Chapter 3.1: Programming Languages

Microprocessors

Software and language levels | Machine language | A sample instruction format

Assembly languages | “Welcome” on different platforms | Compiled languages

Interpreted languages | One program in 14 languages + 8086

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/microprocessors/slides/Chapter_03_01_Programming_Languages.html

Learning Outcomes

Language levels

Explain the difference between machine language, assembly language and high-level languages.

Machine code

Encode and decode binary instructions for a given instruction format.

Assembly

Explain how the assembler and linker turn source code into an executable; read short programs that make system calls.

Platform dependence

Explain why the same program is written differently for x86, 68000, ARM, PowerPC and different operating systems.

Compilation and interpretation

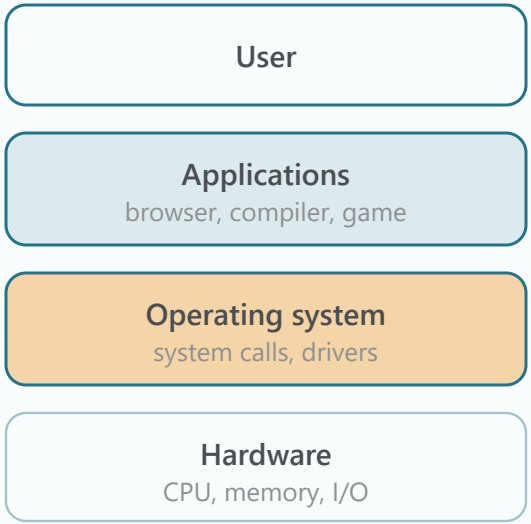
Compare compilers, linkers, interpreters and JIT.

Reading across languages

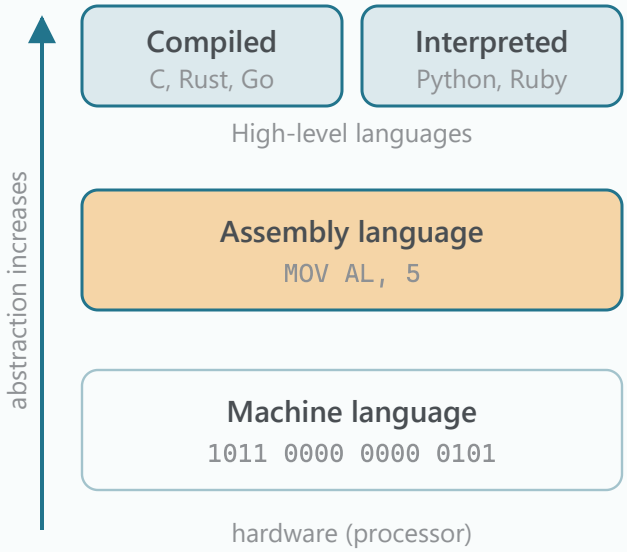
Trace one algorithm in several languages and in 8086 assembly.

Software

- Software makes the hardware work **correctly, in concert** and as intended.
- **The operating system**
  - is the computer's core software;
  - it manages the hardware and controls how other software runs.
- **Programming languages**
  - exist to build software;
  - there are many languages aimed at different fields.
- The evolution of programming languages mirrors the major advances in computer technology.



Programming Languages: Levels



1. **Machine languages**: the 0s and 1s the processor runs directly.
2. **Assembly languages**: a symbolic (mnemonic) notation for machine instructions.
3. **High-level languages**
   - **compiled**: translated to machine language first, then run;
   - **interpreted**: processed line by line at run time.

The higher the level, the more **portability** and **readability**, and the less **direct control** over the hardware.

One Operation at Three Levels: `x = a + b`

High level (C)

```
x = a + b;
```

Where the variables live and which registers to use is left to the compiler.

One Operation at Three Levels: `x = a + b`

High level (C)

```
x = a + b;
```

Where the variables live and which registers to use is left to the compiler.

Assembly language (8086)

```
MOV AX, [a]     ; load a into AX
ADD AX, [b]     ; AX = AX + b
MOV [x], AX     ; store result in x
```

Each line corresponds to one machine instruction.

One Operation at Three Levels: `x = a + b`

High level (C)

```
x = a + b;
```

Where the variables live and which registers to use is left to the compiler.

Assembly language (8086)

```
MOV AX, [a]     ; load a into AX
ADD AX, [b]     ; AX = AX + b
MOV [x], AX     ; store result in x
```

Each line corresponds to one machine instruction.

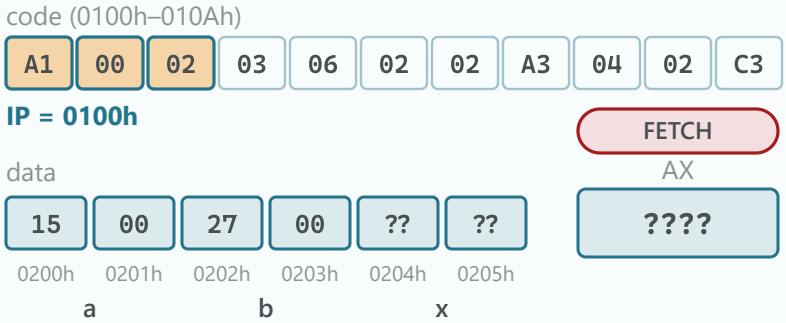
Machine language (a = 0200h, b = 0202h, x = 0204h)

```
A1 00 02        ; MOV AX, [0200h]
03 06 02 02     ; ADD AX, [0202h]
A3 04 02        ; MOV [0204h], AX
```

Addresses are stored low byte first (little-endian): 0200h → `00 02`.

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
▶ mov ax, [0200h]   ; A1 00 02
3 add ax, [0202h]   ; 03 06 02 02
4 mov [0204h], ax   ; A3 04 02
5 ret               ; C3
```



IP = 0100h. **Fetch:** A1 00 02 is read. A1 = “load a word from memory into AX”; the address bytes 00 02 → **0200h** (little-endian).

Step 1 / 7

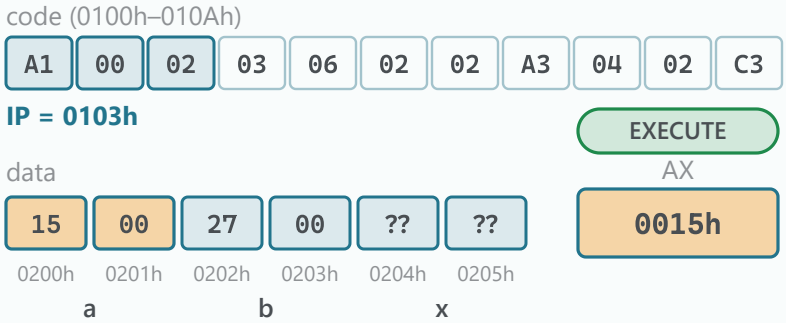
VARIABLES

`IP = 0100h` `AX = ????` `x = ????`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1  org 100h
▶  mov ax, [0200h]   ; A1 00 02
3  add ax, [0202h]   ; 03 06 02 02
4  mov [0204h], ax   ; A3 04 02
5  ret               ; C3
```



Execute: the bytes at 0200h and 0201h are 15 00 → AX = **0015h** (a = 21). IP = 0103h.

Step 2 / 7

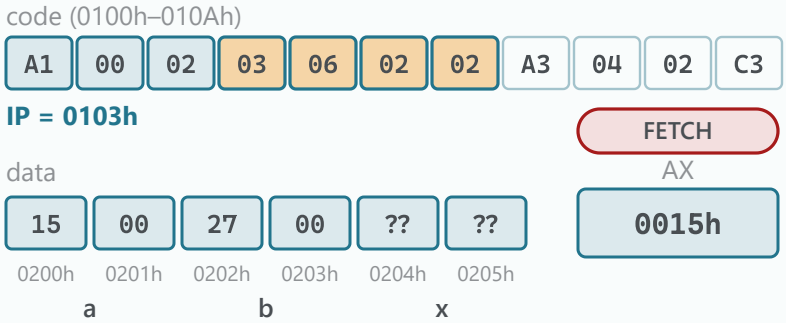
VARIABLES

`IP = 0103h` `AX = 0015h` `x = ????`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
2 mov ax, [0200h]   ; A1 00 02
▶ add ax, [0202h]   ; 03 06 02 02
4 mov [0204h], ax   ; A3 04 02
5 ret               ; C3
```



Fetch: **03 06 02 02** (4 bytes). 03 = ADD r16, r/m16; ModR/M 06 = “AX, direct address”; address 0202h.

Step 3 / 7

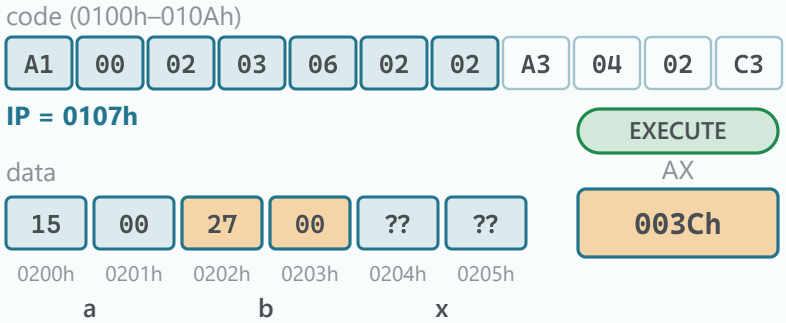
VARIABLES

`IP = 0103h` `AX = 0015h` `x = ????`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
2 mov ax, [0200h]    ; A1 00 02
▶ add ax, [0202h]    ; 03 06 02 02
4 mov [0204h], ax    ; A3 04 02
5 ret                ; C3
```



Execute: b = 0027h (39) is read, AX = 15h + 27h = **003Ch** (60). IP = 0107h.

Step 4 / 7

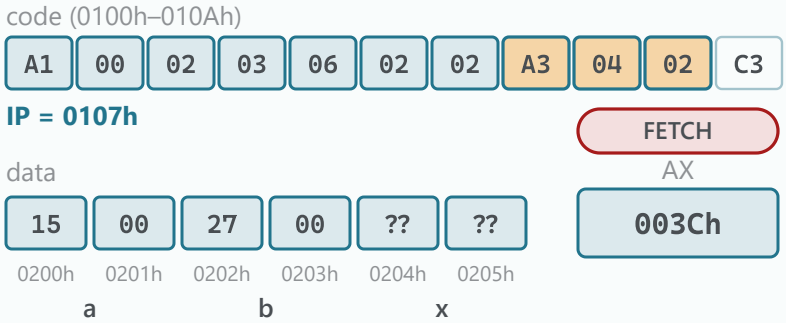
VARIABLES

`IP = 0107h` `AX = 003Ch` `x = ????`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
2 mov ax, [0200h]   ; A1 00 02
3 add ax, [0202h]   ; 03 06 02 02
▶ mov [0204h], ax   ; A3 04 02
5 ret               ; C3
```



Fetch: **A3 04 02**: A3 = “store AX to memory”, target address 0204h.

Step 5 / 7

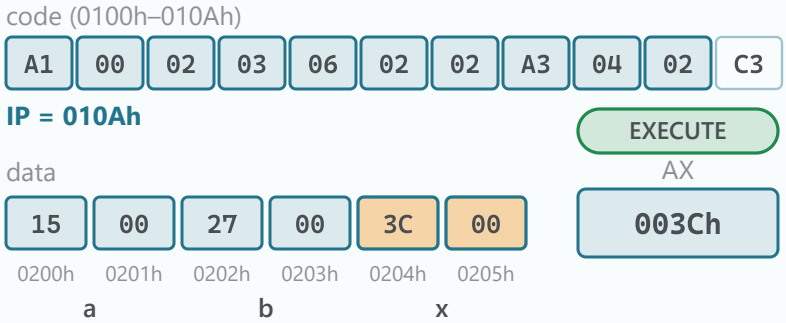
VARIABLES

`IP = 0107h` `AX = 003Ch` `x = ????`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
2 mov ax, [0200h]   ; A1 00 02
3 add ax, [0202h]   ; 03 06 02 02
▶ mov [0204h], ax   ; A3 04 02
5 ret               ; C3
```



Execute: AX is written low byte first: 0204h ← **3C**, 0205h ← **00**. x = 60.

