

INTRODUCTION TO ASSEMBLY LANGUAGE

CS 348

**Implementation of Programming Languages Lab
Computer Science and Engineering Department
Indian Institute of Technology
Guwahati**

The background features three decorative, multi-colored curved lines in the corners. One arc is in the top-left, another in the top-right, and a third in the bottom-left. These arcs are composed of several overlapping, semi-transparent bands of color, including shades of light blue, teal, and light green, creating a soft, layered effect.

Basics

Computer Architecture

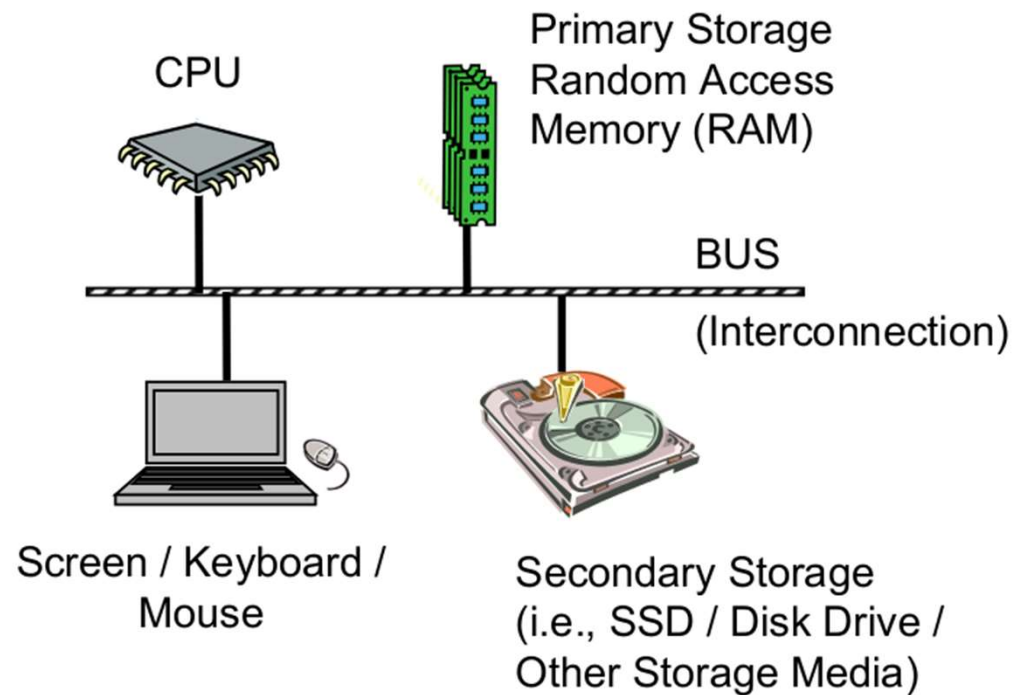
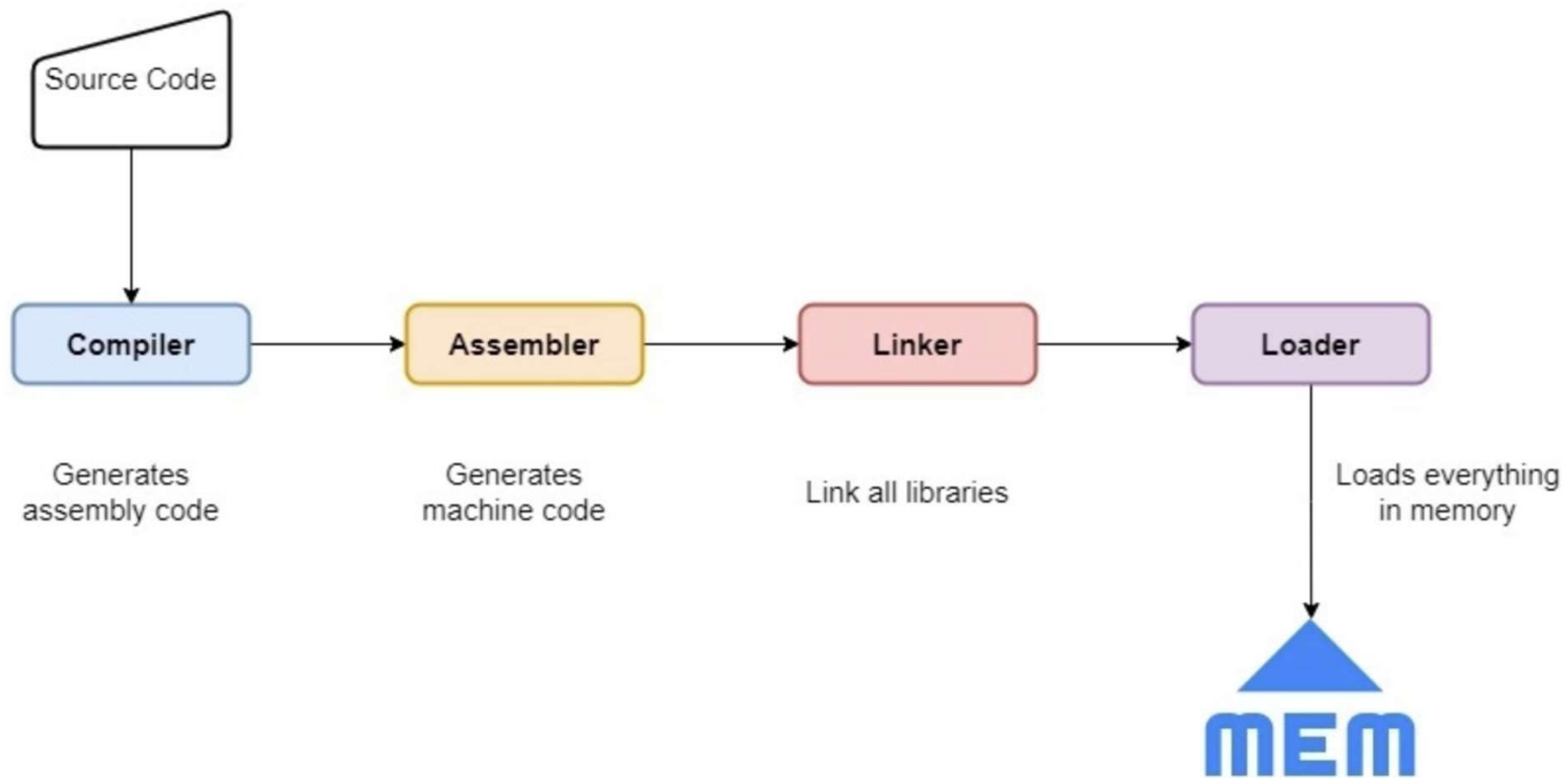


