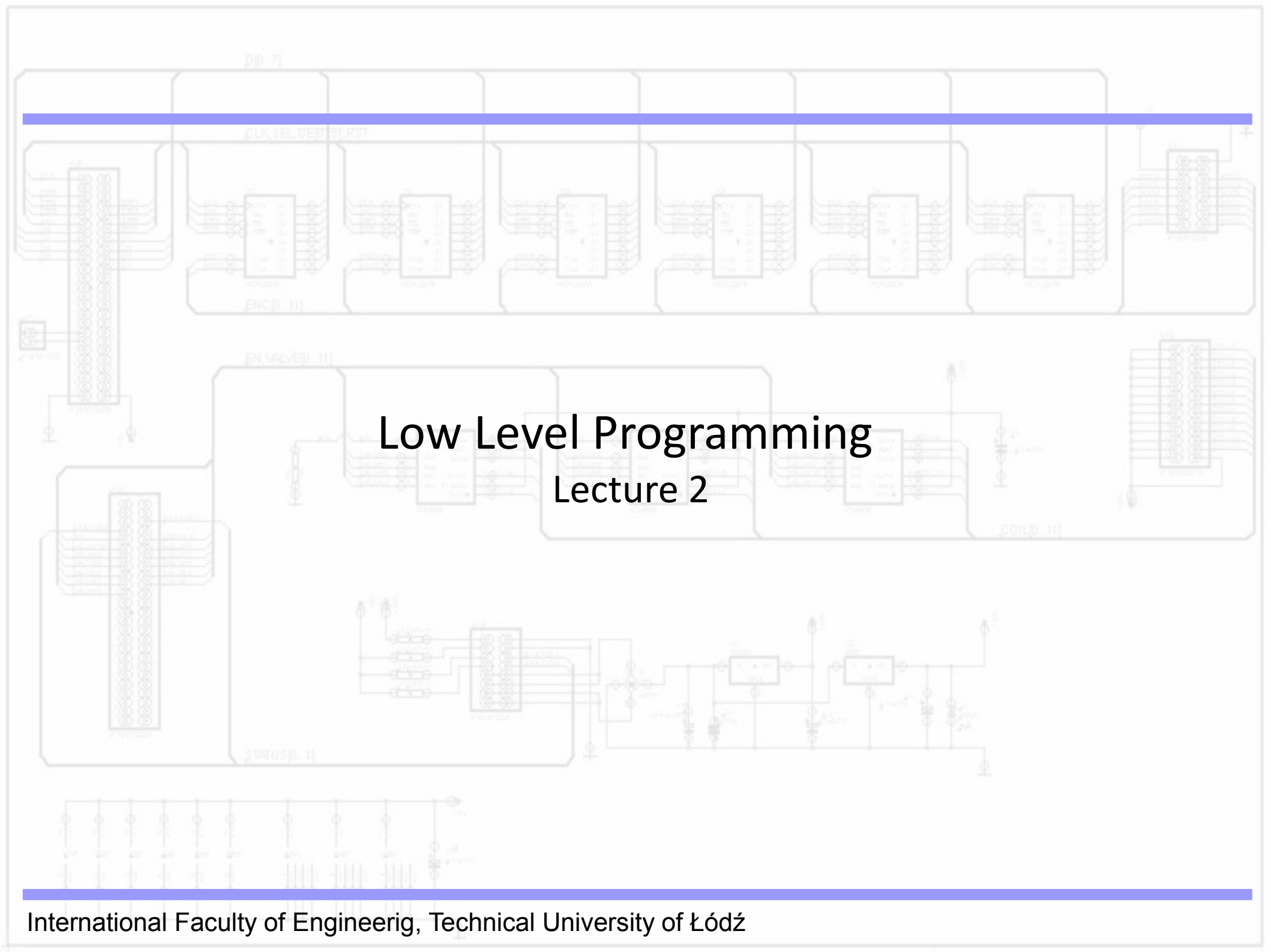


Low Level Programming

Lecture 2

Intel processors' architecture reminder

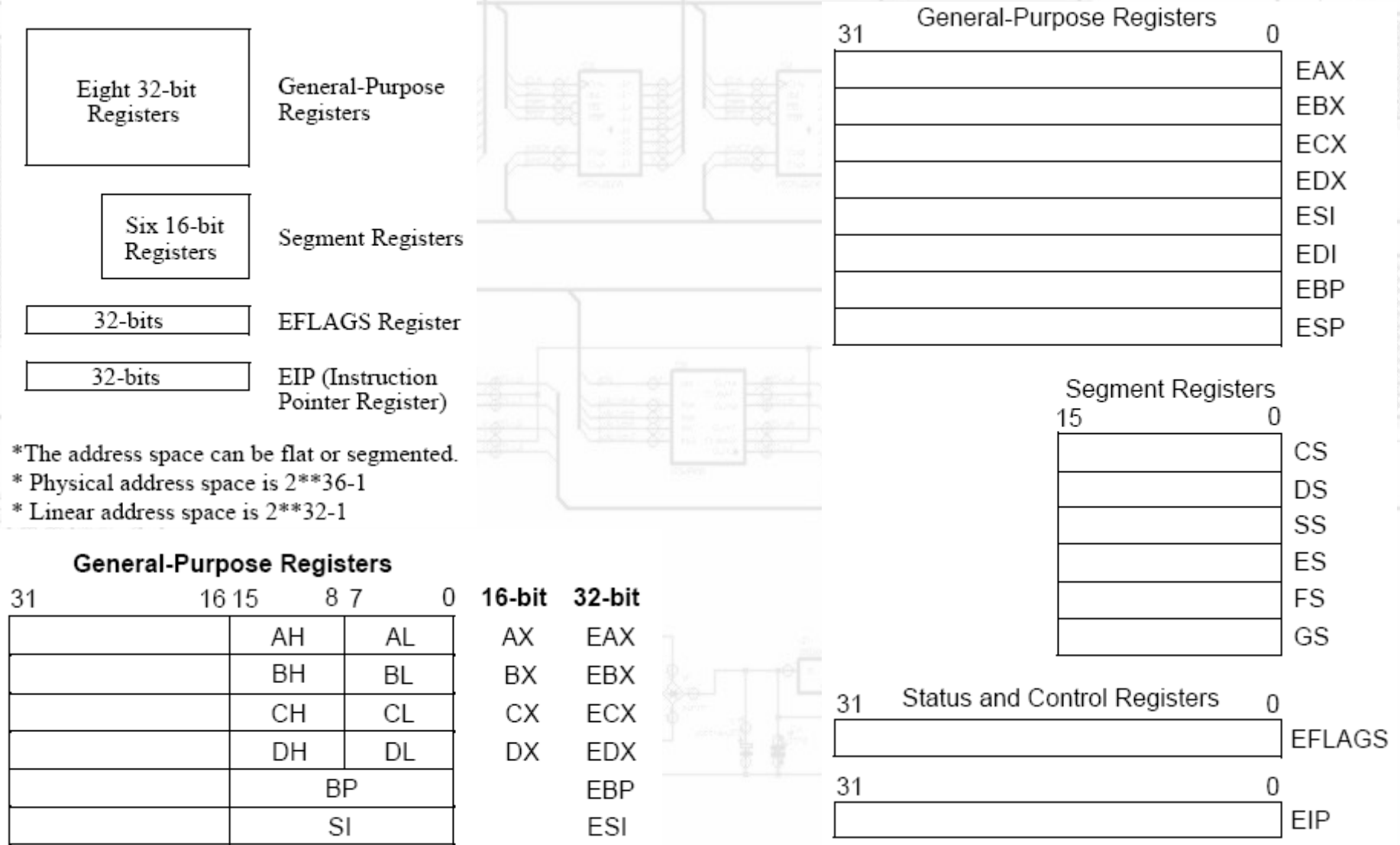


Fig. 1. IA32 Registers

IA general purpose registers

- **EAX**- accumulator, usually used to store results of integer arithmetical or binary operations
- **EBX**- used in memory addressing to hold the base offset,
- **ECX**- as above, also used as a **loop counter**
- **EDX**- as above, also used to store partial results of some 32-bit integer instructions in pair with EAX register
- **ESI** - as above, also used as a **source index** by memory block operations
- **EDI** - as above, also used as a **destination index** by memory block operations
- **ESP** - as above, also it's a processor's **stack pointer**
- **EBP** - as above, also used as a **base pointer** for stack frame operations

IA segment, status and control registers

Segment registers:

- **CS** – code segment register
- **DS** – data segment register
- **SS** – stack segment register
- **ES, FS, GS** – extended (additional) data segment registers

Status registers:

- **EFLAGS** – flag register which stores processor's status and particulary a status of the last arithmetical operation performed by the processor. Flags can be tested and utilizezed for controlling program execution paths.

Control registers:

- **EIP** – processor's instruction pointer used along with the CS register to store next processor's instruction to be executed.

IA system, control, debug and test registers

System registers:

- **GDTR** – global descriptor table register stores the address of a processor's global descriptor table used in protected mode
- **IDTR** – interrupt descriptor table register stores the address of an interrupt descriptor table used in protected mode
- **LDTR** – local descriptor table register stores the address of a current task's local descriptor table used in protected mode
- **TR** – task register stores the current task's state segment pointer

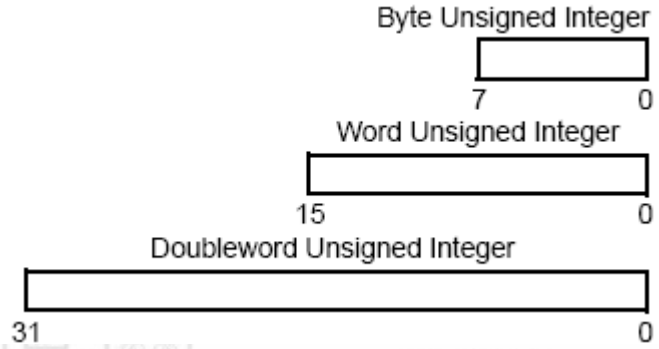
Control registers:

- **CR0-CR3** – processor's control words used to store bit's describing processor's current mode of operation and memory paging mode.