Step 6 / 7

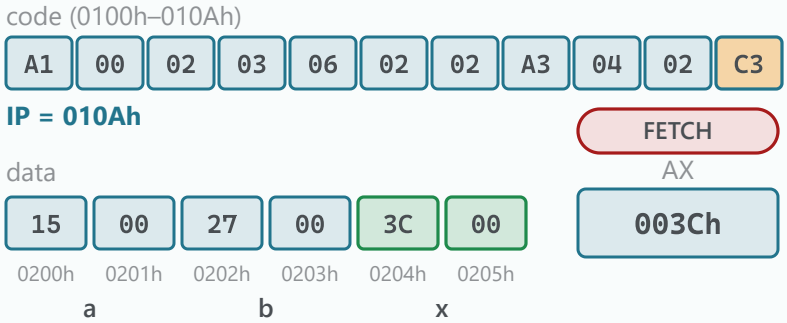
VARIABLES

`IP = 010Ah` `AX = 003Ch` `x = 003Ch`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

Code Trace: How Does `x = a + b` Run in Memory?

```
1 org 100h
2 mov ax, [0200h]   ; A1 00 02
3 add ax, [0202h]   ; 03 06 02 02
4 mov [0204h], ax   ; A3 04 02
▶ ret               ; C3
```



C3 = RET. The single C line `x = a + b;` became 3 instructions, 10 bytes of machine code.

Step 7 / 7

VARIABLES

`IP = 010Ah` `AX = 003Ch` `x = 003Ch`

Assumption: a = 0015h (21), b = 0027h (39); the data shares the code segment (COM program).

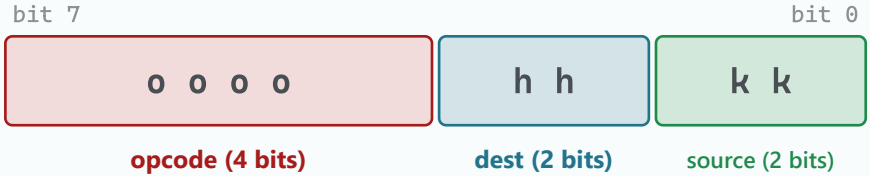
Machine Languages

- The **lowest-level** language, made of 0s and 1s; the only one the processor understands.
- Entirely **processor- and hardware-dependent**: machine code written for the 8086 does not run on ARM.
- **ENIAC** (1945), one of the first general-purpose electronic computers, was programmed directly in hardware with cables and switches.
- Today
  - every processor family has its own machine language (instruction set);
  - it is not used directly because it is impractical to read and write.
- **Assembly languages** reduce the dependence on machine language and make programming more human-friendly.



On the 8086 the bytes `B0 05` mean “load 5 into AL”. On ARM the same job is encoded as a completely different 32-bit word.

Designing a Sample Machine Language



Opcode	Meaning
`0000`	HALT: stop the program
`0001`	ADD: addition
`0010`	SUB: subtraction
`0011`	AND: logical AND
`0100`	OR: logical OR

- Instructions are **8 bits** long.
- 4 registers: R0 = `00`, R1 = `01`, R2 = `10`, R3 = `11`.
- Meaning: **dest ← dest op source**.
- A 4-bit opcode can define at most $2^4 = 16$ operations; 5 are used here.

Example: **`0001`** **`10`** **`11`** → ADD R2, R3 → R2 ← R2 + R3

Sample Program: Splitting the Fields

Machine code	Operation	Dest	Source	Meaning
`0001 0001`	ADD	R0	R1	R0 ← R0 + R1
`0011 1011`	AND	R2	R3	R2 ← R2 AND R3
`0100 0110`	OR	R1	R2	R1 ← R1 OR R2
`0000 0000`	HALT	—	—	stop the program

In this format the first 2-bit field is the **destination**, so the result goes to the **first** register (R0, R2, R1).

In hex: 11h, 3Bh, 46h, 00h.

Code Trace: The Sample Machine Program

```
1 0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
2 0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
3 0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
4 0000 0000   ; HALT
```

Initial values (example). PC = 0: the first instruction is about to be fetched.

Step 1 / 5

VARIABLES

PC = **0** R0 = **0000 0101 (5)**

R1 = **0000 0011 (3)**

R2 = **0000 1100 (12)**

R3 = **0000 1010 (10)**

Code Trace: The Sample Machine Program

```
▶ 0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
2 0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
3 0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
4 0000 0000   ; HALT
```

ADD R0, R1: 5 + 3 = 8 → R0. The source (R1) does not change.

Step 2 / 5

VARIABLES

PC = 1 | R0 = 0000 1000 (8)

R1 = 0000 0011 (3)

R2 = 0000 1100 (12)

R3 = 0000 1010 (10)

Code Trace: The Sample Machine Program

```
1  0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
▶  0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
3  0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
4  0000 0000   ; HALT
```

AND R2, R3: 1100 AND 1010 = **1000** → R2 = 8. Only bits that are 1 on both sides survive.

Step 3 / 5

VARIABLES

PC = **2**

R0 = **0000 1000 (8)**

R1 = **0000 0011 (3)**

R2 = **0000 1000 (8)**

R3 = **0000 1010 (10)**

Code Trace: The Sample Machine Program

```
1  0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
2  0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
▶  0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
4  0000 0000   ; HALT
```

OR R1, R2: 0011 OR 1000 = **1011** → R1 = 11.

Step 4 / 5

VARIABLES

PC = **3**

R0 = **0000 1000 (8)**

R1 = **0000 1011 (11)**

R2 = **0000 1000 (8)**

R3 = **0000 1010 (10)**

Code Trace: The Sample Machine Program

```
1 0001 0001   ; ADD R0, R1 → R0 ← R0 + R1
2 0011 1011   ; AND R2, R3 → R2 ← R2 AND R3
3 0100 0110   ; OR  R1, R2 → R1 ← R1 OR R2
▶ 0000 0000   ; HALT
```

HALT: the processor stops. Result: R0 = 8, R1 = 11, R2 = 8, R3 = 10.

Step 5 / 5

VARIABLES

PC = **3** R0 = **0000 1000 (8)**

R1 = **0000 1011 (11)**

R2 = **0000 1000 (8)**

R3 = **0000 1010 (10)**

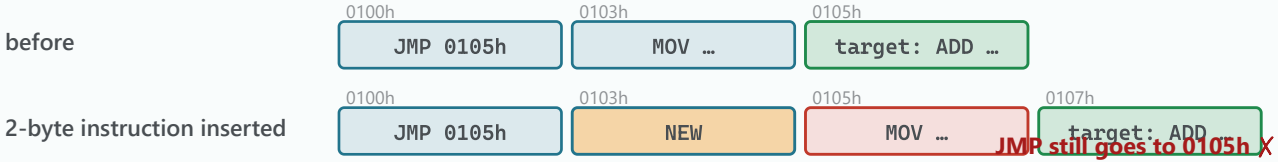
Machine Language: Pros and Cons

Pros

+ No translation needed; the processor runs it directly.
+ Full control over the hardware.
+ The smallest and (if written well) fastest code.

Cons

– Very hard to read, write and debug.
– Addresses are computed by hand; inserting one instruction breaks every jump address.
– Not portable: rewritten from scratch for each processor.
– A single bit error changes the program completely.



To fix these problems, first **assembly languages** (1950s) and then high-level languages such as **FORTRAN** (1957) were developed.

Assembly Languages

- Each machine instruction is written as a memorable **mnemonic**: `MOV AL, 5` instead of `10110000 00000101`.
- Each is tied to an architecture and instruction set: **Intel x86, Motorola 68000, ARM, PIC and AVR** each have their own assembly language.
- Microcontroller assembly languages are common in **embedded systems** and electronics.
- They let the programmer make the most of the hardware.
- The program that translates source code into machine language is the **assembler**.

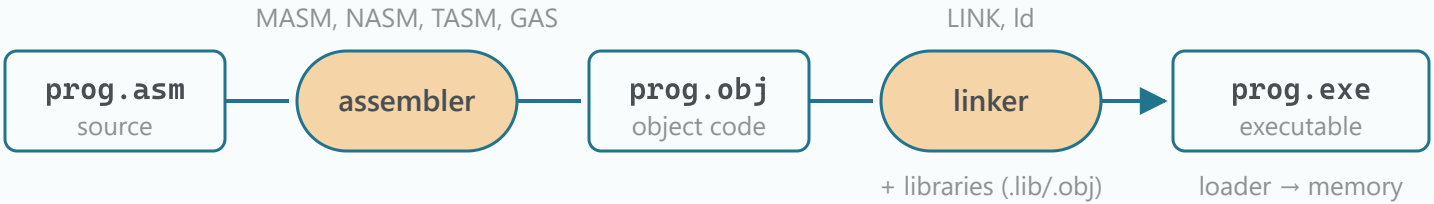


Where is it used?
bootloaders, device drivers, interrupt routines, OS kernels

Why learn it?
understanding how the processor works, debugging, reverse engineering

Performance
special instructions compilers cannot generate (SIMD, crypto) and critical loops

From Source Code to a Running Program



The assembler's **listing** shows each line's address and machine code side by side:

```
Address Machine code  Source
0100    B4 09         mov ah, 09h
0102    BA 0C 01      mov dx, offset msg
0105    CD 21         int 21h
0107    B8 00 4C      mov ax, 4C00h
010A    CD 21         int 21h
010C    57 65 6C ...  msg db 'Welcome!$'
```

emu8086 runs all these steps with one key and produces the COM/EXE file directly.

Code Trace: The Assembler's Two Passes

```
▶ org 100h
2 mov ah, 09h
3 mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
7 msg db 'Welcome!$'
```

PASS 1

location counter (LC)

0100h

generated bytes

(no bytes in pass 1)

symbol table

msg	?

line 1

Pass 1 only counts addresses. `org 100h` sets the location counter (LC) to **0100h**; a directive generates no bytes.

Step 1 / 9

VARIABLES

`pass = 1` `LC = 0100h` `msg = ?`

Code Trace: The Assembler's Two Passes

```
1 org 100h
▶ mov ah, 09h
3 mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
7 msg db 'Welcome!$'
```

PASS 1

location counter (LC)

0102h

generated bytes

(no bytes in pass 1)

symbol table

msg	?

line 2

`mov ah, 09h` takes 2 bytes → LC = **0102h**. The bytes themselves are not written yet.

