Intel x86 Memory Architecture

“The x86 isn't all that complex — it just doesn't make a lot of sense.”

Mike Johnson, Leader of 80x86 Design at AMD, Microprocessor Report (1994)

Program Segments

Data and BSS sections

- Constants
- Initialized and uninitialized variables
- Tables, arrays, strings
- Structures

Text section

- Procedures / functions / control blocks
- Data access
  - Load operations — RAM to register
  - Store operations — register to RAM
  - Peripheral I/O operations
- Calculation
  - Arithmetic operations
  - Logic operations

Stack section

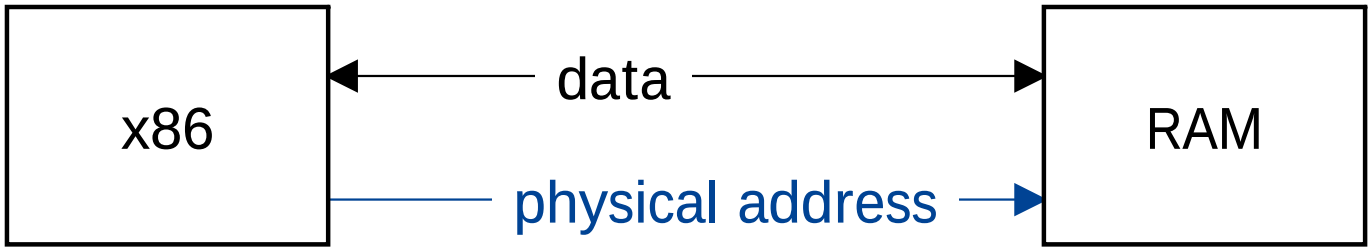
- LIFO buffer accessible by PUSH / POP operations

x86 Data and Address Ranges

Data bits	16	32	64
Unsigned Integer	0 to 64K	0 to 4G	0 to 16E
Signed Integer	–32K to 32K – 1	–2G to +2G - 1	-8E to 8E – 1

	Real Mode	Protected Mode	64-bit Mode
Address bits	20	32	48
Physical Memory	00000 to FFFFF 1 MB	00000000 to FFFFFFFF 4 GB	000000000000 to FFFFFFFFFFFF 256 TB

$K = 2^{10} = 1024$ (Kilo)	$T = 2^{40} = 1024^4$ (Tera)
$M = 2^{20} = 1024^2$ (Mega)	$P = 2^{50} = 1024^5$ (Peta)
$G = 2^{30} = 1024^3$ (Giga)	$E = 2^{60} = 1024^6$ (Exa)



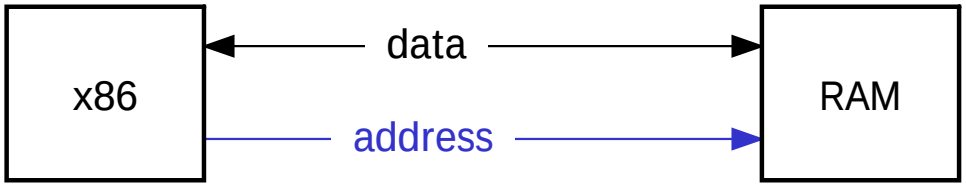
Logical Address CPU

x86 access to physical RAM

CPU accesses external device

Access by physical address

32 bits or 48 bits



Programs do not see physical address

Software address = **logical address** = **SEG**:**OFF**

SEG

Segment selector in CPU segment register

Maps to Physical Base Address

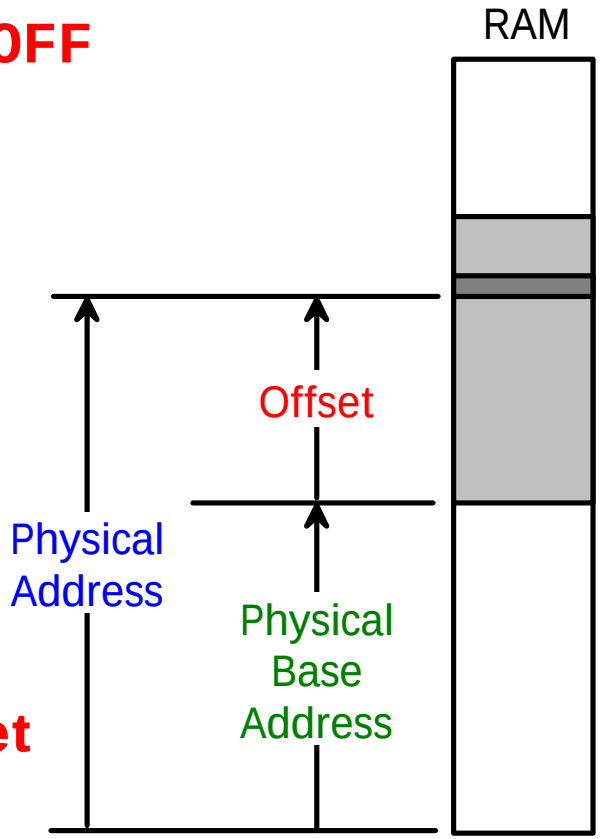
Pointer to start of segment in memory

OFF

Offset formed by pointer arithmetic

Pointer to byte in segment

Physical Address = Physical Base Address + Offset



x86 Segment Registers

x86 segments are sections of physical memory

No one-to-one connection with program segments

Six defined memory pointers

DS (name ~ Data Segment)

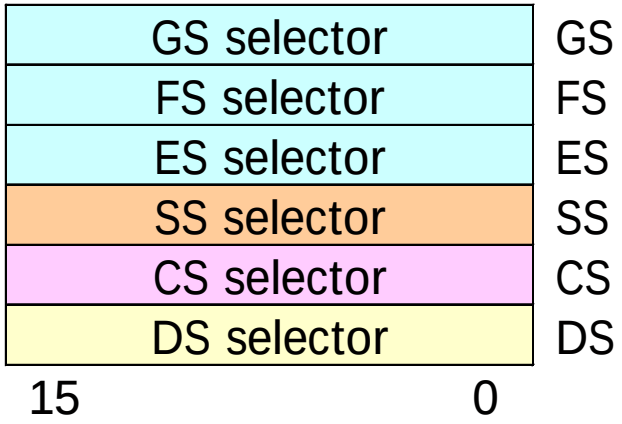
CS (name ~ Code Segment)

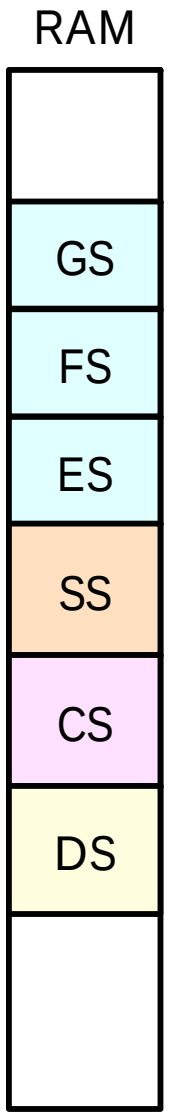
SS (name ~ Stack Segment)

ES (name ~ Extra Segment)

