Basic Intel x86 Assembly Language Programming

Intel IA-32

IA-32

Intel Architecture — 32-bit

32-bit instruction set architecture (ISA)

Master plan for Intel mass market PC processors

Defines

- 32-bit Registers
- 32-bit Addressing modes
- Instruction types and formats

Enables code compatibility between processor models

x86-64

Extends many IA-32 features to 64 bits

Developed in Haifa for 386 processor

Introduced 1985

Updated in 486, Pentium, Pentium II, Pentium III, Pentium 4, Core

Intel IA-32 — x86

x86

- Named for early processor models
  - 386, 486, Pentium (586), Pentium II (686)
  - Called i386 in Linux system files
- 32-bit version of IA-32
  - 32-bit integers
  - 32-bit registers
  - 32-bit address
    - 2^{32} B = 4 GB address space
    - 64 GB with PAE (Physical Address Extension)
- Can run 16-bit (DOS) code

Intel IA-32 — x64

x64 (x86-64)

- 64-bit extensions for IA-32
  - Some instructions use 64-bit integers
  - Some instructions use 64-bit registers
  - 64-bit address
    - 48-bit address implemented
    - 2^{48} B = 256 TB address space (1 TB = 1024 GB)
- Can run 32-bit x86 code
  - x64 CPU running 32-bit operating system equivalent to x86 CPU
- Cannot run 16-bit (DOS) code

Intel IA-64

IA-64

Intel Architecture — 64-bit

64-bit instruction set architecture (ISA)

Master plan for Intel high performance processors

Defines

64-bit registers and integers (used by all instructions)

64-bit address

Instruction types and formats

Not compatible with IA-32 — cannot run PC-oriented software

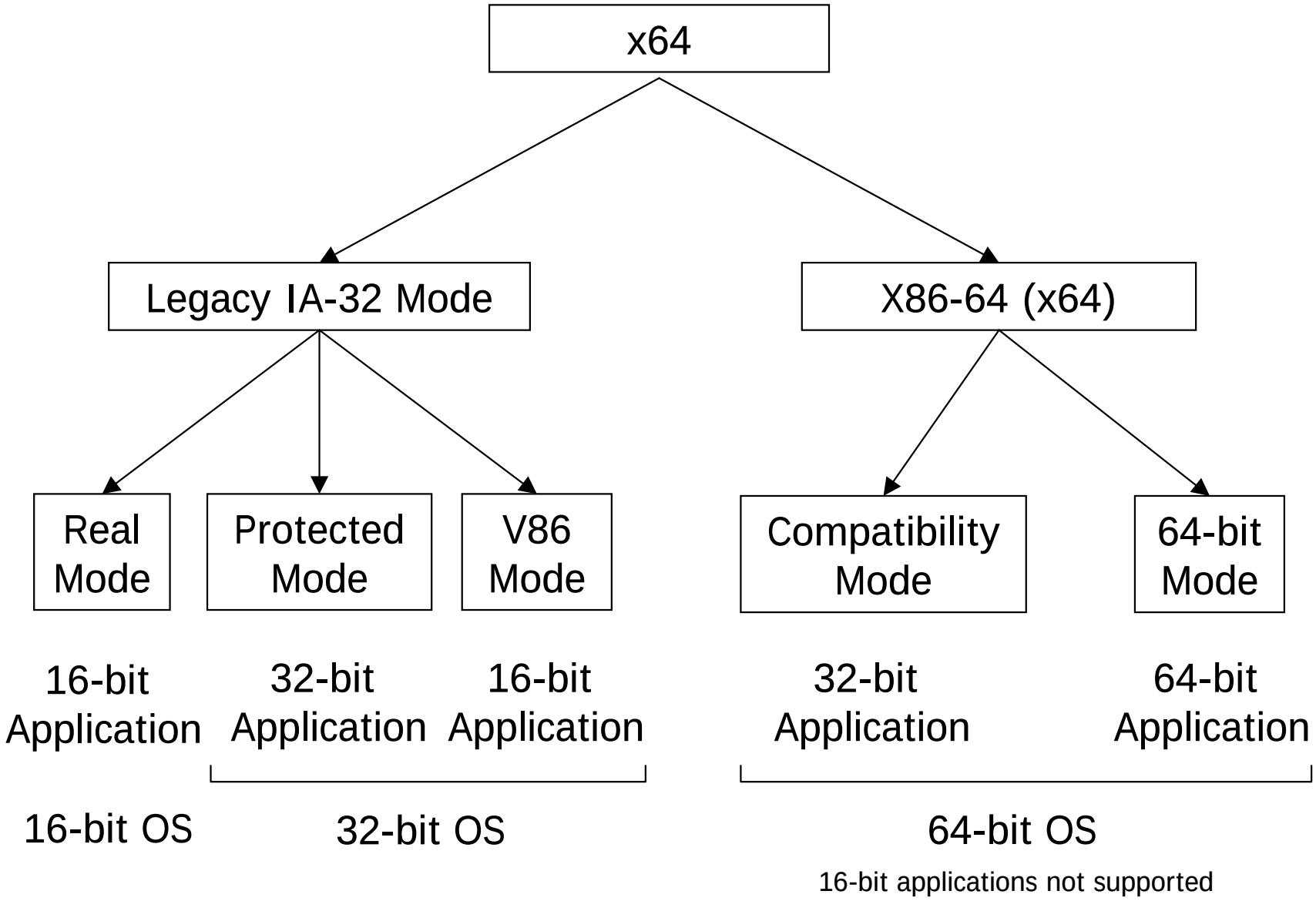
Developed for Itanium processor

Introduced 2001

Intel x86 / x 64 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 x86 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