Debug and test registers:

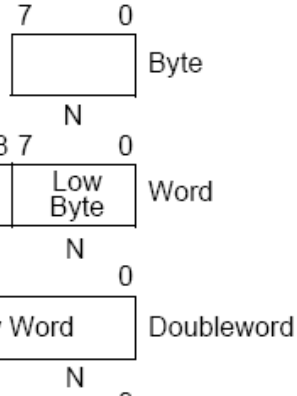
- **DR0 – DR7** - used for debugging purposes for controlling hardware traps
- **TR6 - TR7** - used while testing cosistency of memory descriptors and also while processor's self test

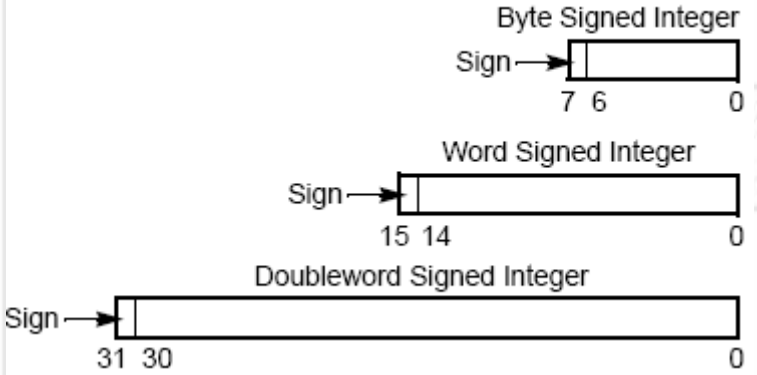
IA integer data types and sizes

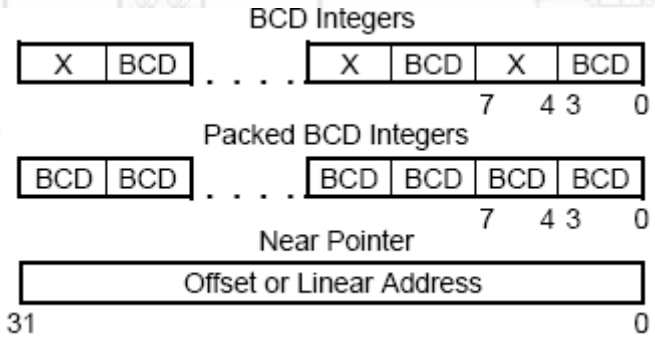


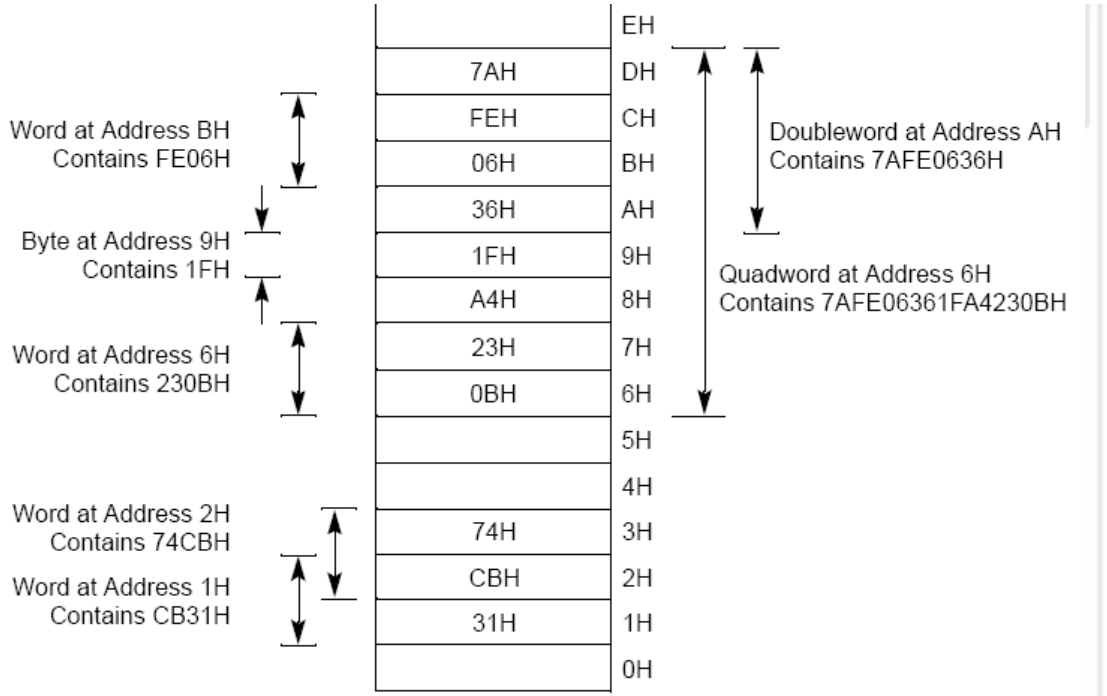
$$U2: \quad (\, a_n\, a_{n-1} \ldots a_1\, a_0\,)_2 = -2^{n-1} \cdot a_n + \sum_{k=0}^{n-1} 2^k \cdot a_k$$

$$BIN: \quad (\, a_n\, a_{n-1} \ldots a_1\, a_0\,)_2 = \sum_{k=0}^{n} 2^k \cdot a_k$$









IA floating point data types and sizes

$$Z = (-1)^{Sign} \cdot 2^{Exponent-127} \cdot \left(1 + \sum_{k=1}^{n} 2^{-k} \cdot a_k\right), \ (a_n\, a_{n-1} \ldots a_1\, a_0)_2 - Significand$$

Note:
Internal FPU registers are 10 bytes long.
In operational memory floating point numbers are stored as 32-bit (single precision), 64-bit (double) or 80-bit (long double) precision numbers with the same format as indicated above.