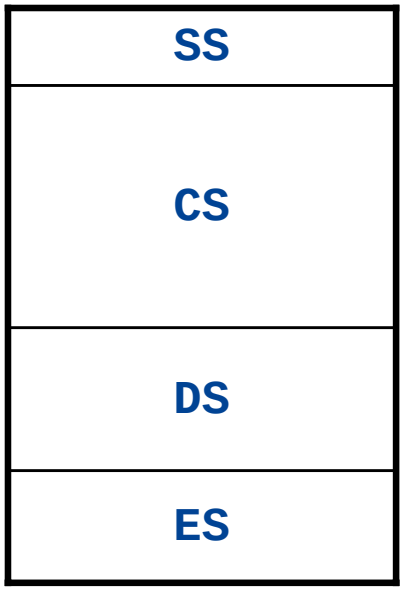
FS, GS

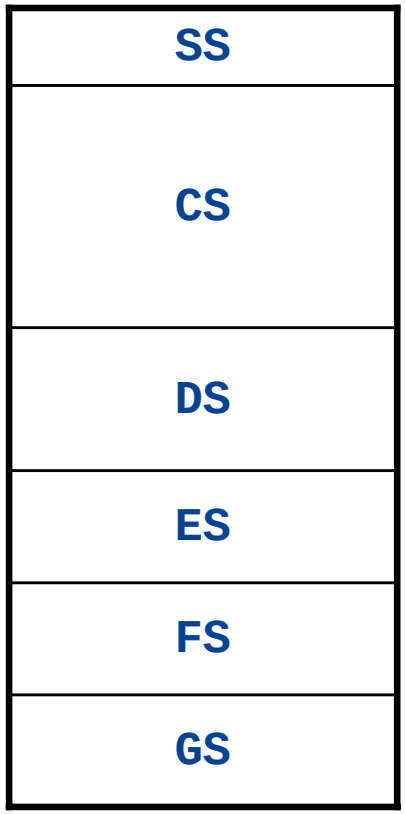
Six segment registers

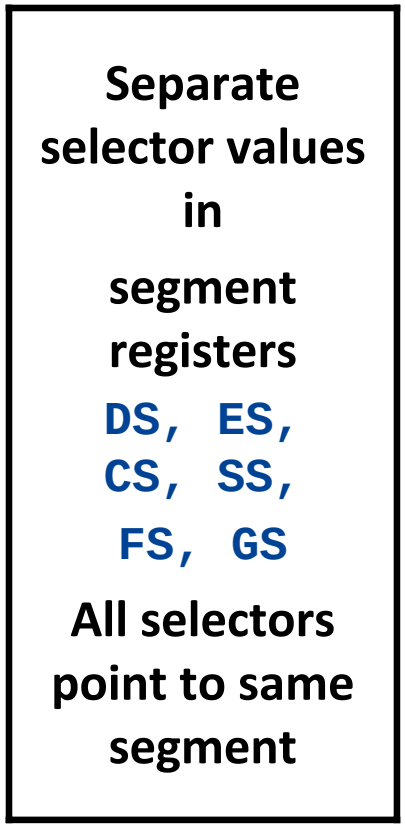


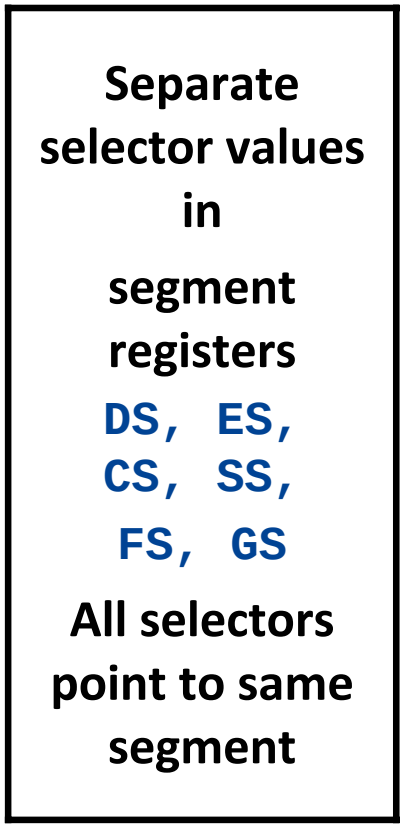


Typical Segment Register Usage









Real mode program

4 defined segments

Segment ≤ 64 KB

32-bit Windows program

6 defined segments

Segment ≤ 4 GB

32-bit Linux program

1 segment

Segment ≤ 4 GB

64-bit program

1 segment

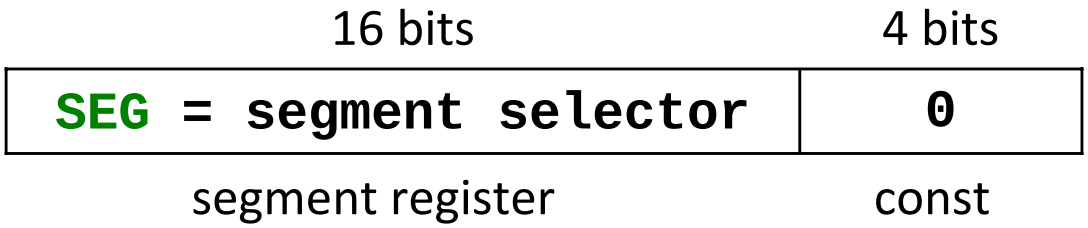
Segment ≤ 256 TB

Mapping Segment

Real mode segment mapping

SEG = 16-bit segment **SELECTOR** in segment register

SEG × **10h** = 20-bit physical base address



Protected mode segment mapping

SEG = 16-bit segment **SELECTOR** in segment register

Selector = index to descriptor table

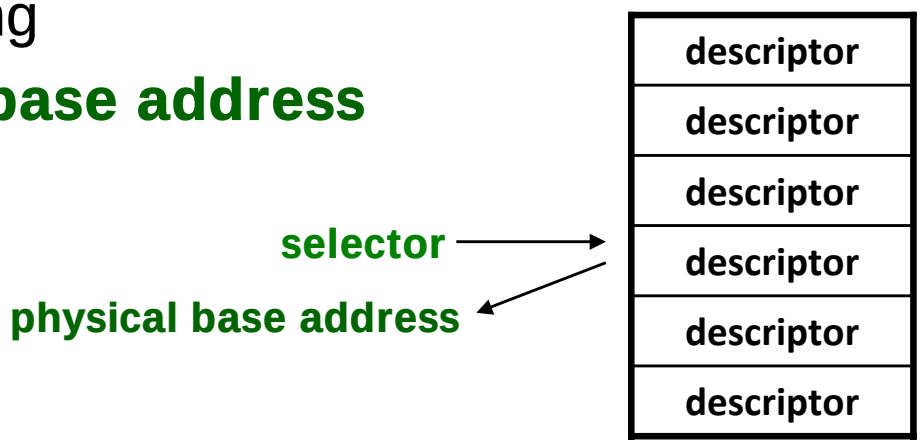
Descriptor = table entry holding

32-bit or 64-bit **physical base address**

Segment size

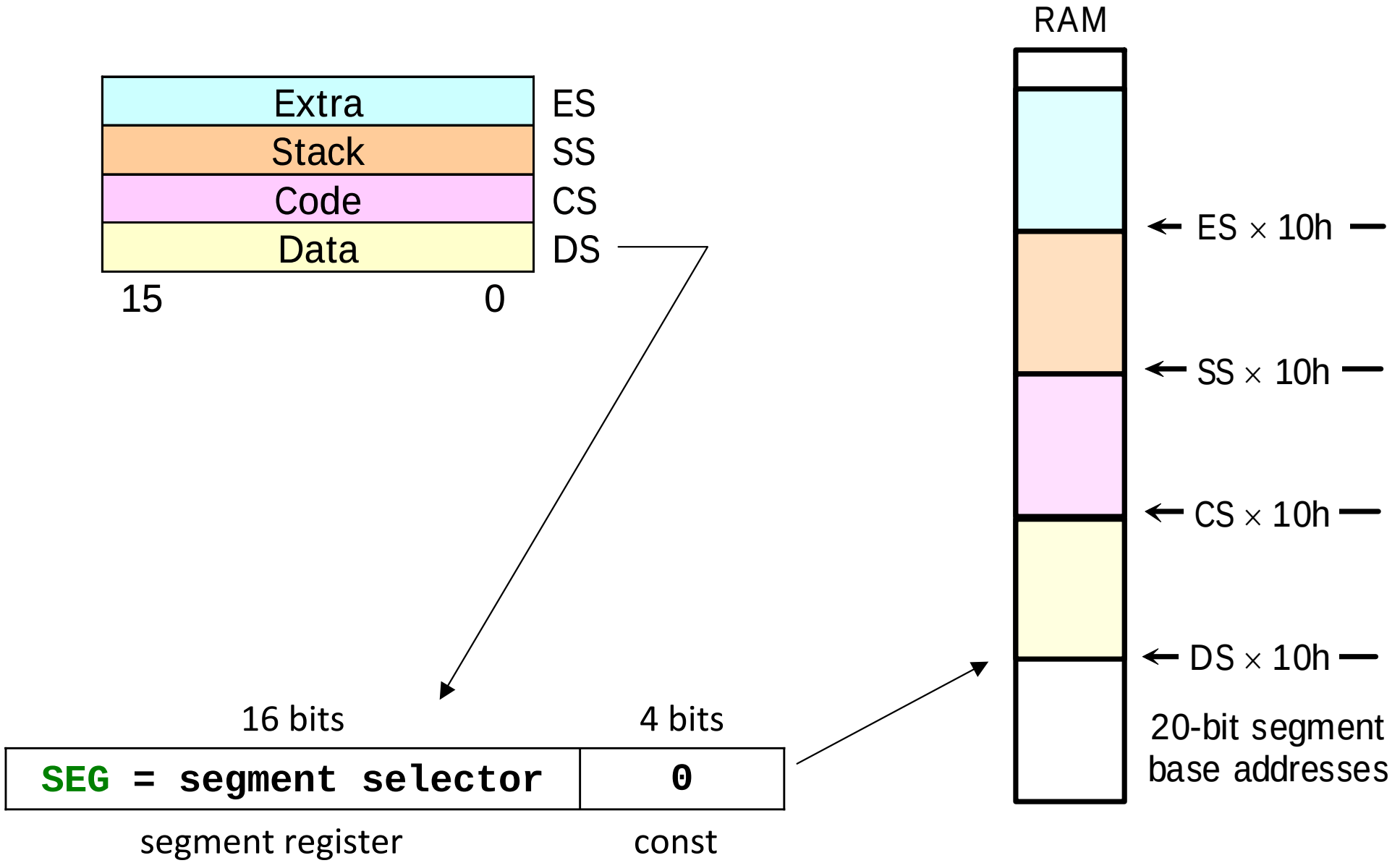
Segment type

Segment access rights



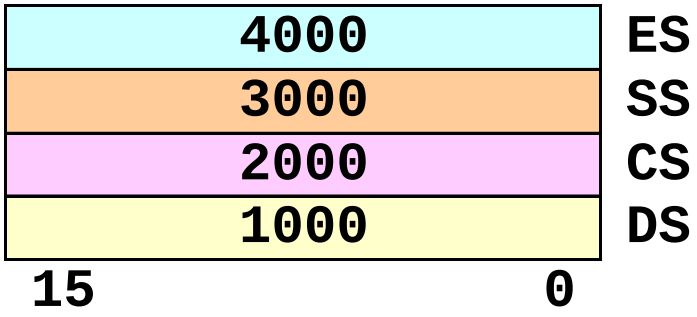
Real Mode Segments Registers

Four segment registers

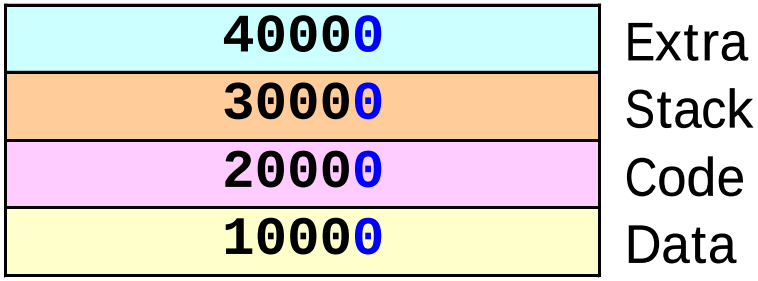


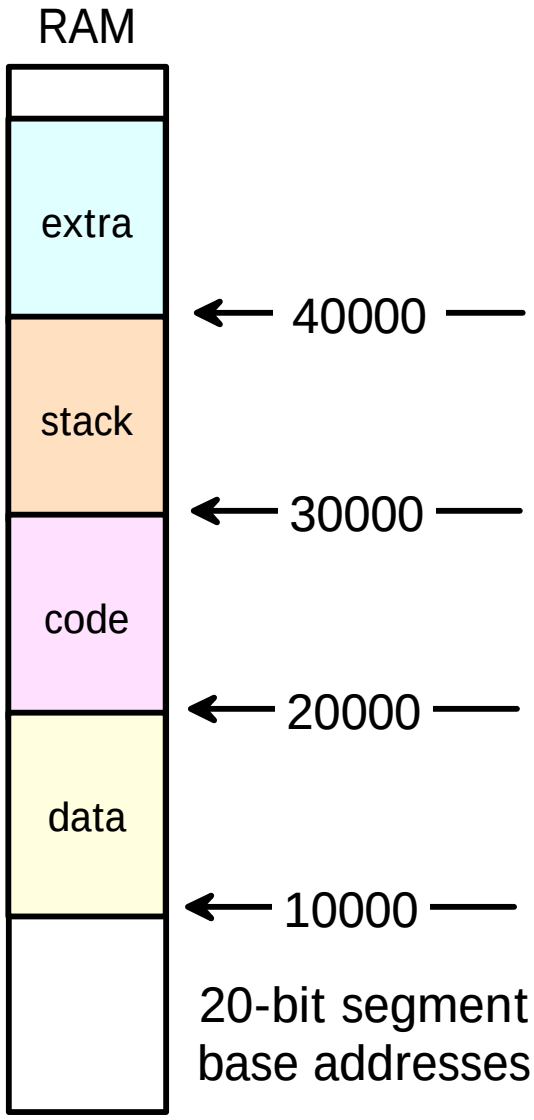
Real Mode Segment Example

16-bit segment registers



20-bit physical base addresses





Real Mode Offset

Offset

16-bit number

Combination of registers and immediate values

Offset ∈ **{0000, 0001, 0002, … , FFFF}**

2^{16} possible offset values

Maximum segment = 2^{16} bytes = 64 KB

byte	FFFF
byte	…
byte	0002
byte	0001
byte	0000

Physical Address = physical base address + offset

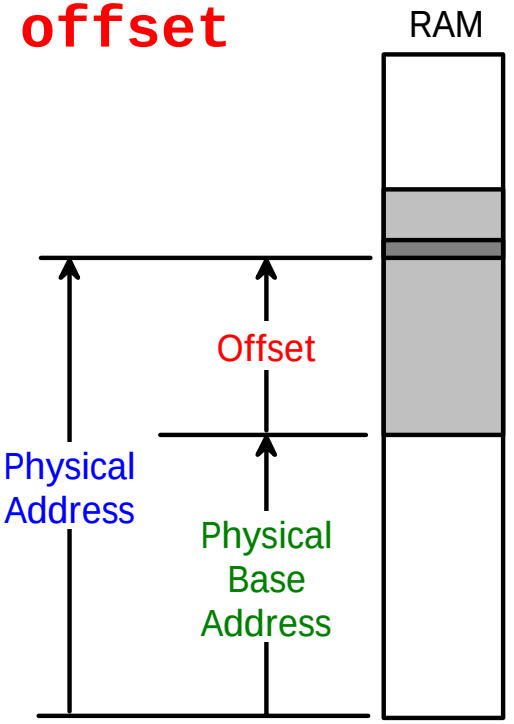
Example

Logical Address = **1234:0005**

Segment selector = **1234**

Physical base address = **12340**

Physical Address = **12340** + **0005** = **12345**



Protected Mode Offset

Offset

- 32-bit number
- Combination of registers and immediate values
- Offset ∈ **{00000000, … , FFFFFFFF}**
- 2^{32} possible offset values
- Maximum segment = 2^{32} bytes = 4 GB