Step 2 / 9

VARIABLES

`pass = 1` `LC = 0102h` `msg = ?`

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
▶ mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
7 msg db 'Welcome!$'
```

PASS 1

location counter (LC)

0105h

generated bytes

(no bytes in pass 1)

symbol table

msg	?

line 3

`offset msg` is a forward reference: msg is not defined yet, but the instruction is known to be 3 bytes → LC = **0105h**.

Step 3 / 9

VARIABLES

`pass = 1` `LC = 0105h` `msg = ?`

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
3 mov dx, offset msg
▶ int 21h
▶ mov ax, 4C00h
▶ int 21h
7 msg db 'Welcome!$'
```

PASS 1

location counter (LC)

010Ch

generated bytes

(no bytes in pass 1)

symbol table

msg	?

lines 4, 5, 6

int 21h (2), mov ax, 4C00h (3), int 21h (2): LC = **010Ch**.

Step 4 / 9

VARIABLES

pass = **1** LC = **010Ch** msg = **?**

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
3 mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
▶ msg db 'Welcome!$'
```

PASS 1

location counter (LC)

0115h

generated bytes

(no bytes in pass 1)

symbol table

msg	010Ch

line 7

The label **msg** enters the **symbol table** with the current LC: msg = 010Ch. A 9-byte string → LC = 0115h.

Step 5 / 9

VARIABLES

`pass = 1` `LC = 0115h` `msg = 010Ch`

Code Trace: The Assembler's Two Passes

```
1 org 100h
▶ mov ah, 09h
3 mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
7 msg db 'Welcome!$'
```

PASS 2

location counter (LC)

0100h

symbol table

msg	010Ch

line 2

generated bytes

B4 09

Pass 2 starts over and generates the bytes: `mov ah, 09h` → **B4 09**.

Step 6 / 9

VARIABLES

pass = 2 | LC = 0100h | msg = 010Ch

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
▶ mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
7 msg db 'Welcome!$'
```

PASS 2

location counter (LC)

0102h

symbol table

msg	010Ch

line 3

generated bytes

B4 09 BA 0C 01

`offset msg` now comes from the table: 010Ch → **BA 0C 01** (address little-endian).

Step 7 / 9

VARIABLES

pass = 2 | LC = 0102h | msg = 010Ch

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
3 mov dx, offset msg
  int 21h
  mov ax, 4C00h
  int 21h
7 msg db 'Welcome!$'
```

PASS 2

location counter (LC)

0105h

symbol table

msg	010Ch

lines 4, 5, 6

generated bytes

B4	09	BA	0C	01	CD	21	B8	00	4C	CD	21

The remaining instructions: **CD 21**, **B8 00 4C**, **CD 21**.

Step 8 / 9

VARIABLES

`pass = 2` `LC = 0105h` `msg = 010Ch`

Code Trace: The Assembler's Two Passes

```
1 org 100h
2 mov ah, 09h
3 mov dx, offset msg
4 int 21h
5 mov ax, 4C00h
6 int 21h
▶ msg db 'Welcome!$'
```

PASS 2

location counter (LC)

`010Ch`

symbol table

msg	010Ch

line 7

generated bytes

B4	09	BA	0C	01	CD	21	B8	00	4C	CD	21	57	65	6C	63
6F	6D	65	21	24											

The string's ASCII codes are written: 57 65 6C … 24 ('$'). The output is a 21-byte COM file; when run it prints **Welcome!**.

Step 9 / 9

VARIABLES

`pass = 2` `LC = 010Ch` `msg = 010Ch`

Assemblers and Their Syntax

Assembler	Syntax · use
MASM	Intel · Microsoft, Windows
TASM	Intel · Borland, DOS
NASM	Intel · open source, cross-platform
GAS (as)	AT&T · GNU/Linux, gcc
emu8086	Intel (MASM-like) · 8086 emulator, this course

One instruction, two notations

```
; Intel:  dest, source
mov eax, 5
add eax, [ebx+4]

# AT&T:  source, dest
movl $5, %eax
addl 4(%ebx), %eax
```

In AT&T, registers take `%` and constants take `$`; the size is given by an instruction suffix (`l` = 32 bits).

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
▶ hello  db 'Welcome, Linux!', 10
▶ len    equ $ - hello      ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4      ; sys_write
8    mov ebx, 1      ; stdout
9    mov ecx, hello  ; string address
10   mov edx, len    ; byte count
11   int 0x80        ; call the kernel
12   mov eax, 1      ; sys_exit
13   xor ebx, ebx    ; exit code 0
14   int 0x80
```

Data section: the string plus a newline (10). `len` is computed as **16** by `$ - hello`.

Step 1 / 9

VARIABLES

EAX = ???????? EBX = ????????

ECX = ???????? EDX = ????????

OUTPUT

“Welcome, Linux!” is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello      ; = 16 bytes
4  section .text
5  global _start
6  _start:
▶    mov eax, 4      ; sys_write
8    mov ebx, 1      ; stdout
9    mov ecx, hello  ; string address
10   mov edx, len    ; byte count
11   int 0x80        ; call the kernel
12   mov eax, 1      ; sys_exit
13   xor ebx, ebx    ; exit code 0
14   int 0x80
```

The program starts at `_start`. EAX = 4: the **sys_write** system call number.

Step 2 / 9

VARIABLES

EAX = 00000004 | EBX = ???????? | ECX = ???????? | EDX = ????????

OUTPUT

“Welcome, Linux!” is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello     ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4     ; sys_write
▶    mov ebx, 1     ; stdout
9    mov ecx, hello ; string address
10   mov edx, len   ; byte count
11   int 0x80       ; call the kernel
12   mov eax, 1     ; sys_exit
13   xor ebx, ebx   ; exit code 0
14   int 0x80
```

EBX = 1: the file descriptor to write to, **stdout** (the screen).

Step 3 / 9

VARIABLES

EAX = 00000004 | EBX = 00000001 | ECX = ???????? | EDX = ????????

OUTPUT

"Welcome, Linux!" is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello      ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4      ; sys_write
8    mov ebx, 1      ; stdout
▶    mov ecx, hello  ; string address
10   mov edx, len    ; byte count
11   int 0x80        ; call the kernel
12   mov eax, 1      ; sys_exit
13   xor ebx, ebx    ; exit code 0
14   int 0x80
```

ECX = the string's **address** (not its contents).

Step 4 / 9

VARIABLES

EAX = **00000004** EBX = **00000001**

ECX = **hello (addr)**

EDX = **????????**

OUTPUT

"Welcome, Linux!" is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello     ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4     ; sys_write
8    mov ebx, 1     ; stdout
9    mov ecx, hello ; string address
▶    mov edx, len   ; byte count
11   int 0x80       ; call the kernel
12   mov eax, 1     ; sys_exit
13   xor ebx, ebx   ; exit code 0
14   int 0x80
```

EDX = 16 (10h): the number of bytes to write.

Step 5 / 9

VARIABLES

EAX = **00000004** EBX = **00000001**

ECX = **hello (addr)**

EDX = **00000010**

OUTPUT

"Welcome, Linux!" is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello     ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4     ; sys_write
8    mov ebx, 1     ; stdout
9    mov ecx, hello ; string address
10   mov edx, len   ; byte count
▶    int 0x80       ; call the kernel
12   mov eax, 1     ; sys_exit
13   xor ebx, ebx   ; exit code 0
14   int 0x80
```

`int 0x80` : the kernel is called and the text is written. On return EAX = bytes written (16).

Step 6 / 9

VARIABLES

EAX = **00000010** EBX = **00000001**

ECX = **hello (addr)**

EDX = **00000010**

OUTPUT

```
Welcome, Linux!
```

“Welcome, Linux!” is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello     ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4     ; sys_write
8    mov ebx, 1     ; stdout
9    mov ecx, hello ; string address
10   mov edx, len   ; byte count
11   int 0x80       ; call the kernel
▶    mov eax, 1     ; sys_exit
13   xor ebx, ebx   ; exit code 0
14   int 0x80
```

EAX = 1: **sys_exit**.

Step 7 / 9

VARIABLES

EAX = 00000001 | EBX = 00000001 | ECX = hello (addr) | EDX = 00000010

OUTPUT

```
Welcome, Linux!
```

"Welcome, Linux!" is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello      ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4      ; sys_write
8    mov ebx, 1      ; stdout
9    mov ecx, hello  ; string address
10   mov edx, len    ; byte count
11   int 0x80        ; call the kernel
12   mov eax, 1      ; sys_exit
▶    xor ebx, ebx    ; exit code 0
14   int 0x80
```

`xor ebx, ebx` : a number XORed with itself is 0 → exit code 0 (success). Shorter than `mov ebx, 0` .

Step 8 / 9

VARIABLES

EAX = **00000001** EBX = **00000000**

ECX = **hello (addr)**

EDX = **00000010** ZF = **1** CF = **0**

OUTPUT

```
Welcome, Linux!
```

“Welcome, Linux!” is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

x86 Assembly: Linux (32-bit, NASM)

```
1  section .data
2  hello  db 'Welcome, Linux!', 10
3  len    equ $ - hello      ; = 16 bytes
4  section .text
5  global _start
6  _start:
7    mov eax, 4      ; sys_write
8    mov ebx, 1      ; stdout
9    mov ecx, hello  ; string address
10   mov edx, len    ; byte count
11   int 0x80        ; call the kernel
12   mov eax, 1      ; sys_exit
13   xor ebx, ebx    ; exit code 0
▶    int 0x80
```

The kernel ends the process with exit code 0; this call never returns.

Step 9 / 9

VARIABLES

EAX = **00000001** EBX = **00000000**

ECX = **hello (addr)**

EDX = **00000010** ZF = **1** CF = **0**

OUTPUT

```
Welcome, Linux!
```

"Welcome, Linux!" is 15 bytes; the trailing newline (10) makes the string 16 bytes. `equ $ - hello` lets the assembler compute the length.

How Does a System Call Work?



Environment	Call	Service number	Parameters
Linux 32-bit	`int 0x80`	EAX (4 = write, 1 = exit)	EBX, ECX, EDX
Linux 64-bit	`syscall`	RAX (1 = write, 60 = exit)	RDI, RSI, RDX
MS-DOS	`int 21h`	AH (09h = print, 4Ch = exit)	DX, AL …
BIOS	`int 10h`	AH (0Eh = print char)	AL, BH …

x86 Assembly: DOS (emu8086, COM)

```
▶ org 100h          ; COM program
2 mov ah, 09h       ; DOS 09h: print string
3 lea dx, msg       ; DS:DX = address
4 int 21h           ; call DOS
5 mov ax, 4C00h     ; DOS 4Ch: exit
6 int 21h
7 msg db 'Welcome, DOS!$'
```

`org 100h` : a COM program is loaded at offset 100h of its segment (the 256 bytes before it are the PSP). CS = DS = ES = SS.

Step 1 / 6

VARIABLES

IP = **0100h** AH = **??** DX = **????**

OUTPUT

Service 09h expects a **'$'** terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: DOS (emu8086, COM)

```
1 org 100h          ; COM program
▶ mov ah, 09h       ; DOS 09h: print string
3 lea dx, msg       ; DS:DX = address
4 int 21h           ; call DOS
5 mov ax, 4C00h     ; DOS 4Ch: exit
6 int 21h
7 msg db 'Welcome, DOS!$'
```

AH = 09h: the DOS service “print a string **ending in $**”.

Step 2 / 6

VARIABLES

`IP = 0102h` `AH = 09h` `DX = ????`

OUTPUT

Service 09h expects a **'$'** terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: DOS (emu8086, COM)