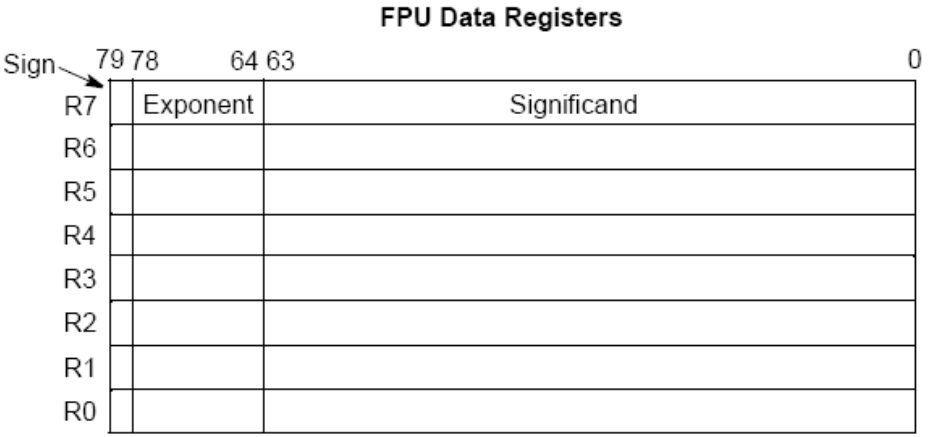
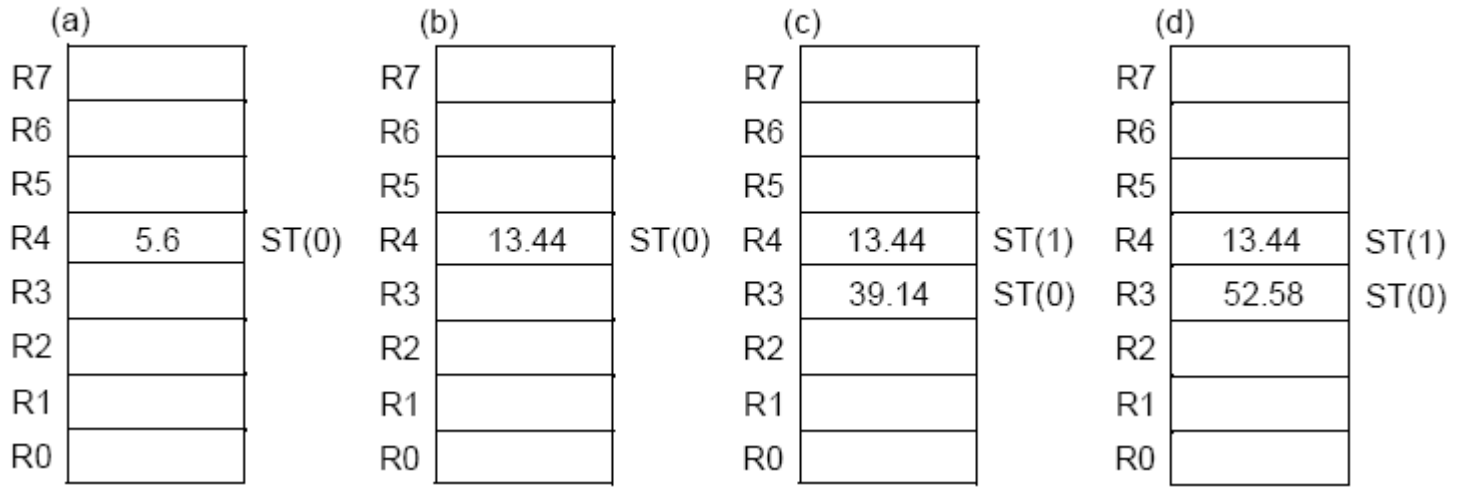