byte	FFFFFFFF
byte	…
byte	00000002
byte	00000001
byte	00000000

Physical Address = physical base address + offset

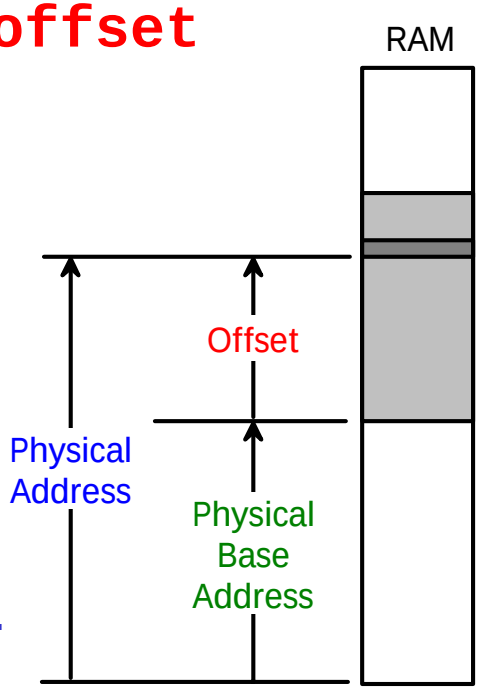
Example

Logical Address = **1234**:**11223344**

Segment selector = **1234** → descriptor table

Physical base address = **00000000**

Physical Address = **0** + **11223344** = **11223344**



Program Segments versus Memory Segments

Segment

Segment = any section, part, portion, subdivision

Segment ≠ specific, 1-to-1 technical meaning

Program segments (sections)

Sections of assembly source code or executable program

Data segment

- Space allocated for storing program data

Text (code) segment

- Space allocated for program instructions

Stack segment

- Space allocated for user stack operations

x86 memory segments

`DS, CS, SS, ES, FS, GS`

Pointers to sections of physical memory (RAM)

No one-to-one connection to program segments

x86 Addressing Modes

Addressing mode

Location of data in register, memory, or instruction

Mode	Example	Operation / Addressing Mode
Register	`MOV EAX, EBX`	`regs[EAX] ← regs[EBX]`
Immediate	`MOV EAX, 12h`	`regs[EAX] ← #12`
Absolute	`MOV EAX, [11223344]`	`regs[EAX] ← mem[DS:11223344]`
Effective Address	`MOV EAX, [EA]`	`regs[EAX] ← mem[SEG:EA]`
String	`LODSB`	`AL ← mem[DS:ESI]` `ESI ← ESI + 1`
Stack	`POP EAX`	`EAX ← mem[SS:ESP]` `ESP ← ESP + 4`

Effective Address (EA)

Combination of register values and literal values

32-bit Effective Address

Effective Address (EA)

EA = base + index * scale + displacement

Base register — 0 or 1 register name

base ∈ {EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI}

Index register — 0 or 1 register name

index ∈ {EAX, EBX, ECX, EDX, EBP, ESI, EDI}

Scale

scale ∈ {1, 2, 4, 8}

Displacement — 0 or 1 literal value

Examples

MOV EAX,[EAX]	Base
MOV EAX,[EBX+EDI]	Base + Index
MOV EAX,[ECX+4*EDI]	Base + Index * Scale
MOV EAX,[EDX+8*ESI+1122h]	Base + Index * Scale + Disp
MOV EAX,[11223344h]	Displacement

32-bit Effective Address ⇒ Segment

Effective Address

```
MOV REG, [EA]    ⇒    regs[REG] ← mem[SEG:EA]
EA = base + index * scale + displacement
```

Default segment

```
base = ESP or EBP
    SEG = SS
```

Otherwise

```
    SEG = DS
```

Examples

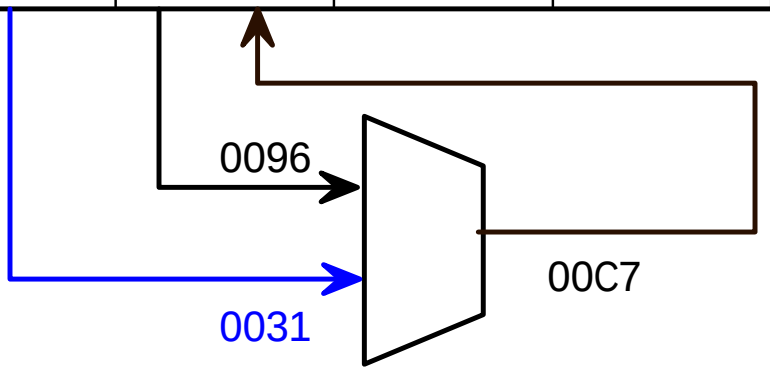
```
MOV EAX, [EAX]          regs[EAX] ← mem[DS:EAX]
MOV EAX, [ESP]          regs[EAX] ← mem[SS:ESP]
MOV EAX, [EBP + ESI]    regs[EAX] ← mem[SS:EBP + ESI]
MOV EAX, [EAX + 4*EPB]  regs[EAX] ← mem[DS:EAX + 4*EPB]
MOV EAX, [1123344]      regs[EAX] ← mem[DS:1123344]
```

x86 Register Addressing Mode

Addressing Mode	Example Instruction	Meaning
Register	**ADD BX,AX**	**REGS[BX] ← REGS[BX] + REGS[AX]**
	Used for operands that the ALU is using now	

Registers

ESI	EDI	EBP	EAX	EBX	ECX	EDX	ESP
00001025	00001004	00000003	00000031	00000096	0000D400	00009008	00000000

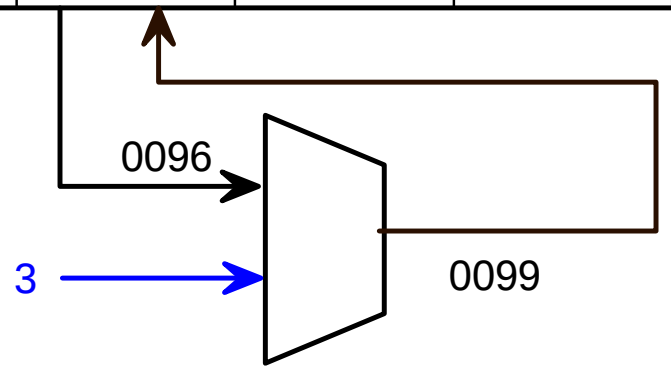


x86 Immediate Addressing Mode

Addressing Mode	Example Instruction	Meaning
Immediate	**ADD BX,3**	**REGS[BX] ← REGS[BX] + 3**
	Used for constant operands	

Registers

ESI	EDI	EBP	EAX	EBX	ECX	EDX	ESP
00001025	00001004	00000003	00000031	00000096	0000D400	00009008	00000000

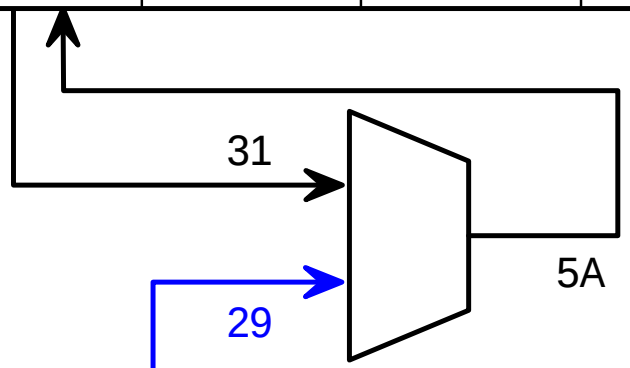


x86 Absolute Addressing Mode

Addressing Mode	Example Instruction	Meaning
Direct Or Absolute	**ADD AL,[1001]**	**REGS[AL] ← REGS[AL] + MEM[DS_start + 1001]**
	Used for storing general data in main memory	

ESI	EDI	EBP	EAX	EBX	ECX	EDX	ESP
00001025	00001004	00000003	00000031	00000096	0000D400	00009008	00000000

Registers



Main Memory (RAM)

1007	00
1006	58
1005	00
1004	FF
1003	9E
1002	10
1001	29
1000	45

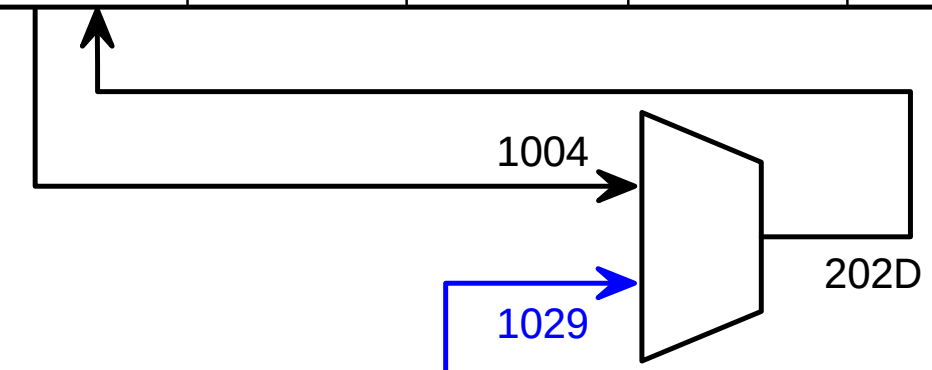
x86 Absolute Addressing Mode

Addressing Mode	Example Instruction	Meaning
Direct Or Absolute	**ADD DI,[1001]**	**REGS[DI] ← REGS[DI] + MEM[DS_start + 1001]**
	Used for storing general data in main memory	