```
1 org 100h          ; COM program
2 mov ah, 09h       ; DOS 09h: print string
▶ lea dx, msg       ; DS:DX = address
4 int 21h           ; call DOS
5 mov ax, 4C00h     ; DOS 4Ch: exit
6 int 21h
7 msg db 'Welcome, DOS!$'
```

`lea dx, msg` : DS:DX points to the string. The code is 12 bytes (0100h–010Bh), so msg is at **010Ch**.

Step 3 / 6

VARIABLES

IP = **0105h** AH = **09h** DX = **010Ch**

OUTPUT

Service 09h expects a **'$'** terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: DOS (emu8086, COM)

```
1 org 100h          ; COM program
2 mov ah, 09h       ; DOS 09h: print string
3 lea dx, msg       ; DS:DX = address
▶ int 21h           ; call DOS
5 mov ax, 4C00h     ; DOS 4Ch: exit
6 int 21h
7 msg db 'Welcome, DOS!$'
```

`int 21h`: DOS prints characters until it sees '$' (the '$' itself is not printed).

Step 4 / 6

VARIABLES

IP = 0107h AH = 09h DX = 010Ch

OUTPUT

```
Welcome, DOS!
```

Service 09h expects a **'$'** terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: DOS (emu8086, COM)

```
1 org 100h          ; COM program
2 mov ah, 09h       ; DOS 09h: print string
3 lea dx, msg       ; DS:DX = address
4 int 21h           ; call DOS
▶ mov ax, 4C00h     ; DOS 4Ch: exit
6 int 21h
7 msg db 'Welcome, DOS!$'
```

AX = 4C00h: AH = 4Ch “terminate the program”, AL = 00h exit code.

Step 5 / 6

VARIABLES

`IP = 010Ah` `AH = 4Ch` `AL = 00h` `DX = 010Ch`

OUTPUT

```
Welcome, DOS!
```

Service 09h expects a **'$'** terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: DOS (emu8086, COM)

```
1 org 100h          ; COM program
2 mov ah, 09h       ; DOS 09h: print string
3 lea dx, msg       ; DS:DX = address
4 int 21h           ; call DOS
5 mov ax, 4C00h     ; DOS 4Ch: exit
▶ int 21h
7 msg db 'Welcome, DOS!$'
```

The program ends and control returns to DOS. The data sits **after the code**, so the processor never mistakes it for instructions.

Step 6 / 6

VARIABLES

IP = 010Ch | AH = 4Ch | AL = 00h | DX = 010Ch

OUTPUT

```
Welcome, DOS!
```

Service 09h expects a '$' terminator; the data goes after the code; placed before it, it would be executed. `mov ax, 4C00h` also clears AL (the exit code).

x86 Assembly: Windows (Win32 API, NASM)

```
extern MessageBoxA    ; user32.dll
extern ExitProcess    ; kernel32.dll

section .data
msg db 'Welcome, Windows!', 0

section .text
global _start
_start:
    push 0          ; uType = MB_OK
    push msg        ; lpCaption (title)
    push msg        ; lpText (text)
    push 0          ; hWnd = NULL
    call MessageBoxA
    push 0          ; uExitCode = 0
    call ExitProcess
```

Differences

- No system call: OS services are called as **DLL functions**.
- Parameters are pushed **onto the stack** right to left: `MessageBoxA(hWnd, lpText, lpCaption, uType)` in C.
- **stdcall**: the called function cleans up the stack.
- Strings end **with 0** (C strings).
- Linking may need the decorated name: `_MessageBoxA@16` (16 = parameter bytes).

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
▶  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
12   call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



Start: ESP = **0012FF80h** (sample value). The stack grows toward lower addresses.

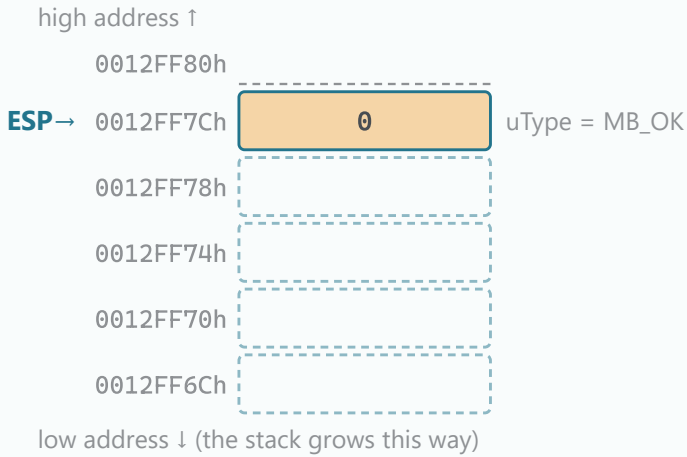
Step 1 / 8

VARIABLES

`ESP = 0012FF80h` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
▶    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
12   call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



`push 0` : ESP drops by 4; the **last** parameter (uType) is pushed first.

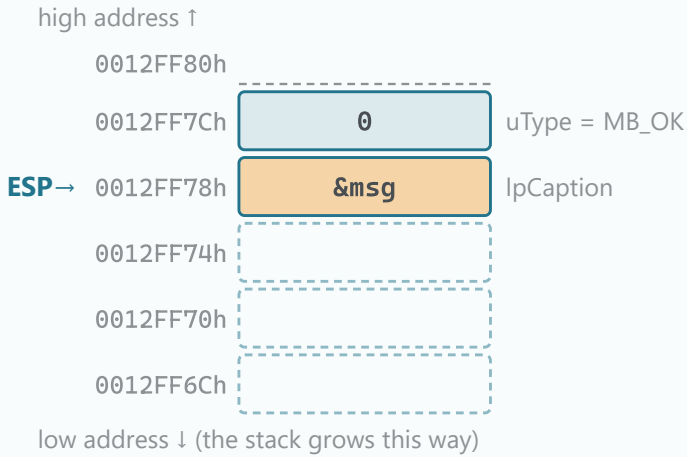
Step 2 / 8

VARIABLES

`ESP = 0012FF7Ch` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
▶    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
12   call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



`push msg` : the **address** of the caption string is pushed.

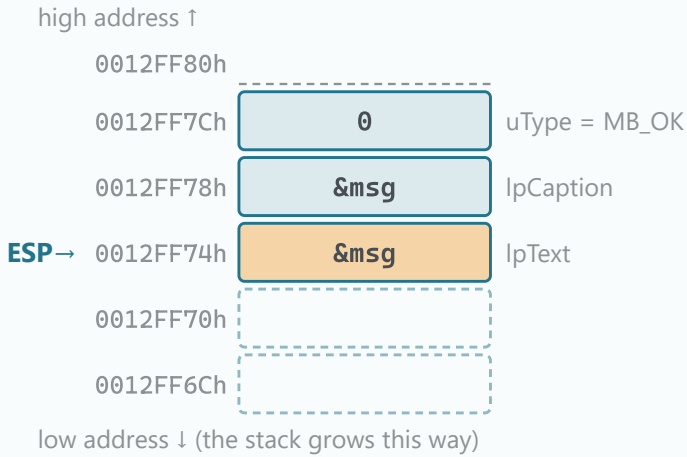
Step 3 / 8

VARIABLES

`ESP = 0012FF78h` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
▶    push msg      ; lpText
11   push 0        ; hWnd
12   call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



`push msg` : the address of the text string.

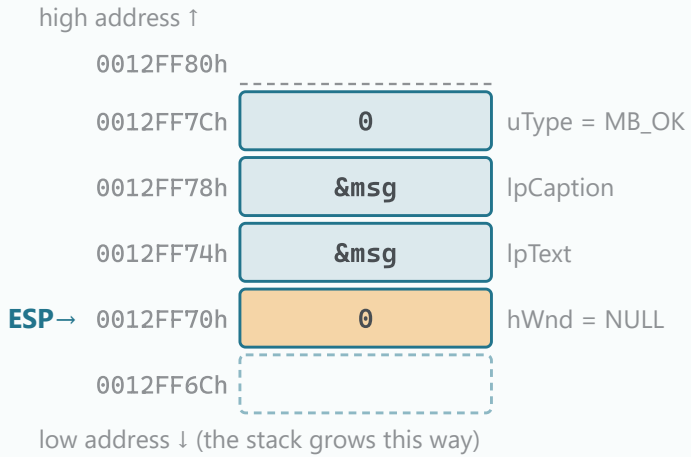
Step 4 / 8

VARIABLES

`ESP = 0012FF74h` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
▶    push 0        ; hWnd
12   call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



`push 0` : the **first** parameter (hWnd) is pushed last → it sits on top. Parameters go on **right to left**.

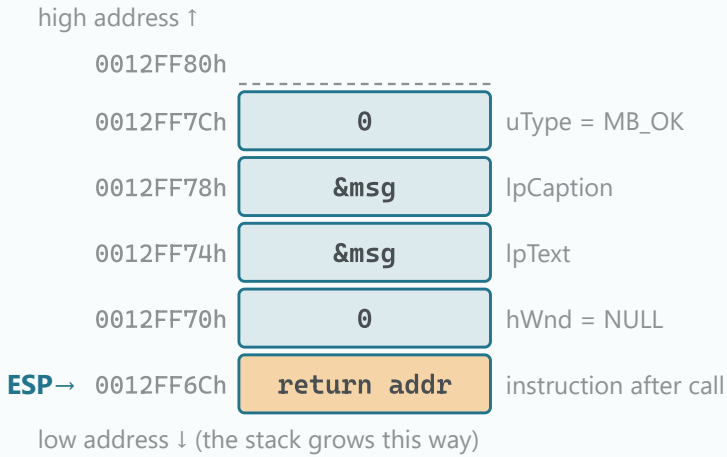
Step 5 / 8

VARIABLES

`ESP = 0012FF70h` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
▶    call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



`call MessageBoxA` : the return address is pushed and control jumps to the function; a message box opens.

Step 6 / 8

VARIABLES

`ESP = 0012FF6Ch` `EAX = ????????h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
▶    call MessageBoxA
13   push 0        ; uExitCode
14   call ExitProcess
```



When the user clicks OK, the function returns with `ret 16`: under **stdcall** the callee cleans the stack, so ESP is 0012FF80h again. EAX = 1 (IDOK).

Step 7 / 8

VARIABLES

`ESP = 0012FF80h` `EAX = 00000001h`

Code Trace: Parameters onto the Stack

```
1  extern MessageBoxA
2  extern ExitProcess
3  section .data
4  msg db 'Welcome...', 0
5  section .text
6  global _start
7  _start:
8    push 0        ; uType
9    push msg      ; lpCaption
10   push msg      ; lpText
11   push 0        ; hWnd
12   call MessageBoxA
▶    push 0        ; uExitCode
▶    call ExitProcess
```



`push 0` + `call ExitProcess` : the process ends with exit code 0; this call never returns.

