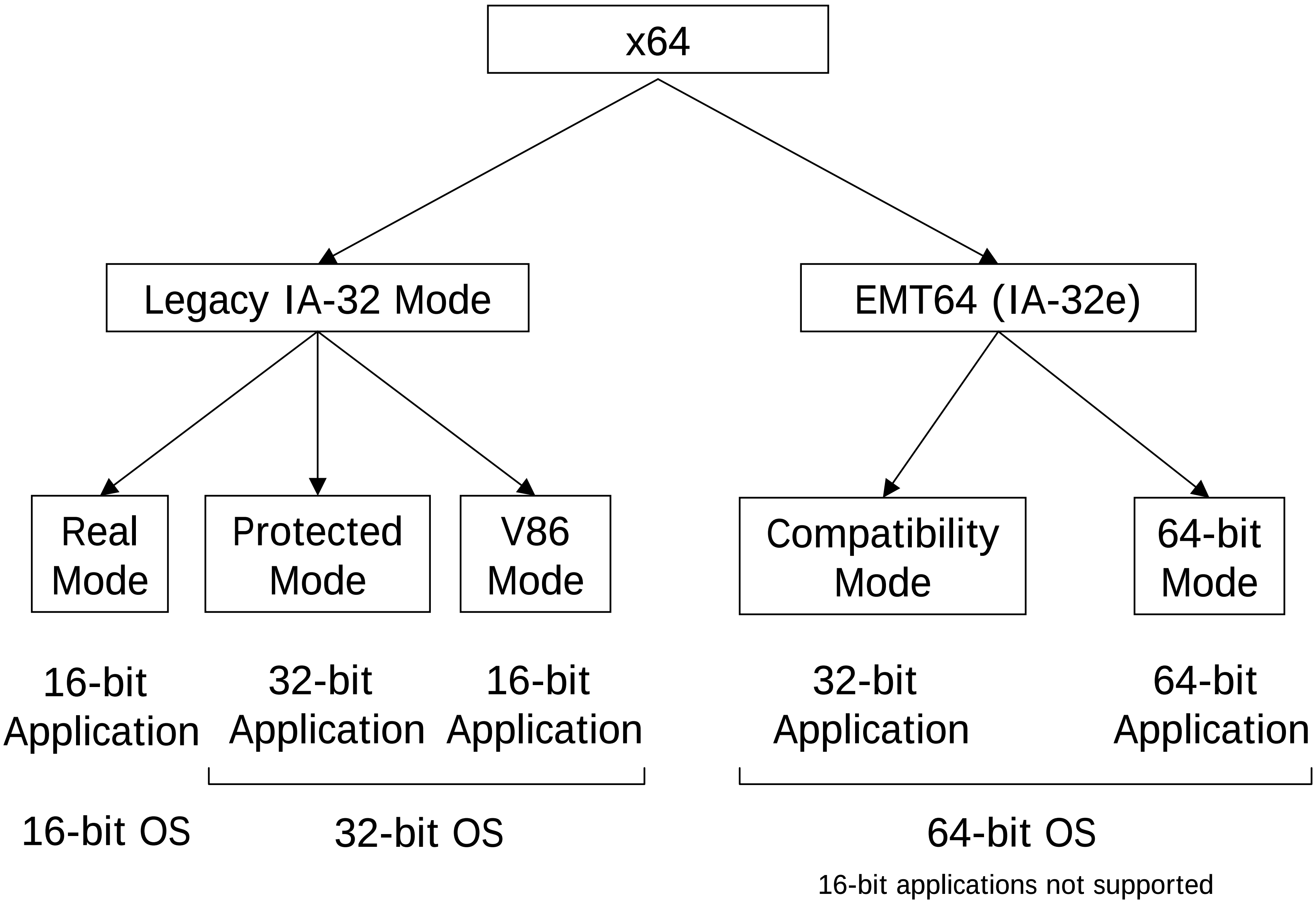
Basic Intel x86 Assembly Language Programming

Intel x86 Processor Family

Processor	Integer Width (bits)	Physical Address (bits)	Year	Features
8086	16	20	1978	16-bit data I/O bus
8088	16	20	1979	8-bit data I/O bus Used in IBM PC-XT
286	32	24	1984	Used in IBM PC-AT
386	32	32	1985	First IA-32 processor Intel design standard Extensive OS support
486	32	32	1989	FPU + cache
Pentium	32	32	1993	Dual ALUs
Pentium II – III – 4	32 or 64	36 to 64	1995 to 2000	Multiple ALUs Reorders instructions for optimum efficiency
Multicore	32 or 64	36 to 64	2005	Multiple P4s on one chip

Operating Modes for Intel x64 Processors



16-bit Assembly in Popular Operating Systems

Native support for 16-bit DOS programs

- Run at command line interface (CLI)
- 16-bit versions of DOS
- 32-bit versions of Windows (95/98/2k/XP/Vista/7)

No native support for 16-bit programs

- 32-bit and 64-bit versions of Linux
- 32-bit and 64-bit versions of MacOS
- 64-bit versions of Windows (XP/Vista/7)

Emulation environment for DOS

- Application program provides standard DOS CLI window
- Supports 16-bit programs and DOS system calls
- DOSbox
  - Freeware available for host operating systems Windows, Linux, Mac, …
  - Download from http://www.dosbox.com
  - Instructions and utilities on course website

Executable Programs Types for x86

8086

16-bit integers / 20-bit addresses

DOS

COMMON file (`file.com`)

Simple program up to 64 KB in size

DOS EXE file (`file.exe`)

Complex program up to 640 KB in size

Program links together separate file modules

IA-32 (i386)

32-bit or 64-bit integers and addresses

Linux

ELF file (`a.out`)

Executable and Linkable Format

Windows

WIN32 file (`file.exe, file.dll, ...`)

Running Machine Language Program

Program list

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
...

CPU

- Fetches next instruction in list
- Decodes fetched instruction
- Executes decoded instruction

Instructions in RAM

Instruction	byte	Address
Instruction 5	byte	A+11
	byte	A+10
Instruction 4	byte	A+9
	byte	A+8
Instruction 3	byte	A+7
Instruction 2	byte	A+6
	byte	A+5
	byte	A+4
Instruction 1	byte	A+3
	byte	A+2
	byte	A+1
	byte	A

Instruction Set Architecture

Typical Intel machine instruction

Operation	destination	source

Examples

```
MOV destination, source
    destination ← source (copy)

ADD destination, source
    destination ← destination + source

SUB destination, source
    destination ← destination - source
```