ESI	EDI	EBP	EAX	EBX	ECX	EDX	ESP
00001025	00001004	00000003	00000031	00000096	0000D400	00009008	00000000

Registers

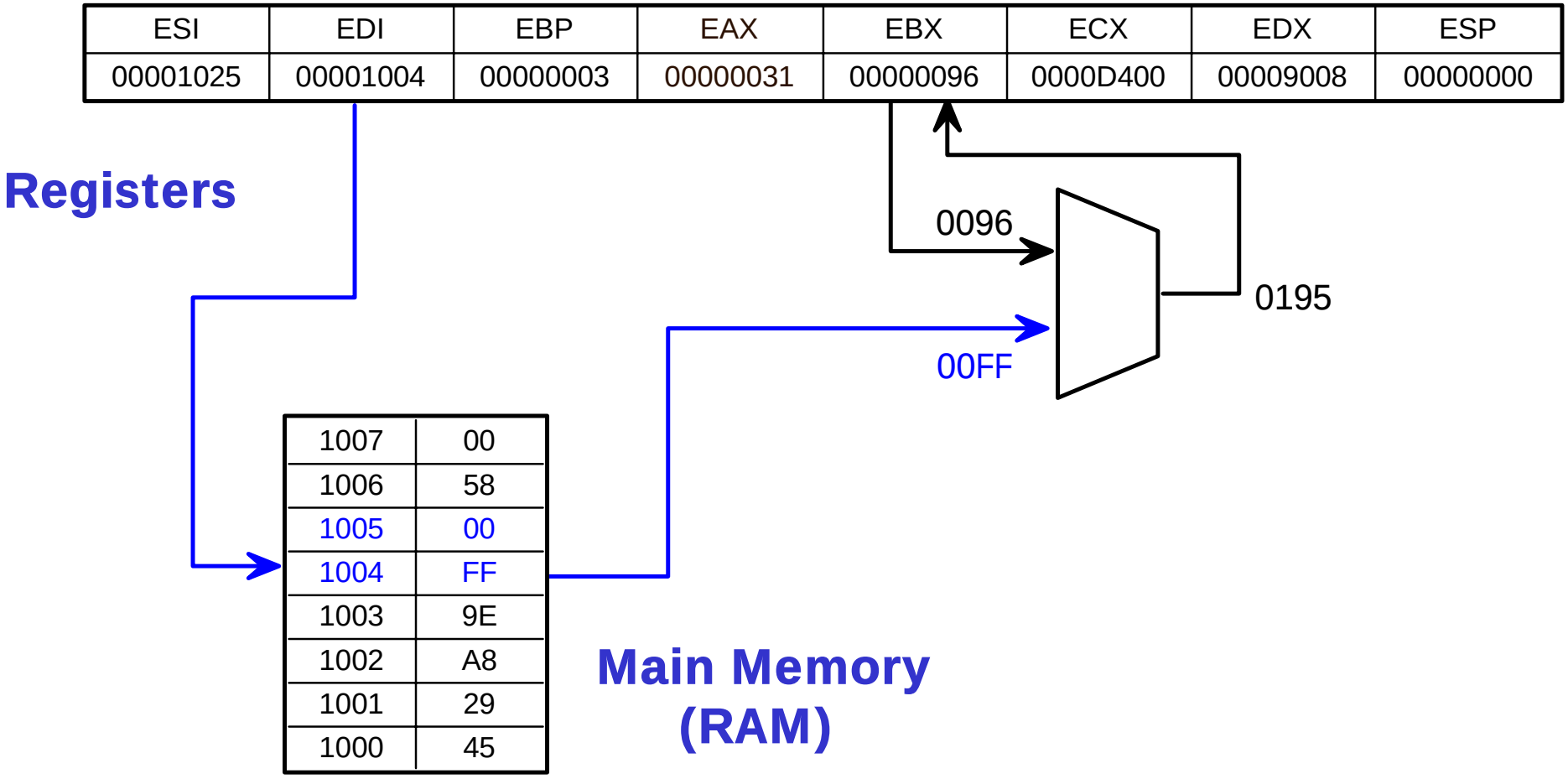
Main Memory (RAM)



1007	00
1006	58
1005	00
1004	FF
1003	9E
1002	10
1001	29
1000	45

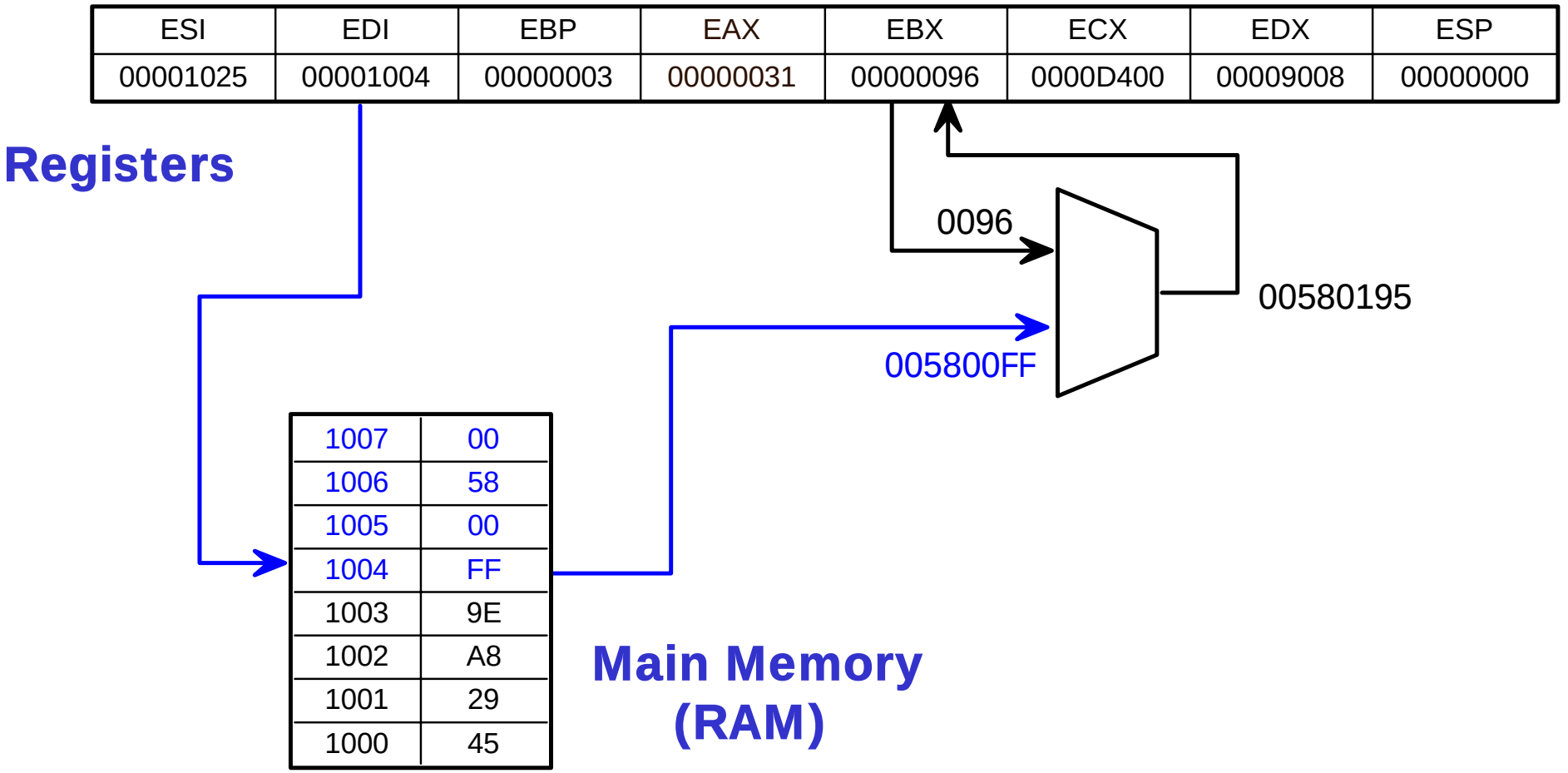
x86 Based Addressing Mode

Addressing Mode	Example Instruction	Meaning
Register Deferred	**ADD BX,[EDI]**	**REGS[BX] ← REGS[BX] + MEM[DS_start + REGS[EDI]]**
		A pointer to data is kept in the register



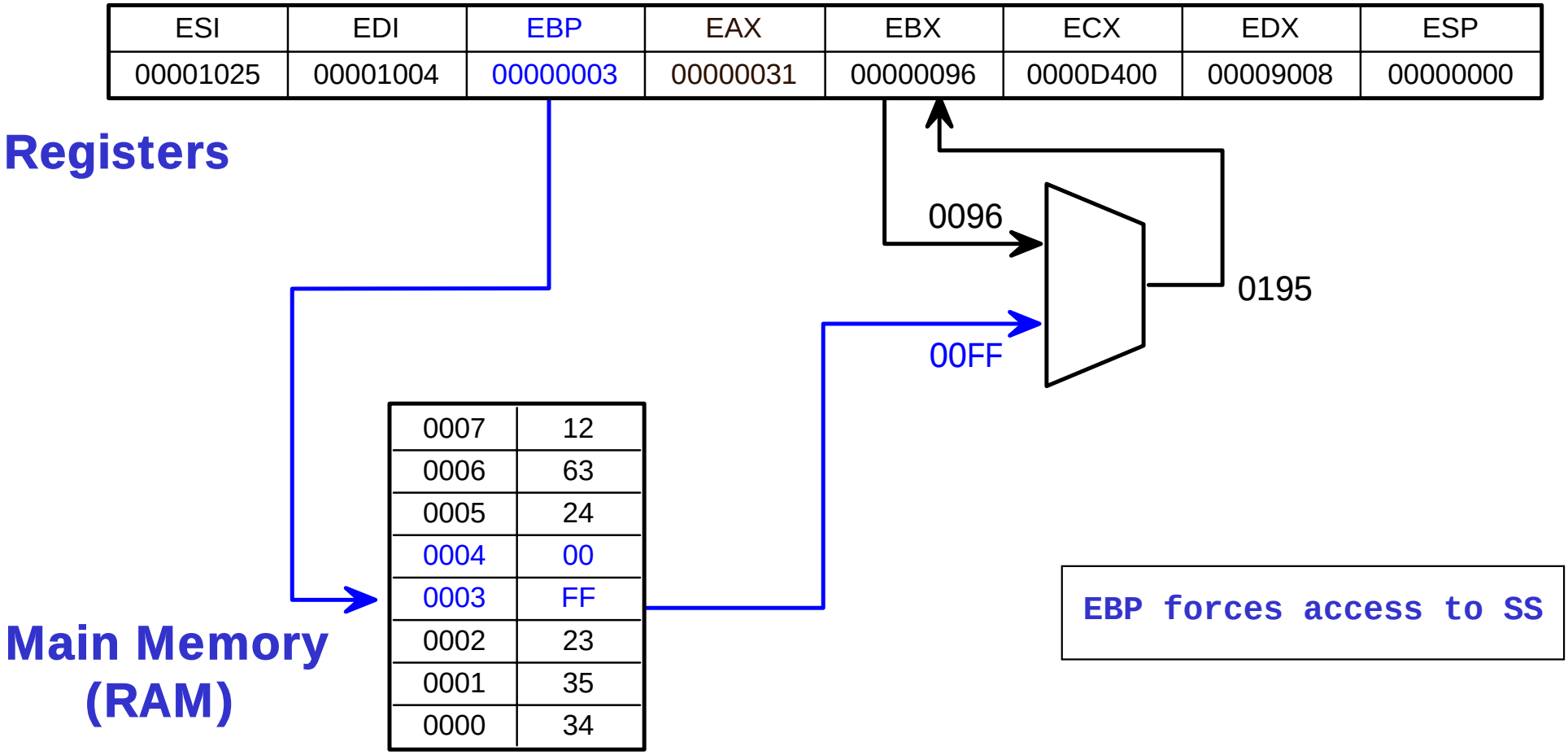
x86 Based Addressing Mode

Addressing Mode	Example Instruction	Meaning
Register Deferred	**ADD EBX,[EDI]**	**REGS[EBX] ← REGS[EBX] + MEM[DS_start + REGS[EDI]]**
	A pointer to data is kept in the register	