Step 8 / 8

VARIABLES

`ESP = 0012FF7Ch` `EAX = 00000001h`

Motorola 68000 Assembly: Amiga

```
    move.l  4.w,a6          ; a6 = ExecBase
    lea     dosname(pc),a1  ; library name
    moveq   #0,d0           ; any version
    jsr     -552(a6)        ; OpenLibrary()
    tst.l   d0
    beq.s   quit            ; failed → quit
    move.l  d0,a6           ; a6 = DOSBase
    jsr     -60(a6)         ; Output() → d0
    move.l  d0,d1           ; file handle
    move.l  #msg,d2         ; string address
    moveq   #16,d3          ; byte count
    jsr     -48(a6)         ; Write(d1,d2,d3)
    move.l  a6,a1
    move.l  4.w,a6
    jsr     -414(a6)        ; CloseLibrary()
quit moveq  #0,d0           ; return code 0
    rts
dosname dc.b 'dos.library',0
msg     dc.b 'Welcome, Amiga!',10
```

Differences

- 8 data registers (`d0–d7`) and 8 address registers (`a0–a7`); **source, destination** order.
- Size suffixes: `.b` , `.w` , `.l` ; constants take `#` .
- Services are called at negative offsets from the library base (`-48(a6)`).
- The library is opened, used and closed.

ARM Assembly: Linux (32-bit, GNU as)

```
    .data
msg:    .ascii "Welcome, ARM!\n"
len = . - msg          @ 14 bytes

    .text
    .global _start
_start:
    mov  r7, #4        @ sys_write
    mov  r0, #1        @ stdout
    ldr  r1, =msg      @ string address
    mov  r2, #len      @ byte count
    svc  #0            @ call the kernel
    mov  r7, #1        @ sys_exit
    mov  r0, #0        @ exit code
    svc  #0
```

GNU as uses `.ascii`, not `db`; `. - msg` lets the assembler compute the length.

Differences

- RISC: 16 registers (`r0–r15`), fixed 32-bit instructions.
- The service number goes in **r7**, parameters in **r0–r2**.
- Entering the kernel: `svc #0` (formerly `swi`).
- Memory is read and written only with `ldr` / `str` .

PowerPC (PPC) Assembly: Linux (32-bit)

```
    .data
msg:    .ascii "Welcome, PowerPC!\n"
len = . - msg           # 18 bytes

    .text
    .globl _start
_start:
    li   0, 4           # r0 = sys_write
    li   3, 1           # r3 = stdout
    lis  4, msg@ha      # r4 = addr (high)
    addi 4, 4, msg@l    #    + low half
    li   5, len         # r5 = byte count
    sc                  # call the kernel
    li   0, 1           # r0 = sys_exit
    li   3, 0           # r3 = exit code
    sc
```

Differences

- RISC: 32 general registers; GNU as writes registers as plain numbers (`3` = r3).
- The service number goes in **r0**, parameters in **r3, r4, r5**; the trap is `sc`.
- A 32-bit address is loaded in two 16-bit halves (`@ha` , `@l`).

One “Welcome”, Six Environments

Environment	Processor	Entering the OS	Service no.	Parameters	String end
Linux	x86 (32)	`int 0x80`	EAX	EBX, ECX, EDX	length given
DOS	8086	`int 21h`	AH	DX (address)	`'$'`
Windows	x86 (32)	`call` to a DLL	function name	stack	`0`
AmigaOS	68000	`jsr -48(a6)`	library offset	d1, d2, d3	length given
Linux	ARM	`svc #0`	r7	r0, r1, r2	length given
Linux	PowerPC	`sc`	r0	r3, r4, r5	length given

Assembly code depends on both the **processor** (instruction set, registers) and the **operating system** (call mechanism, numbers). Even the comment character depends on the syntax: `;` on x86, `@` in ARM GNU, `#` on PowerPC. The big win of high-level languages is handing this dependence to the compiler or interpreter.

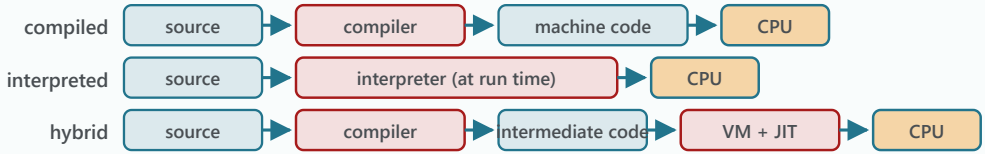
High-Level Languages

Compiled languages

- Source code is translated to **machine language** ahead of time.
- Most errors are caught **at compile time**.
- The code runs **faster**.
- Stages: **compiling** (compiler) and **linking** (linker).
- The output is specific to one processor + operating system.

Interpreted languages

- Source code is processed by an interpreter **at run time**.
- Allows **fast development** and experimentation.
- Source code is **platform-independent**: it runs wherever an interpreter exists.
- Errors may only appear when the line in question runs.
- Usually slower.



The hybrid approach: Java and C# are first compiled to intermediate code (bytecode/IL); the virtual machine translates it to machine language while running (**JIT**, just-in-time). Python and JavaScript engines also use bytecode/JIT internally.

The Compilation Process (C, gcc)

1 · Preprocessor `#include` and `#define` are expanded → `oddeven.i` (gcc -E)

Step 1 / 5

In one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly: `gcc -S -masm=intel oddeven.c`.

The Compilation Process (C, gcc)

1 · Preprocessor `#include` and `#define` are expanded → `oddeven.i` (gcc -E)

2 · Compiler C code is syntax- and type-checked and translated to **assembly** → `oddeven.s` (gcc -S)

In one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly: `gcc -S -masm=intel oddeven.c`.

The Compilation Process (C, gcc)

1 · Preprocessor `#include` and `#define` are expanded → `oddeven.i` (gcc -E)

2 · Compiler C code is syntax- and type-checked and translated to **assembly** → `oddeven.s` (gcc -S)

3 · Assembler Assembly is translated to machine code → object file `oddeven.o` (gcc -c)

In one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly: `gcc -S -masm=intel oddeven.c`.

The Compilation Process (C, gcc)

1 · Preprocessor `#include` and `#define` are expanded → `oddeven.i` (gcc -E)

2 · Compiler C code is syntax- and type-checked and translated to **assembly** → `oddeven.s` (gcc -S)

3 · Assembler Assembly is translated to machine code → object file `oddeven.o` (gcc -c)

4 · Linker The object file is combined with library code such as `printf` / `scanf` → executable `oddeven` / `oddeven.exe`

In one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly: `gcc -S -masm=intel oddeven.c`.

The Compilation Process (C, gcc)

1 · Preprocessor `#include` and `#define` are expanded → `oddeven.i` (gcc -E)

2 · Compiler C code is syntax- and type-checked and translated to **assembly** → `oddeven.s` (gcc -S)

3 · Assembler Assembly is translated to machine code → object file `oddeven.o` (gcc -c)

4 · Linker The object file is combined with library code such as `printf` / `scanf` → executable `oddeven` / `oddeven.exe`

5 · Loader The operating system loads the program into memory and jumps to `main`.

In one command: `gcc oddeven.c -o oddeven`. To see the intermediate assembly: `gcc -S -masm=intel oddeven.c`.

Code Trace: A Loop the Compiler Generated

```
1  org 100h
▶  mov bx, 0          ; sum = 0
▶  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
0	10

10 + 7 + 4 + 1 = 22

The compiler put the variables in registers: sum → **BX**, i → **CX**. The while condition is tested at the top of the loop.

Step 1 / 11

VARIABLES

BX = 0 | CX = 10 | ZF = 0 | SF = 0

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
▶  cond: cmp cx, 0    ; i > 0 ?
▶  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
0	10

`cmp: i − 0 → ZF=0 SF=0 OF=0`

jle not taken → body

10 + 7 + 4 + 1 = 22

`cmp cx, 0` + `jle` : i = 10 > 0, so no jump; the body runs (the C condition was inverted into an “exit” jump).

Step 2 / 11

VARIABLES

BX = 0 | CX = 10 | ZF = 0 | SF = 0

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
▶  add bx, cx         ; sum += i
▶  sub cx, 3          ; i -= 3
▶  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
10	7

10 + 7 + 4 + 1 = 22

Body: sum = **10**, i = **7**; `jmp cond` goes back to the test.

Step 3 / 11

VARIABLES

BX = **10** CX = **7** ZF = **0** SF = **0**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
▶  cond: cmp cx, 0    ; i > 0 ?
▶  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
10	7

`cmp: i − 0 → ZF=0 SF=0 OF=0`

jle not taken → body

10 + 7 + 4 + 1 = 22

`cmp cx, 0` + `jle` : i = 7 > 0, so no jump; the body runs (the C condition was inverted into an “exit” jump).

Step 4 / 11

VARIABLES

BX = 10 | CX = 7 | ZF = 0 | SF = 0

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
▶  add bx, cx         ; sum += i
▶  sub cx, 3          ; i -= 3
▶  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
17	4

10 + 7 + 4 + 1 = 22

Body: sum = **17**, i = **4**; `jmp cond` goes back to the test.

Step 5 / 11

VARIABLES

BX = **17** CX = **4** ZF = **0** SF = **0**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
▶  cond: cmp cx, 0    ; i > 0 ?
▶  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
17	4

`cmp: i − 0 → ZF=0 SF=0 OF=0`

jle not taken → body

10 + 7 + 4 + 1 = 22

`cmp cx, 0` + `jle` : i = 4 > 0, so no jump; the body runs (the C condition was inverted into an “exit” jump).

Step 6 / 11

VARIABLES

BX = **17** CX = **4** ZF = **0** SF = **0**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
▶  add bx, cx         ; sum += i
▶  sub cx, 3          ; i -= 3
▶  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
21	1

10 + 7 + 4 + 1 = 22

Body: sum = **21**, i = **1**; `jmp cond` goes back to the test.

Step 7 / 11

VARIABLES

BX = **21** CX = **1** ZF = **0** SF = **0**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
▶  cond: cmp cx, 0    ; i > 0 ?
▶  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
21	1

`cmp: i − 0 → ZF=0 SF=0 OF=0`

jle not taken → body

10 + 7 + 4 + 1 = 22

`cmp cx, 0` + `jle` : i = 1 > 0, so no jump; the body runs (the C condition was inverted into an “exit” jump).

Step 8 / 11

VARIABLES

BX = **21** CX = **1** ZF = **0** SF = **0**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
▶  add bx, cx         ; sum += i
▶  sub cx, 3          ; i -= 3
▶  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
22	−2

10 + 7 + 4 + 1 = 22

Body: sum = **22**, i = **-2**; `jmp cond` goes back to the test.

Step 9 / 11

VARIABLES

BX = **22** CX = **−2** ZF = **0** SF = **1**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
▶  cond: cmp cx, 0    ; i > 0 ?
▶  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
9  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
22	−2

`cmp: i − 0 → ZF=0 SF=1 OF=0`

jle taken → done

10 + 7 + 4 + 1 = 22

i = -2: the condition i > 0 is false (i ≤ 0, SF ≠ OF) → `jle done` **jumps** and the loop exits.

Step 10 / 11

VARIABLES

BX = **22** CX = **−2** ZF = **0** SF = **1**

Code Trace: A Loop the Compiler Generated

```
1  org 100h
2  mov bx, 0          ; sum = 0
3  mov cx, 10         ; i = 10
4  cond: cmp cx, 0    ; i > 0 ?
5  jle done
6  add bx, cx         ; sum += i
7  sub cx, 3          ; i -= 3
8  jmp cond
▶  done: mov sum, bx
10 ret
11 sum dw ?
```

C source

```
sum = 0;
i = 10;
while (i > 0) {
sum += i;
i -= 3;
}
```

sum (BX)	i (CX)
22	−2

sum ← BX = 22

10 + 7 + 4 + 1 = 22

The result is stored: sum = **22**. One C loop became a compare, a conditional jump and an unconditional jump.

Step 11 / 11

VARIABLES

BX = **22** CX = **−2** ZF = **0** SF = **1**

Compiled Languages

- The **compiler** examines the whole source before it runs.
- Compiled code runs directly on the processor, so it is **faster**.
- Type and syntax errors are caught **at compile time**.

C and C++
systems programming, operating systems, embedded software, high-performance applications

Rust
safe, fast systems programming (memory safety checked by the compiler)

Fortran
scientific and numerical computing; the first widely used high-level language (1957)

Go
server and network software; fast compilation, concurrency

Ada
safety-critical systems such as aviation, defense and railways

Java, C# (hybrid)
compiled to intermediate code, run with a JIT on a virtual machine

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

Every sample program does the same job: read a number, then print whether each number from 1 to it is odd or even.

Step 1 / 10

VARIABLES

`number = ?`

OUTPUT

```
Enter a number:
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

The user enters 3. `scanf` gets the variable's **address** (`&number`).