Running Machine Language Program

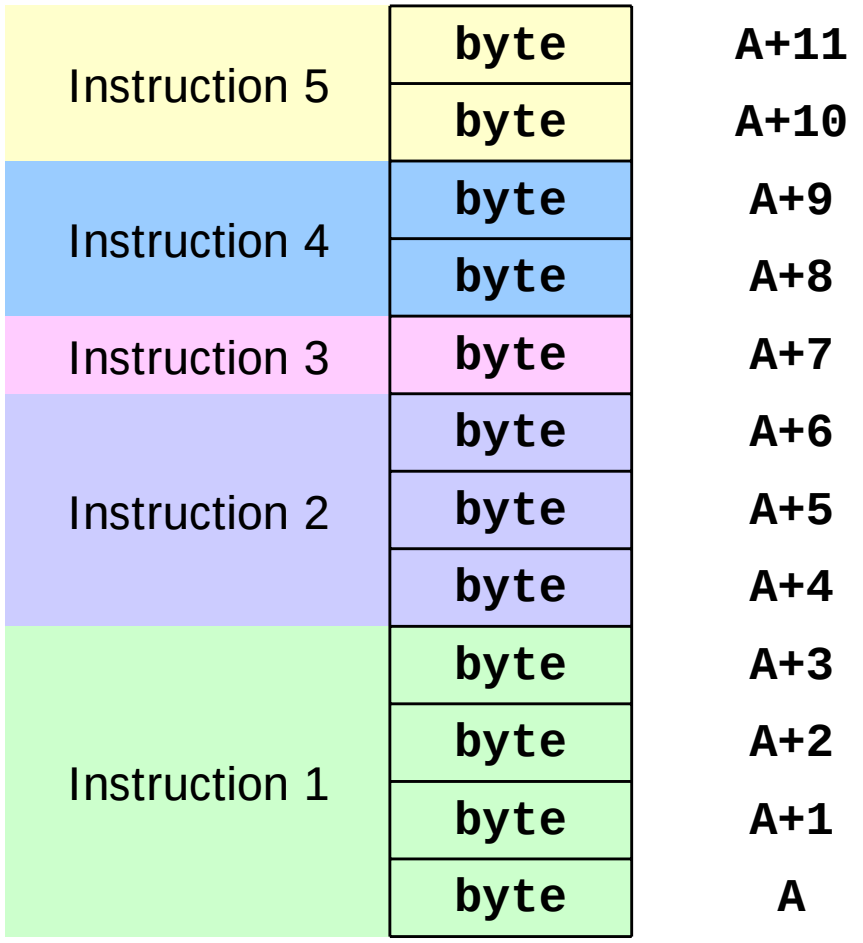
Program list

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 Set Architecture

Typical Intel machine instruction



Examples

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

ADD destination, source
    destination ← destination + source

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

Intel x86 General Registers

8-bit access

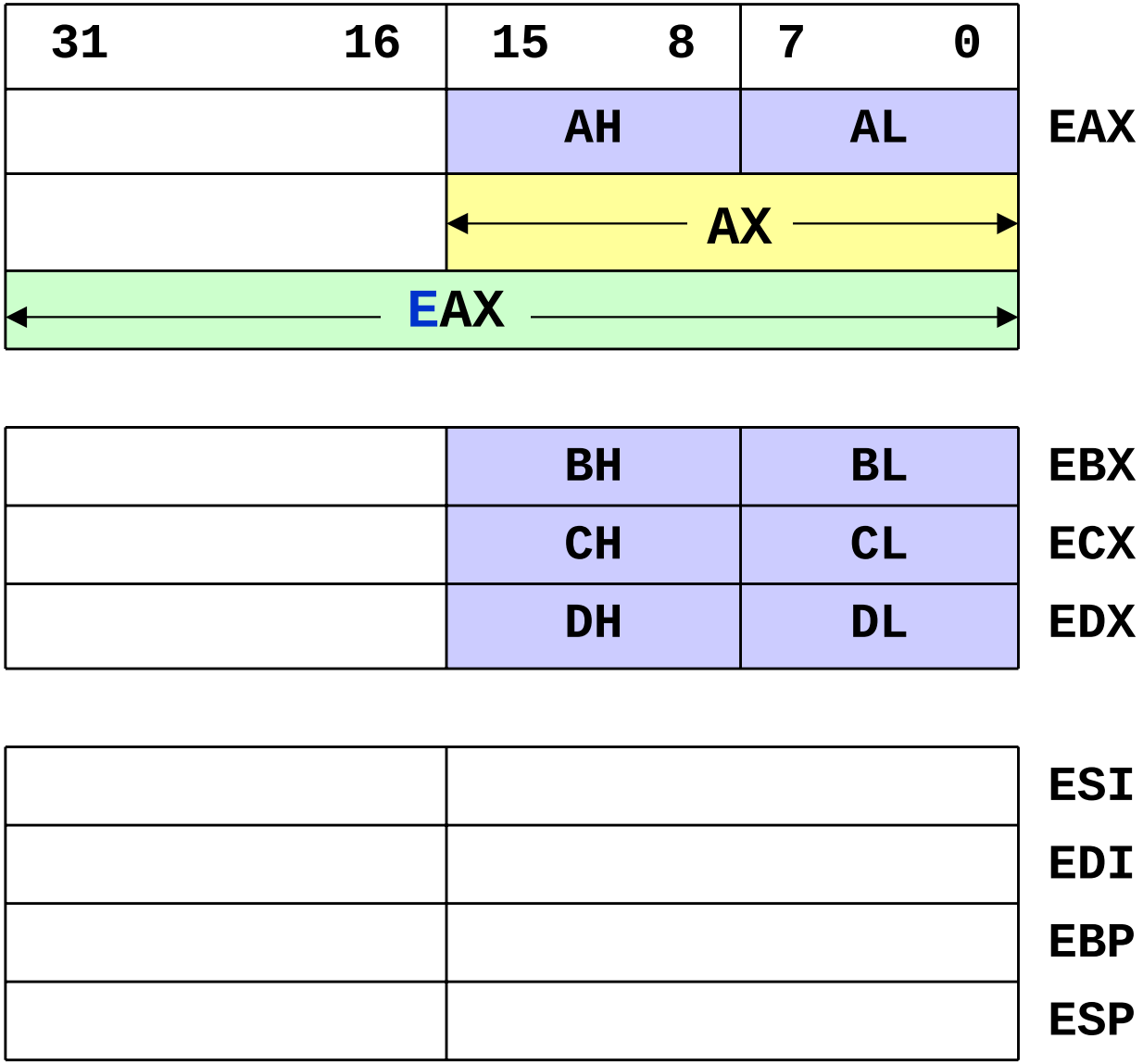
AL, BL, CL, DL,
AH, BH, CH, DH

16-bit access

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

32-bit access

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



Register Examples

8-bit operation

```
ADD AL, BL
    AL ← AL + BL
```

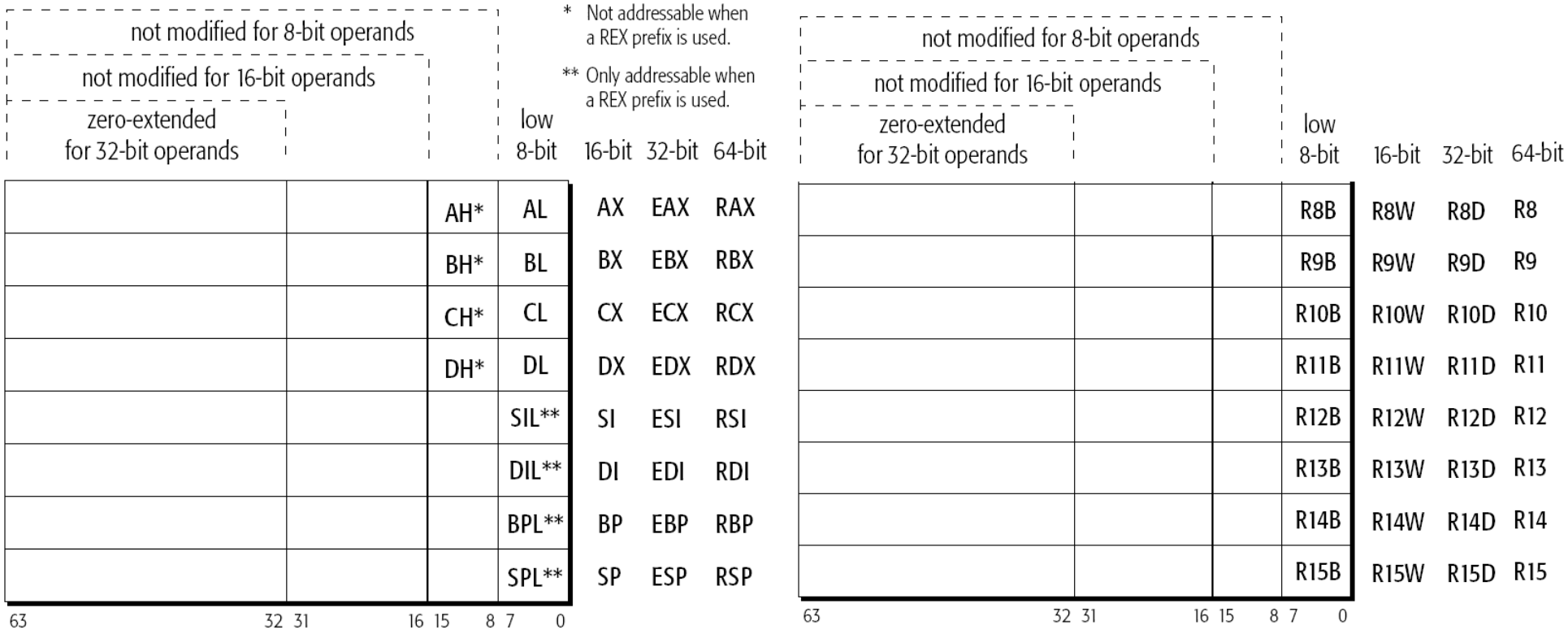
16-bit operation

```
ADD AX, BX
    AX ← AX + BX
```

32-bit operation

```
ADD EAX, EBX
    EAX ← EAX + EBX
```

Intel x64 General Registers



Register accesses

64-bit operations access entire register

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

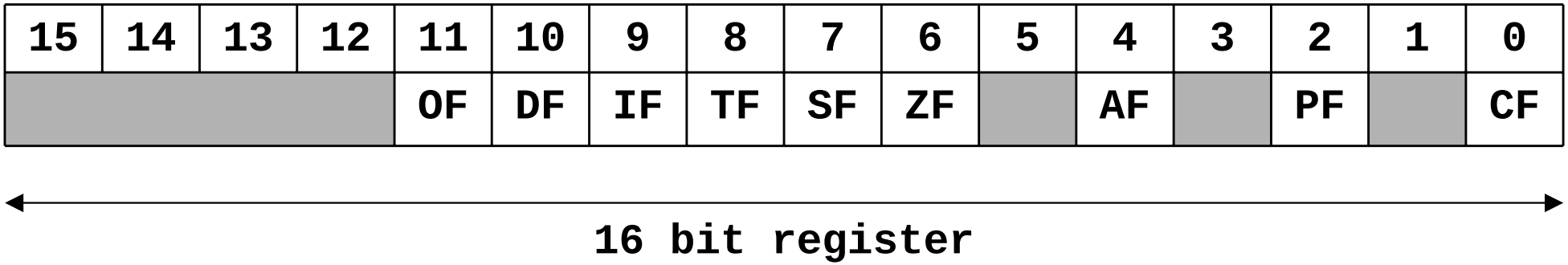
16-bit operations access lower 16-bits of 64-bit registers (where permitted)

8-bit operations access lower 8-bits of 64-bit registers (where permitted)

ADD RAX, RBX performs **RAX ← RAX + RBX**

Status Register

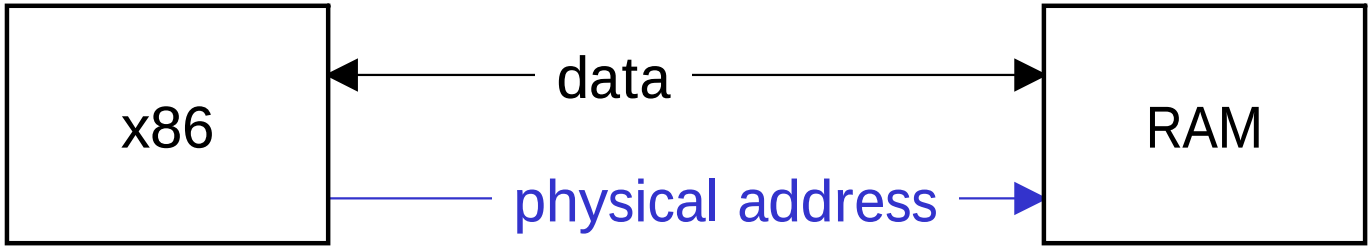
Bit Position	Name	Function
0	CF	Carry Flag — set on unsigned overflow
2	PF	Parity Flag
4	AF	Auxiliary Flag — set on overflow from 4 low bits of AL
6	ZF	Zero Flag — set on zero result
7	SF	Sign Flag — set on negative result
8	TF	Stops after next instruction and resets TF
9	IF	Interrupt Enable
10	DF	Autodecrement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



x86 Data and Address Ranges

Data			
Unsigned Integer		Signed Integer (2's complement)	
0	4,294,967,295	-2,147,483,648	2,147,483,647
0	FFFFFFFF	80000000	EFFFFFFF

Physical Addresses (4 GB)	
0	4,294,967,295
0	FFFFFFFF



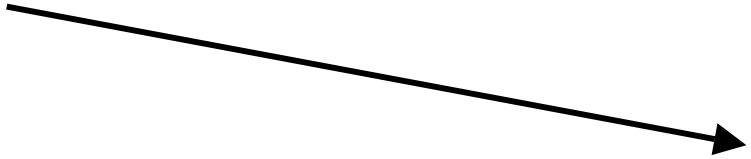
Program in Memory

Program data

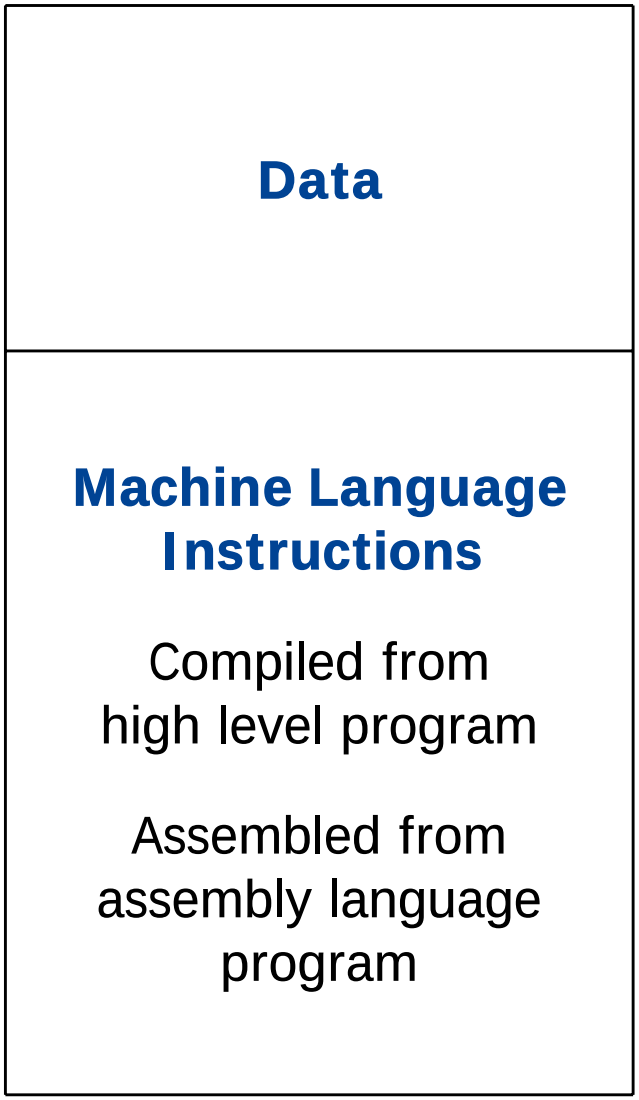
- Constants
- Variables
- Tables
- Structures
- User stack