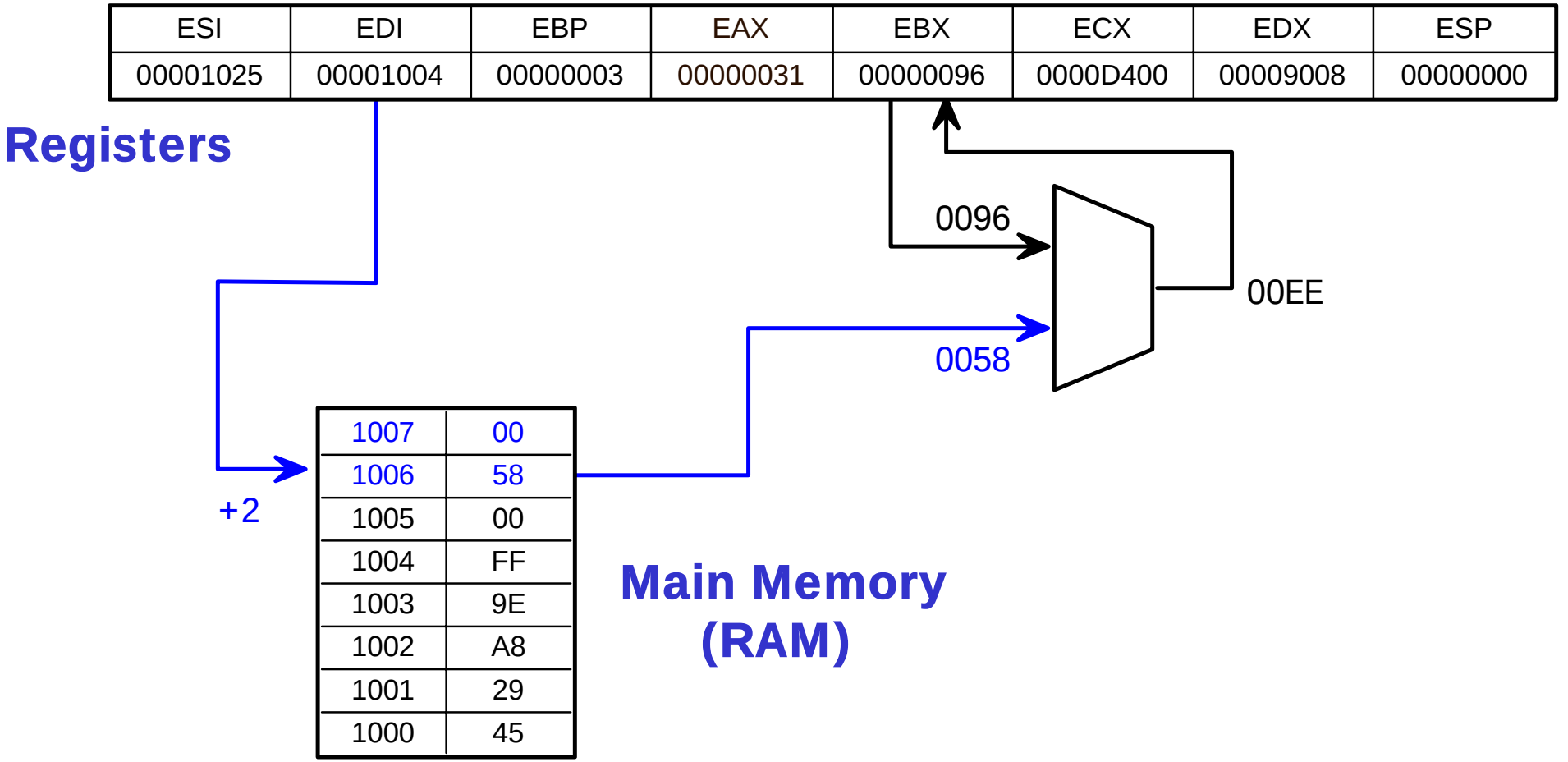
x86 Based Addressing Mode

Addressing Mode	Example Instruction	Meaning
Register Deferred	**ADD BX, [EBP]**	**REGS[BX] ← REGS[BX] + MEM[SS_start + REGS[EBP]]**
	A pointer to data is kept in the register	



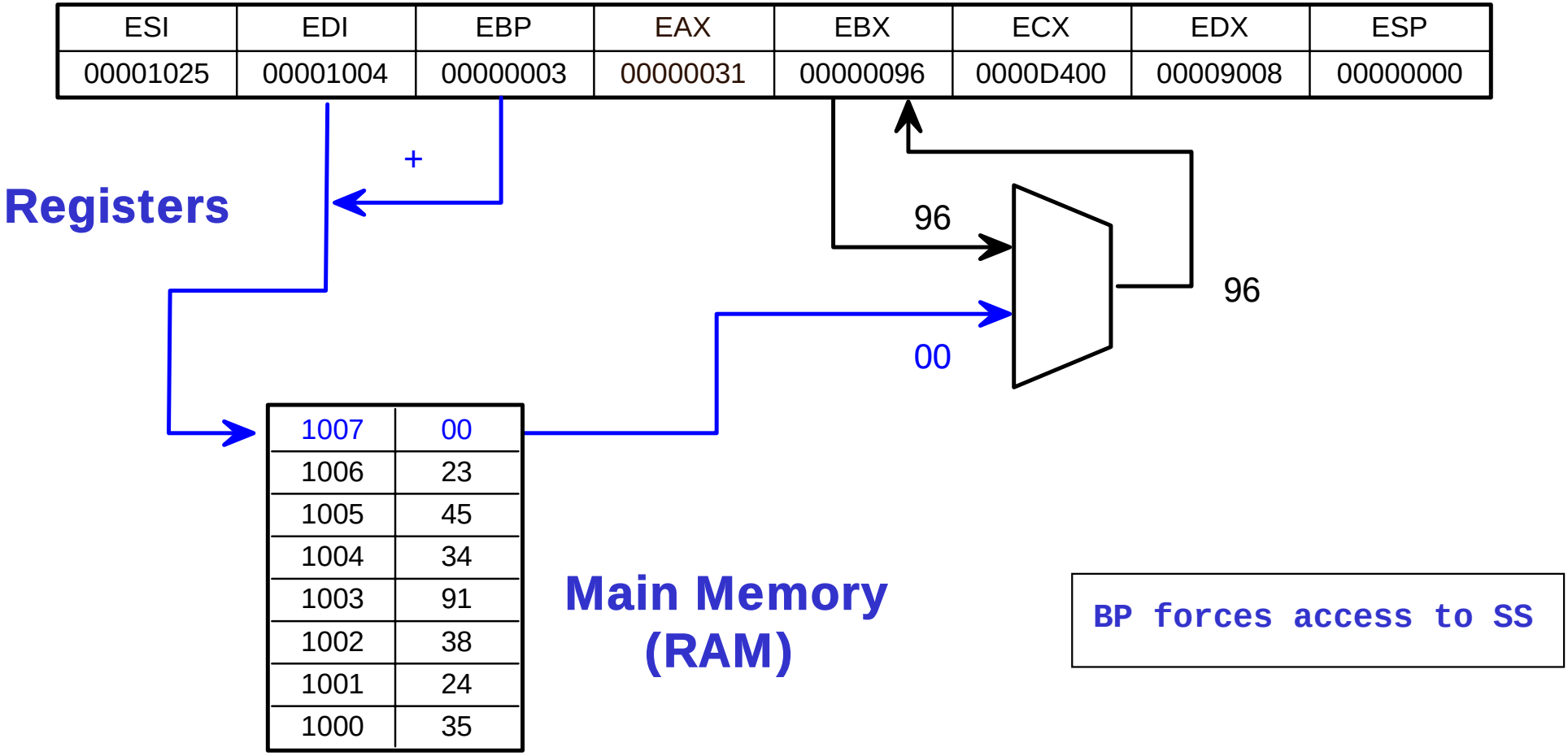
x86 Based with Displacement Addressing Mode

Addressing Mode	Example Instruction	Meaning
Displacement	`ADD BX,[EDI+2]`	`REGS[BX] ← REGS[BX] + MEM[DS_start + REGS[DI] + 2]`
	A constant offset is added to the pointer	