Fig. 2. Example of dot product computation

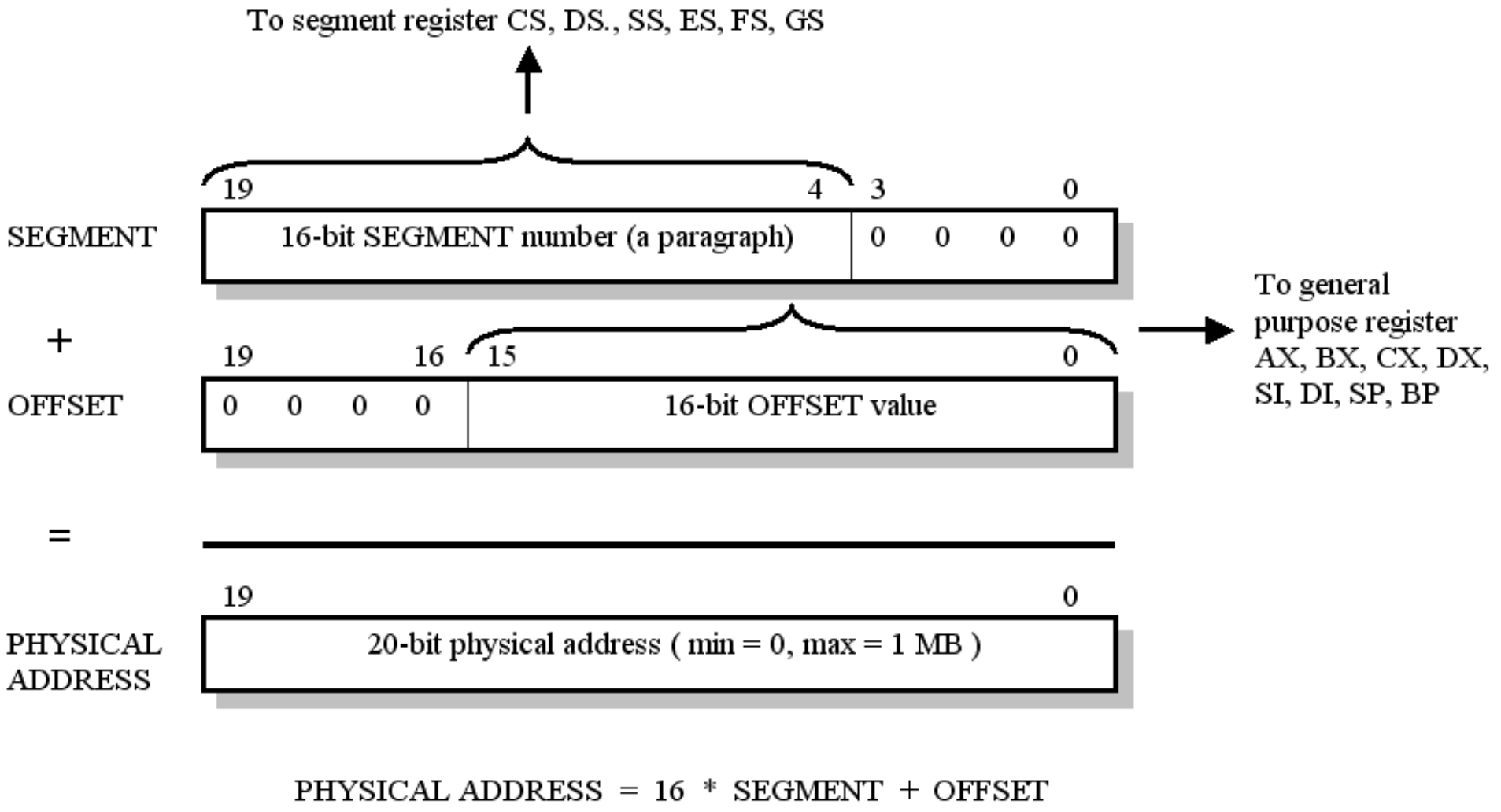
Computation
Dot Product = (5.6 x 2.4) + (3.8 x 10.3)

Code:

```
FLD   value1 ;(a) value1=5.6
FMUL  value2 ;(b) value2=2.4
FLD   value3 ; value3=3.8
FMUL  value4 ;(c)value4=10.3
FADD  ST(1)  ;(d)
```