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







Simple 32-Bit Assembly Program

Instructions written as list

```
MOV EDX, value_1          ; EDX ← value_1
MOV EAX, [address_1]      ; EAX ← [address_1]
ADD EAX, EDX              ; EAX ← EAX + EDX
MOV EBX, address_2        ; EBX ← address_2
MOV [EBX], EAX            ; [address_2] ← EAX
```

CPU

Fetches next instruction in list

Decodes fetched instruction

Executes decoded instruction



Simple Assembly Language Programming

Write program

Combine assembly language instructions

Source file

Save program listing in text file

Typical name: `source.asm`

Assemble program

Similar to compiling C program

Run assembler using `source.asm` as input

Assembler converts `source.asm` to object file `source.o`

Link object file

Linker converts `source.o` to `a.out` (or assigned name)

View executable file

Disassembler converts executable to assembly language

NASM Assembler

Assembler system

Free open source system

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

Pre-installed on Ubuntu Linux

Assembler — **`nasm`**

Disassembler — **`ndisasm`**

Typically **`/usr/bin/nasm`** and **`/usr/bin/ndisasm`**

Installation for Ubuntu Linux

```
sudo apt-get install nasm
```

Help

```
man nasm
```

Linker

```
gcc
```

NASM — Usual Command Line Options

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

Valid output formats for `-f` :

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

`-o <output>`

By Linux default NASM converts **`source.asm`** to **`source.o`**

Option allows choice of output file name

Example

```
~/asm$ nasm -f elf example.asm <ENTER>
```

`~/asm$` — Linux prompt

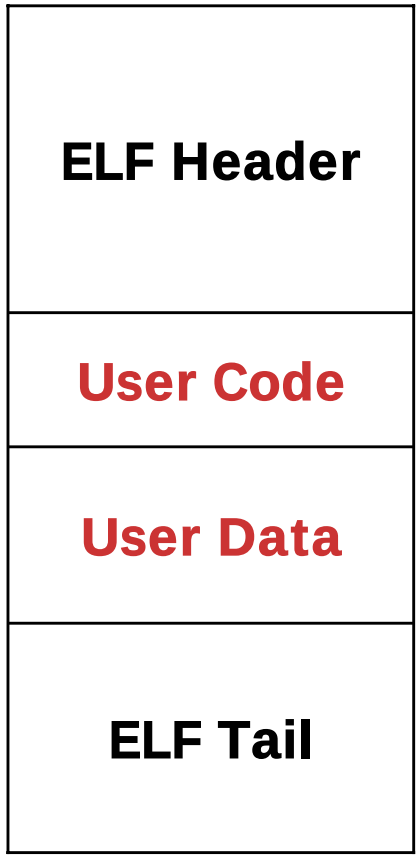
`nasm -f elf example.asm <ENTER>` — user command

ELF

Executable and Linking Format

Standard format for Linux executable files

Structure of executable file stored on system

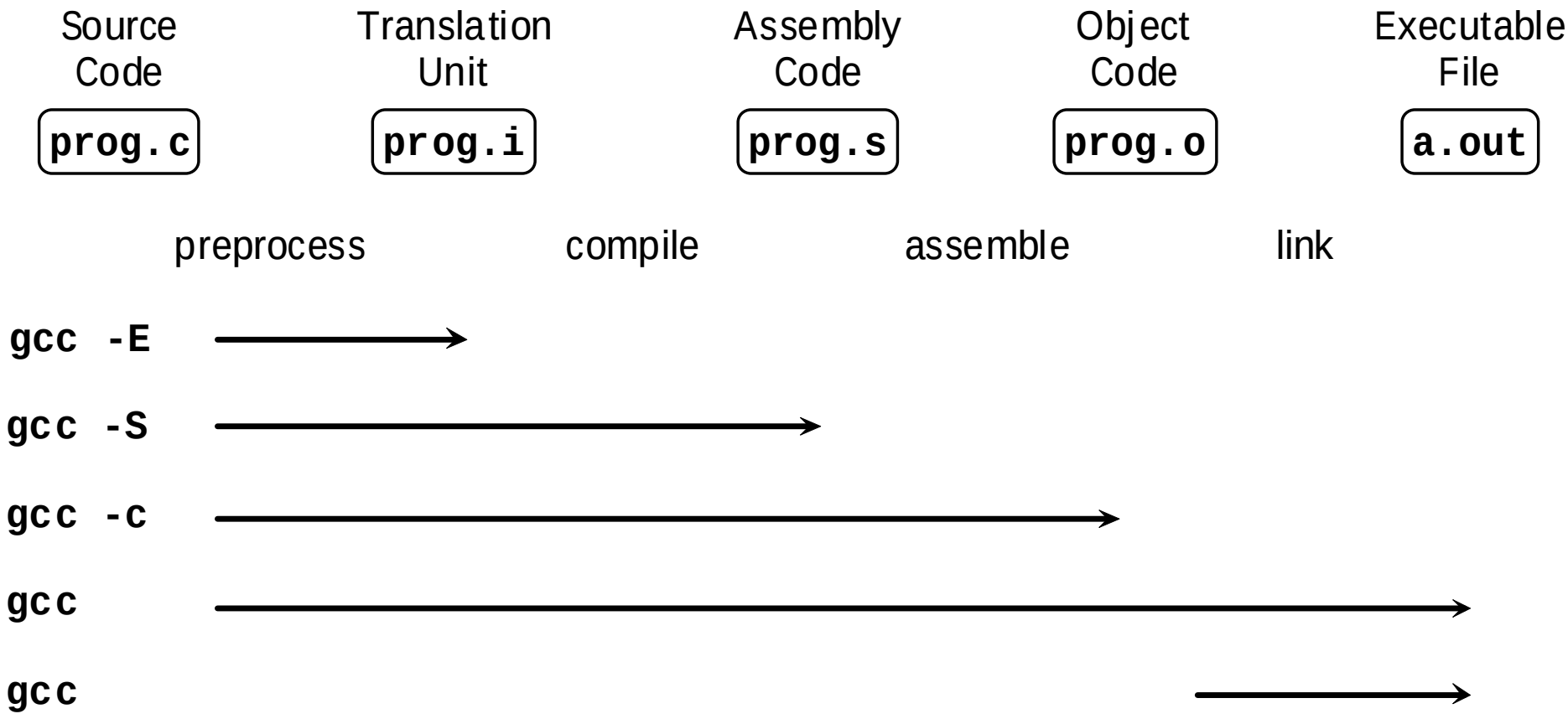


Information for Linux
Describes user code and locations of program data
Required for OS to load and execute program

Instructions and data specified in source program

Information for Linux
Required for OS to load and execute program

Stages of Gnu C Compilation (gcc)



gcc — Usual Command Line Options

```
gcc [-nostartfiles] [-nostdlib] [-o <output>]
    <filename>
```

-nostdlib

No Linux or C standard libraries linked

For pure assembly code without Linux system calls or C function calls

-nostartfiles

No C standard libraries linked

For assembly code without standard C function calls

-o <output>

By Linux default gcc converts **source.o** to **a.out**

Option allows choice of output file name

Example

```
~/asm$ gcc –nostdlib -o example example.o <ENTER>
```

Assembly Example

Write example.asm in text editor

Start Text Editor, vi, gvim, emacs, ...

Enter assembly instructions

```
MOV EAX, EBX
ADD EAX, ECX
```

Save file in convenient directory **~/asm/example.asm**

Open terminal in directory ~/asm

Assemble file

```
~/asm$ nasm -f elf example.asm <ENTER>
```

Produces default object file **example.o**

Link file

```
~/asm$ gcc –nostdlib -o ex0 example.o <ENTER>
```

Produces Linux executable file **ex0**

MOV – reg / reg – reg / imm

```
MOV dest, src          dest ← src

    src = reg / mem / imm        dest = reg / mem

    NOT LEGAL:  MOV mem, mem

MOV EAX, EBX           EAX ←————— EBX
                           32-bits

MOV AX, BX             AX ←————— BX
                          16-bits

MOV EAX, 11223344      EAX ←————— 11223344
                           32-bits

MOV AX, 1234           AX ←————— 1122
                          16-bits

MOV AL, 11             AL ←————— 11
                          8-bits
```

MOV — reg / mem

MOV dest, src dest ← src

src = reg / mem / imm dest = reg / mem

NOT LEGAL: MOV mem, mem

MOV EAX, [address] EAX ←(32-bits)— [address]

MOV AL, [address] AL ←(8-bits)— [address]

32-bit address

LEGAL ADDRESS = BASE + INDEX * SCALE + DISPLACEMENT

BASE = any 32-bit register

INDEX = any 32-bit register except ESP

SCALE = 1, 2, 3, 4, 8

DISPLACEMENT = 32-bit immediate

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 EAX,EAX          ; EAX ← 0

MOV EAX,11223344     ; EAX ← 11223344
AND EAX,000000FF     ; EAX ← 00000044

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

MOV EAX, 1           ; AX ← 00000001
NOT EAX              ; AX ← FFFFFFFE
```

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	-1
			+ 001	+ 1
0	1	1	000	0

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

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

Data Conversion

CBW — convert **byte** to **word** with sign extension

CWD — convert **word** to **double** with sign extension

CDQ — convert **double** to **quad** 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
CDQ	**If EAX < 80000000H, then DX ← 0**
	If EAX > 7FFFFFFFH, then DX ← FFFFFFFFH

Add/Subtract

ADD dest, src	ADD EBX,ECX	EBX ← EBX + ECX
ADC dest, src	ADC ESI,EDX	ESI ← ESI + EDX + CF
SUB dest ,src	SUB ESI,EDX	ESI ← ESI - EDX
SBB dest ,src	SBB ESI,EDX	ESI ← ESI – EDX - 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 EDX.EAX and EDI.ESI

```
ADD EAX, ESI        EAX ← EAX + ESI
                    CF ← carry
ADC EDX, EDI        EDX ← EDX + EDI + CF
                    CF ← overflow

SUB EAX, ESI        EAX ← EAX - ESI
                    CF ← borrow
SBB EDX, EDI        EDX ← EDX - EDI - CF
                    CF ← overflow
```

Unsigned Multiplication / Division

MUL source	MUL BL	AX ← AL*BL
	MUL BX	DX.AX ← AX*BX
	MUL EBX	EDX.EAX ← EAX*EBX
DIV source	DIV BL	AL ← AX / BL AH ← AX % BL
	DIV BX	AX ← DX.AX / BX DX ← DX.AX % BX
	DIV EBX	EAX ← EDX.EAX / EBX EDX ← EDX.EAX % EBX

Signed Multiplication / Division

IMUL source	IMUL BL	AX ← AL*BL
	IMUL BX	DX.AX ← AX*BX
	IMUL EBX	EDX.EAX ← EAX*EBX
IDIV source	IDIV BL	AL ← AX / BL AH ← AX % BL
	IDIV BX	AX ← DX.AX / BX DX ← DX.AX % BX
	IDIV EBX	EAX ← EDX.EAX / EBX EDX ← EDX.EAX % EBX

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
```