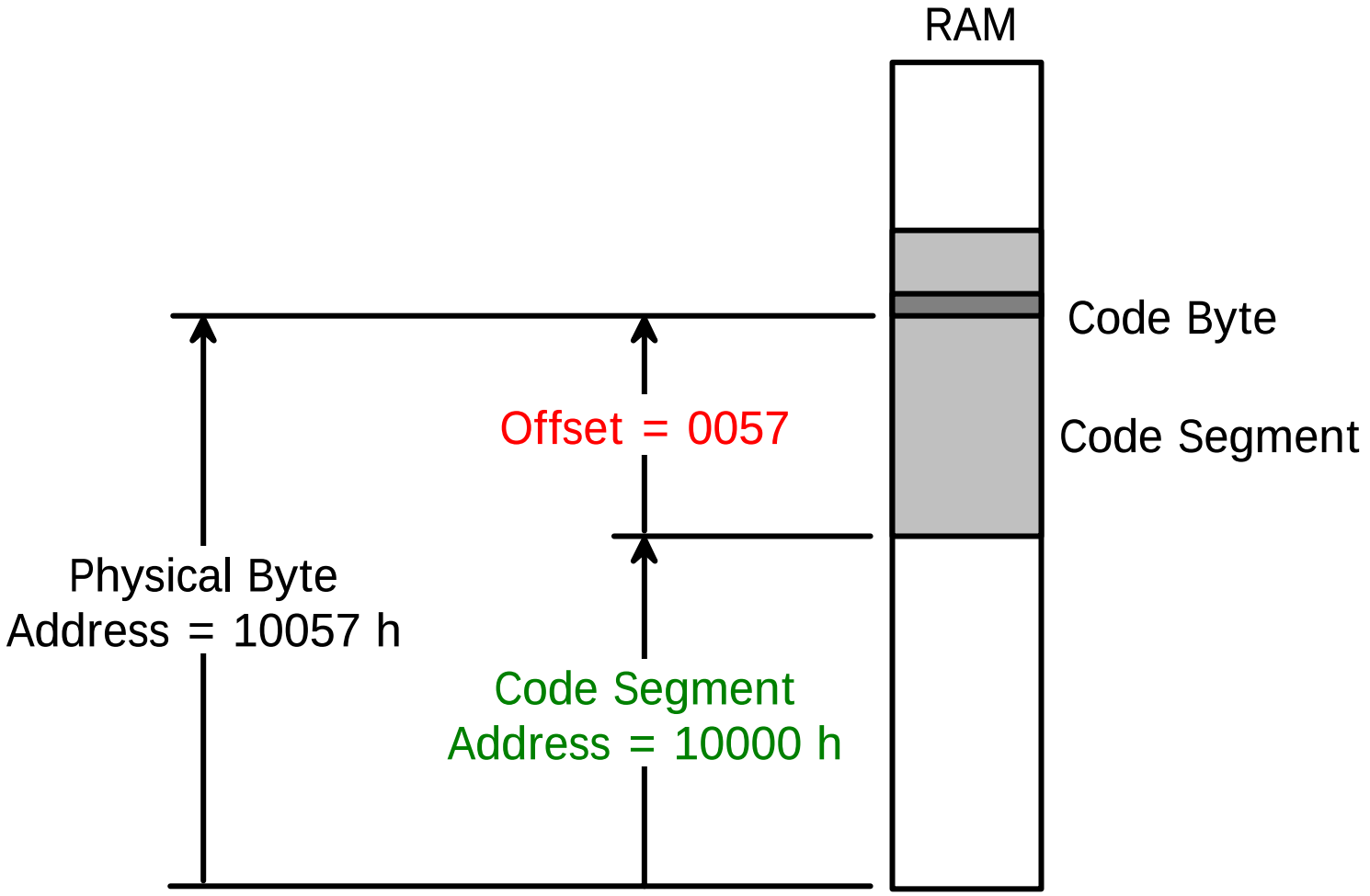
x86 Based and Indexed Addressing Mode

Addressing Mode	Example Instruction	Meaning
Indexed	**ADD BL,[EBP+EDI]**	**REGS[BL]← REGS[BL]+ MEM[SS_start + REGS[EBP]] + REGS[EDI]**
	A stored offset or table index is added to the pointer	



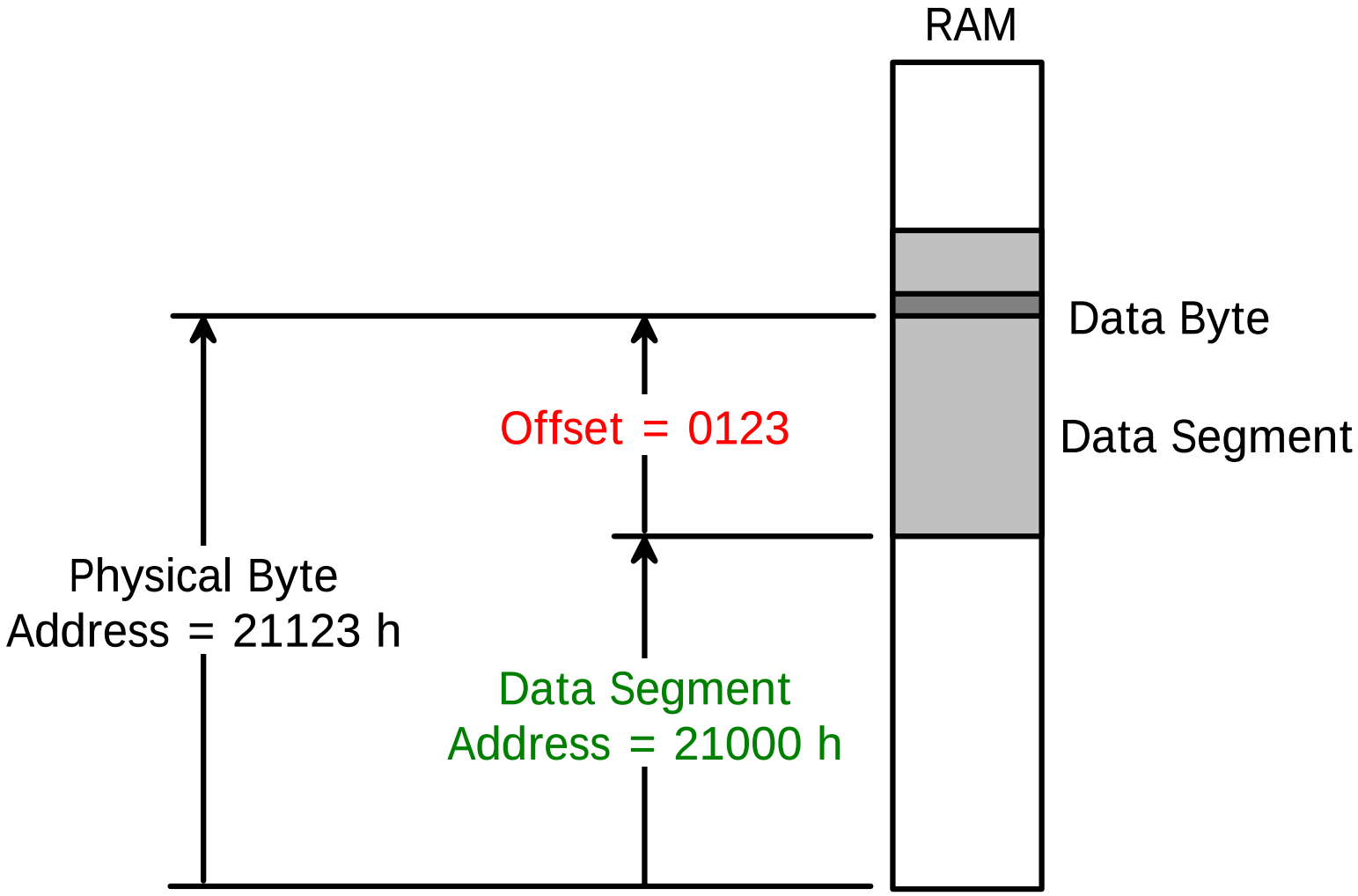
Segmentation Example: Instruction Fetch

Instruction Fetch -- Logical Address = CS:IP
CS = 0012
IP = 0057
Logical Address = CS:IP = 0012:0057
Code Segment Physical Base Address = 10000 h



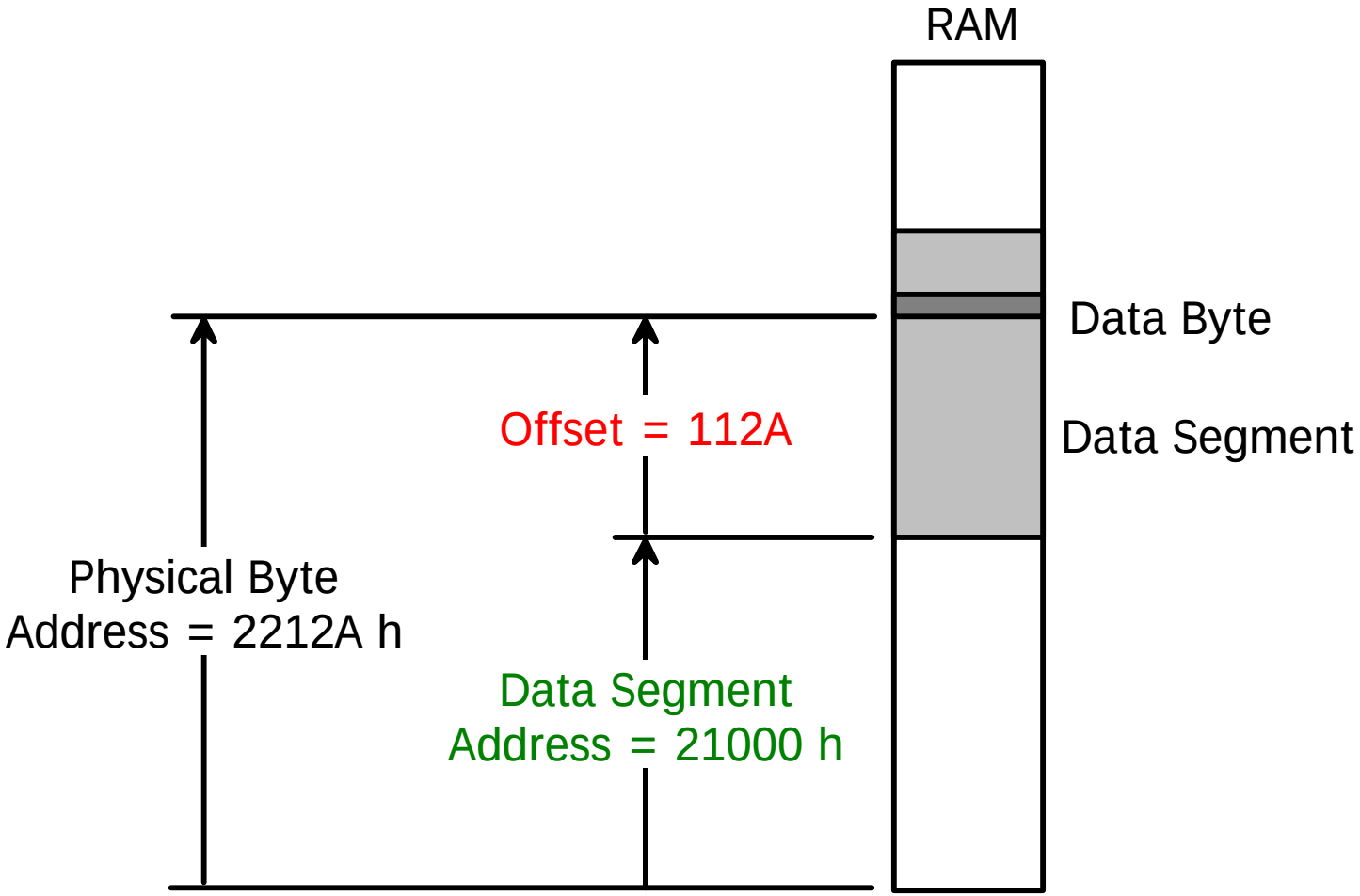
8086 Segmentation Example: MOV AX,[BX]

```
MOV Instruction -- DS
Effective Address = EBX
Logical Address = DS:EBX = 0047:0123
Data Segment physical base address = 21000h
```



8086 Segmentation Example: MOV AX,[BX+SI+7]

```
MOV Instruction -- DS
Effective Address = EBX+ESI+7
Logical Address = DS:EBX+ESI+7
                = 1234:0123 + 1000 + 7
                = 1234:112A
Data Segment physical base address = 21000
```