Illustration 1: Computer Architecture

Executable Generation



Assemble code steps

`$nasm -f elf32 ex1.asm -o ex1.o`

- Elf32: 32 bit executable linking format.
- -f: flag to create elf

`$ld -m elf_i386 ex1.o -o ex1`

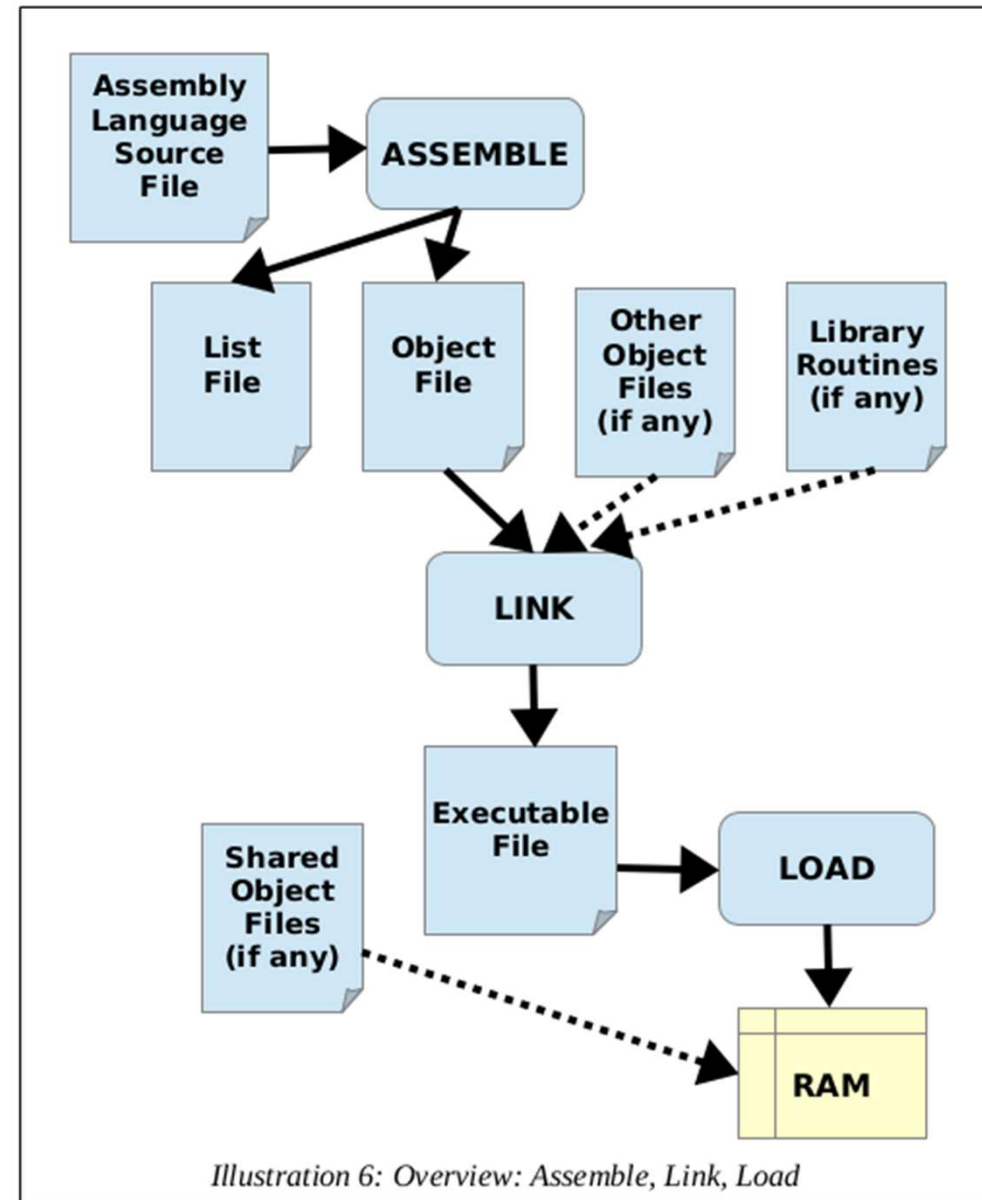
- -m: flag to create linker executable from the object file

`$/ex1`

- Execute the final code.

`$echo $?`

- Inspects exit status



High Level language

- Characteristics:
 - Portable: Code from one CPU can be used in other CPUs.
 - Complex: One statement can perform multiple tasks.
 - Human Readable: Structured and can be easily understood.

Program to find if a number is even or odd:

Let, n is the number to be checked.

```
If (n%2 == 0)
{
    Print("Number is even");
}
Else
{
    Print("Number is odd");
}
```

Assembly Language

- Characteristics:
 - Not Portable: Each assembly language instruction maps to one machine language instruction.
 - Simple: Each instruction does a simple task.
 - Human Readable: Structured but not as easily understood as HLL.

```
        mov     w1, 0
loop:
        cmp     w0, 1
        ble     endloop
        add     w0, w0, #1
        ands    wzr, w0, #1
        beq     else
        add     w2, w0, w0
        add     w0, w0, w2
        add     w0, w0, 1
        b       endif
else:
        asr     w0, w0, 1
endif:
        b       loop
endloop:
```

Machine Language

- Characteristics:
 - Not Portable: Processor specific code.
 - Simple: Each instruction does a simple task.
 - Not Human Readable: Not Structured and requires lot of effort.

0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
9222	9120	1121	A120	1121	A121	7211	0000
0000	0001	0002	0003	0004	0005	0006	0007
0008	0009	000A	000B	000C	000D	000E	000F
0000	0000	0000	FE10	FACE	CAFE	ACED	CEDE
1234	5678	9ABC	DEF0	0000	0000	F00D	0000
0000	0000	EEEE	1111	EEEE	1111	0000	0000
B1B2	F1F5	0000	0000	0000	0000	0000	0000

Why learn assembly language?

