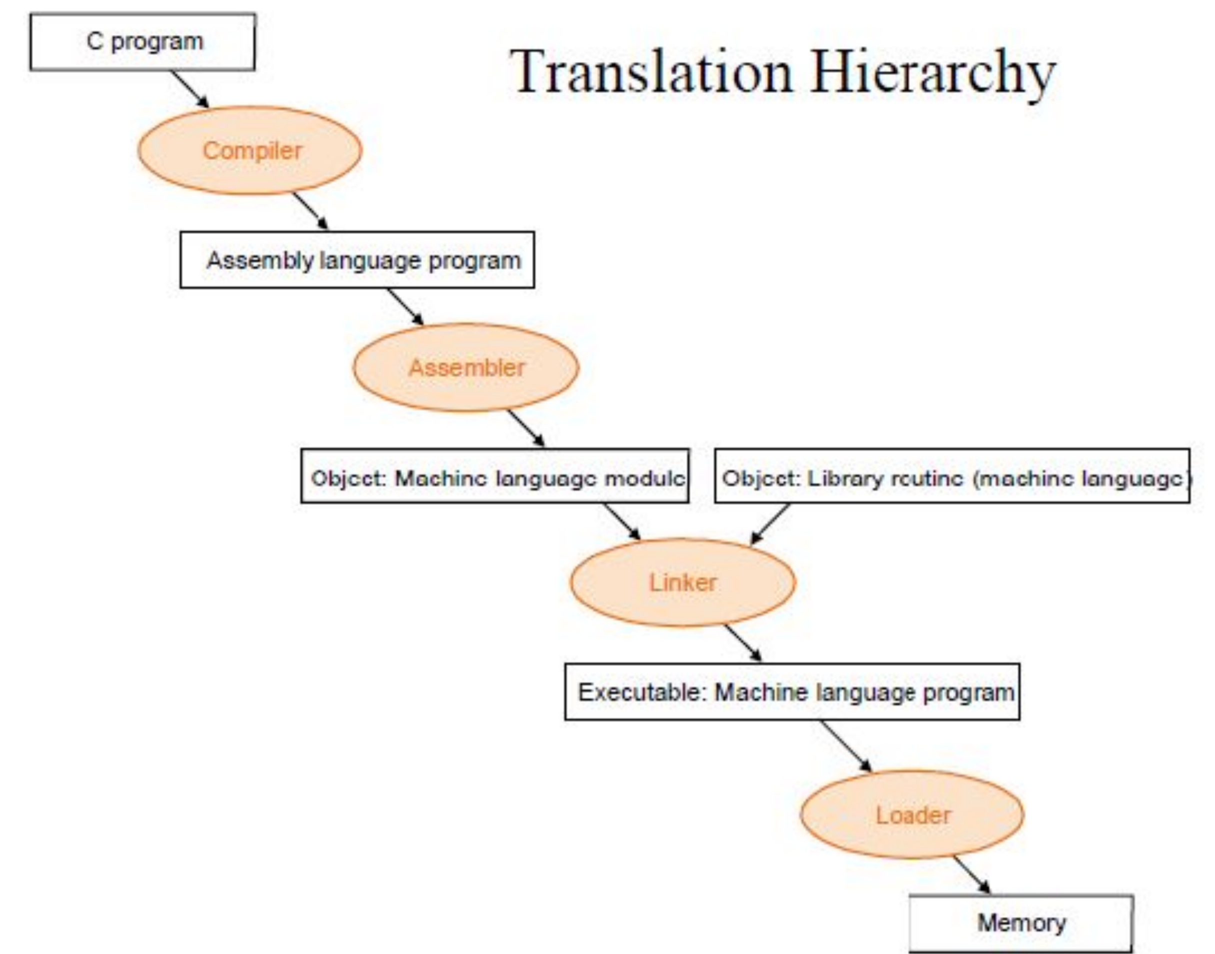
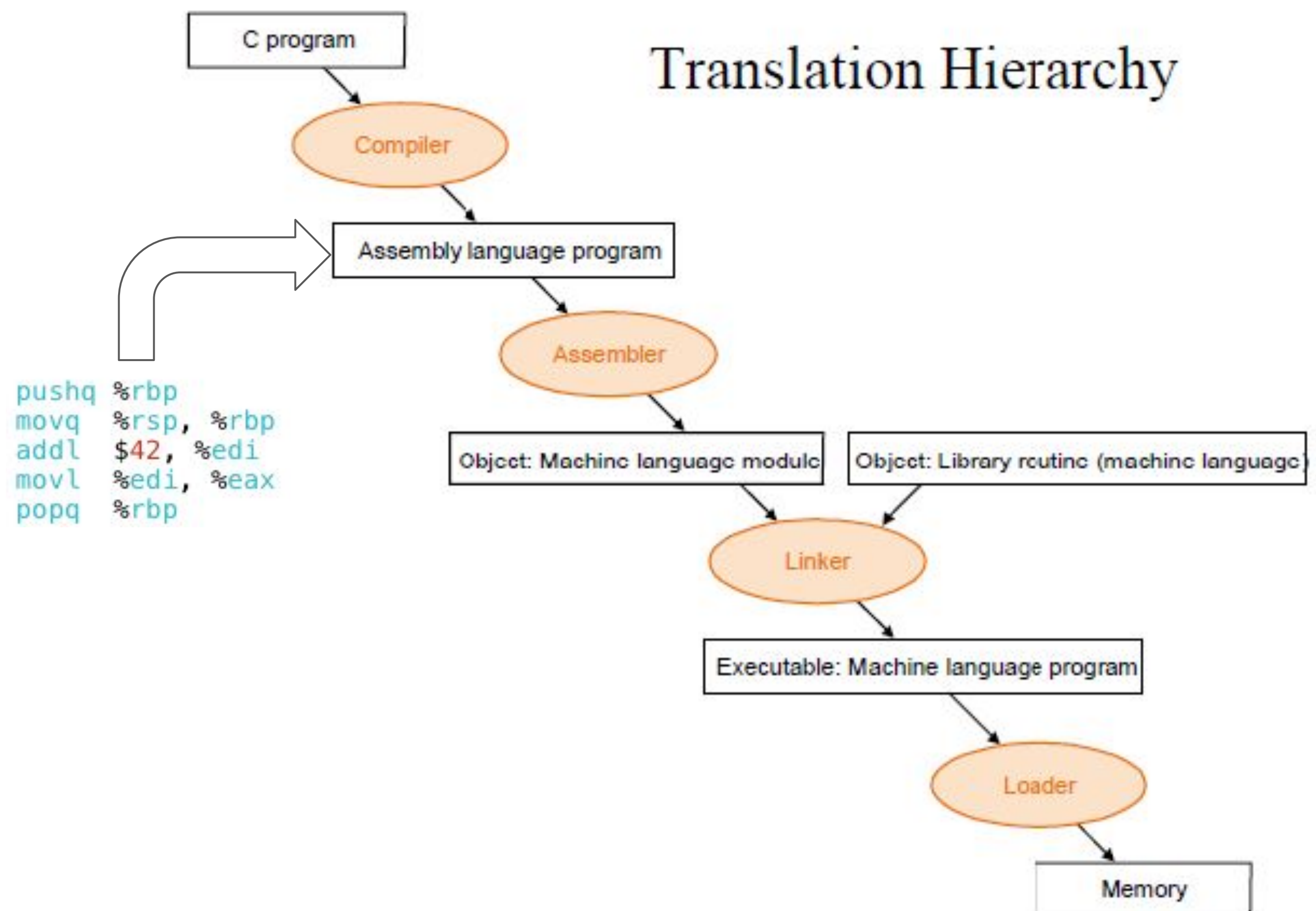