MOV in x86

```
MOV dest, src          dest ← src

MOV AX, [1234]         REGS[AX] ←(16-bits)— MEM[DS_start + 00001234]

                       REGS[AL] ←(8-bits)— MEM[DS_start + 00001234]

                       REGS[AH] ←(8-bits)— MEM[DS_start + 00001235]

MOV AL, [1234]         REGS[AL] ←(8-bits)— MEM[DS_start + 00001234]

MOV AX, [EBX]          REGS[AX] ←(16-bits)— MEM[DS_start + REGS[EBX] ]

MOV EAX, [EBX+ESI]

            REGS[AX] ←(32-bits)— MEM[DS_start + REGS[EBX] + REGS[ESI] ]

MOV EAX, [EBX+ESI+12]

            REGS[AX] ←(32-bits)— MEM[DS_start + REGS[EBX] + REGS[ESI] + 12]
```

x86 Stack Operations

PUSH src

$$ESP \leftarrow ESP - 4$$

$$[SS:ESP] \xleftarrow[32\text{-bits}]{} src$$

POP dest

$$dest \xleftarrow[32\text{-bits}]{} [SS:ESP]$$

$$ESP \leftarrow ESP + 4$$

x86 Segment Override

```
CS:   MOV [EBP],ECX     [CS:EBP] ← ECX
ES:   MOV [EBP],ECX     [ES:EBP] ← ECX
DS:   MOV [EBP],ECX     [DS:EBP] ← ECX
SS:   MOV [ESI],ECX     [SS:ESI] ← ECX
```

x86 String Instructions — 1

STOSB	Store String Byte	[ES:EDI] ← AL EDI ← EDI+1 (default) (EDI ← EDI-1 if flag DF = 1)
STOSW	Store String Word	[ES:EDI] ← AL [ES:EDI+1] ← AH EDI ← EDI+2 (default) (EDI ← EDI-2 if flag DF = 1)
STOSD	Store String Double	[ES:EDI] ← EAX EDI ← EDI+4 (default) (EDI ← EDI-4 if flag DF = 1)
LODSB	Load String Byte	AL ← [DS:ESI] ESI ← ESI+1 (default) (ESI ← ESI-1 if flag DF = 1)
LODSW	Load String Word	AL ← [DS:ESI] AH ← [DS:ESI+1] ESI ← ESI+2 (default) (ESI ← ESI-2 if flag DF = 1)
LODSD	Load String Double	EAX ← [DS:ESI] ESI ← ESI+4 (default) (ESI ← ESI-4 if flag DF = 1)

x86 String Instructions — 2

MOVSB	Move String Byte	[ES:EDI] ← [DS:ESI] (byte) EDI ← EDI+1 (default) ESI ← ESI+1
MOVSW	Move String Word	[ES:EDI] ← [DS:ESI] (word) EDI ← EDI+2 (default) ESI ← ESI+2
MOVSD	Move String Double	[ES:EDI] ← [DS:ESI] (dword) EDI ← EDI+4 (default) ESI ← ESI+4
SCASB	Scan String Byte	AL-[ES:DI]; update flags (byte) EDI ← EDI+1 (default)
SCASW	Scan String Word	AL-[ES:DI]; update flags (word) DI ← DI+2 (default)
SCASD	Scan String Double	AL-[ES:DI]; update flags (dword) DI ← DI+4 (default)

x86 String Instructions — 3

CMPSB	**Compare String Byte**	**[DS:ESI]-[ES:EDI] (byte)** **Update flags** **EDI ← EDI+1 (default)** **ESI ← ESI+1**
CMPSW	**Compare String Word**	**[DS:ESI]-[ES:EDI] (word)** **Update flags** **EDI ← EDI+2 (default)** **ESI ← ESI+2**
CMPSD	**Compare String Double**	**[DS:ESI]-[ES:EDI] (dword)** **Update flags** **EDI ← EDI+4 (default)** **ESI ← ESI+4**

8086 String Instructions — 4

REP STOSB	STOSB ECX ← ECX - 1 Repeat until ECX = 0
REP MOVSB	MOVSB ECX ← ECX - 1 Repeat until ECX = 0

Working with Strings

```
PUSH ES             ; ESP ← ESP - 4
                    ; [SS:ESP] ← ES
PUSH DS             ; ESP ← ESP - 4
                    ; [SS:ESP] ← DS
POP ES              ; ES ← DS
                    ; ESP ← ESP + 4
MOV SI,0            ; SI ← 0
MOV DI,1000         ; DI ← 1000
MOV ECX,200         ; ECX ← 200
REP MOVSB           ; COPY 200 H BYTES FROM
                    ;  DS:0000 - DS:01FF to
                    ;  DS:1000 - DS:11FF
POP ES              ; restore old ES
                    ; ESP ← ESP + 4
```

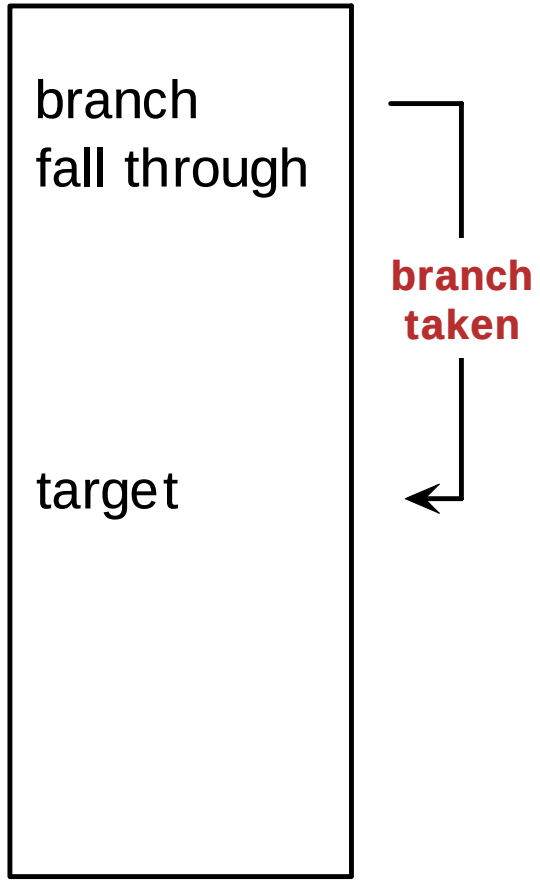
Branch Instructions

Fall-through

Instruction following branch in program listing

Next instruction if branch **not taken**

`CS:EIP` points to fall-through



Target

Next instruction if branch **taken**

`CS:EIP` points to target

Displacement

Displacement = **`target EIP – fall-through EIP`**

Displacement > 0 is forward jump

Displacement < 0 is backward jump

Jump Distance

Displacement

target EIP – fall-through EIP

Short Jump

Target in same code segment

Displacement is byte

$\mathbf{-128_{10} = 80 \leq displacement \leq 7F = 127_{10}}$

Near Jump

Target in same code segment

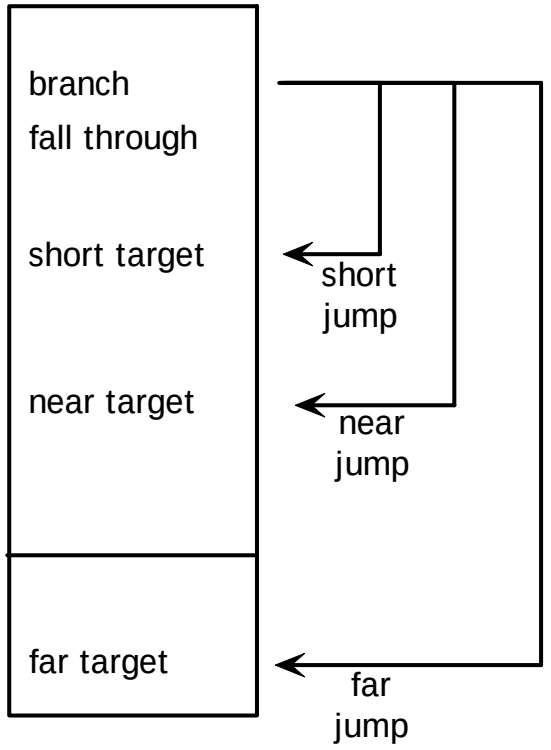
Pointer is **EIP**

Far Jump

Target in different code segment

Pointer is **CS:EIP**

Displacement not relevant



Jump Instruction

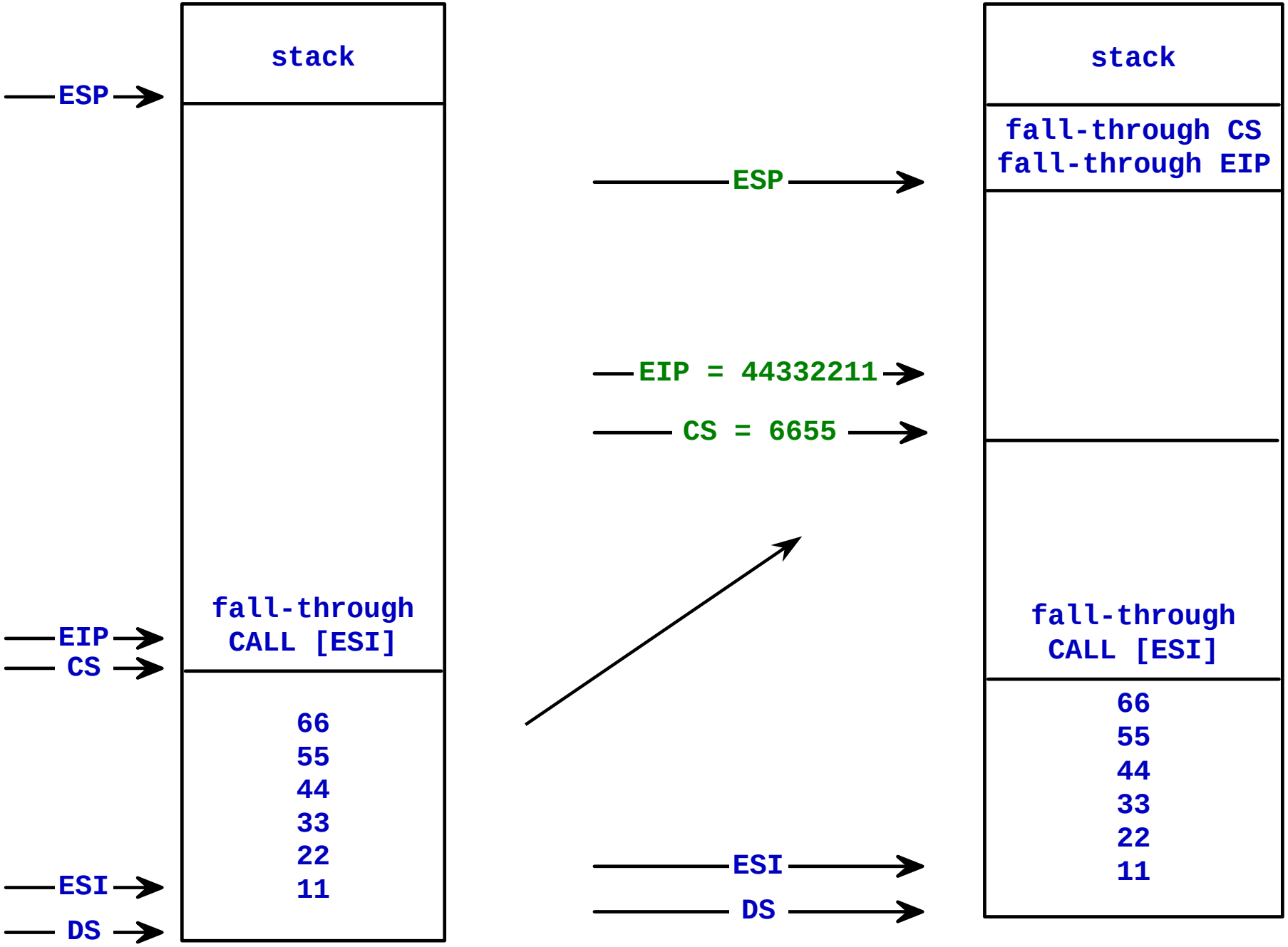
JMP target	JMP 1024	EIP $\leftarrow_{32}$ 1024
	JMP [1024]	EIP $\leftarrow_{32}$ [1024]
	JMP [ESI]	EIP $\leftarrow_{32}$ [DS:ESI]
JMP far target	JMP FAR [1024]	CS $\leftarrow_{16}$ [1028] EIP $\leftarrow_{32}$ [1024]
	JMP FAR [SI]	CS $\leftarrow_{16}$ [DS:ESI+4] EIP $\leftarrow_{32}$ [DS:ESI]