Intel x86 General Registers

8086

8-bit access

AL, BL, CL, DL,
AH, BH, CH, DH

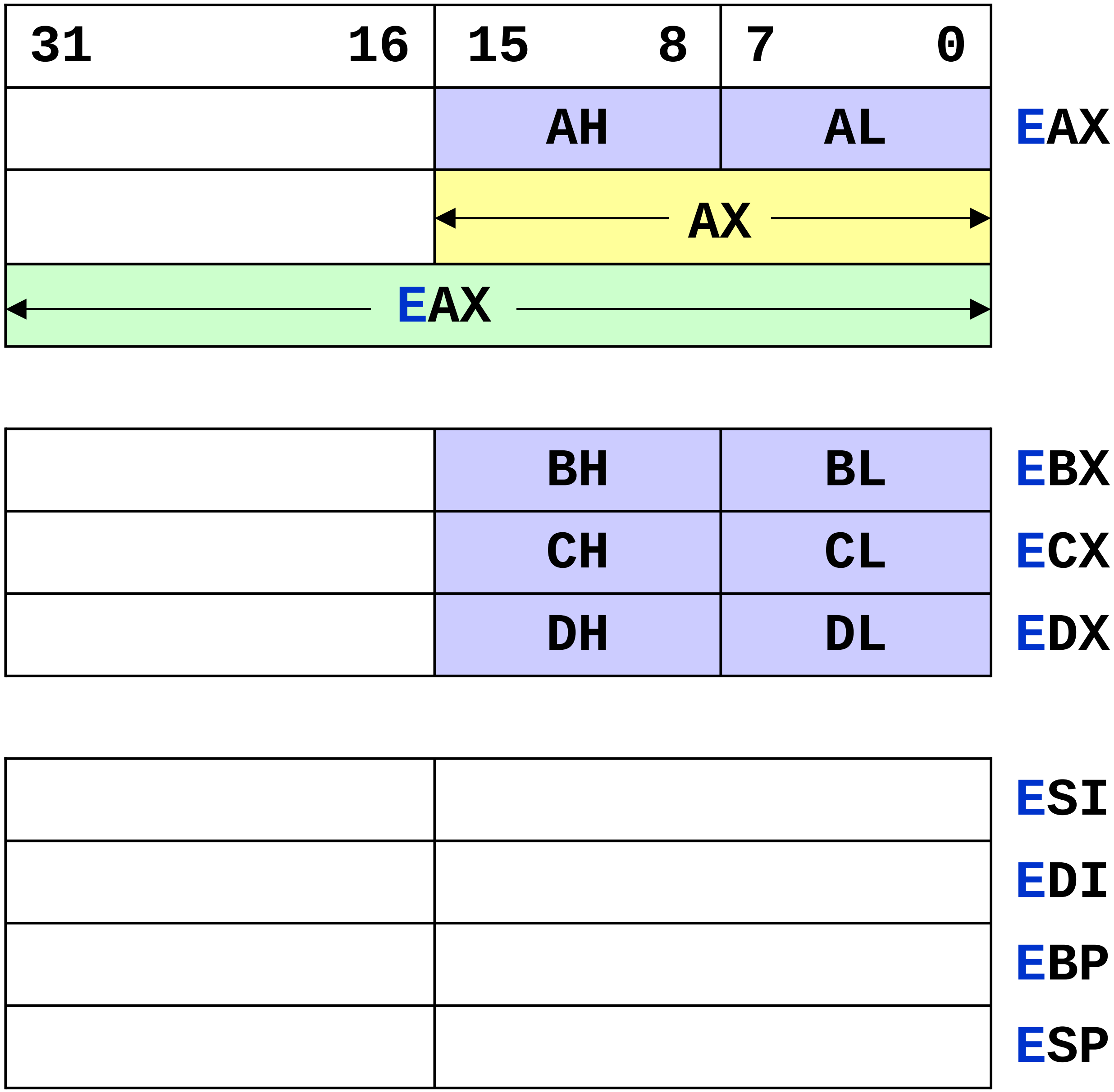
16-bit registers

AX, BX, CX, DX,
SI, DI, BP, SP

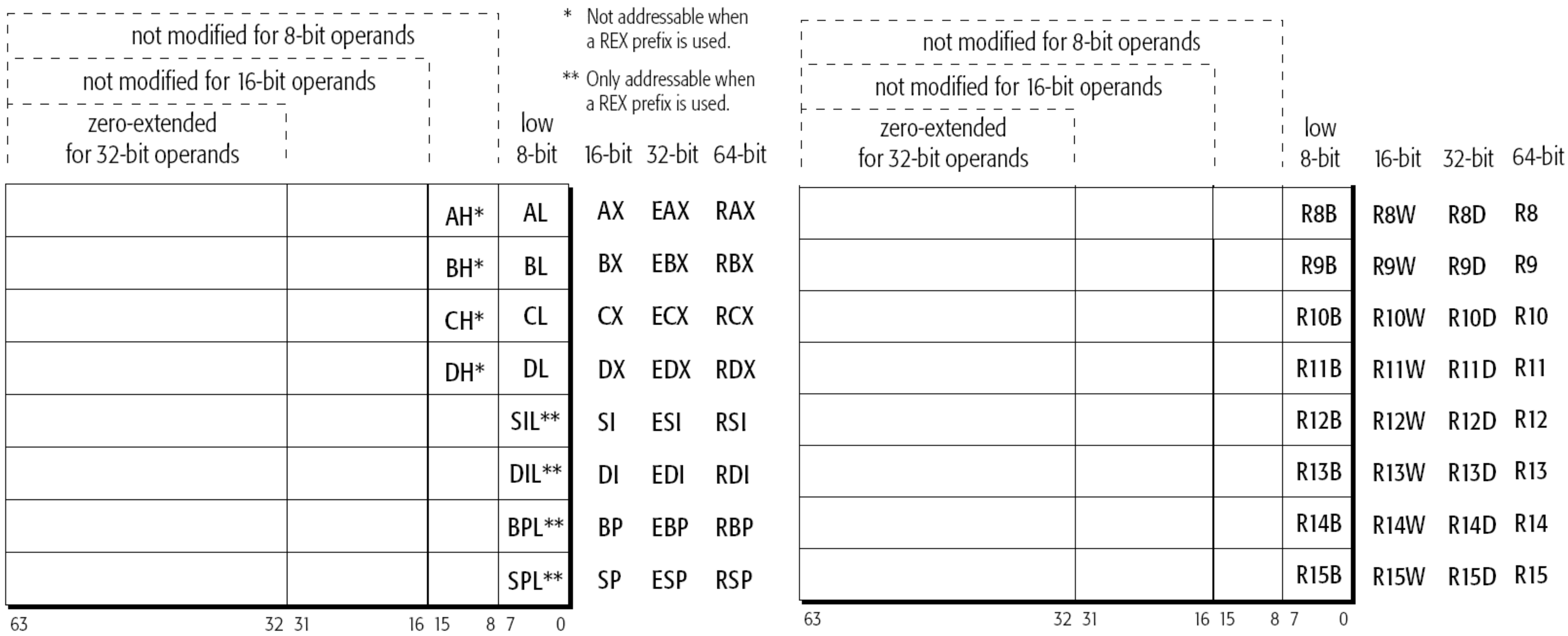
386

32-bit registers

EAX, EBX,
ECX, EDX,
ESI, EDI,
EBP, ESP



Intel x64 General Registers



Register accesses

64-bit operations access entire register

32-bit operations access lower 32-bits of 64-bit registers (default)

16-bit operations access lower 16-bits of 64-bit registers (where permitted)

8-bit operations access lower 8-bits of 64-bit registers

8086 Status Register

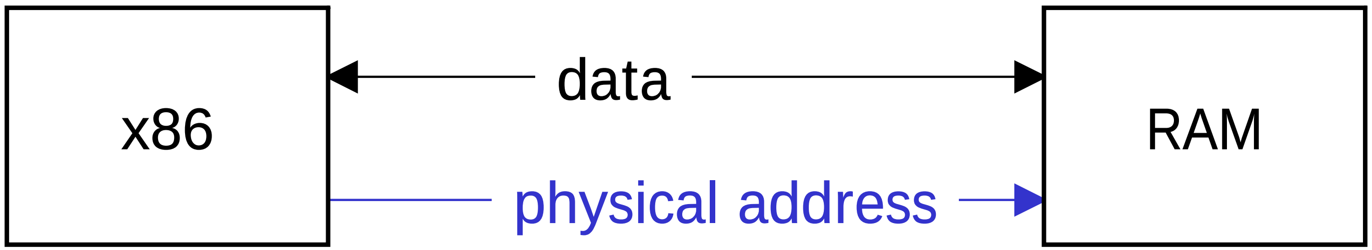
Bit Position	Name	Function
0	CF	Carry Flag — set on unsigned overflow
2	PF	Parity Flag
4	AF	Auxiliary Flag — set on overflow from 4 low bits of AL
6	ZF	Zero Flag — set on zero result
7	SF	Sign Flag — set on negative result
8	TF	Stops after next instruction and resets TF
9	IF	Interrupt Enable
10	DF	Autodecrement string instructions
11	OF	Overflow Flag — set on signed overflow

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

16 bit register

x86 Data and Address Ranges

OS	Data bits	Unsigned Integer	Signed Integer	Physical Memory Addresses
DOS	16	0 to 65,535	-32768 to +32767	00000 to FFFFF 1,048,576B (1 MB)
Windows Linux	32	0 to 4,294,967,295	-2,147,483,648 to 2,147,483,647	00000000 to FFFFFFFF 4,294,967,296B (4 GB)



Program in Memory

Program data

- Constants
- Variables
- Tables
- Structures
- User stack

Procedures / functions

- Control blocks
- Data access
  - Load operations — RAM to register
  - Store operations — register to RAM
  - I/O operations
- Calculation (ALU)
  - Arithmetic operations
  - Logic operations

Main Memory

Main Memory
Data
Machine Language Instructions Compiled from high level program Assembled from assembly language program

Simple 16-Bit Assembly Program

Instructions written as list

```
MOV DX, value_1          ; DX ← value_1
MOV AX, [address_1]      ; AX ← [address_1]
ADD AX, DX               ; AX ← AX + DX
MOV BX, address_2        ; BX ← address_2
MOV [BX], AX             ; [address_2] ← AX
```

CPU

Fetches next instruction in list

Decodes fetched instruction

Executes decoded instruction

repeat

Simple DOS Assembly Language Programming

1. **Write program**

   Combination of assembly language instructions

2. **Source file**

   Save program listing in text file

   Typical name — **`program.asm`**

3. **Assemble program**

   Run assembler on **`source.asm`**

   Similar to compiling **`C++`** program

   Convert **`source.asm`** to

   Executable file **`program.com`** (for **`com`** program)

   Object file **`program.obj`** — (for **`exe`** program)

4. **Link object files (for exe program)**

   Convert **`program.obj`** to **`program.exe`**

Disassembler — converts executable to assembly language

NASM Assembler

Assembler system

Free open source assembler + disassembler

Versions for DOS, Windows, Linux, Unix, MAC, ...