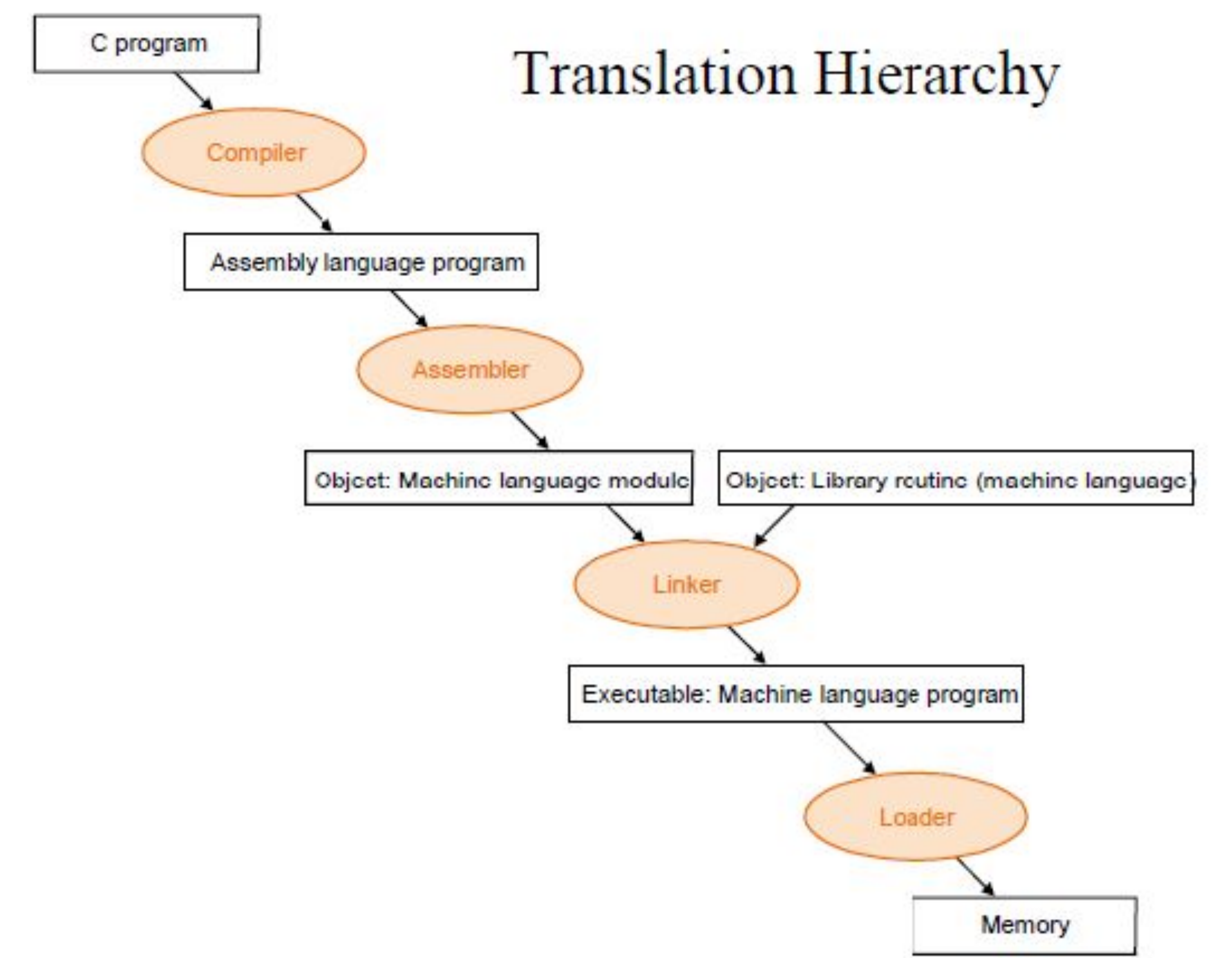
A Beginners guide to Assembly

By Rishiraj , Varun and Shreyas