Step 2 / 10

VARIABLES

`number = 3`

OUTPUT

```
Enter a number: 3
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

The loop starts: i = 1, the condition 1 ≤ 3 is true.

Step 3 / 10

VARIABLES

number = **3** i = **1**

OUTPUT

```
Enter a number: 3
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

1 % 2 = 1 ≠ 0 → the `else` branch: “odd”.

Step 4 / 10

VARIABLES

`number = 3` `i = 1` `i % 2 = 1`

OUTPUT

```
Enter a number: 3
1 is odd.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

`i++` → 2; 2 ≤ 3 is true.

Step 5 / 10

VARIABLES

`number = 3` `i = 2`

OUTPUT

```
Enter a number: 3
1 is odd.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

2 % 2 = 0 → “even”.

Step 6 / 10

VARIABLES

`number = 3` `i = 2` `i % 2 = 0`

OUTPUT

```
Enter a number: 3
1 is odd.
2 is even.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

i = 3; 3 ≤ 3 is true.

Step 7 / 10

VARIABLES

`number = 3` `i = 3`

OUTPUT

```
Enter a number: 3
1 is odd.
2 is even.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

3 % 2 = 1 → “odd”.

Step 8 / 10

VARIABLES

`number = 3` `i = 3`

`i % 2 = 1`

OUTPUT

```
Enter a number: 3
1 is odd.
2 is even.
3 is odd.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

i = 4; 4 ≤ 3 is **false** → the loop ends.

Step 9 / 10

VARIABLES

`number = 3` `i = 4`

OUTPUT

```
Enter a number: 3
1 is odd.
2 is even.
3 is odd.
```

Task: Odd/Even from 1 to n — C

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            printf("%d is even.\n", i);
        } else {
            printf("%d is odd.\n", i);
        }
    }
    return 0;
}
```

`return 0` : a “success” exit code for the operating system.

Step 10 / 10

VARIABLES

`number = 3` `return = 0`

OUTPUT

```
Enter a number: 3
1 is odd.
2 is even.
3 is odd.
```

HIGH-LEVEL LANGUAGES · COMPILED

C++

```cpp
#include <iostream>

int main() {
    int number;
    std::cout << "Enter a number: ";
    std::cin >> number;
    for (int i = 1; i <= number; i++) {
        if (i % 2 == 0) {
            std::cout << i << " is even."
                      << std::endl;
        } else {
            std::cout << i << " is odd."
                      << std::endl;
        }
    }
    return 0;
}
```

C++ (1985, Bjarne Stroustrup)

- A superset of C; classes, templates, exceptions.
- Streams for I/O: `std::cout <<` writes, `std::cin >>` reads.
- Build: `g++ oddeven.cpp -o oddeven`
- `std::endl` adds a newline and flushes the buffer; for many lines `'\n'` is faster.

C#

```
using System;

class Program {
  static void Main() {
    Console.Write("Enter a number: ");
    int number = int.Parse(Console.ReadLine());
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        Console.WriteLine($"{i} is even.");
      } else {
        Console.WriteLine($"{i} is odd.");
      }
    }
  }
}
```

C# (2000, Microsoft)

- The .NET platform; compiled to the **IL** intermediate language, which the CLR JIT-compiles to machine code at run time.
- `$"{i} …"` : string interpolation.
- Run: `dotnet run`
- `int.Parse` throws an exception on non-numeric input; `int.TryParse` is safer.

Java

```java
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int number = scanner.nextInt();
    for (int i = 1; i <= number; i++) {
      if (i % 2 == 0) {
        System.out.println(i + " is even.");
      } else {
        System.out.println(i + " is odd.");
      }
    }
  }
}
```

Java (1995, Sun Microsystems)

- `javac` → platform-independent **bytecode** (`.class`) → JIT on the JVM.
- "Write once, run anywhere."
- Run: `javac Main.java` and `java Main`
- Everything lives in a class; the file name = the `public` class name.

Go

```go
package main

import "fmt"

func main() {
    var number int
    fmt.Print("Enter a number: ")
    fmt.Scan(&number)
    for i := 1; i <= number; i++ {
        if i%2 == 0 {
            fmt.Printf("%d is even.\n", i)
        } else {
            fmt.Printf("%d is odd.\n", i)
        }
    }
}
```

Go (2009, Google)

- Compiles straight to machine code; one self-contained executable.
- A garbage collector and lightweight threads (**goroutines**).
- Run: `go run oddeven.go`
- No parentheses around conditions and loops; `:=` infers the type.

Rust

```rust
use std::io;

fn main() {
    let mut input = String::new();
    println!("Enter a number:");
    io::stdin().read_line(&mut input)
        .expect("read error");
    let number: i32 = input.trim().parse()
        .expect("parse error");
    for i in 1..=number {
        if i % 2 == 0 {
            println!("{} is even.", i);
        } else {
            println!("{} is odd.", i);
        }
    }
}
```

Rust (2015, Mozilla)

- **Memory safety** without a garbage collector: ownership rules are checked by the compiler.
- Variables are immutable by default; you need `mut`.
- `1..=number` : a range that includes the upper bound.
- Run: `rustc oddeven.rs` or `cargo run`

Fortran

```
program Main
    implicit none
    integer :: number, i
    write(*,*) 'Enter a number:'
    read(*,*) number
    do i = 1, number
        if (mod(i, 2) == 0) then
            write(*,*) i, ' is even.'
        else
            write(*,*) i, ' is odd.'
        end if
    end do
end program Main
```

Fortran (1957, IBM · John Backus)

- The first widely used high-level language: FORmula TRANslation.
- Still common in scientific computing (weather forecasting, physics).
- `implicit none`: every variable must be declared explicitly.
- Run: `gfortran oddeven.f90 -o oddeven`
- `write(*,*)` uses free format; numbers may print with leading spaces.

Ada

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is
   Number : Integer;
begin
   Put ("Enter a number: ");
   Get (Number);
   for I in 1 .. Number loop
      Put (I, Width => 0);
      if I mod 2 = 0 then
         Put_Line (" is even.");
      else
         Put_Line (" is odd.");
      end if;
   end loop;
end Main;
```

Ada (1983, US Department of Defense)

- Strong type checking; aviation and defense software.
- Run: `gnatmake main.adb`
- Integer `Get` / `Put` need `Ada.Integer_Text_IO`; `Put_Line` adds a newline.

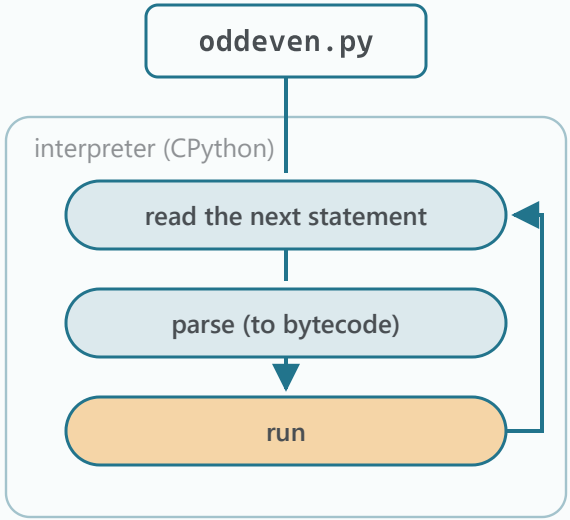
Interpreted Languages

- Source code is interpreted **at run time**; there is no separate compile step.
- Fast development and experimentation (interactive shell, REPL).

Python
general-purpose, high-level; data science, AI

JavaScript
web development; runs in the browser and in Node.js

Ruby
ease of use and flexibility; Ruby on Rails



HIGH-LEVEL LANGUAGES · INTERPRETED

Python

```python
# Read input from the user
number = int(input("Enter a number: "))

# Check the numbers from 1 to number
for i in range(1, number + 1):
    if i % 2 == 0:
        print(f"{i} is even.")
    else:
        print(f"{i} is odd.")
```

Python (1991, Guido van Rossum)

- Blocks are marked by **indentation**, not braces.
- `range(1, number + 1)`: the upper bound is **excluded**, hence the +1.
- `input` returns text; `int(...)` converts it to a number.
- Run: `python oddeven.py`
- Dynamically typed: a variable's type is decided at run time.

The job that takes 15 lines in C takes 6 here: the **expressiveness** of high-level languages. The price: CPython runs the same loop tens of times slower than C.

HIGH-LEVEL LANGUAGES · INTERPRETED (JIT)

JavaScript

```
// Read input from the user
let number = Number(prompt("Enter a number:"));

// Check the numbers from 1 to number
for (let i = 1; i <= number; i++) {
    if (i % 2 === 0) {
        console.log(i + " is even.");
    } else {
        console.log(i + " is odd.");
    }
}
```

JavaScript (1995, Brendan Eich)

- The language of browsers; runs on servers too with Node.js.
- Modern engines (V8) JIT-compile the code to machine code.
- `prompt` returns **text**; convert it with `Number(...)`.
- `===`: equality without type conversion.

Ruby

```ruby
# Read input from the user
print "Enter a number: "
number = gets.chomp.to_i

# Check the numbers from 1 to number
for i in 1..number do
  if i % 2 == 0
    puts "#{i} is even."
  else
    puts "#{i} is odd."
  end
end
```

Ruby (1995, Yukihiro Matsumoto)

- Everything is an object: `gets.chomp.to_i` is a chain of method calls.
- `chomp` drops the newline, `to_i` converts to an integer.
- `1..number`: a range that includes the upper bound.
- `"#{i}"`: string interpolation.
- Run: `ruby oddeven.rb`

PHP

```php
<?php
// Read input from the user
echo "Enter a number: ";
$number = (int) trim(fgets(STDIN));

// Check the numbers from 1 to number
for ($i = 1; $i <= $number; $i++) {
    if ($i % 2 == 0) {
        echo $i . " is even.\n";
    } else {
        echo $i . " is odd.\n";
    }
}
?>
```

PHP (1995, Rasmus Lerdorf)

- A server-side web language; can be embedded in HTML.
- Variables start with `$`; `.` concatenates strings.
- `fgets(STDIN)` reads a line from the command line; `trim` drops the newline.
- Run: `php oddeven.php`

Perl

```perl
use strict;
use warnings;