Installation for Windows XP

Download **`nasmXXX.zip`** from course web site

Includes full documentation

Unpack to some directory, for example **`c:\nasm`**

`nasm.exe` is assembler

`Nasm -h` for help

Installation for Ubuntu Linux

`sudo apt-get install nasm`

`man nasm` for help

`ndisasm.exe` is disassembler

Usual Command Line Options

```
nasm [-f <format>] <filename> [-o <output>]
```

Valid output formats for `-f` :

bin	(default) flat-form binary files (DOS .com)
obj	MS-DOS 16-bit/32-bit OMF object files
elf	UNIX/Linux object files
win32	Microsoft Win32 (i386) object files

The `-o` option

NASM chooses default name of output file

Depends on object file format

Example — DOS Common File

Write example.asm in text editor

Start notepad, notepad++, vim, emacs, ...

Enter assembly instructions

```
MOV AX, BX
ADD AX, CX
```

Save file in directory **`...\nasm\example.asm`**

Open command window

Change to directory **`...\nasm>`**

Assemble file

`…\nasm>nasm example.asm -o example.com <ENTER>`

Produces default flat-form binary file **`example.com`**

MOV

MOV dest, src $\qquad$ dest ← src

src = reg / mem / imm $\qquad$ dest = reg / mem

NOT LEGAL: MOV mem, mem

```
MOV AX, 1234           AX <--16-bits-- 1234

MOV AX, BX             AX <--16-bits-- BX

MOV AX, [address]      AX <--16-bits-- [address]

MOV AL, [address]      AL <--8-bits--  [address]
```

Bitwise Logical Operations

NOT dest	dest ← not dest NOT 11001111 → 00110000
AND dest, src	dest ← dest AND src 10110000 AND 11001111 → 10000000
OR dest, src	dest ← dest OR src 10110000 OR 11001111 → 11111111
XOR dest, src	dest ← dest XOR src 10110000 XOR 11001111 → 01111111
TEST dest, src	dest AND src (no write to dest) set flags SF and ZF

Using Boolean Operations

```
XOR AX,AX         ; AX ← 0

MOV AX,1122       ; AX ← 1122
AND AX,00FF       ; AX ← 0022

MOV AX,1122       ; AX ← 1122
TEST AX,8000      ; tests high order bit
                  ; ZF ← 1 , SF = 0

MOV AX,0001       ; AX ← 0001
NOT AX            ; AX ← FFFE
```

Unsigned Integers

n-bit number in usual binary representation

Represents value from **0** to 2^n-1

Integers determined modulo 2^n

n=3	
7	111
6	110
5	101
4	100
3	011
2	010
1	001
0	000

Overflow

$a + b > 2^n-1$

Carry Flag is set

CF	3-Bit Integer
0	111 + 001
1	000

CF	3-Bit Integer
0	000 - 001
1	111

Signed Numbers — 1

n-bit number in 2's complement representation

Represents value from -2^{n-1} **to** $+2^{n-1}-1$

Integers determined modulo 2^n

Overflow

Carry-in not equal to carry-out at highest order

Overflow Flag is set

n=3	
+3	011
+2	010
+1	001
0	000
-1	111
-2	110
-3	101
-4	100

Signed Numbers — 2

Upper bit = 0 for positive numbers

Upper bit = 1 for negative numbers

n = 3	
+3	011
+2	010
+1	001
0	000
-1	111
-2	110
-3	101
-4	100

n = 4	
+7	0111
…	…
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
…	…
-8	1000

Signed Numbers — 3

OF	CO	CI	3-Bit Integer	Decimal
			111 + 001	-1 + 1
0	1	1	000	0

OF	CO	CI	3-Bit Integer	Decimal
			111 - 001	-1 - 1
0	0	0	110	-2

OF	CO	CI	3-Bit Integer	Decimal
			011 + 001	3 + 1
1	0	1	100	4

Data Conversion

CBW — convert byte to word with sign extension

CWD — convert word to double with sign extension

CBW	**If AL < 80H, then AH ← 0** **00000000 0xxxxxxx ← xxxxxxxx 0xxxxxxx**
	If AL > 7F, then AH ← FFH **11111111 1xxxxxxx ← xxxxxxxx 1xxxxxxx**
CWD	**If AX < 8000H, then DX ← 0**
	If AX > 7FFFH. then DX ← FFFFH

Add/Subtract

ADD dest, src	**ADD BX,CX**	**BX ← BX + CX**
ADC dest, src	**ADC SI,DX**	**SI ← SI + DX + CF**
SUB dest ,src	**SUB SI,DX**	**SI ← SI - DX**
SBB dest ,src	**SBB SI,DX**	**SI ← SI – DX - CF**
INC dest	**INC BL**	**BL ← BL + 1**
DEC dest	**DEC BL**	**BL ← BL - 1**
NEG dest	**NEG BL**	**BL ← 0 - BL**
CMP dest ,src	**CMP AL,AH**	**AL – AH (no write to dest)** **Set flags CF,OF,SF,ZF**

Long Integer Add/Sub

Long integers in **DX.AX** and **DI.SI**

ADD AX, SI	**AX ← AX + SI**
	CF ← carry
ADC DX, DI	**DX ← DX + DI + CF**
	CF ← overflow
SUB AX, SI	**AX ← AX - SI**
	CF ← borrow
SBB DX, DI	**DX ← DX - DI - CF**
	CF ← overflow

Multiplication / Division

MUL source	MUL BL	AX ← AL*BL
	MUL CX	DX.AX ← AX*CX
IMUL source	IMUL BL	AX ← AL*BL
	IMUL CX	DX.AX ← AX*CX
DIV source	DIV BL	AL ← AX / BL AH ← AX % BL
	DIV CX	AX ← DX.AX / CX DX ← DX.AX % CX
IDIV source	IDIV BL	AL ← AX / BL AH ← AX % BL
	IDIV CX	AX ← DX.AX / CX DX ← DX.AX % CX

Program Fragment

```
MOV     AX,1122
MOV     BX,3344
SUB     BX,AX
MOV     CX,0003
IMUL    CX
```

		AX=0000	BX=0000	CX=0000
MOV	AX,1122	AX=1122	BX=0000	CX=0000
MOV	BX,3344	AX=1122	BX=3344	CX=0000
SUB	BX,AX	AX=1122	BX=2222	CX=0000
MOV	CX,0003	AX=1122	BX=2222	CX=0003
IMUL	CX	AX=3366	BX=2222	CX=0003

Shift Instructions

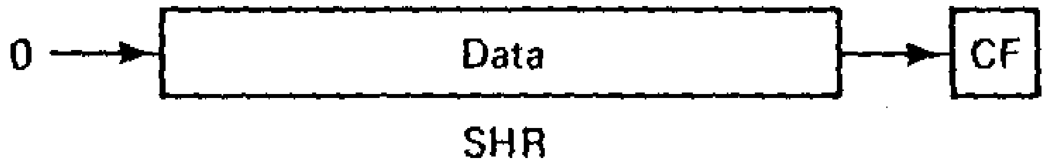
SHR dest, times

dest = register or memory

times

Number or **CL**

8086 — not used (illegal)