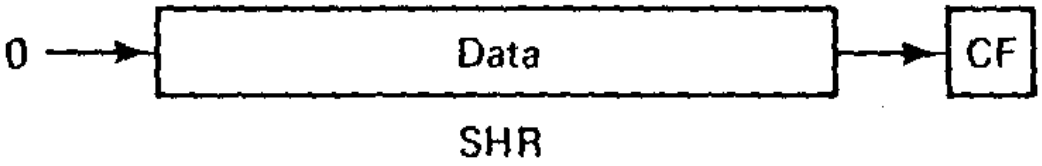
Shift Instructions

SHR dest, times

dest = register or memory

times

Number or **CL**

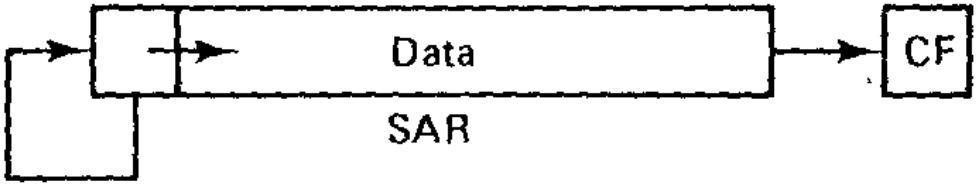


SAR dest, times

Shift Arithmetic Right

Shift bits right

Preserves sign



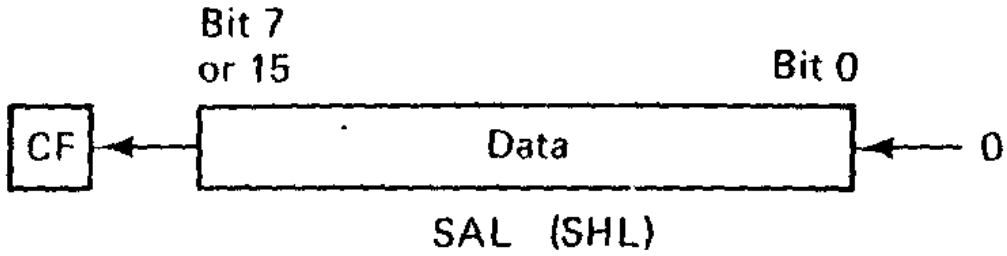
SAL dest, times

Shift Arithmetic Left

Shift left

Copies sign bit to CF

OF = 1 if sign bit changes



Rotate Instructions

ROL dest, times

Rotate Left

ROR dest, times

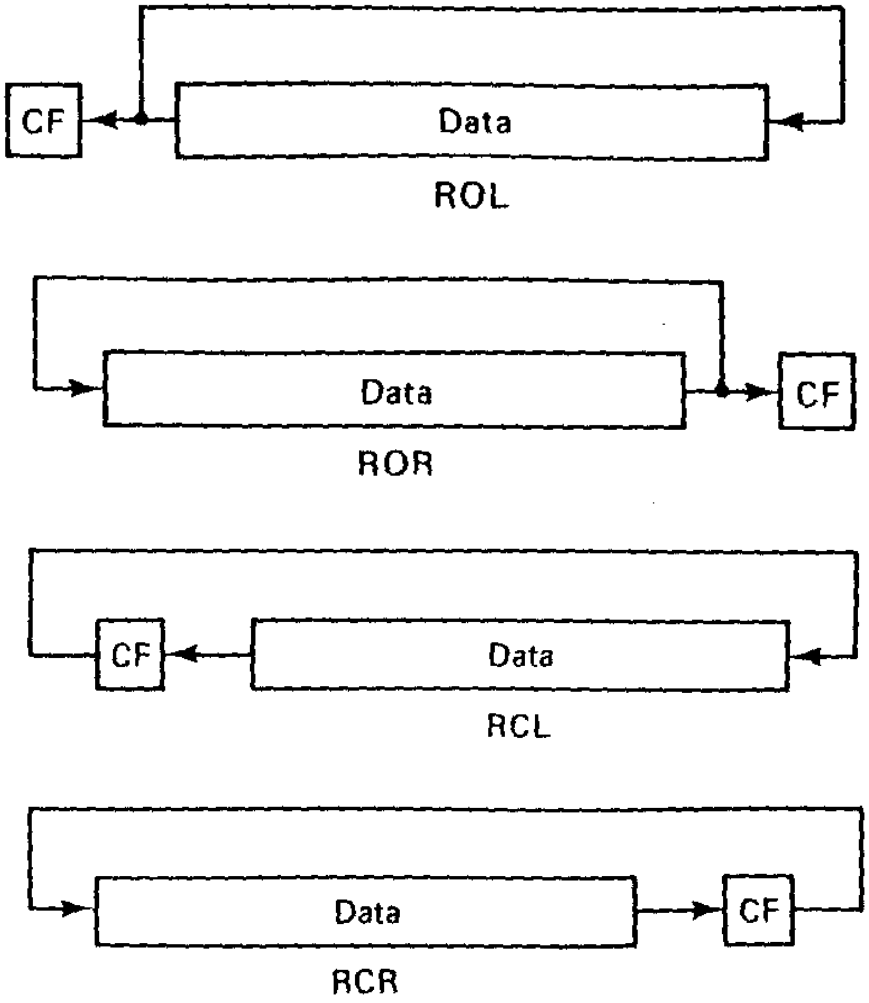
Rotate right

RCL dest, times

Rotate carry left

RCR dest, times

Rotate carry right



Example

```
SHL EAX, 2
MOV CL, 4
ROR BH, CL
```

Branch Instructions

Changes program execution order

Program chooses next instruction

Used to build control blocks

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

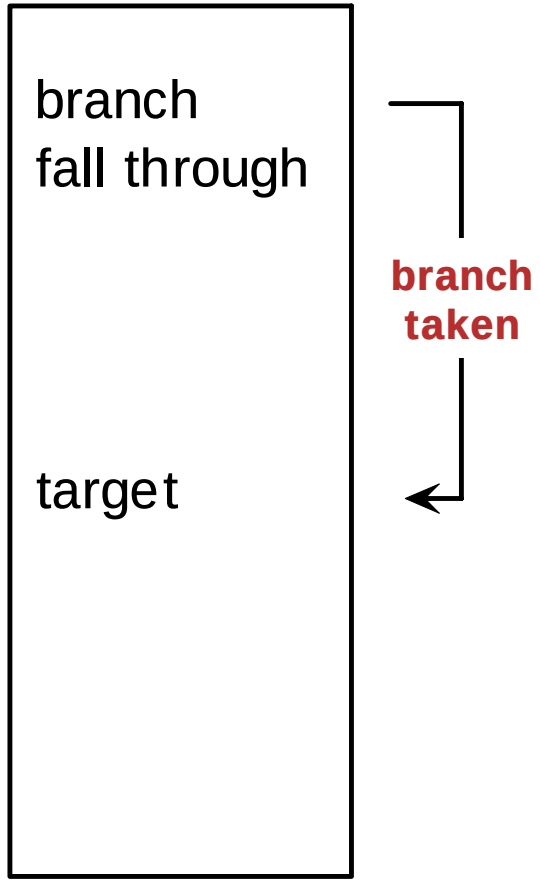
Fall-through

Instruction following branch in program listing

Next instruction if branch **not taken**

Target

Next instruction if branch **taken**



Jump Instruction

Unconditional branch

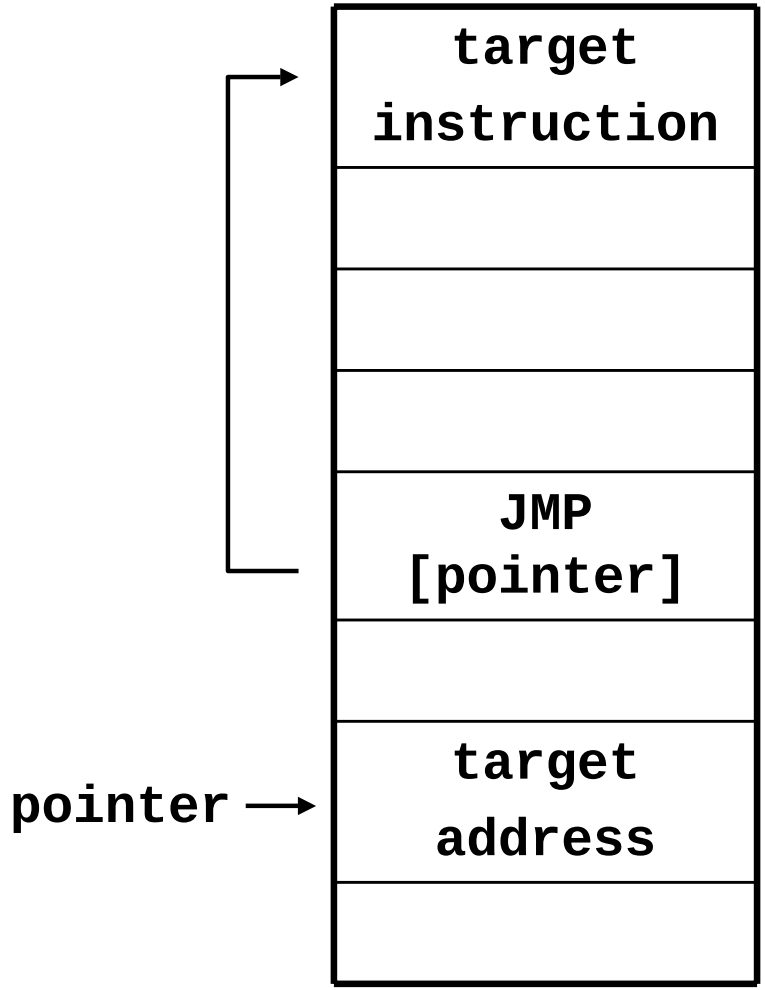
```
JMP target
```

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

Assembler converts label `L1` to 32-bit address

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 EAX,EAX   ; zero accumulator (AX ← 0)
        MOV EBX,0001  ; BX ← 1
L1:     ADD EAX,EBX   ; add BX to accumulator
        INC EBX       ; EBX++
        CMP EBX,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 EAX,EAX    ; zero accumulator
    MOV ECX,0005   ; ECX ← 5
L1: ADD EAX,ECX    ; add ECX to accumulator
    LOOP L1        ; ECX-- and loop if ECX > 0
```

Program Sections

NASM programs include

x86 assembly instructions

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

Linux 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`**

Quoted null-terminated string data: **`'this is a string',0`**

NASM Template for Simple Assembly Code