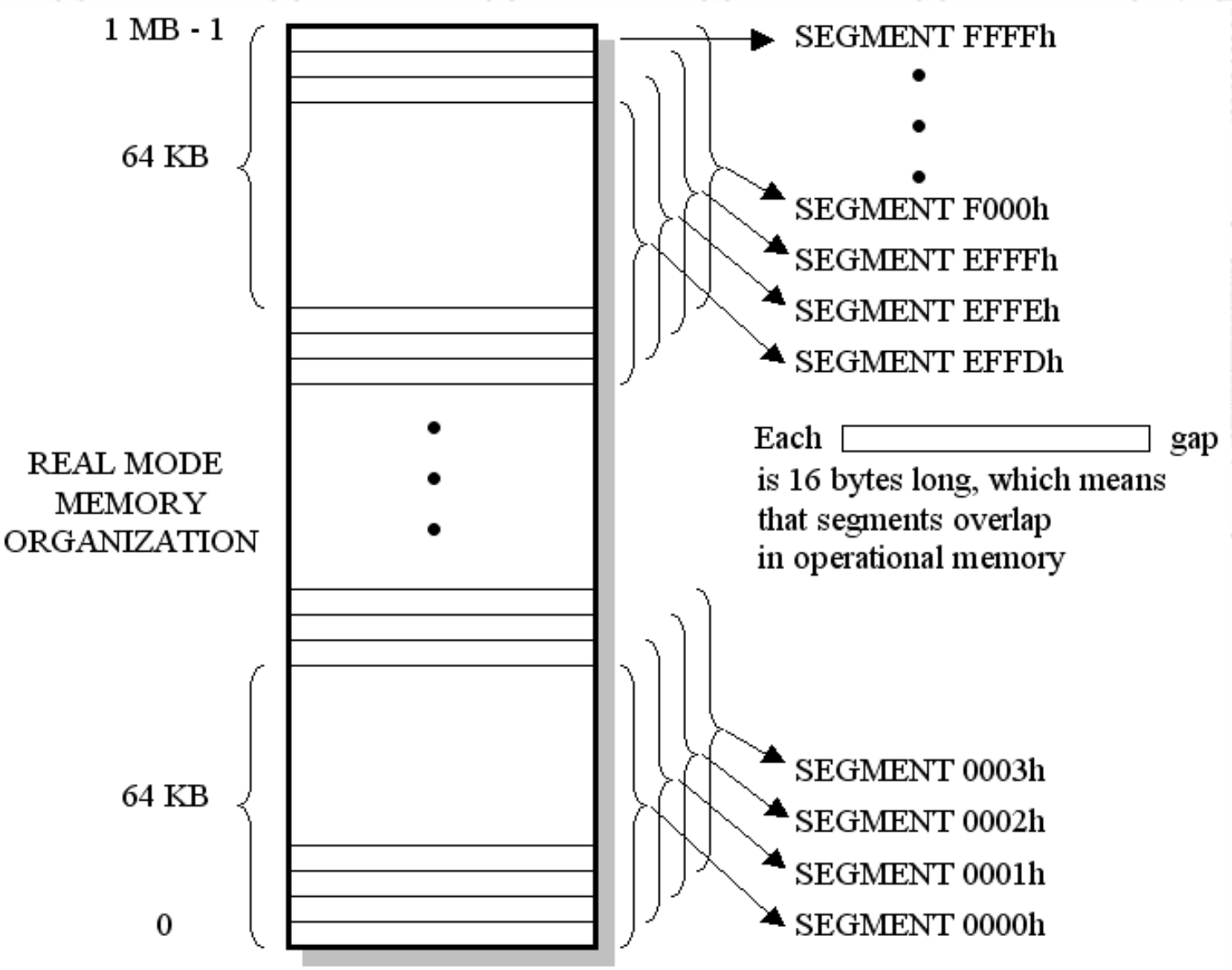


IA Real Mode Addressing

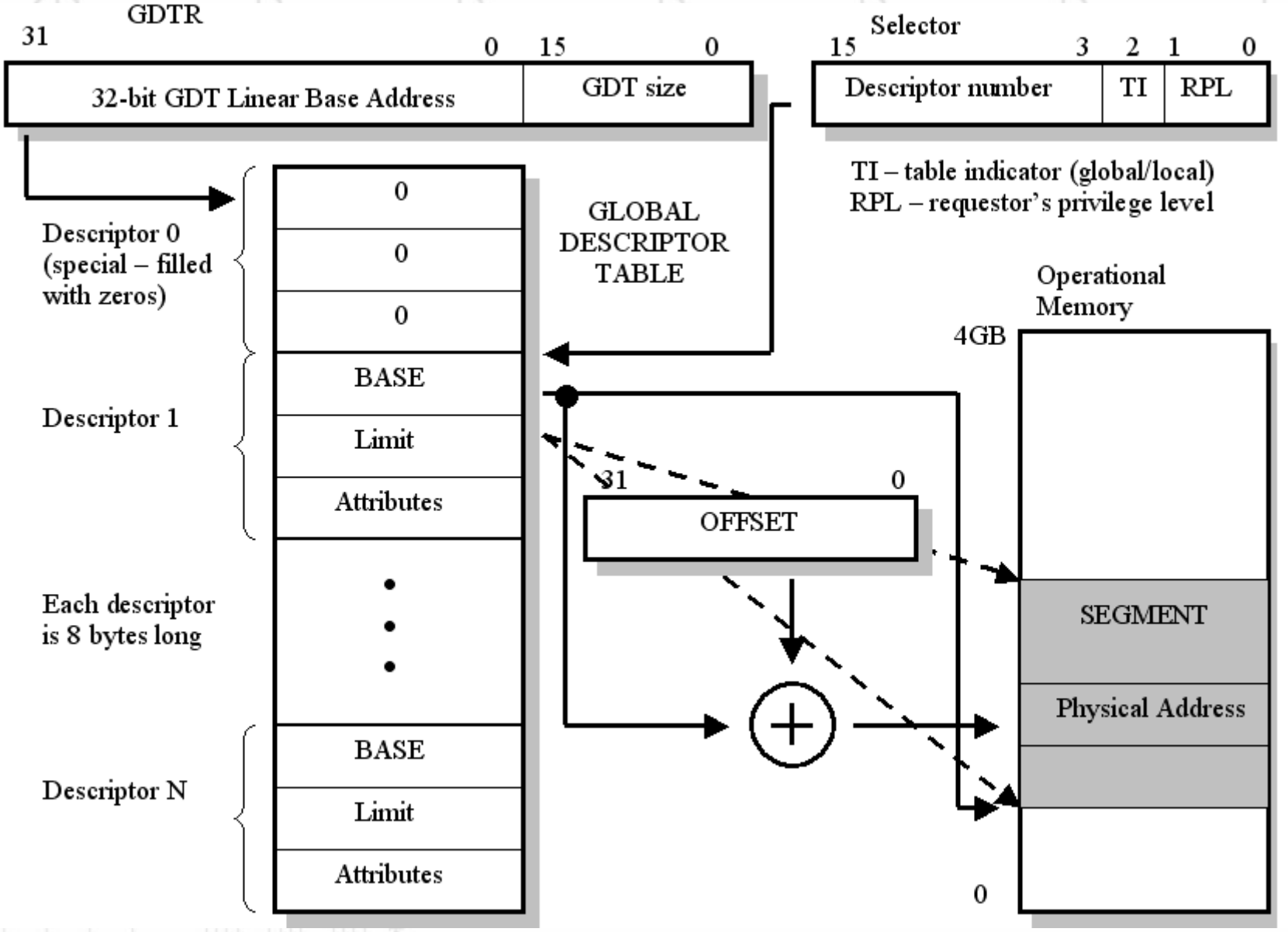


PHYSICAL ADDRESS = 16 * SEGMENT + OFFSET

IA Real Mode Addressing



IA Protected Mode Addressing



Assemby language DOS program structure

DOS COM program has a simplified format and is totally *unrelocable*. It means that when it's compiled to a binary file it must be loaded at specific address without any modification of it's binary form. It must have only one segment reserved for code an data and no stack segment (it uses DOS system stack). The code of a binary file must start at 256 (100h) byte offset from the beginning of the program segment.

Program with only one, no more than 64KB segment

```
.model tiny

.code
org 100h
start:  jmp begin
text    db "Hello world!", 10, 13, '$'
begin:  mov ax, 0900h
        mov dx, offset text
        int 21h
        mov ax, 4c00h
        int 21h
end start
```

256 byte offset

Omit data, so that the data bytes won't be treated as the instruction codes by the processor

One and only program segment