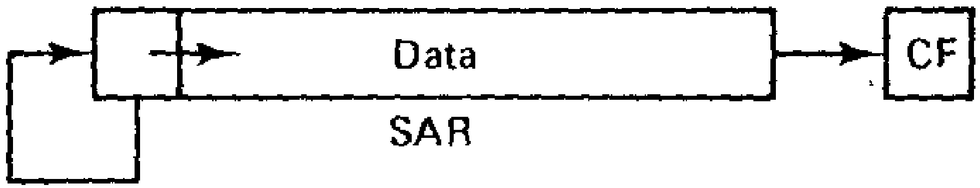


SAR dest, times

Shift Arithmetic Right

Shift bits right

Preserves sign



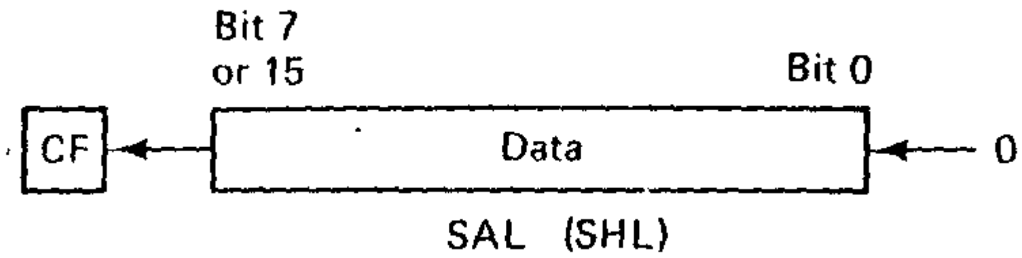
SAL dest, times

Shift Arithmetic Left

Shift left

Copies sign bit to CF

OF = 1 if sign bit changes



Rotate Instructions

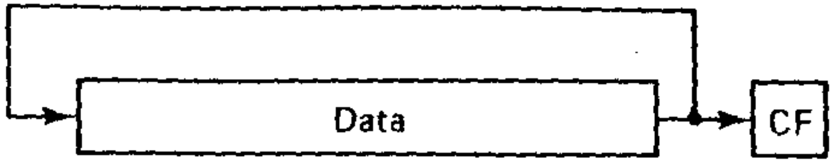
ROL dest, times

Rotate Left

ROL

ROR dest, times

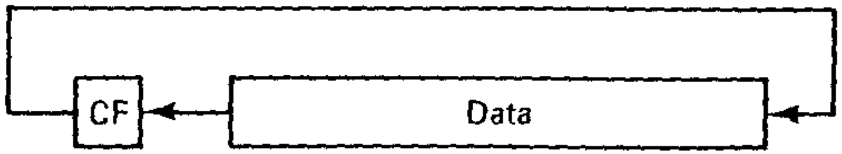
Rotate right



ROR

RCL dest, times

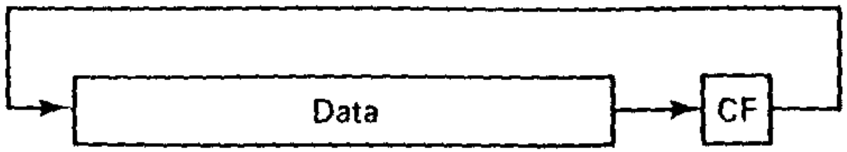
Rotate carry left



RCL

RCR dest, times

Rotate carry right



RCR

Example

```
SHL AX, 2       ; not legal in 8086
MOV CL, 4
ROR BX, CL      ; legal in all x86
```

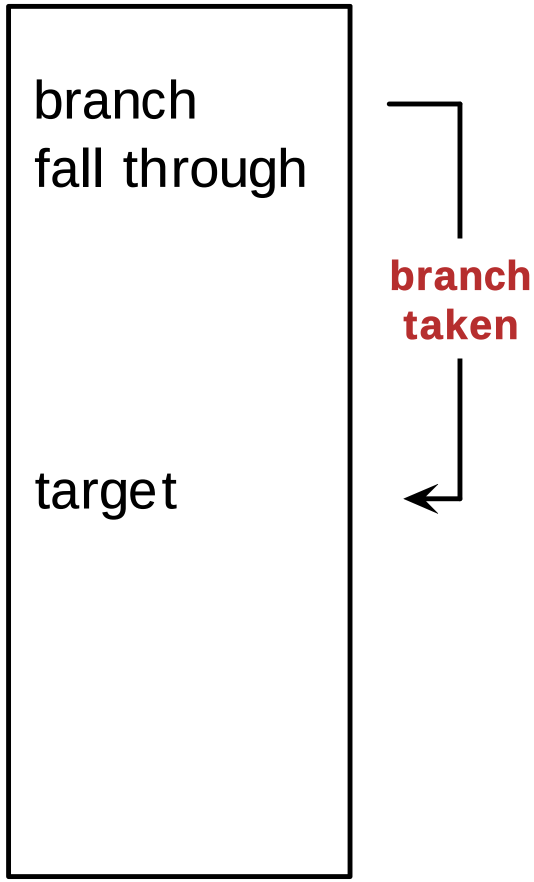
Branch Instructions

Changes program execution order

Program chooses next instruction

Used to build control blocks

`for`, **`while`**, **`if`**, **`switch`**, ...



Fall-through

Instruction following branch in program listing

Next instruction if branch **not taken**

Target

Next instruction if branch **taken**

Jump Instruction

Unconditional branch

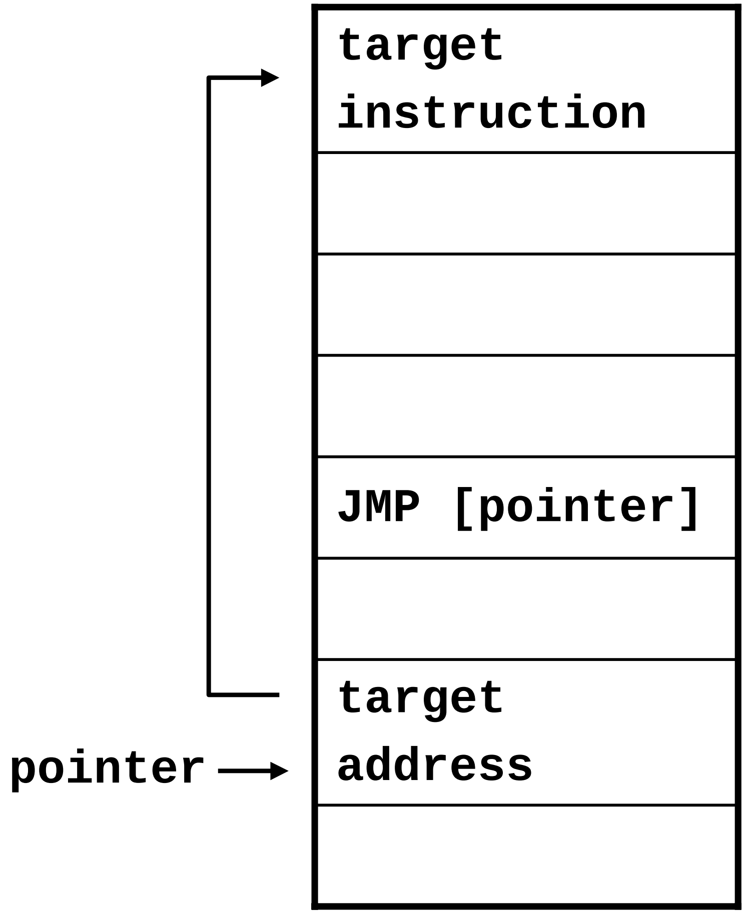
```
JMP target

        MOV     AX,1122
        MOV     BX,3344
        JMP     L1
        SUB     BX,AX
L1:     MOV     CX,0003
        IMUL    CX
```

Indirect unconditional branch

```
JMP [pointer]
```

; **pointer** = memory address of stored target address



Conditional Branch

ALU operations set flags in status word

Conditional branch

Test flags

Jumps if **`flag condition = 1`**

General form

```
Jcc target      ; Jcc = any conditional branch
                ;     = JC, JE, JNC, ...
```