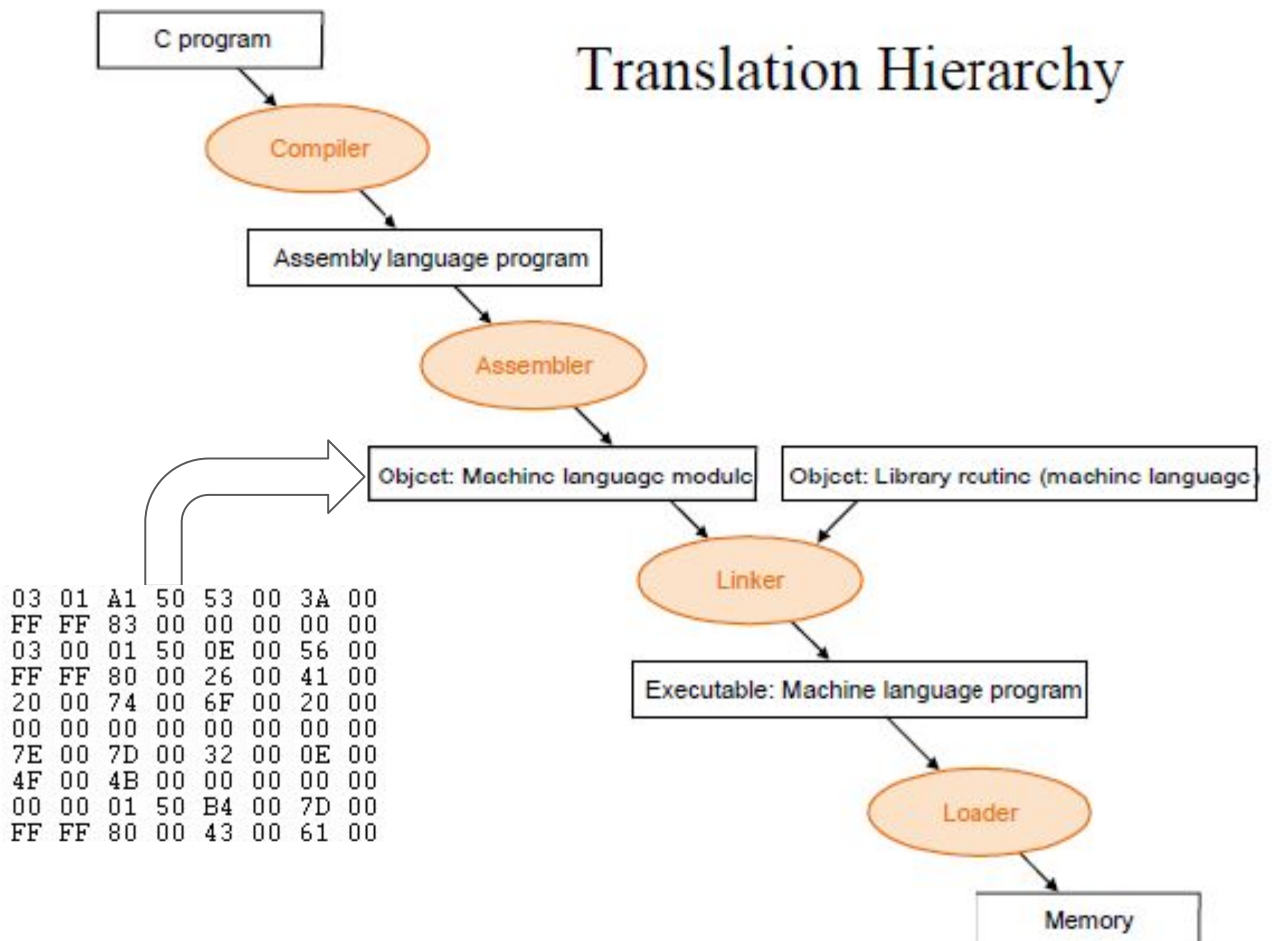

Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