# Read input from the user
print "Enter a number: ";
my $number = <STDIN>;
chomp $number;

# Check the numbers from 1 to number
for my $i (1..$number) {
    if ($i % 2 == 0) {
        print "$i is even.\n";
    } else {
        print "$i is odd.\n";
    }
}
```

Perl (1987, Larry Wall)

- Text processing and system administration; powerful regular expressions.
- `<STDIN>` reads a line, `chomp` drops the newline.
- In a double-quoted string `$i` is replaced by its value.
- `use strict; use warnings;` catch common mistakes.
- Run: `perl oddeven.pl`

MATLAB

```
% Read input from the user
number = input('Enter a number: ');

% Check the numbers from 1 to number
for i = 1:number
    if mod(i, 2) == 0
        fprintf('%d is even.\n', i);
    else
        fprintf('%d is odd.\n', i);
    end
end
```

MATLAB (1984, MathWorks)

- Matrix-based numerical computing; common in engineering, especially signal processing and control.
- `1:number`: the vector 1, 2, ..., number.
- A `;` at the end of a line suppresses printing the result.
- A free, compatible alternative: GNU Octave.

One Program, Fourteen Languages

Language	Model	Loop header	Remainder	Output
C	compiled	`for (int i = 1; i <= n; i++)`	`i % 2`	`printf`
C++	compiled	`for (int i = 1; i <= n; i++)`	`i % 2`	`std::cout <<`
C#	IL + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`Console.WriteLine`
Java	bytecode + JIT	`for (int i = 1; i <= n; i++)`	`i % 2`	`System.out.println`
Go	compiled	`for i := 1; i <= n; i++`	`i%2`	`fmt.Printf`
Rust	compiled	`for i in 1..=n`	`i % 2`	`println!`
Fortran	compiled	`do i = 1, n`	`mod(i, 2)`	`write(*,*)`
Ada	compiled	`for I in 1 .. N loop`	`I mod 2`	`Put_Line`
Python	interpreted	`for i in range(1, n + 1):`	`i % 2`	`print`
JavaScript	interpreted/JIT	`for (let i = 1; i <= n; i++)`	`i % 2`	`console.log`
Ruby	interpreted	`for i in 1..n do`	`i % 2`	`puts`
PHP	interpreted	`for ($i = 1; $i <= $n; $i++)`	`$i % 2`	`echo`
Perl	interpreted	`for my $i (1..$n)`	`$i % 2`	`print`
MATLAB	interpreted/JIT	`for i = 1:n`	`mod(i, 2)`	`fprintf`

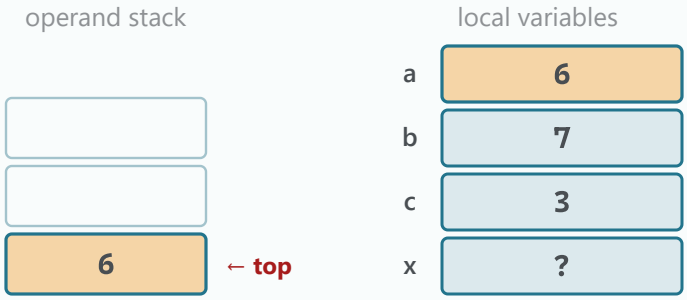
Compiled, Interpreted, Hybrid

Criterion	Compiled	Interpreted	Hybrid (bytecode + JIT)
Translation time	once, before running	on every run	first to intermediate code, then at run time
Speed	highest	lowest	high (after warm-up)
What ships	machine code	source code	intermediate code (.class, .dll)
Portability	recompile for each platform	anywhere with an interpreter	anywhere with a virtual machine
Error detection	mostly at compile time	mostly at run time	type errors at compile time
Examples	C, C++, Go, Rust, Fortran, Ada	Python, Ruby, PHP, Perl, MATLAB	Java, C#, JavaScript (V8)

"Compiled/interpreted" is a property of the **implementation**, not the language: there are interpreters for C and compilers for Python (Cython, PyPy JIT).

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
▶ iload a       // push a
3 iload b       // push b
4 iadd          // add the top two
5 iload c
6 imul          // multiply the top
  two
7 istore x      // pop into x
8 return
```



For `x = (a + b) * c` the compiler generated stack-based **intermediate code** (bytecode). Locals: a = 6, b = 7, c = 3. `iload a` : local a = 6 is pushed onto the stack.

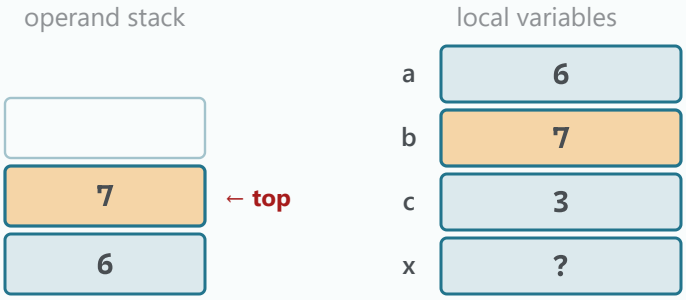
Step 1 / 7

VARIABLES

`stack = [6]` `x = ?`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
▶ iload b       // push b
4 iadd          // add the top two
5 iload c
6 imul          // multiply the top
  two
7 istore x      // pop into x
8 return
```



`iload b` : local b = 7 is pushed onto the stack.

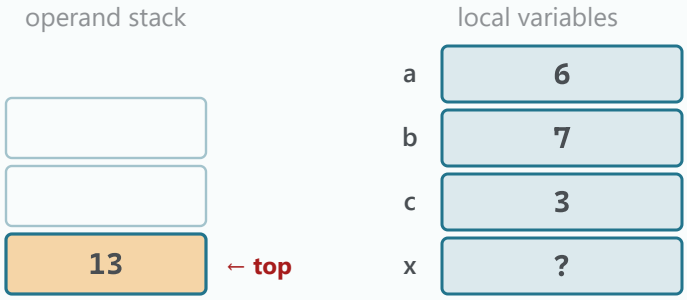
Step 2 / 7

VARIABLES

`stack = [6, 7]` `x = ?`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
3 iload b       // push b
▶ iadd          // add the top two
5 iload c
6 imul          // multiply the top
  two
7 istore x      // pop into x
8 return
```



`iadd` : the top two items (6, 7) are popped and the sum **13** is pushed. No operand names; everything is on the stack.

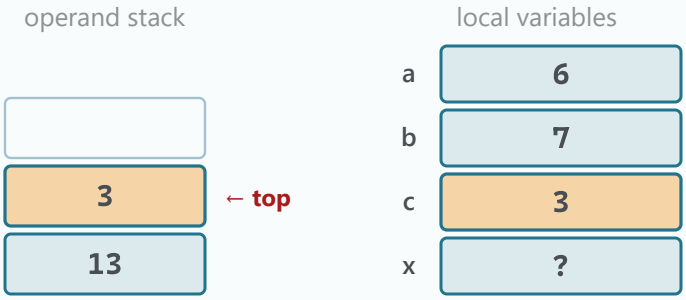
Step 3 / 7

VARIABLES

`stack = [13]` `x = ?`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
3 iload b       // push b
4 iadd          // add the top two
▶ iload c
6 imul          // multiply the top
  two
7 istore x      // pop into x
8 return
```



`iload c` : local c = 3 is pushed onto the stack.

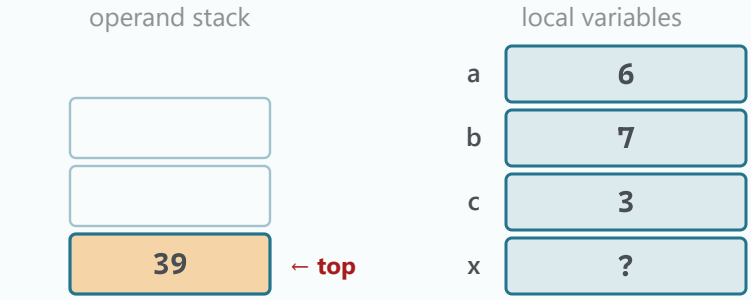
Step 4 / 7

VARIABLES

`stack = [13, 3]` `x = ?`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
3 iload b       // push b
4 iadd          // add the top two
5 iload c
▶ imul          // multiply the top
  two
7 istore x      // pop into x
8 return
```



`imul` : 13 × 3 = **39** is pushed.

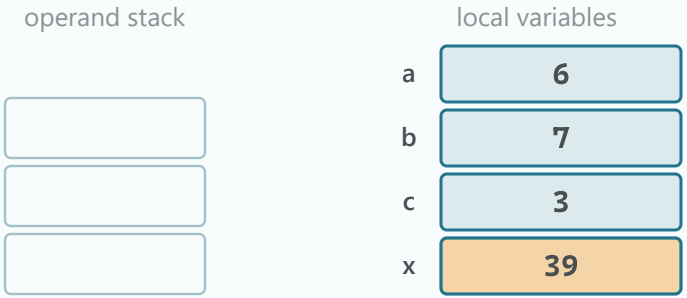
Step 5 / 7

VARIABLES

`stack = [39]` `x = ?`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
3 iload b       // push b
4 iadd          // add the top two
5 iload c
6 imul          // multiply the top
  two
▶ istore x      // pop into x
8 return
```



`istore x` : the top of the stack is stored in x → x = **39**; the stack is empty.

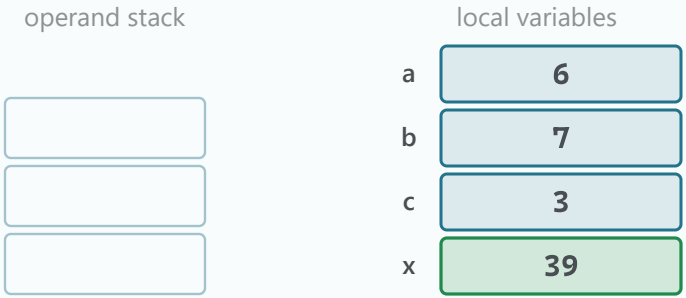
Step 6 / 7

VARIABLES

`stack = []` `x = 39`

Code Trace: How Does Bytecode Run?