Mnemonic	Condition	Test
JC	Carry	CF = 1
JE	Equal	ZF = 1
JNC	Not carry	CF = 0
JNE	Not equal	ZF = 0

Unsigned Compare

Mnemonic	Condition
JA	Unsigned greater than
JAE	Unsigned greater than or equal
JB	Unsigned less than
JBE	Unsigned less than or equal

Signed Compare

Mnemonic	Condition
JG	Greater than
JGE	Greater than or equal
JL	Less than
JLE	Less than or equal
JO	Overflow
JS	Sign
JNO	Not overflow
JNS	Not sign

Simple Loop

```
      XOR AX,AX    ; zero accumulator (AX ← 0)
      MOV BX,0001  ; BX ← 1
L1:   ADD AX,BX    ; add BX to accumulator
      INC BX       ; BX++
      CMP BX,0005  ; set flags according to value
                   ;     of BX – 5
      JLE L1       ; loop to L1 if BX ≤ 5
```

Loop Instruction

LOOP target	CX ← CX - 1 IF CX ≠ 0, JMP target
LOOPZ target	CX ← CX - 1 IF (CX ≠ 0) AND (ZF = 1), JMP target
LOOPNZ target	CX ← CX - 1 IF (CX ≠ 0) AND (ZF = 0), JMP target

```
    XOR AX,AX      ; zero accumulator
    MOV CX,0005    ; CX ← 5
L1: ADD AX,CX      ; add CX to accumulator
    LOOP L1        ; CX-- and loop if CX > 0
```

NASM Syntax for *.com Programs

NASM programs include

x86 assembly instructions

```
label: instruction operands ; comment
```

System calls

Section declarations

`section .data`	Initialized data
`section .bss`	Non-initialized data and buffers
`section .text`	Instructions

Variable names and line labels (case sensitive)

Number values

Default = decimal (written as **`123`** or **`123d`**)

Hexadecimal value written as **`0x1234`** or **`1234h`**

Binary value written as **`01101100b`**

Null-terminated string data in quotes: 'this is a string', 0

Template for *.**com** Program

```
      ORG 0x100             ; common file organization
section .data
    ; initialized data and variables
section .bss
    ; uninitialized data and variables
section .text
    ; user instructions
    mov ax,4C00h            ; exit code
    int 21h                 ; DOS system call to end
                            ; program
```

Data Declarations

`label: db 0x55,0x56,0x57`

Stores HEX bytes **`55 56 57`** in memory

`label` points to **`0x55`**, **`label+1`** points to **`0x56`**, ...

`label: db 'ABC'`

Stores HEX bytes **`41 42 43`** (ASCII codes) in memory

`label: dw 0x22, 'ABC'`

Stores string **`22 00 41 42 43 00`** in memory

1 byte per character — appends **`00`** to odd-length string

`label: dw 0x1234, 0x5678`

Stores HEX bytes **`34 12 78 56`** in memory

In order of word definition (little-endian order in each word)

`label: dd 0x12345678,2`

Stores HEX bytes **`78 56 34 12 02 00 00 00`** in memory

In order of dword definition (little-endian order in each dword)

Uninitialized Data

; in data section

```
zerobuf: times 64 db 0
```

; Writes 64 0-bytes in variable named **zerobuf**

; in BSS section

```
buffer:  resb  64
```

; Reserves 64 bytes in variable named **buffer**

```
wordvar:       resw  1
```

; Reserves 1 word in variable named **wordvar**

; assign value to variable in text section

```
MOV [wordvar], 0x1122
```

Disassembler

```
C:\nasm\programs\examples>ndisasm -h

usage: ndisasm [-a] [-i] [-h] [-r] [-u] [-b bits] [-o origin]
               [-s sync...] [-e bytes] [-k start,bytes]
               [-p vendor] file
   -a or -i activates auto (intelligent) sync
   -u sets USE32 (32-bit mode)
   -b 16 or -b 32 sets number of bits
   -h displays this text
   -r or -v displays the version number
   -e skips <bytes> bytes of header
   -k avoids disassembling <bytes> bytes from position <start>
   -p selects the preferred vendor instruction set (intel, amd,
      cyrix, idt)
```

Disassembler

```
C:\nasm\programs\examples>ndisasm -e200h ex9.exe
00000000  B80000            mov ax,0x0
00000003  8ED8              mov ds,ax
00000005  55                push bp
00000006  89E5              mov bp,sp
00000008  81EC0400          sub sp,0x4
0000000C  B8004C            mov ax,0x4c00
0000000F  CD21              int 0x21
```

*.com Example

Edit `ex1.asm` in text editor

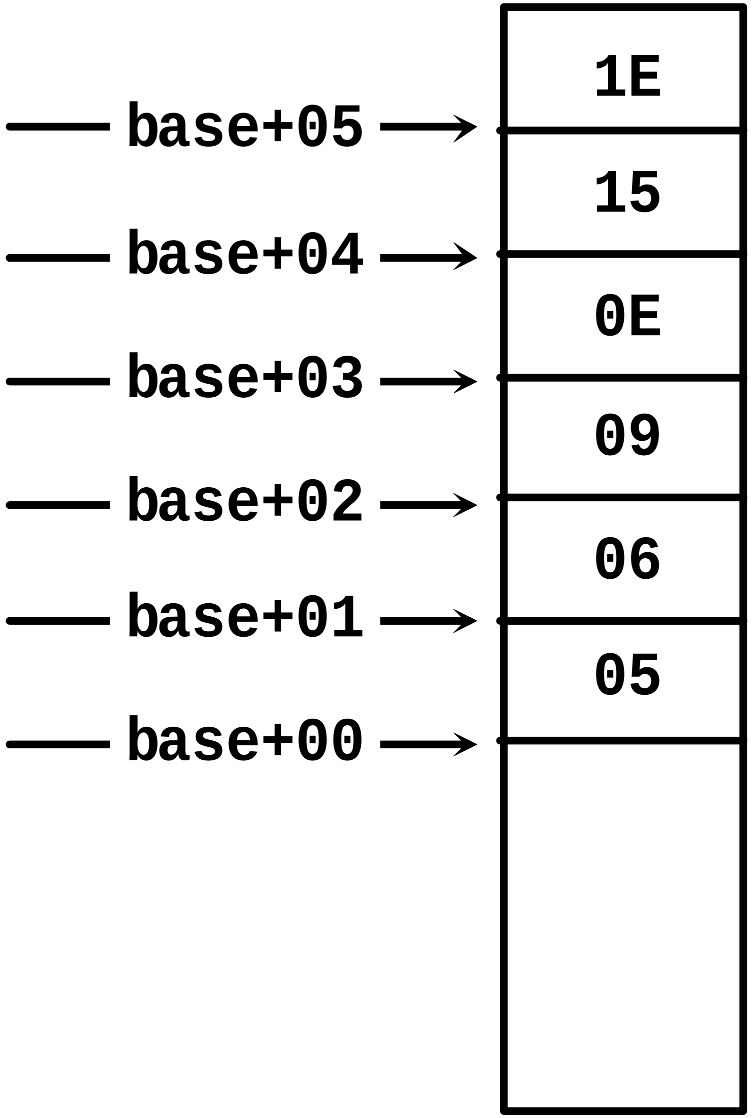
```
        ORG 0x100
    section .data
        v1: dw 0x0005   ; v1 = name of pointer to integer
    section .text
        XOR AX,AX       ; zero accumulator (AX ← 0)
        MOV BX,0001     ; BX ← 1
L1:     ADD AX,BX       ; add BX to accumulator
        INC BX          ; BX++
        CMP BX,[v1]     ; set CF,OF,SF,ZF
                        ; according to value of BX – 5
        JLE L1          ; loop to L1 if BX ≤ 5
        mov ax, 4C00h
        int 21h
```

```
C:\nasm>nasm ex1.asm –o ex1.com
```

NASM places executable program file `ex1.com` in directory `C:\nasm>`

Writing Values into a Table

```
    ORG 0x100
section .bss
    base resb 6
section .text
    MOV     SI, 0
L1: MOV     AX,SI
    IMUL    AL
    ADD     AL,5
    MOV     [base + SI],AL
    INC     SI
    CMP     SI,5
    JLE     L1
    mov     ax,4C00h
    int     21h
```