- It is used for direct hardware manipulation.
- It helps us access specialized processor instructions.
- It helps us address critical performance issues.
- Many real time systems, low level embedded systems and device drivers use Assembly Language.
- We learn how the OS interacts with input/output, system calls such as interrupt.
- It helps us improves algorithm development skills.
- It helps us improve the understanding of functions/procedures.
- It helps us gain an understanding of I/O buffering.
- It helps us adapt to changes in the computing world.

The background features decorative curved lines in the corners. In the top-left and bottom-left corners, there are light blue and green curved lines. In the top-right corner, there is a thick, multi-layered curved line in shades of blue and green. The text "X86 Architecture" is centered in the middle of the page.

X86 Architecture

Data Storage Sizes

- The x86 architecture supports a specific set of data storage size elements.
- The sizes are based on powers of 2.
- These storage sizes have a direct correlation to variable declarations in HLL (e.g., C, C++, Java, etc).

Storage	Size (bits)	Size (bytes)
Byte	8-bits	1 byte
Word	16-bits	2 bytes
Double-word	32-bits	4 bytes
Quadword	64-bits	8 bytes
Double quadword	128-bits	16 bytes

CPU Registers

- A CPU register is a temporary storage built into the CPU itself.
- Computations are typically performed on them.
- The types of such registers are:
 - General Purpose Registers (GPRs)
 - Stack Pointer Register (RSP)
 - Base Pointer Register (RBP)
 - Instruction Pointer Register (RIP)
 - Flag Registers (rFlags)
 - XMM Registers

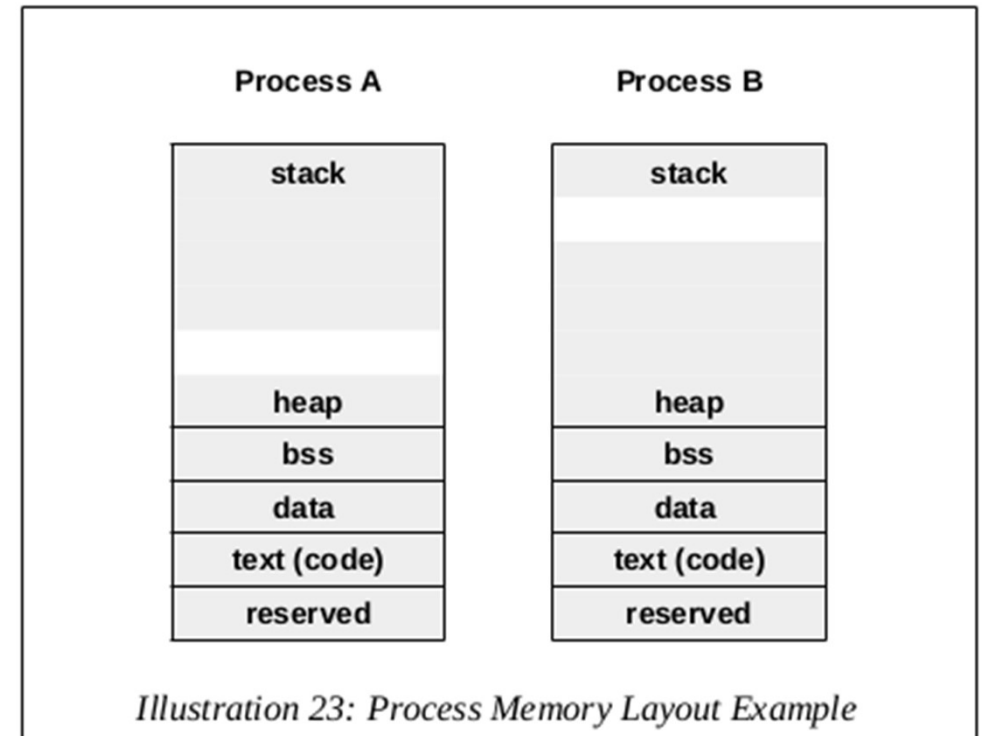
64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Name	Symbol	Bit	Use
Carry	CF	0	Used to indicate if the previous operation resulted in a carry.
Parity	PF	2	Used to indicate if the last byte has an even number of 1's (i.e., even parity).
Adjust	AF	4	Used to support Binary Coded Decimal operations.
Zero	ZF	6	Used to indicate if the previous operation resulted in a zero result.
Sign	SF	7	Used to indicate if the result of the previous operation resulted in a 1 in the most significant bit (indicating negative in the context of signed data).
Direction	DF	10	Used to specify the direction (increment or decrement) for some string operations.
Overflow	OF	11	Used to indicate if the previous operation resulted in an overflow.