```
; define initialized data structures here
    section .data
; define uninitialized data structures here
; minimum allocation size is 1 dword = 32 bits
; cannot pass BSS pointers to all system calls
    section .bss
; user code
    section .text
    global _start           ; permits access to _start
_start:                     ; symbolic pointer to code
    ; place user code here
    ;
    mov eax, 1              ; Linux terminate code
    mov ebx, 0              ; exit code = 0
    int 0x80                ; call Linux kernel
```

Number Representations

```
mov     ax,200          ; decimal
mov     ax,0200         ; still decimal
mov     ax,0200d        ; explicitly decimal
mov     ax,0d200        ; also decimal
mov     ax,0c8h         ; hex
mov     ax,$0c8         ; hex again: the 0 is required
mov     ax,0xc8         ; hex yet again
mov     ax,0hc8         ; still hex
mov     ax,310q         ; octal
mov     ax,310o         ; octal again
mov     ax,0o310        ; octal yet again
mov     ax,0q310        ; octal yet again
mov     ax,11001000b    ; binary
mov     ax,1100_1000b   ; same binary constant
mov     ax,1100_1000y   ; same binary constant once more
mov     ax,0b1100_1000  ; same binary constant yet again
mov     ax,0y1100_1000  ; same binary constant yet again
```

Data Declarations — Numbers

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

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

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

`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)

Data Declarations — Strings

`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: dd 'ABCDE'`

Stores string **`41 42 43 44 45 00 00 00`** in memory

1 byte per character — appends **`00`** to form 4-byte 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
    dwordvar:    resd 1
        ; Reserves 1 dword in variable named dwordvar


; assign value to variable in text section
    MOV [wordvar], 0x11223344
```

Program Example — ex1

```
section .data
    s1 db "this is a string",0
section .text
global _start
_start:
    mov eax, 0x11223344 ; eax ← 0x11223344
    mov ebx, 0x55667788 ; ebx ← 0x55667788
    mov esi, s1         ; esi ← pointer to data s1
;
mov eax, 1              ; Linux terminate code
mov ebx, 0              ; exit code
int 0x80                ; call Linux kernel
```

Running — ex1

Assemble

```
~/asm$ nasm -f elf ex1.asm
```

Produces object file `ex1.o`

Link

```
~/asm$ gcc -nostdlib -o ex1 ex1.o
```

Produces executable file `ex1`

Run

```
~/asm$ ex1
```

Display exit code

```
~/asm$ echo $?
```

Run and display exit code

```
~/asm$ ex1 ; echo $?
```

```
~/asm$ ex1 ; echo $?
0
~/asm$
```

Program Example — ex1a

```
section .data
    s1 db "this is a string",0
section .text
global _start
_start:
    mov eax, 0x1          ; eax ← 0x1
    mov ebx, 0x2          ; ebx ← 0x2
    add ebx, eax          ; ebx ← 0x2 + 0x1 = 0x3
;
mov eax, 1                ; Linux terminate code
mov ebx, 0                ; exit code
int 0x80                  ; call Linux kernel
```

Running — ex1a

Assemble

```
~/asm$ nasm -f elf ex1a.asm
```

Produces object file **ex2.o**

Link

```
~/asm$ gcc -nostdlib -o ex1a ex1a.o
```

Produces executable file **ex1a**

Run

```
~/asm$ ex1a
```

Display exit code

```
~/asm$ echo $?
```

Run and display exit code

```
~/asm$ ex1a ; echo $?
```

```
~/asm$ ex1a ; echo $?
3
~/asm$
```

Linux Debugging Utilities — 1

Object dump

`objdump -d ex1`

Displays disassembly of executable code

`.text section`

Uses AT&T **`gas`** format

`operation source, destination`

`$literal (fixed number)`

`%register`

`objdump -s ex1`

Displays hexadecimal content of file with ASCII translation

Linux Debugging Utilities — 2

```
objdump -d ex1

ex1:     file format elf32-i386

Disassembly of section .text:

080480c0 <_start>:
 80480c0:   b8 44 33 22 11      mov $0x11223344,%eax
 80480c5:   bb 88 77 66 55      mov $0x55667788,%ebx
 80480ca:   be dc 90 04 08      mov $0x80490dc,%esi
 80480cf:   b8 01 00 00 00      mov $0x1,%eax
 80480d4:   bb 00 00 00 00      mov $0x0,%ebx
 80480d9:   cd 80               int $0x80
```

Linux Instruction Address

Machine Code

Assembly Code

Linux Debugging Utilities — 3

objdump –s ex1

```
ex1:     file format elf32-i386

Contents of section .note.gnu.build-id (header):
 8048094 04000000 14000000 03000000 474e5500  ............GNU.
 80480a4 a8e51408 341cdc6f e94a200d 6e43b9e8  ....4..o.J .nC..
 80480b4 e9df2553                             ..%S
Contents of section .text:
 80480c0 b8443322 11bb8877 6655bedc 900408b8  .D3"...wfU......
 80480d0 01000000 bb000000 00cd80             ...........
Contents of section .data:
 80490dc 74686973 20697320 61207374 72696e67  this is a string
 80490ec 00
```

Linux Debugging Utilities — 4

Read ELF

`readelf -h ex1`

Displays summary of ELF header information

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x80480c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          296 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         7
  Section header string table index: 4
```

NASM Disassembler — 1

```
~/asm$ 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 same as -b 32
   -b 16, -b 32 or -b 64 sets the processor mode
   -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)
```

Example

Disassemble 32-bit executable **`ex1`** skipping 0xC0 header bytes

```
~/asm$ ndisasm –b32 –ec0h ex1
```

NASM Disassembler — 2

```
ndisasm -b32 ex1
00000000  7F45            jg 0x47
00000002  4C              dec esp
00000003  46              inc esi
00000004  0101            add [ecx],eax
00000006  0100            add [eax],eax
00000008  0000            add [eax],al
          ...
000000BC  0000            add [eax],al
000000BE  0000            add [eax],al
000000C0  B844332211      mov eax,0x11223344
000000CA  BEDC900408      mov esi,0x80490dc
000000CF  B801000000      mov eax,0x1
000000D4  BB00000000      mov ebx,0x0
000000D9  CD80            int 0x80
```

header

.text section

```
ndisasm -b32 -e0c0h ex1
00000000  B844332211        mov eax,0x11223344
00000005  BB88776655        mov ebx,0x55667788
0000000A  BEDC900408        mov esi,0x80490dc
0000000F  B801000000        mov eax,0x1
00000014  BB00000000        mov ebx,0x0
00000019  CD80              int 0x80
```

Linux Text Utilities — 5

grep

Get **R**egular **E**xpression and **P**rint

Displays lines containing search text

```
readelf -h ex1 | grep address
Entry point address:   0x80480c0
```

awk

String manipulation language

```
readelf -h ex1 | grep address | awk '{print
    substr($4,7,3)}'
0c0
```

Displays characters 7 to 10 in 4th word of address line

`0c0` = offset from start of `ex1` header

= Entry point address – Linux default load address

= `0x80480c0 - 0x8048000`

less

Display text by page

Convenient Script for NASM Disassembler

Script file

Combine **`readelf + grep + awk + less + ndisasm`**

Edit file

`nd`

Write following line into file

```
ndisasm -b32 -e`readelf -h $1 | grep address
    | awk '{print substr($4,7,3)}'`h $1 | less
```

Save file and make executable

```
~/asm$ chmod u+x nd
```

Run

```
~/asm$ nd ex1
```

```
00000000  B844332211        mov eax,0x11223344
00000005  BB88776655        mov ebx,0x55667788
0000000A  BEDC900408        mov esi,0x80490dc
0000000F  B801000000        mov eax,0x1
00000014  BB00000000        mov ebx,0x0
00000019  CD80              int 0x80
```

Introduction to **debug** for Linux

Port to Linux of Microsoft debug for DOS

Download package from course website

Executable = **`trace`**

Sources written in assembly language for NASM

Running debugger

```
~/asm$ trace program_file arg_1 arg_2 ...
```

Loads **`program_file`** into buffer

Emulates running program with list of arguments

Basic features

Disassembly — shows instructions in NASM-style assembly language

Run program

- Normally — start to finish
- Trace mode — one instruction at a time

Displays register values

Displays memory values

debug — Opening Display

Start

```
~/asm$ trace ex1

[ Trace ] - ver 0.3 Beta
Copyleft (C) 2001-2005 Amjads. Made in Taiwan

EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BFD6A000 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF

080480C0 B844332211              MOV     EAX,11223344
-
```

Register values

```
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BFD6A000 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF
```

First instruction

```
080480C0 B844332211              MOV     EAX,11223344
```

Command prompt

```
-
```

debug — help

```
-h
Command:
        d <address>             memory dump
        u <address>             disassemble
        t                       step one instruction
        g                       run program
        c                       cont program
        r                       display registers
        xr                      display fp/mmx registers
        xrs                     switch xr=on/off
        cr                      display control registers
        dr                      display debug registers
        tr                      task register
        ldtr                    LDT register
        idtr                    IDT register
        gdtr                    GDT register
        idt                     Interrupt Descriptor Table
        gdt                     Global Descriptor Table
        pg <address>            list page table
        bp[1-4] <address>       set no.1-4 breakpoint
        bpl                     breakpoint list
        ?/h/help                command help
        q/quit                  exit
-
```

basic

Advanced OS functions

Advanced debugging

debug — Disassembly

```
[ Trace ] - ver 0.3 Beta
Copyleft (C) 2001-2005 Amjads. Made in Taiwan

EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BF97D550 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF

080480C0 B844332211          MOV     EAX,11223344
-u
080480C0 B844332211          MOV     EAX,11223344
080480C5 BB88776655          MOV     EBX,55667788
080480CA BEDC900408          MOV     ESI,080490DC
080480CF B801000000          MOV     EAX,00000001
080480D4 BB00000000          MOV     EBX,00000000
080480D9 CD80                INT     80
080480DB 00746869            ADD     BYTE [EAX+EBP*2+69],DH
080480DF 7320                JAE     08048101
-u
080480E1 69732061207374      IMUL    ESI,DWORD [EBX+20],74732061
080480E8 7269                JB      08048153
```

Begins at value of EIP

continues

debug — Display Memory