Arithmetic With Stored Data

```
    ORG 0x100
section .data
    table dw 1,2,3,4,5,6,7,8,9,10,0
section .text
    MOV     SI,table
    MOV     AX,0000
L1: ADD     AX,[SI]
    ADD     SI,2
    CMP     WORD [SI],0        ; WORD – operate on
                               ; 16-bit word = 2 bytes
                               ; from [SI+1].[SI]
    JNZ     L1
    mov     ax,4C00h
    int     21h
```

AX = 0 → 1 → 3 → 6 → 10 → 15 → 21 → 28 → 36 → 45 → 55

`if` Control Block

```
main()
{
  int x = 0, y = 2;
  if (x == y) {
    x = 2 * y;
    y = 2 * x;
  }
}
```

```
     ORG 0x100
section .bss
     x resw               ; allocate memory
                          ;  for integer x
     y resw               ; allocate memory
                          ;  for integer y
section .text
     MOV word [x],0       ; x ← 0
     MOV word [y],2       ; y ← 2
     MOV AX,[x]           ; AX ← x
     CMP AX,[y]           ; AX - y
     JNZ end              ; JMP IF NOT EQUAL
     MOV AX,[y]           ; AX ← y
     SHL AX,1             ; AX ← AX * 2
     MOV [x],AX           ; x ← AX
     MOV AX,[x]           ; AX ← x
     SHL AX,1             ; AX ← AX * 2
     MOV [y],AX           ; y ← AX
end: mov        ax,4C00h
     int        21h
```

for Loop

```
main()
{
   int i,j;
   for (i = 0; i < 10; i++){
      j = 3 * i;
   }
}
```

```
    ORG 0x100
section .bss
    i resw
    j resw
section .text
     MOV WORD [i],0
     MOV BX,3
L1:  CMP WORD [i],10
     JGE end              ; break on i ≥ 10
     MOV AX,[i]           ; AX ← i
     IMUL BX              ; AX ← AX * 3
     MOV [j],AX           ; j ← AX
     INC WORD [i]         ; i++
     JMP L1               ; loop
end: mov ax,4C00h
     int 21h
```

Compound `if` Control Block

```
main()
{
  int x = 0, y = 2, z = 3;
  if (x == y || y > z && x > z) {
    x = 2 * y;
  }
}
```

```
  ORG 0x100
section .bss
     x resw
     y resw
     z resw
section .text
     MOV WORD [x],0      ; x ← 0
     MOV WORD [y],2      ; y ← 2
     MOV WORD [z],3      ; z ← 3
     MOV AX,[x]          ; AX ← x
     CMP AX,[y]          ; CMP AX = x, y
     JZ L1               ; if x = y, JMP L1
     MOV AX,[y]          ; AX ← y
     CMP AX,[z]          ; CMP AX = y, z
     JLE end             ; if y <= z, JMP end
     MOV AX,[x]          ; AX ← x
     CMP AX,[z]          ; CMP AX = x, z
     JLE end             ; if x <= z, JMP end
L1:  MOV AX,[y]          ; AX ← y
     SHL AX,1            ; AX ← 2 * AX
     MOV [x],AX          ; x ← AX
end: mov ax,4C00h
     int 21h
```

C versus Assembly

```
main()
{
  int N, M = 1;
  for (N = 2 ; N <= 7 ; ++N){
    M = N * M;
  }
}
```

compile

```
      MOV WORD [M],1          ; M ← 1
      MOV WORD [N],2          ; N ← 2
L1:   CMP WORD [N],7          ; compare N, 7
      JG end                  ; break if N > 7
      MOV AX,[N]              ; AX ← N
      IMUL WORD [M]           ; AX ← AX * M
      MOV [M],AX              ; M ← AX
      INC WORD [N]            ; N++
      JMP L1                  ; loop
end:  mov ax,4c00h
      int 21h
```

rewrite in assembly language

```
     MOV CX,0007; counter CX ← 7
     MOV AX,0001; accumulator AX ← 1
L1:  IMUL CX    ; AX ← AX * CX
     LOOP L1    ; CX-- → loop if CX != 0
     MOV [M],AX ; M ← AX
```

String Instructions

Simple Byte Transfers

LODSB	Load String Byte	AL ← [SI] SI ← SI+1
STOSB	Store String Byte	[DI] ← AL DI ← DI+1

Simple Word Transfers

LODSW	Load String Word	AL ← [SI] AH ← [SI+1] SI ← SI+2
STOSW	Store String Word	[DI] ← AL [DI+1] ← AH DI ← DI+2

String Example

```
      org 0x100
section .data
     s_1 db "0123456789",0
     v_1 times 11 db 0
section .text
     MOV SI, s_1     ; SI ← pointer to s_1
     MOV DI, v_1     ; DI ← pointer to v_1
L1:  LODSB           ; AL ← [SI]
                     ; SI ← SI + 1
     CMP AL, 0
     JZ end
     SUB AL, 30      ; AL ← AL - 30 = numerical value of ASCII char
     STOSB           ; [DI] ← AL
                     ; DI ← DI + 1
     JMP L1          ; loop
end: STOSB           ; store \null
     mov ax,4C00h
     int 21h
```

XCGH

XCHG dest, src	**dest ↔ src**
XCHG AX, BX	**AX ↔ BX**
XCHG AL, AH	**AL ↔ AH**
XCHG AX, [SI]	**AX ↔ [SI]**

LEA

Load Effective Address

Similar to **MOV**

Copies pointer to data

Does not access memory

LEA dest, [EA] 　　 dest $\xleftarrow{\text{16-bits}}$ EA

LEA BX, [x] 　　 BX $\xleftarrow{\text{16-bits}}$ &(x)

Same as

MOV BX, x 　　 BX $\xleftarrow{\text{16-bits}}$ &(x)

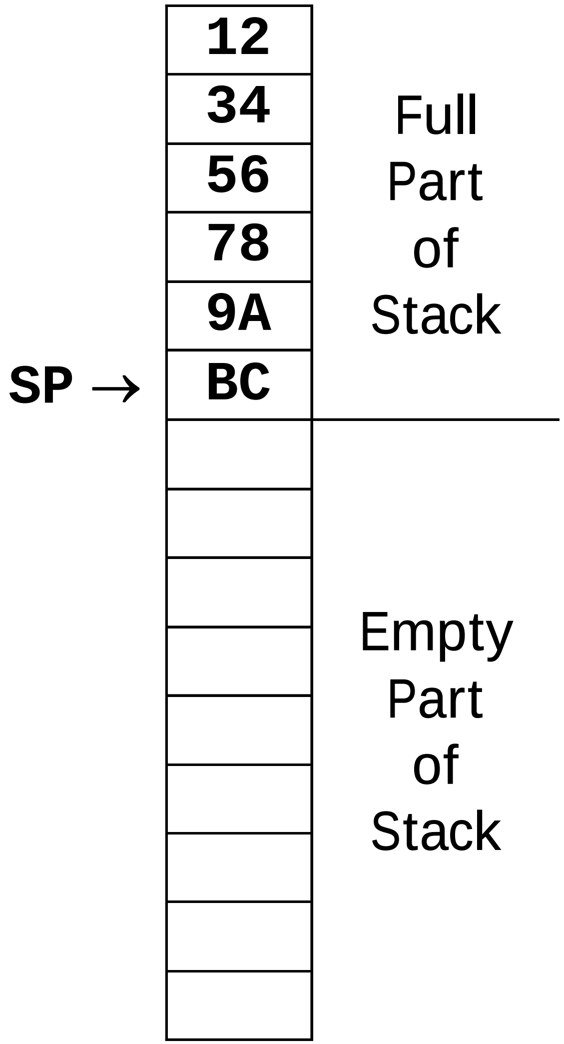
User Stack

x86 provides user stack mechanism

- Last In First Out (**LIFO**) buffer
- Store/Load full-length integer
  - DOS: 16-bit integer
  - Linux: 32-bit integer

Stack expands down

- Fills from top
- Register **SP** points to **TOP OF STACK**
- Last in location = **`[SP]`**
- CPU auto-updates **SP** after stack operation