Call and Return

CALL near target	PUSH EIP EIP ← target
CALL 1024	PUSH EIP EIP ← 1024
CALL ESI	PUSH EIP EIP ← ESI
CALL [ESI]	PUSH EIP EIP ← [DS:ESI]
RET	POP EIP

CALL far target	PUSH CS PUSH EIP CS ← SEG EIP ← OFF
CALL [ESI]	PUSH CS PUSH EIP CS ← [DS:ESI+4] EIP ← [DS:ESI]
RETF	POP EIP POP CS

Indirect Far Call



Interrupt

Interrupt

- Pushes values of **`flags`** and **`CS:EIP`** onto stack
- Transfers control to Interrupt Service Routine (ISR)
  - ISR can be stored anywhere in memory
  - **`CS:EIP`** for ISR found in table

Interrupt Vector Descriptor Table (IDT)

- Managed by operating system
- Interrupt descriptor
  - Target address of ISR
  - Permissions for running ISR

Hardware interrupt

- Initiated by system hardware and peripheral devices

Trap

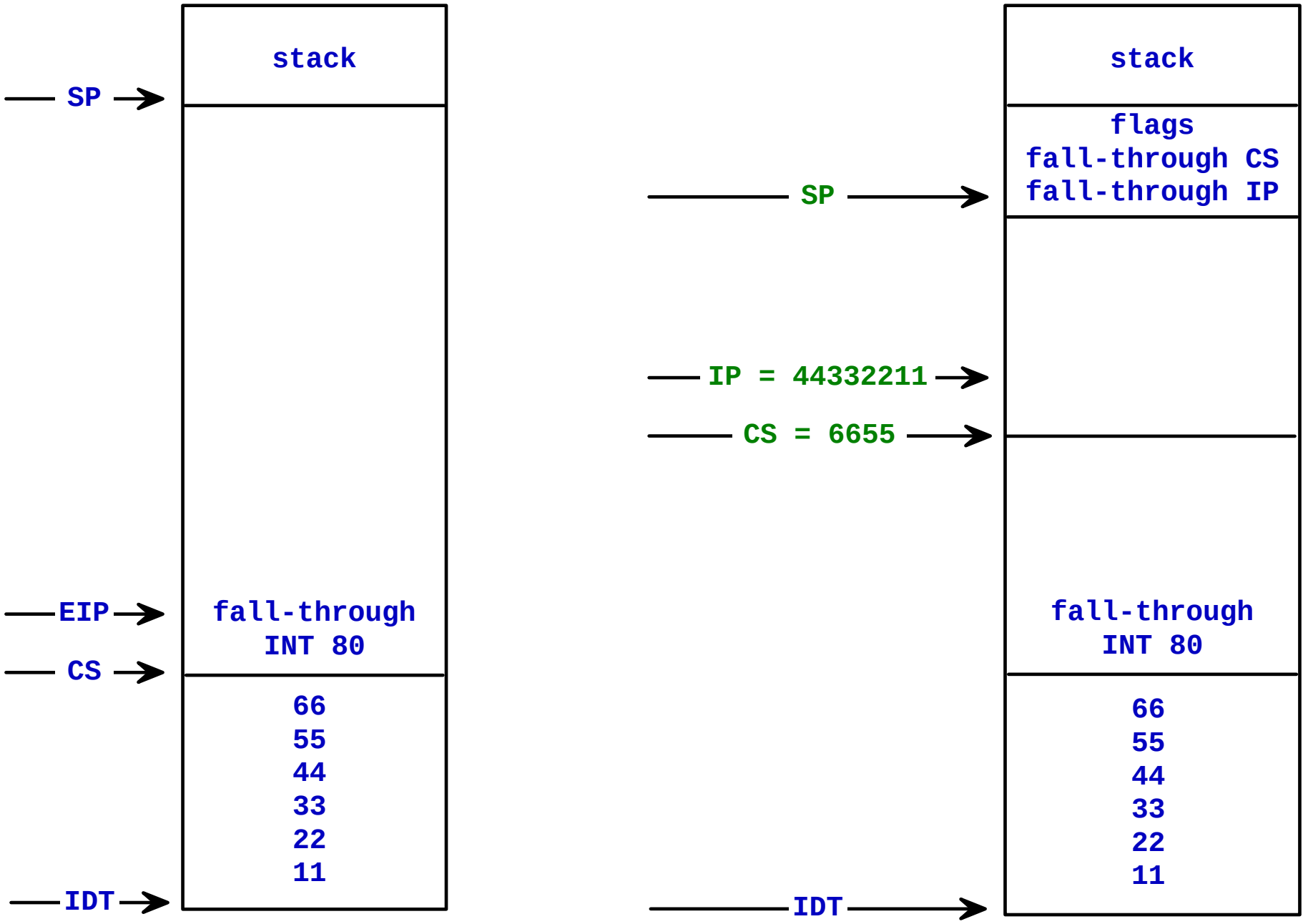
- **`INT`** instruction
- Operates exactly as hardware interrupt

Software Interrupt Instructions

`INT type`	`INT 80H`	`PUSH flags`
		`IF ← 0 ; block further interrupts`
		`TF ← 0 ; no debug mode`
		`PUSH CS`
		`PUSH EIP`
		`CS:EIP ← IDT values for interrupt 80`

`IRET none`	`IRET`	`POP EIP`
		`POP CS`
		`POP flags`

Interrupt


stack
SP
EIP
fall-through
INT 80
CS
66
55
44
33
22
11
IDT
stack
flags
fall-through CS
fall-through IP
SP
IP = 44332211
CS = 6655
fall-through
INT 80
66
55
44
33
22
11
IDT

Processor Control Instructions

STC	CF ← I	Set carry flag
CLC	CF ← 0	Clear carry flag
CMC	CF ← not(CF)	Complement carry flag
STD	DF ← 1	Set direction flag
CLD	DF ← 0	Clear direction flag
STI	IF ← I	Set interrupt flag
CLI	IF ← 0	Clear interrupt flag
HLT	None	CPU stops – requires reset
WAIT	None	Enter wait state
NOP	None	No operation

LEA

Load Effective Address

Similar to **MOV**

Copies address (pointer) of memory location

Does not access memory

```
LEA dest,[EA]
```

dest $\xleftarrow[\text{32-bits}]{}$ EA

```
LEA BX,[x]
```

BX $\xleftarrow[\text{32-bits}]{}$ &(x)

```
LEA AX,[1234]
```

AX $\xleftarrow[\text{32-bits}]{}$ 1234

```
LEA CX,[SI+12]
```

CX $\xleftarrow[\text{32-bits}]{}$ SI+12

```
LEA DX,[BX+DI]
```

DX $\xleftarrow[\text{32-bits}]{}$ BX+DI

Moving Data Around

```
MOV ESI,1122                ; ESI ← 1122
MOV [0000],ESI              ; [DS:0000] ← 1122
MOV EBX,3344                ; EBX ← 3344
MOV [EBX],ESI               ; [DS:3344] ← 1122
MOV [EBX+ESI],EBX           ; [DS:4466] ← 3344
LEA EBX,[EBX+ESI]           ; EBX ← 4466
MOV EDI,[EBX]               ; EDI ← [DS:4466]
MOV ES,EDI                  ; ES ← [DS:4466]
MOV EAX,[EBX+2]             ; EAX ← [DS:4468]
```

LDS and LES

Loads 48-bit logical address of type DS:EA

LDS dest, [EA]

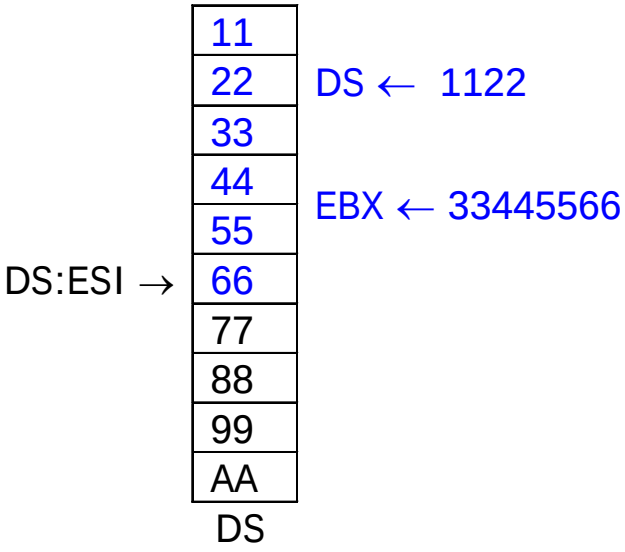
dest $\xleftarrow[\text{32-bits}]{}$ **[EA]**

DS $\xleftarrow[\text{16-bits}]{}$ **[EA+4]**

LDS EBX, [ESI]

EBX $\xleftarrow[\text{32-bits}]{}$ **[DS:ESI]**

DS $\xleftarrow[\text{16-bits}]{}$ **[DS:ESI+4]**



LES dest, [EA]

dest $\xleftarrow[\text{32-bits}]{}$ **[EA]**

ES $\xleftarrow[\text{16-bits}]{}$ **[EA+4]**

Switching Data Tables

```
/* DO ARITHMETIC WITH DS:BX = 1111:2222 */
/* SWITCH DATA TABLES */
MOV [ESI], 4444                 ; [ESI] ← 4444
MOV [ESI+4], 3333               ; [ESI+4] ← 3333
PUSH DS                         ; ESP ← ESP - 4
                                ; [SS:ESP] ← 1111
PUSH BX                         ; ESP ← ESP - 4
                                ; [SS:ESP] ← 2222
LDS EBX,[SI]                    ; EBX ← 4444
                                ; DS ← 3333
/* DO ARITHMETIC WITH DS:BX = 3333:4444 */
/* SWITCH BACK TO FIRST DATA TABLE */
POP [ESI]                       ; [ESI] ← 2222
                                ; ESP ← ESP + 4
POP [ESI+4]                     ; [ESI+4] ← 1111
                                ; ESP ← ESP + 4
LDS EBX,[ESI]                   ; EBX ← 2222
                                ; DS ← 1111
```