Define which instruction starts the program

Assemby language DOS program structure

DOS EXE program has universal structure. It can have as many code, data and stack segments as one wishes. It can occupy as much memory as one wants. It is relocable, which means that the actual binary form of the program can be freely modified by the operating system during program loading process. It usually has it's own stack, but it can also use DOS system stack, which is quite risky.

Program with one code, data and at most one stack segments

```
                .model small
                .data
text            db "Hello world!", 10, 13, '$'
                .code
start:          mov ax, @data
                mov ds, ax
                mov ax, 0900h
                mov dx, offset text
                int 21h
                mov ax, 4c00h
                int 21h
                .stack 1024
                end start
```

Data segment

Code segment

1024 byte long Stack segment

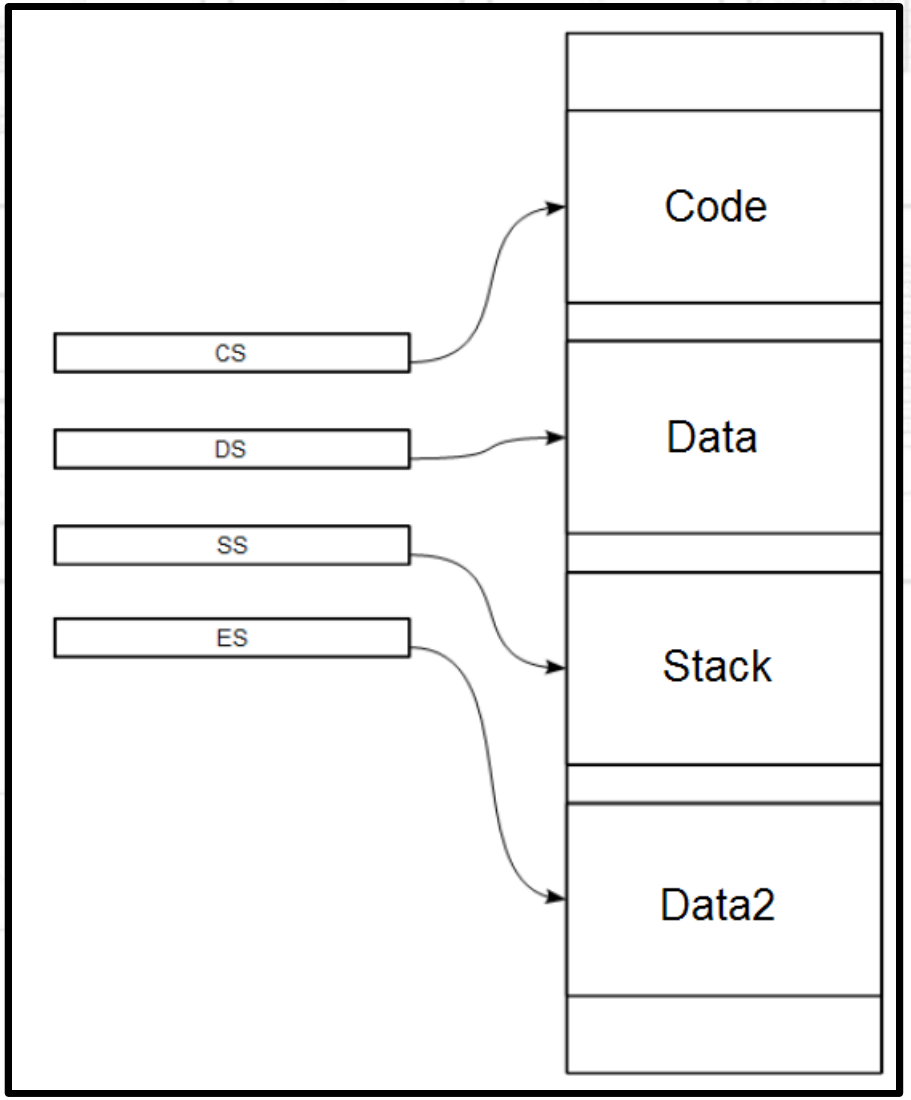
Define which instruction starts the program