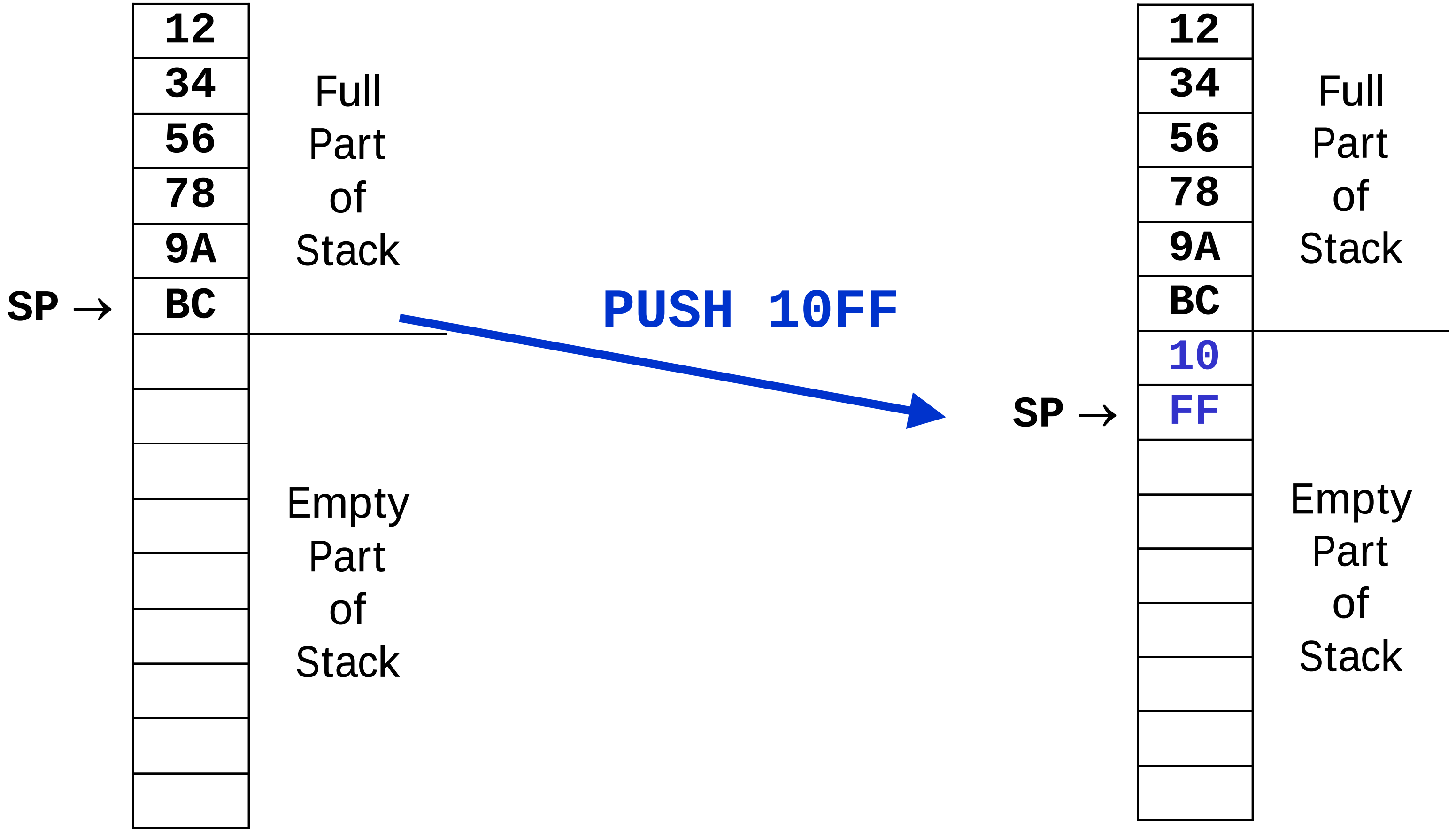
PUSH Operation — 1

PUSH src

$SP \leftarrow SP - 2$

$[SP] \xleftarrow[\text{16-bits}]{} src$

Example



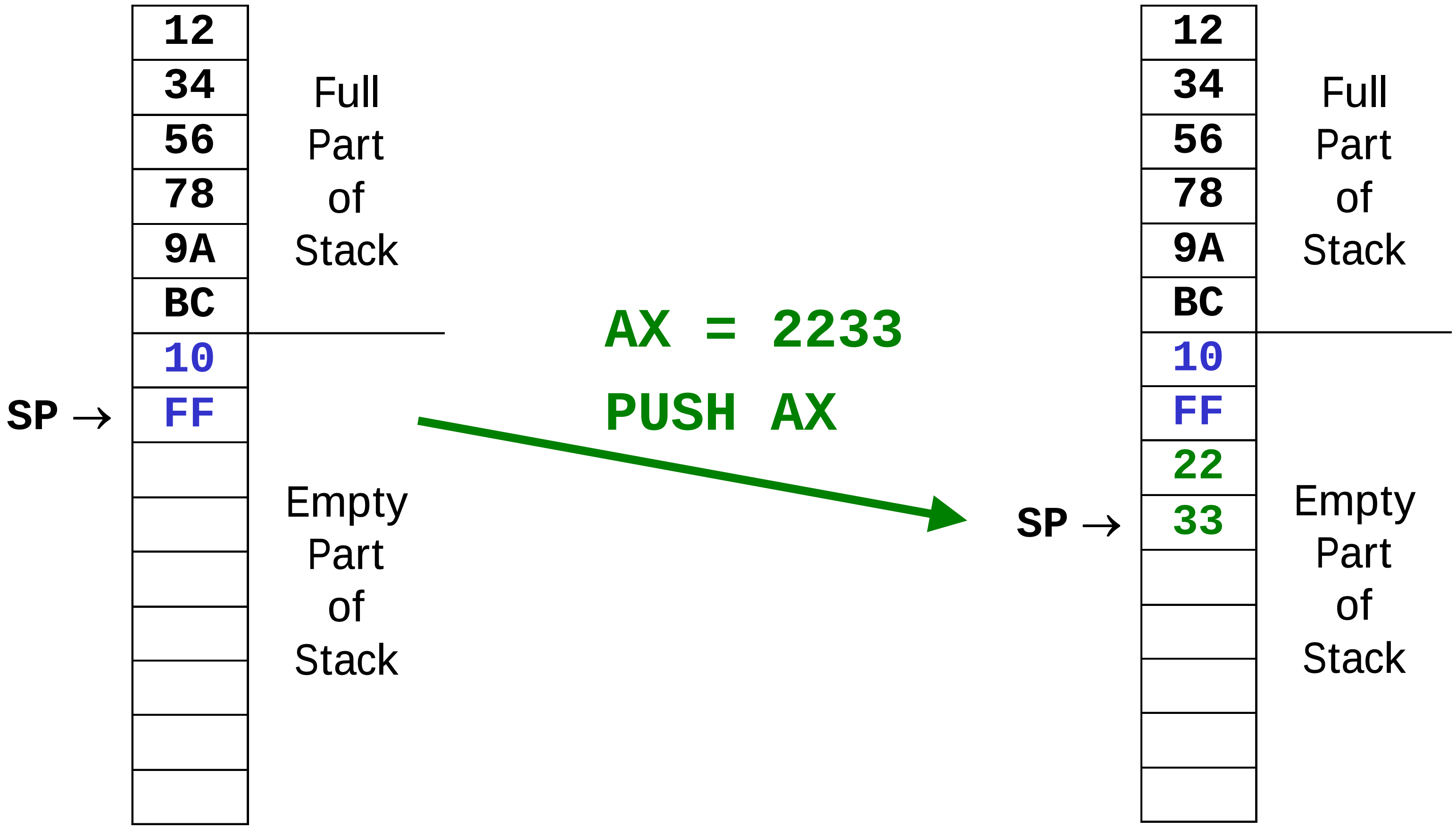
PUSH Operation — 2

PUSH src

SP ← SP - 2

[SP] $\xleftarrow[\text{16-bits}]{}$ src

Example



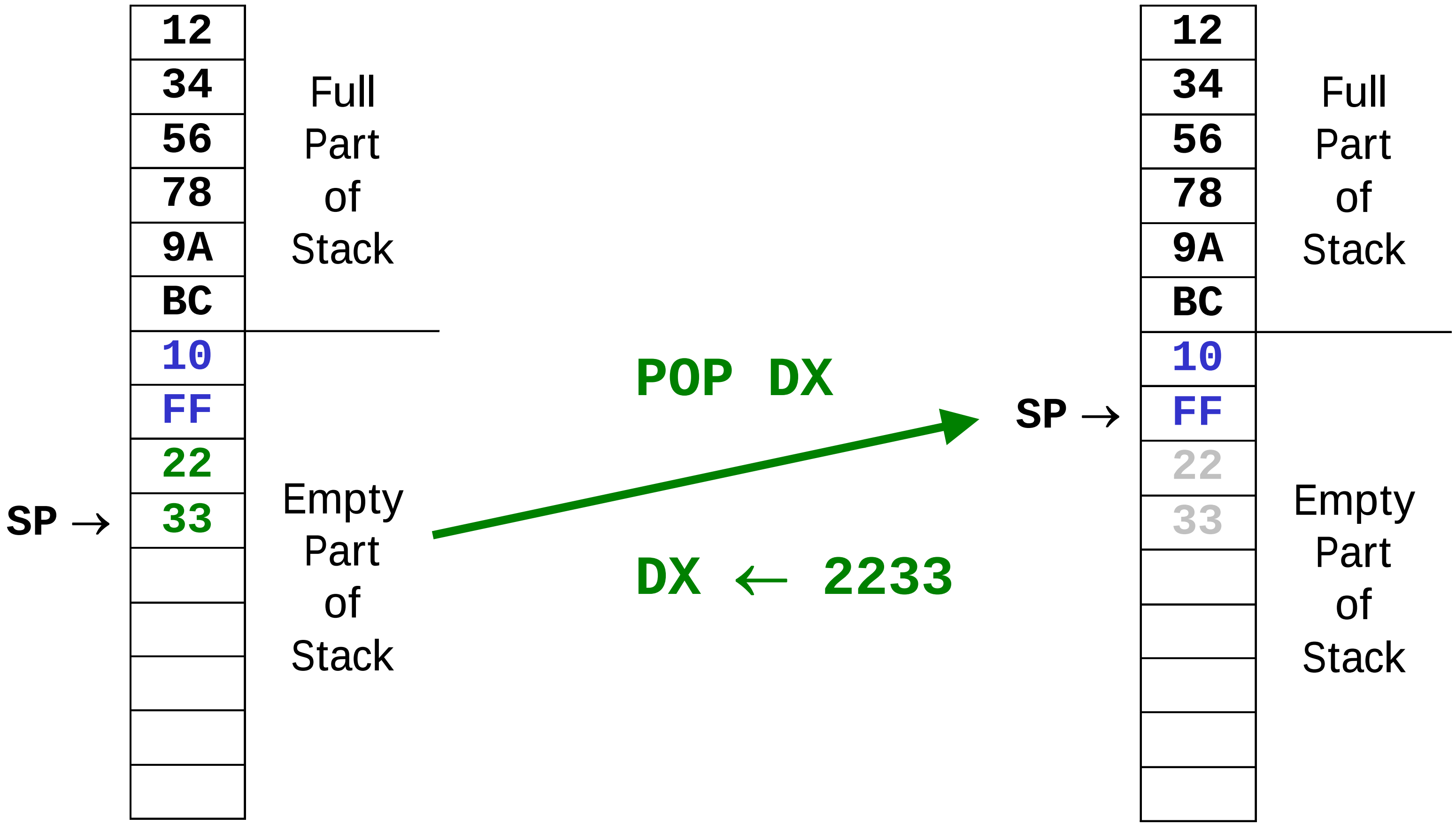
POP Operation

POP dest

$$\text{dest} \xleftarrow{\text{16-bits}} [\text{SP}]$$

$$\text{SP} \leftarrow \text{SP} + 2$$

Example



Using Push and Pop

Common application

Save current value of register or memory

Change value

Perform operations

Restore old value

Example

```
MOV AX, [x]      ; copy [x] into AX
SHL AX, 1        ; AX ← AX * 2
CMP AX, [y]
JLE end
PUSH AX          ; save value of AX
MOV AX, 1234     ; overwrite AX
IMUL [y]         ; AX ← AX * [y] (requires AX)
MOV [y], AX
POP AX           ; restore previous value of AX
```

Call and Return

Instruction	Equivalent instructions
CALL target	PUSH address of fall-through onto stack JMP target