```
-u
080480C0 B844332211              MOV     EAX,11223344
080480C5 BB88776655              MOV     EBX,55667788
080480CA BEDC900408              MOV     ESI,080490DC
080480CF B801000000              MOV     EAX,00000001
080480D4 BB00000000              MOV     EBX,00000000
080480D9 CD80                    INT     80
080480DB 00746869                ADD     BYTE [EAX+EBP*2+69],DH
080480DF 7320                    JAE     08048101
-d 080490DC
080490DC  74 68 69 73 20 69 73 20-61 20 73 74 72 69 6E 67 this is a string
080490EC  00 00 2E 73 79 6D 74 61-62 00 2E 73 74 72 74 61 ...symtab..strta
080490FC  62 00 2E 73 68 73 74 72-74 61 62 00 2E 6E 6F 74 b..shstrtab..not
0804910C  65 2E 67 6E 75 2E 62 75-69 6C 64 2D 69 64 00 2E e.gnu.build-id..
0804911C  74 65 78 74 00 2E 64 61-74 61 00 00 00 00 00 00 text..data......
0804912C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0804913C  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0804914C  00 00 00 00 1B 00 00 00-07 00 00 00 02 00 00 00 ................
```

debug — Run Normally

Program runs

Normal termination (error free) ⟶ re-initializes registers

```
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BFB5F670 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF

080480C0 B844332211              MOV     EAX,11223344
-go
Program terminated normally

EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BFE34A90 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF

080480C0 B844332211              MOV     EAX,11223344
-
```

debug — Run With Breakpoints

```
-u
080480C0 B844332211              MOV     EAX,11223344
080480C5 BB88776655              MOV     EBX,55667788
080480CA BEDC900408              MOV     ESI,080490DC
080480CF B801000000              MOV     EAX,00000001
080480D4 BB00000000              MOV     EBX,00000000
080480D9 CD80                    INT     80
080480DB 00746869                ADD     BYTE [EAX+EBP*2+69],DH
080480DF 7320                    JAE     08048101
-bp1 080480CF
-bpl
 Break Point #1=080480CF
 Break Point #2=00000000
 Break Point #3=00000000
 Break Point #4=00000000
-go

EAX=11223344 EBX=55667788 ECX=00000000 EDX=00000000 ESI=080490DC EDI=00000000
SS =0000007B ESP=BFF79890 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480CF Flags=00000246 PF ZF IF

080480CF B801000000              MOV     EAX,00000001
```

If **EIP = breakpoint**
Program stops before execution of instruction

debug — Trace Mode

```
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BF821E90 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C0 Flags=00000246 PF ZF IF

080480C0 B844332211              MOV     EAX,11223344
-t

EAX=11223344 EBX=00000000 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BF821E90 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480C5 Flags=00000246 PF ZF IF

080480C5 BB88776655              MOV     EBX,55667788
-t

EAX=11223344 EBX=55667788 ECX=00000000 EDX=00000000 ESI=00000000 EDI=00000000
SS =0000007B ESP=BF821E90 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480CA Flags=00000246 PF ZF IF

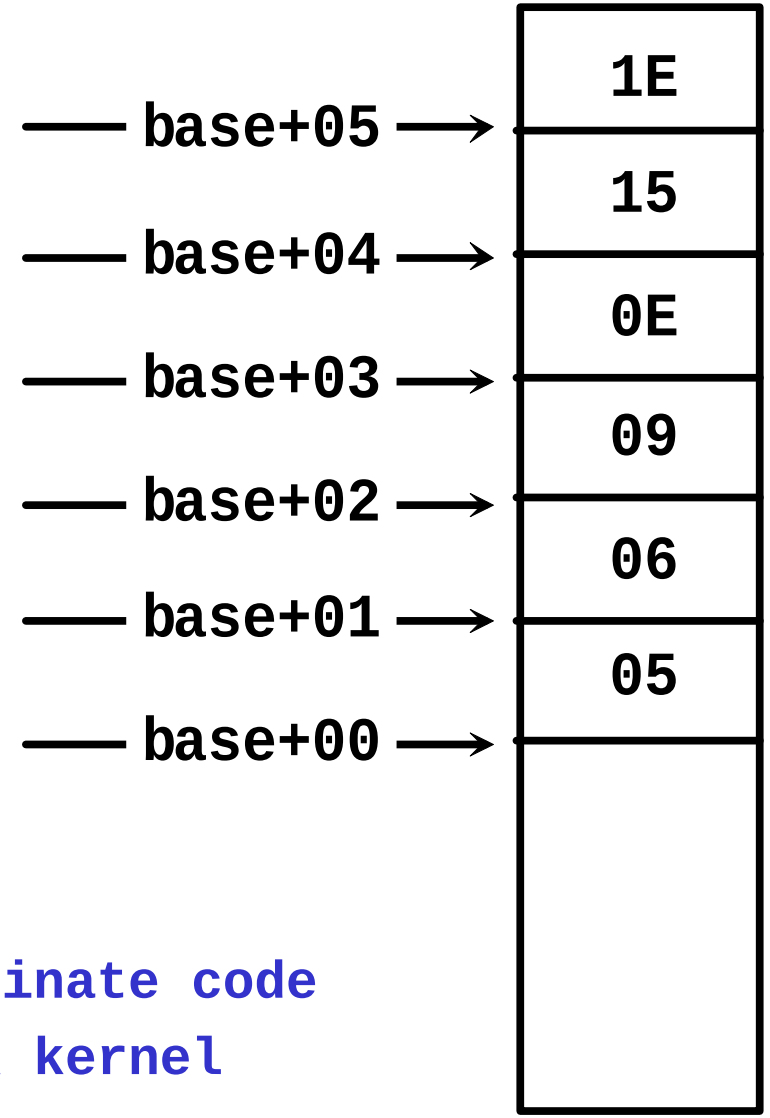
080480CA BEDC900408              MOV     ESI,080490DC
-t

EAX=11223344 EBX=55667788 ECX=00000000 EDX=00000000 ESI=080490DC EDI=00000000
SS =0000007B ESP=BF821E90 EBP=00000000 DS =0000007B ES =0000007B FS =00000000
GS =00000000 CS =00000073 EIP=080480CF Flags=00000246 PF ZF IF

080480CF B801000000              MOV     EAX,00000001
```

Example ex2 — Writing Values into a Table

```
section .bss
     base resb 6
section .text
global _start
_start:
     MOV       ESI, 0
L1:  MOV       EAX,ESI
     IMUL      AL
     ADD       AL,5
     MOV       [base + ESI],AL
     INC       ESI
     CMP       ESI,5
     JLE       L1
;
mov ebx, EAX                ; exit code
mov eax, 1                  ; Linux terminate code
int 0x80                    ; call Linux kernel
```



Example ex3 — Arithmetic With Stored Data

```
section .data
     table dw 1,2,3,4,5,6,7,8,9,10,0
section .text
global _start
_start:
     MOV      ESI,table
     MOV      AX,0
L1:  ADD      AX,[ESI]
     ADD      ESI,2
     CMP      WORD [ESI],0    ; operate on 16-bit word
                              ; [SI+1].[SI]
     JNZ      L1
;
mov ebx, EAX                  ; exit code
mov eax, 1                    ; Linux terminate code
int 0x80                      ; call Linux kernel
```

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

Example ex4 — `if` Control Block

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

```
section .bss
     x resd 1          ; integer x
     y resd 1          ; integer y
section .text
global _start
_start:
     MOV dword [x],0   ; x ← 0
     MOV dword [y],2   ; y ← 2
     MOV EAX,[x]       ; EAX ← x
     CMP EAX,[y]       ; EAX - y
     JNZ end           ; JMP IF NOT EQUAL
     MOV EAX,[y]       ; EAX ← y
     SHL EAX,1         ; EAX ← EAX * 2
     MOV [x],EAX       ; x ← EAX
     MOV EAX,[x]       ; EAX ← x
     SHL EAX,1         ; EAX ← AX * 2
     MOV [y],EAX       ; y ← EAX
end:  mov eax, 1
      mov ebx, 0
      int 0x80
```

Example ex5 — `for` Loop

```
section .bss
     i resd 1
     j resd 1
section .text
global _start
_start: MOV DWORD [i],0
        MOV EBX,3
L1:     CMP DWORD [i],10
        JGE end                 ; break on i ≥ 10
        MOV EAX,[i]             ; EAX ← i
        IMUL EBX                ; EAX ← EAX * 3
        MOV [j],EAX             ; j ← EAX
        INC DWORD [i]           ; i++
        JMP L1                  ; loop
end:    mov eax, 1
        mov ebx, [j]
        int 0x80
```

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

Example ex6 — Compound `if` Control Block

```
section .bss
      x resd 1
      y resd 1
      z resd 1
section .text
global _start
_start: MOV DWORD [x],0    ; x ← 0
        MOV DWORD [y],2    ; y ← 2
        MOV DWORD [z],3    ; z ← 3
        MOV EAX,[x]        ; EAX ← x
        CMP EAX,[y]        ; CMP EAX = x, y
        JZ L1              ; if x = y, JMP L1
        MOV EAX,[y]        ; EAX ← y
        CMP EAX,[z]        ; CMP EAX = y, z
        JLE end            ; if y <= z, JMP end
        CMP EAX,[z]        ; CMP EAX = x, z
        JLE end            ; if x <= z, JMP end
L1:     MOV EAX,[y]        ; EAX ← y
        SHL EAX,1          ; EAX ← 2 * EAX
        MOV [x],EAX        ; x ← EAX
```

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

```
end:    mov eax, 1
        mov ebx, 0
        int 0x80
```

Example ex7 — C versus Assembly

compile

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

rewrite in assembly language

```
      MOV DWORD [M],1          ; M ← 1
      MOV DWORD [N],2          ; N ← 2
L1:   CMP DWORD [N],7          ; compare N, 7
      JG end                   ; break if N > 7
      MOV EAX,[N]              ; EAX ← N
      IMUL DWORD [M]           ; EAX ← EAX * M
      MOV [M],EAX              ; M ← EAX
      INC DWORD [N]            ; N++
      JMP L1                   ; loop
end:  mov eax,1
      mov ebx,0
      int 80h
```

```
      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

Byte	LODSB	Load String Byte	AL ← [ESI] ESI ← ESI+1
	STOSB	Store String Byte	[EDI] ← AL EDI ← EDI+1
Word	LODSW	Load String Word	AL ← [ESI] AH ← [ESI+1] ESI ← ESI+2
	STOSW	Store String Word	[EDI] ← AL [EDI+1] ← AH EDI ← EDI+2
Dword	LODSD	Load String Double	EAX ← [ESI] ESI ← ESI+4
	STOSD	Store String Double	[EDI] ← AL [EDI+1] ← AH EDI ← EDI+2

Example ex8 — String Manipulation

```
section .data
     s_1 db "0123456789",0
     v_1 times 11 db 0
section .text
global _start
_start: MOV ESI, s_1          ; ESI ← pointer to s_1
        MOV EDI, v_1          ; EDI ← pointer to v_1
L1:     LODSB                 ; AL ← [ESI]
                              ; ESI ← ESI + 1
        CMP AL, 0
        JZ end
        SUB AL, 30            ; AL ← AL - 30
                              ; numerical value of ASCII char
        STOSB                 ; [EDI] ← AL
                              ; EDI ← EDI + 1
        JMP L1                ; loop
end:    STOSB                 ; store \null
        mov eax, 1
        mov ebx, 0
        int 0x80
```