```
1 // x = (a + b) * c;
2 iload a       // push a
3 iload b       // push b
4 iadd          // add the top two
5 iload c
6 imul          // multiply the top
  two
7 istore x      // pop into x
▶ return
```



This intermediate code is tied to no processor: the virtual machine (JVM, CLR) either interprets each instruction or JIT-compiles it into real machine code such as 8086/x86.

Step 7 / 7

VARIABLES

`stack = []` `x = 39`

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

CX serves as the counter (i): i = 1. n = 3 is hard-coded; it is not read from the keyboard.

Step 1 / 19

VARIABLES

CX = **0001h**

OUTPUT

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov   cx, 1        ; i = 1
3  again:  cmp   cx, 3        ; compare i with n
4          ja    done         ; done if i > n
5          mov   dl, cl
6          add   dl, '0'      ; digit → ASCII
7          mov   ah, 02h
8          int   21h          ; print DL
9          test  cl, 1        ; lowest bit
10         jnz   odd          ; 1 means odd
11         lea   dx, even_m
12         jmp   print
13 odd:    lea   dx, odd_m
14 print:  mov   ah, 09h
15         int   21h          ; print string
16         inc   cx           ; i++
17         jmp   again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

`cmp cx, 3` : computes 1 − 3 (the result is discarded) → ZF=0, CF=1 (borrow). `ja` jumps only if CF=0 and ZF=0 → **no jump**.

Step 2 / 19

VARIABLES

CX = **0001h** ZF = **0** CF = **1**

OUTPUT

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov   cx, 1        ; i = 1
3  again:  cmp   cx, 3        ; compare i with n
4          ja    done         ; done if i > n
▶          mov   dl, cl
▶          add   dl, '0'      ; digit → ASCII
7          mov   ah, 02h
8          int   21h          ; print DL
9          test  cl, 1        ; lowest bit
10         jnz   odd          ; 1 means odd
11         lea   dx, even_m
12         jmp   print
13 odd:    lea   dx, odd_m
14 print:  mov   ah, 09h
15         int   21h          ; print string
16         inc   cx           ; i++
17         jmp   again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

DL = 1 + 30h = 31h: the ASCII code of the digit 1. ADD updates the flags: CF=0.

Step 3 / 19

VARIABLES

CX = **0001h** DL = **31h ('1')** ZF = **0** CF = **0**

OUTPUT

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1          ; i = 1
3  again:  cmp  cx, 3          ; compare i with n
4          ja   done           ; done if i > n
5          mov  dl, cl
6          add  dl, '0'        ; digit → ASCII
▶          mov  ah, 02h
▶          int  21h            ; print DL
9          test cl, 1          ; lowest bit
10         jnz  odd            ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h            ; print string
16         inc  cx             ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

DOS service 02h prints the character in DL.

Step 4 / 19

VARIABLES

CX = **0001h** | DL = **31h ('1')** | AH = **02h** | ZF = **0** | CF = **0**

OUTPUT

```
1
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
▶          test cl, 1        ; lowest bit
▶          jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

`test cl, 1` : 1 AND 1 = 1 ≠ 0 → ZF=0 → `jnz odd` **jumps**. (The lowest bit instead of i % 2.)

Step 5 / 19

VARIABLES

CX = **0001h** DL = **31h ('1')** AH = **02h** ZF = **0** CF = **0**

OUTPUT

```
1
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
▶ odd:     lea  dx, odd_m
▶ print:   mov  ah, 09h
▶          int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

DX = the address of the odd_m string; service 09h prints up to the '$'. 0Dh, 0Ah are carriage return + line feed.

Step 6 / 19

VARIABLES

CX = **0001h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
▶          inc  cx           ; i++
▶          jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

`inc cx` → 2 (INC does not change CF); back to the top of the loop.

Step 7 / 19

VARIABLES

CX = **0002h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1          ; i = 1
▶ again:   cmp  cx, 3          ; compare i with n
▶          ja   done           ; done if i > n
5          mov  dl, cl
6          add  dl, '0'        ; digit → ASCII
7          mov  ah, 02h
8          int  21h            ; print DL
9          test cl, 1          ; lowest bit
10         jnz  odd            ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h            ; print string
16         inc  cx             ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

2 − 3: CF=1 → continue.

Step 8 / 19

VARIABLES

CX = **0002h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **1**

OUTPUT

```
1 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1          ; i = 1
3  again:  cmp  cx, 3          ; compare i with n
4          ja   done           ; done if i > n
▶          mov  dl, cl
▶          add  dl, '0'        ; digit → ASCII
▶          mov  ah, 02h
▶          int  21h            ; print DL
9          test cl, 1          ; lowest bit
10         jnz  odd            ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h            ; print string
16         inc  cx             ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

DL = 32h ('2') is printed.

Step 9 / 19

VARIABLES

CX = 0002h | DL = 32h ('2') | AH = 02h | ZF = 0 | CF = 0

OUTPUT

```
1 is odd.
2
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov   cx, 1        ; i = 1
3  again:  cmp   cx, 3        ; compare i with n
4          ja    done         ; done if i > n
5          mov   dl, cl
6          add   dl, '0'      ; digit → ASCII
7          mov   ah, 02h
8          int   21h          ; print DL
▶          test  cl, 1        ; lowest bit
▶          jnz   odd          ; 1 means odd
11         lea   dx, even_m
12         jmp   print
13 odd:    lea   dx, odd_m
14 print:  mov   ah, 09h
15         int   21h          ; print string
16         inc   cx           ; i++
17         jmp   again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

2 AND 1 = 0 → ZF=1 → `jnz` **does not jump**: the number is even.

Step 10 / 19

VARIABLES

CX = **0002h** | DL = **32h ('2')** | AH = **02h** | ZF = **1** | CF = **0**

OUTPUT

```
1 is odd.
2
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
▶          lea  dx, even_m
▶          jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

DX = even_m; `jmp print` goes to the shared print code.

Step 11 / 19

VARIABLES

CX = **0002h**

DX = **offset even_m**

AH = **02h**

ZF = **1**

CF = **0**

OUTPUT

```
1 is odd.
2
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
▶ print:   mov  ah, 09h
▶          int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

The “even” message is printed.

Step 12 / 19

VARIABLES

CX = **0002h**

DX = **offset even_m**

AH = **09h**

ZF = **1**

CF = **0**

OUTPUT

```
1 is odd.
2 is even.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
▶          inc  cx           ; i++
▶          jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

CX = 3.

Step 13 / 19

VARIABLES

CX = **0003h**

DX = **offset even_m** AH = **09h**

ZF = **0** CF = **0**

OUTPUT

```
1 is odd.
2 is even.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

3 − 3 = 0 → ZF=1, CF=0. Since ZF=1, `ja` **does not jump** (3 ≤ 3).

VARIABLES

CX = **0003h**

DX = **offset even_m** AH = **09h**

ZF = **1** CF = **0**

OUTPUT

```
1 is odd.
2 is even.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov   cx, 1          ; i = 1
3  again:  cmp   cx, 3          ; compare i with n
4          ja    done           ; done if i > n
▶          mov   dl, cl
▶          add   dl, '0'        ; digit → ASCII
▶          mov   ah, 02h
▶          int   21h            ; print DL
▶          test  cl, 1          ; lowest bit
▶          jnz   odd            ; 1 means odd
11         lea   dx, even_m
12         jmp   print
13 odd:    lea   dx, odd_m
14 print:  mov   ah, 09h
15         int   21h            ; print string
16         inc   cx             ; i++
17         jmp   again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

'3' is printed; 3 AND 1 = 1 → ZF=0 → odd.

Step 15 / 19

VARIABLES

CX = 0003h | DL = 33h ('3') | AH = 02h | ZF = 0 | CF = 0

OUTPUT

```
1 is odd.
2 is even.
3
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1          ; i = 1
3  again:  cmp  cx, 3          ; compare i with n
4          ja   done           ; done if i > n
5          mov  dl, cl
6          add  dl, '0'        ; digit → ASCII
7          mov  ah, 02h
8          int  21h            ; print DL
9          test cl, 1          ; lowest bit
10         jnz  odd            ; 1 means odd
11         lea  dx, even_m
12         jmp  print
▶ odd:     lea  dx, odd_m
▶ print:   mov  ah, 09h
▶          int  21h            ; print string
16         inc  cx             ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

The “odd” message is printed.

Step 16 / 19

VARIABLES

CX = **0003h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
2 is even.
3 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1          ; i = 1
3  again:  cmp  cx, 3          ; compare i with n
4          ja   done           ; done if i > n
5          mov  dl, cl
6          add  dl, '0'        ; digit → ASCII
7          mov  ah, 02h
8          int  21h            ; print DL
9          test cl, 1          ; lowest bit
10         jnz  odd            ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h            ; print string
▶          inc  cx             ; i++
▶          jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

CX = 4.

Step 17 / 19

VARIABLES

CX = **0004h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
2 is even.
3 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov  cx, 1        ; i = 1
3  again:  cmp  cx, 3        ; compare i with n
4          ja   done         ; done if i > n
5          mov  dl, cl
6          add  dl, '0'      ; digit → ASCII
7          mov  ah, 02h
8          int  21h          ; print DL
9          test cl, 1        ; lowest bit
10         jnz  odd          ; 1 means odd
11         lea  dx, even_m
12         jmp  print
13 odd:    lea  dx, odd_m
14 print:  mov  ah, 09h
15         int  21h          ; print string
16         inc  cx           ; i++
17         jmp  again
18 done:   ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

4 − 3 = 1 → ZF=0, CF=0 → `ja done` **jumps** (4 > 3).

VARIABLES

CX = **0004h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
2 is even.
3 is odd.
```

The Same Task in 8086 Assembly (n = 3)

```
1  org 100h
2          mov   cx, 1        ; i = 1
3  again:  cmp   cx, 3        ; compare i with n
4          ja    done         ; done if i > n
5          mov   dl, cl
6          add   dl, '0'      ; digit → ASCII
7          mov   ah, 02h
8          int   21h          ; print DL
9          test  cl, 1        ; lowest bit
10         jnz   odd          ; 1 means odd
11         lea   dx, even_m
12         jmp   print
13 odd:    lea   dx, odd_m
14 print:  mov   ah, 09h
15         int   21h          ; print string
16         inc   cx           ; i++
17         jmp   again
▶ done:    ret
19 odd_m   db ' is odd.',0Dh,0Ah,'$'
20 even_m  db ' is even.',0Dh,0Ah,'$'
```

`ret` : the COM program returns to DOS. C's `for` + `if` became pairs of **compare + conditional jump**.

Step 19 / 19

VARIABLES

CX = **0004h** | DX = **offset odd_m** | AH = **09h** | ZF = **0** | CF = **0**

OUTPUT

```
1 is odd.
2 is even.
3 is odd.
```

Summary

- Software runs the hardware; the operating system is the core software, and programming languages are used to build software.
- **Machine language** is the only language the processor understands: binary instructions made of opcode and operand fields; fully hardware-dependent.
- **Assembly language** writes machine instructions as mnemonics; the assembler and linker turn it into an executable. The code depends on both the processor and the operating system (int 0x80, int 21h, DLL calls, svc, sc).
- **Compiled languages** (C, C++, Go, Rust, Fortran, Ada) are translated to machine code ahead of time: fast, but platform-specific.
- **Interpreted languages** (Python, JavaScript, Ruby, PHP, Perl, MATLAB) are processed at run time: fast development, portable source, slower execution. Java/C# sit in between with intermediate code + JIT.
- A high-level `for` / `if` is implemented in assembly as **compare + conditional jump**.

SUMMARY

Exercises

1. Encode `SUB R3, R0` and `OR R0, R2` in binary and hex.
2. Decode the instruction `0010 0111`. If R1 = 9 and R3 = 4, what is the result?
3. How many opcodes fit in this design? How must the format change for 8 registers?
4. What happens if the DOS program's string lacks the final `'$'`?
5. In the Linux example, what would be printed if `len` were hard-coded as 15?
6. Windows example: why are parameters pushed in reverse? What is ESP right after the `call`?
7. Why can't Java be labeled simply “compiled” or “interpreted”?

▶ **Hints / answers**

References

- Tanenbaum and Bos, *Modern Operating Systems*, 3rd ed., Pearson.
- Brey, *The Intel Microprocessors*, 8th ed., Pearson.
- Irvine, *Assembly Language for x86 Processors*, 7th ed., Pearson.
- NASM documentation (nasm.us) and the GNU Binutils *as* manual.
- Ralf Brown's Interrupt List (DOS INT 21h services); Microsoft Win32 API documentation (`MessageBoxA`, `ExitProcess`).
- emu8086 documentation and 8086 instruction set reference.

The End

Chapter 3.1: Programming Languages

Previous: Chapter 2.2: 8086 Registers · Next: Chapter 3.2: Assembly Language