RET	POP instruction address from stack

Example

```
      MOV AX,[x]
      CALL function
      MOV [x],AX    ; fall-through
      MOV AX,[y]
      CALL function
      MOV [y],AX    ; fall-through
end:
function:   IMUL AX
            SUB AX,75
            RET
```

Interrupt

Software Interrupts (Trap)

Interrupt instruction **INT N**

PUSH fall-through address

PUSH status register

Branch to service routine N

Hardware Interrupt

Initiated by external hardware

CPU gets signal from motherboard

Interrupt Service Routine (ISR)

Interrupt causes branch to ISR

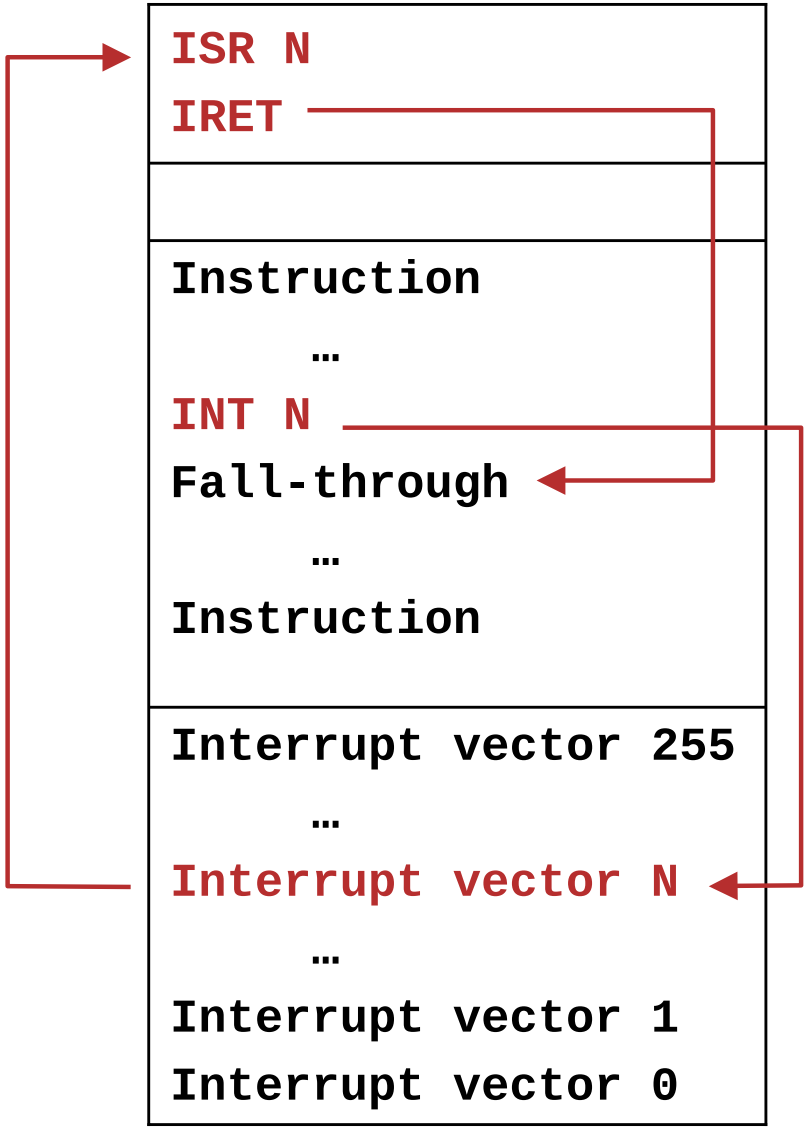
ISR address stored in OS table

Interrupt number N = Index into table

Return from interrupt

IRET instruction

POP fall-through address and **status** register from stack



Interrupt Example

DOS service to terminate program

```
mov AH, 0x4C          ; DOS service code number
mov AL, exit_code     ; exit code message
int 0x21              ; interrupt calls DOS
```