Flags

Process Stack

- The `rsp` is used to point to the current top of stack in memory.
- The stack is implemented growing down in the memory.
- The heap grows upwards, while the stack grows downward, to ensure most effective use of memory.
- If the stack and heap meet, the program will crash.



The background features decorative curved lines in the corners. In the top-left and bottom-left corners, there are light blue and green curved lines. In the top-right corner, there are light blue and green curved lines. The text "Program Format" is centered in the middle of the page.

Program Format

Program Format

- A properly formatted assembly source file consists of several main parts:
 - Data section: Initialized data is declared and defined.
 - BSS section: Where uninitialized data is declared.
 - Text section: Where code is placed.
- Comments can be placed after semicolon (;). Any instruction after ; are ignored by the compiler for that line.
- Numerical values may be specified in decimal, hex or octal.
 - '0x' preceeds hex values
 - 'q' preceeds octal values
 - Decimal doesnt require any notation.

Data Section

- The initialized data must be declared in the "section .data" section.
- Variable name must start with a letter, followed by letters or numbers, and some special characters.
- The general format is:
<variable name> <dataType> <initialValue>

Supported Data Types

Declaration	
db	8-bit variable(s)
dw	16-bit variable(s)
dd	32-bit variable(s)
dq	64-bit variable(s)
ddq	128-bit variable(s) → integer
dt	128-bit variable(s) → float

Some simple examples include:

bVar	db	10	; byte variable
cVar	db	"H"	; single character
strng	db	"Hello World"	; string
wVar	dw	5000	; 16-bit variable
dVar	dd	50000	; 32-bit variable
arr	dd	100, 200, 300	; 3 element array
flt1	dd	3.14159	; 32-bit float
qVar	dq	1000000000	; 64-bit variable

BSS Section

- The uninitialized data is declared in the "section .bss" section.
- Variable name must start with a letter, followed by letters or numbers, and some special characters.
- The general format is:
<variable name> <resType> <count>

The supported data types are as follows:

Declaration	
resb	8-bit variable(s)
resw	16-bit variable(s)
resd	32-bit variable(s)
resq	64-bit variable(s)
resdq	128-bit variable(s)

Some simple examples include:

```
bArr    resb    10        ; 10 element byte array
wArr    resw    50        ; 50 element word array
dArr    resd    100       ; 100 element double array
qArr    resq    200       ; 200 element quad array
```

The allocated array is not initialized to any specific value.

Text Section

- The code is placed in the "section .text" section.
- The instructions are specified one per line.
- Each instruction must be valid with appropriate operands.
- It includes some headers or labels that define initial program entry.
- Assuming a basic program using the standard system linked, the following declarations must be included.

```
global _start  
_start:
```

- No special label is required to terminate the program.
- The OS should be informed so that resources can be recovered and re-utilized.

Text Section key terms

- Global- It is the keyword used to make identifier accessible to linker.
- `_start`- The processor starts compiling from this. It is equivalent to `main()` in C.
- Label: -Any word with a colon in the text section is a label. It can be accessed via jumps.
- `Int 0x__` – It is an interrupt specified by the hex code following it.
- `0x80` – hex code used to call system calls based on the value in the 'eax' register.

Program Example

- A program to calculate 2^x

```
1 section .text
2 _start:
3 mov ebx,1      ;starts ebx to 1
4 mov ecx,4      ;no of iterations
5
6 label:
7 add ebx,ebx    ;ebx = ebx + ebx
8 dec ecx       ;ecx = ecx - 1
9 cmp ecx,0      ;compare ecx with 0
10 jg label      ;jump to label if ecx is greater than 0
11 mov eax,1     ;sys_exit system call
12 int 0x80      ;interrupt system call
```

The output of the program is 16.