XCGH

`XCHG dest, src`	dest ↔ src
`XCHG EAX, EBX`	EAX ↔ EBX
`XCHG AX, BX`	AX ↔ BX
`XCHG AL, AH`	AL ↔ AH
`XCHG AX,[SI]`	AX ↔ [SI]

LEA

Load Effective Address

Effective Address = pointer to data in memory segment

Similar to MOV

- Copies pointer to data
- Does not access memory

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

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

In NASM

```
LEA EBX,[x]
```

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

identical to

```
MOV EBX, x
```

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

Other possibilities

```
LEA EBX,[EAX+ESI+16]
```

EBX $\xleftarrow[\text{32-bits}]{}$ EAX+ESI+16

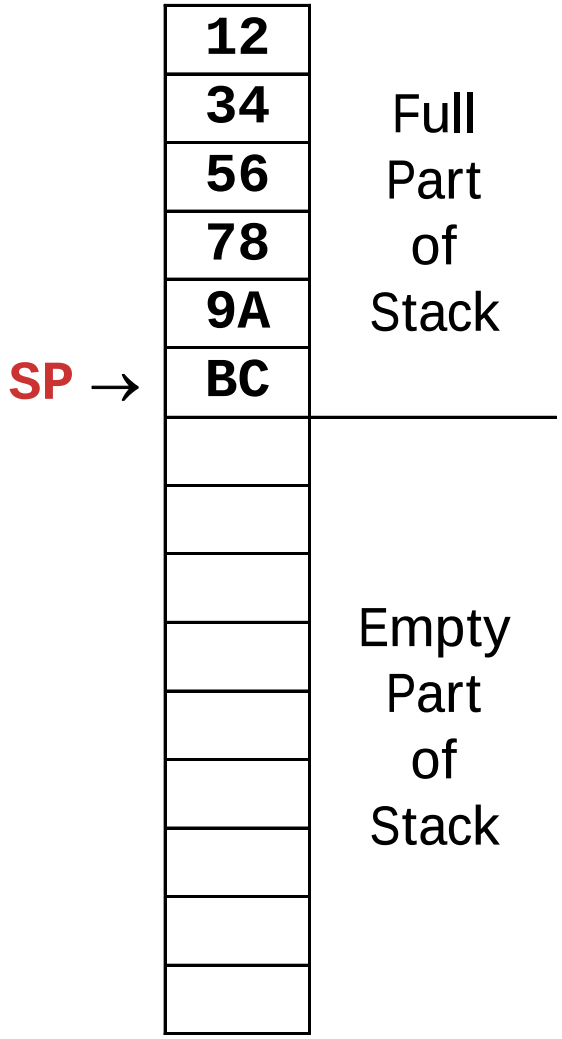
User Stack

x86 provides user stack mechanism

- Last In First Out **(LIFO)** buffer
- Store/Load 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



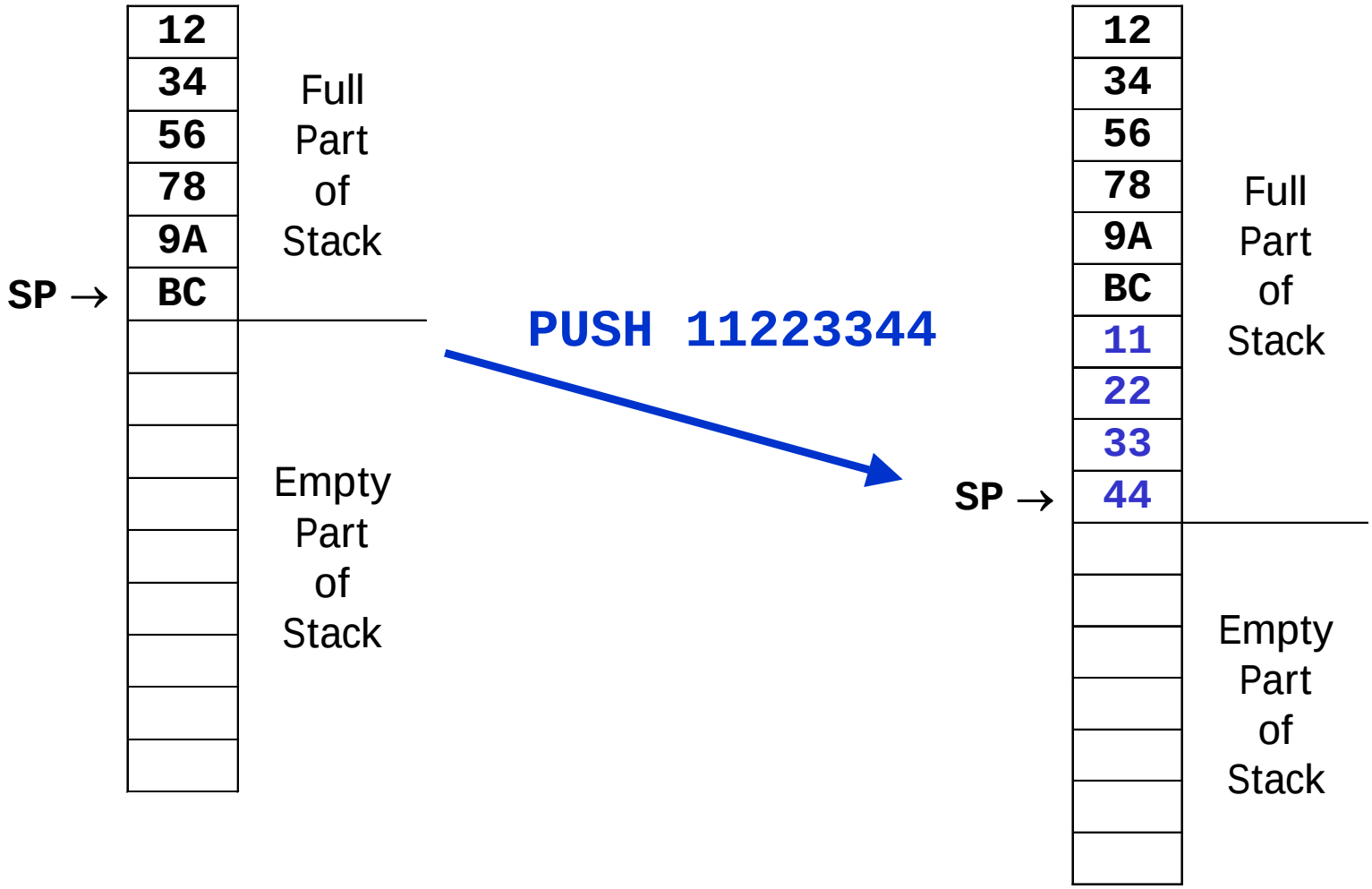
PUSH Operation — 1

PUSH src

$$SP \leftarrow SP - 4$$

$$[SP] \xleftarrow[\text{32-bits}]{} src$$

Example



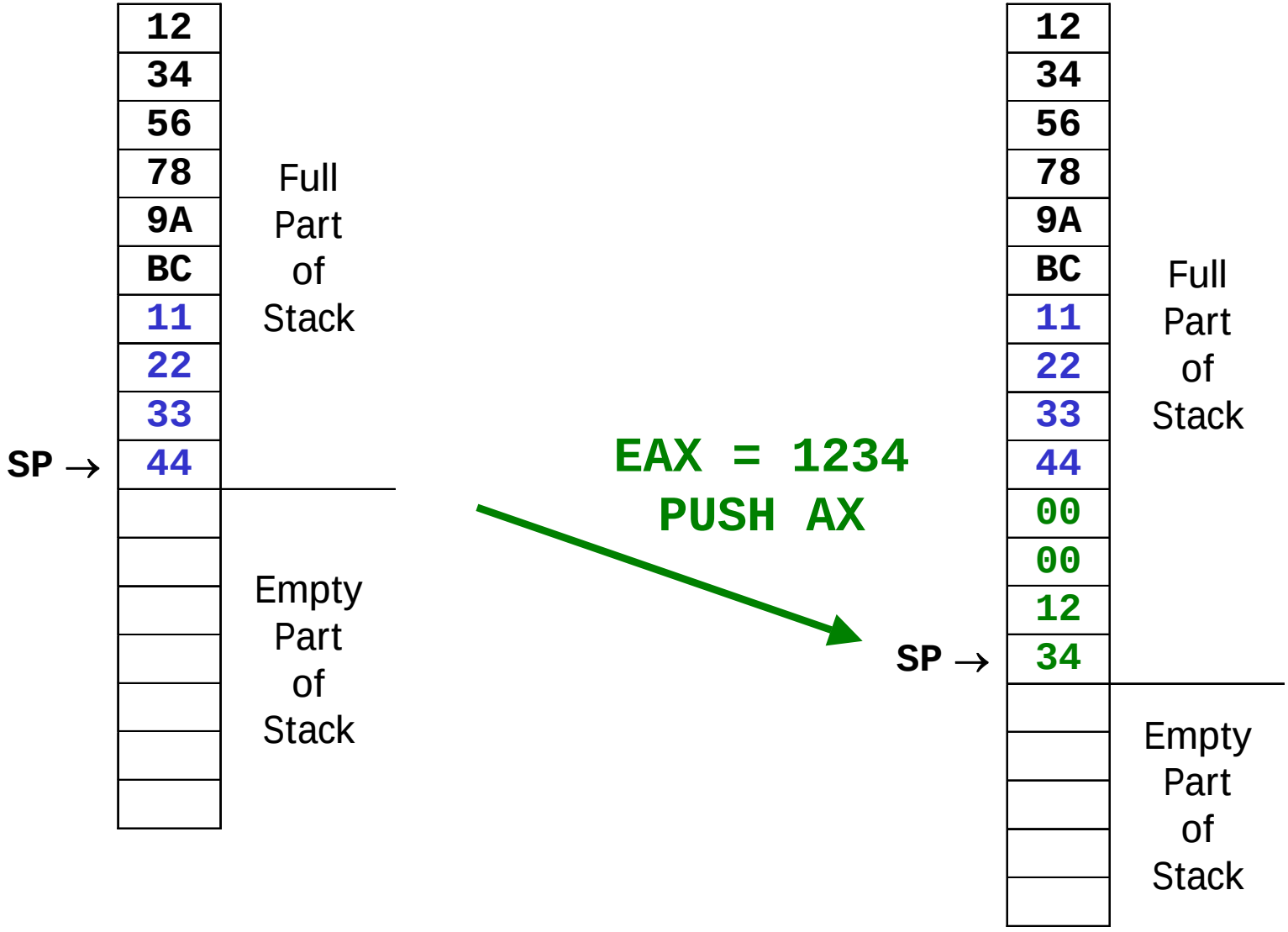
PUSH Operation — 2

PUSH src

SP ← SP - 4

[SP] ← src (32-bits)