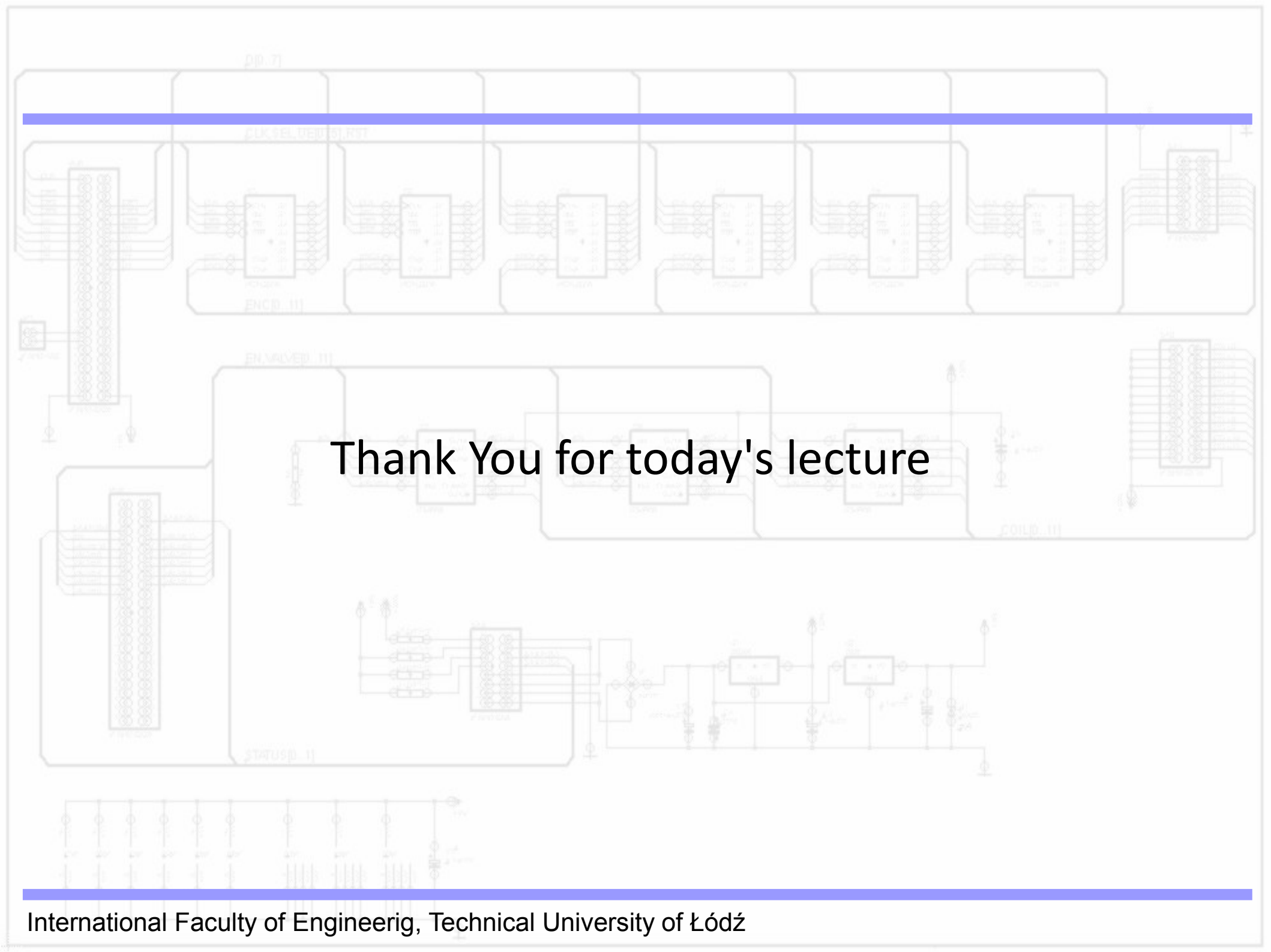
Assemby language DOS program structure

Hence the typical DOS program can be composed of the following segments:
- Code segment,
- Data segment,
- Stack segment,
- Additional data segment.

It is understood that a code segment is obligatory.

In the following figure the typical usage of segment registers to address program segments is presented.





Thank You for today's lecture