Example



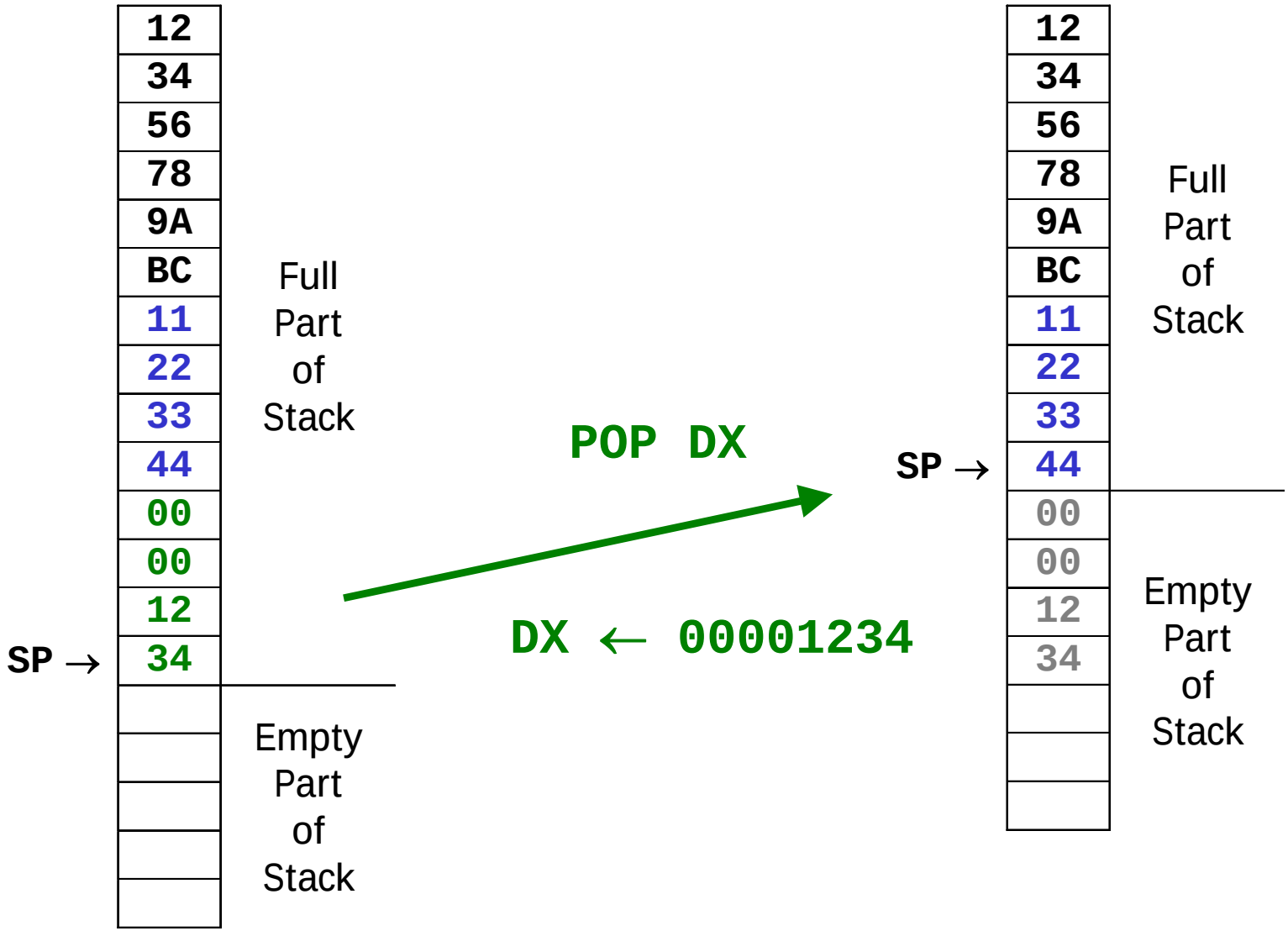
POP Operation

POP dest

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

$$SP \leftarrow SP + 4$$

Example



Using Push and Pop

Common application

Save current value of register or memory

Change value

Perform operations

Restore old value

Example

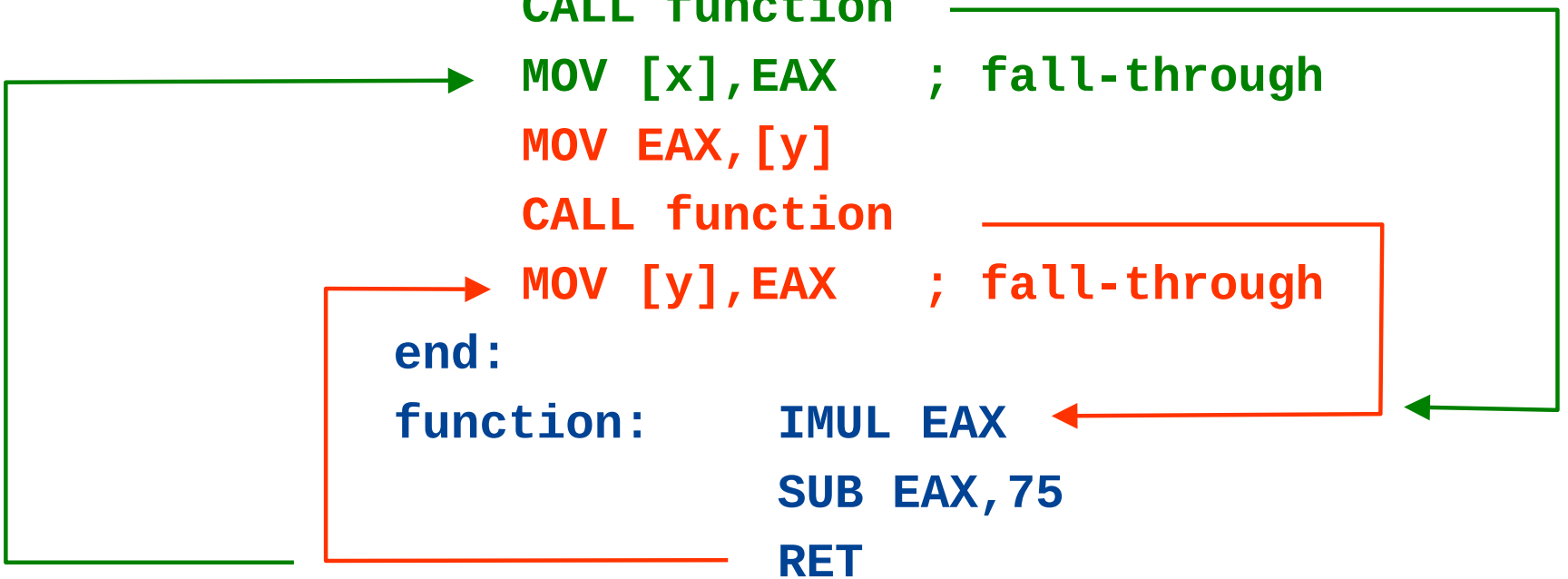
```
MOV EAX, [x]    ; copy [x] into EAX
SHL EAX, 1      ; EAX ← EAX * 2
CMP EAX, [y]
JLE end
PUSH EAX        ; save value of EAX
MOV EAX, 1234   ; overwrite EAX
IMUL [y]        ; EAX ← EAX * [y] (requires EAX)
MOV [y], EAX
POP EAX         ; restore previous value of EAX
```

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 EAX,[x]
     CALL function
     MOV [x],EAX    ; fall-through
     MOV EAX,[y]
     CALL function
     MOV [y],EAX    ; fall-through
end:
function:    IMUL EAX
             SUB EAX,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

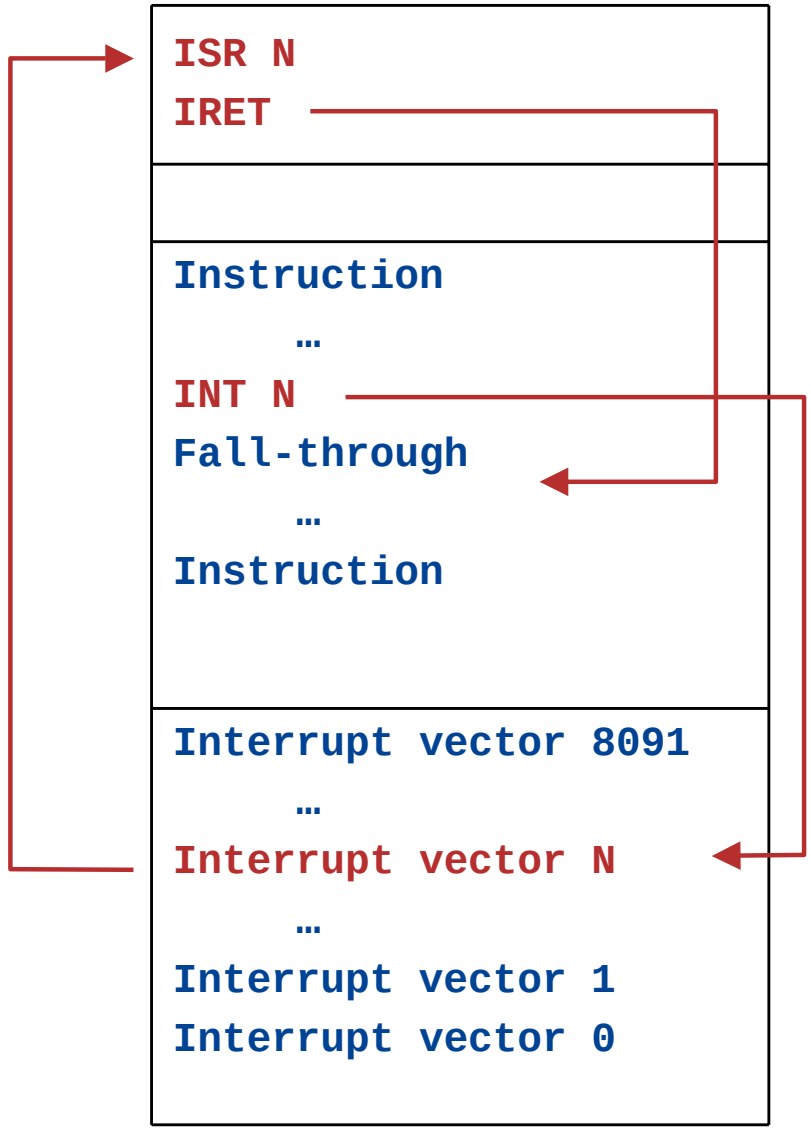
ISR address stored in OS table

Interrupt number **`N`** = Index into table

Return from interrupt

`IRET` instruction

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



Interrupt Example

Linux service to terminate program

```
mov eax, 1                  ; Linux terminate code
mov ebx, 0                  ; exit code 0
int 0x80                    ; call Linux kernel
```