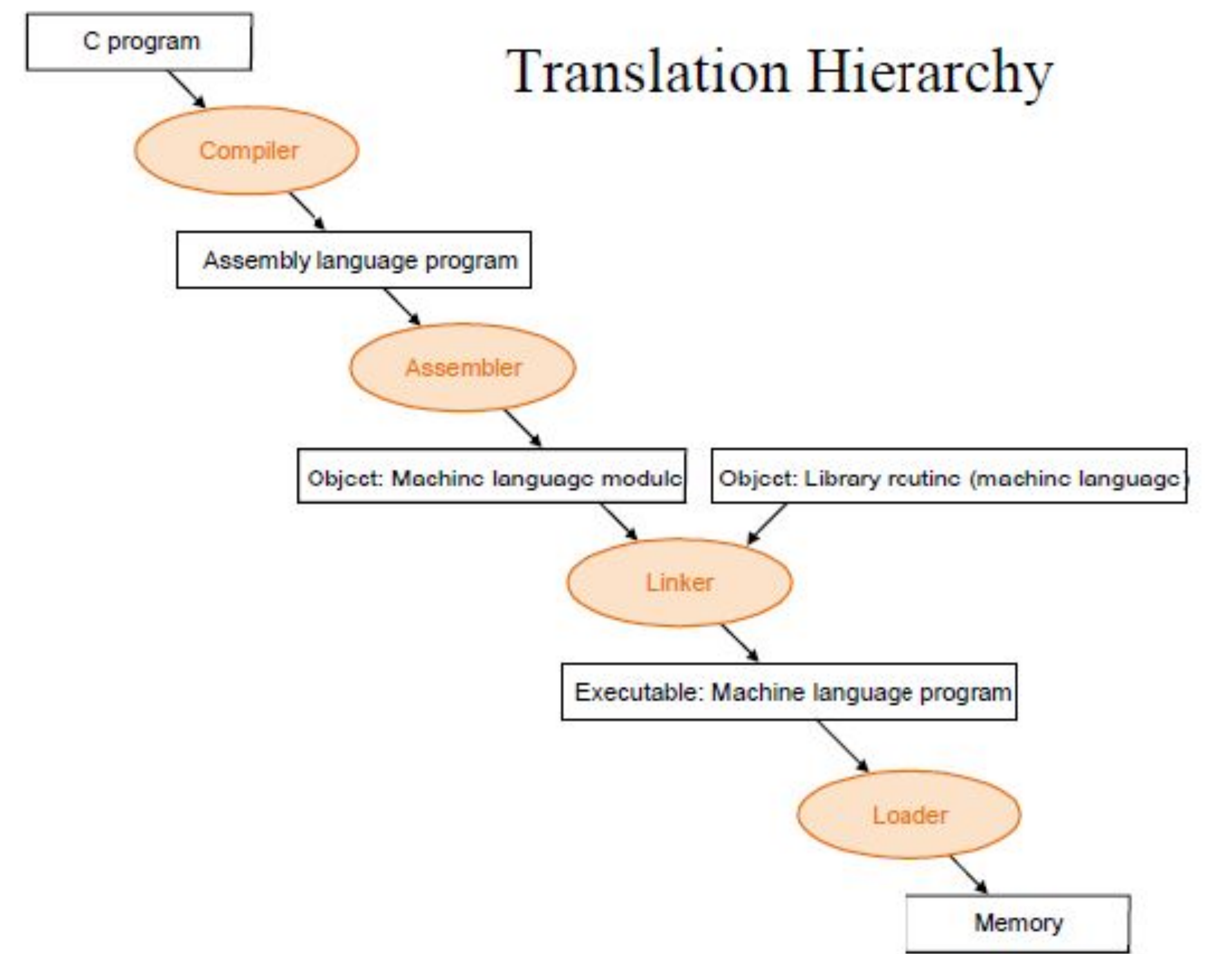
Translation Hierarchy



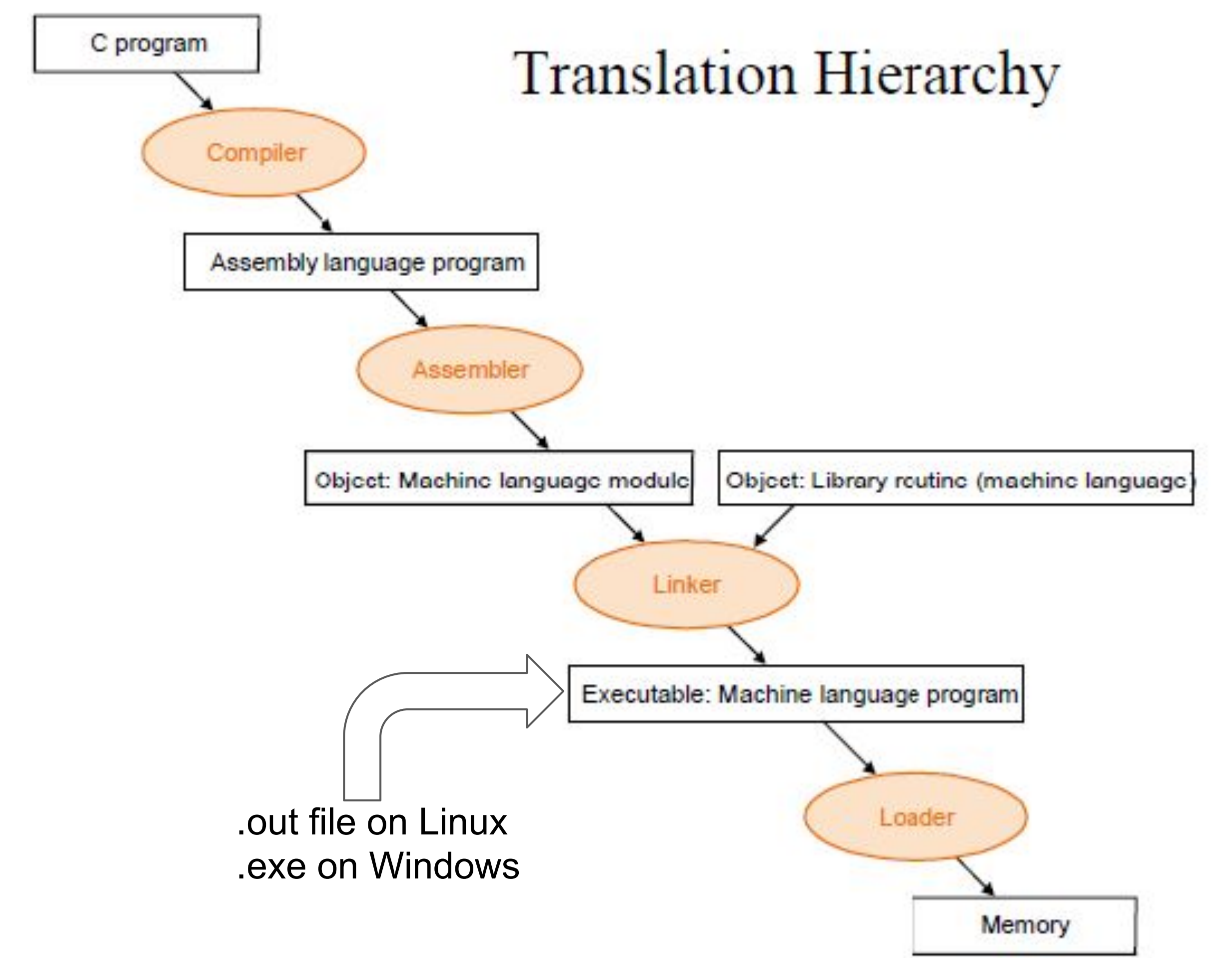

Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Translation Hierarchy

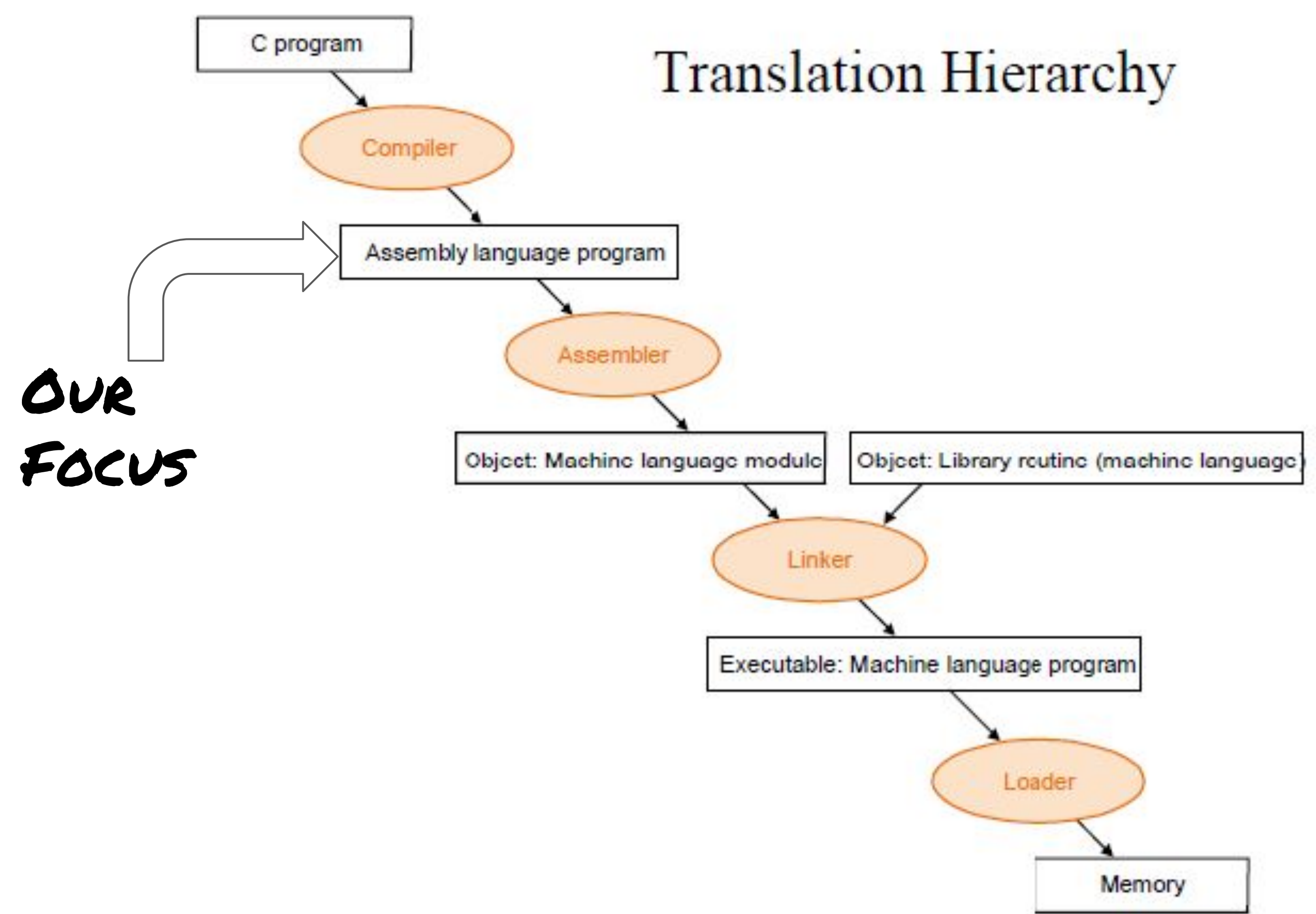




Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory



Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
.out file on Linux
.exe on Windows



Translation Hierarchy

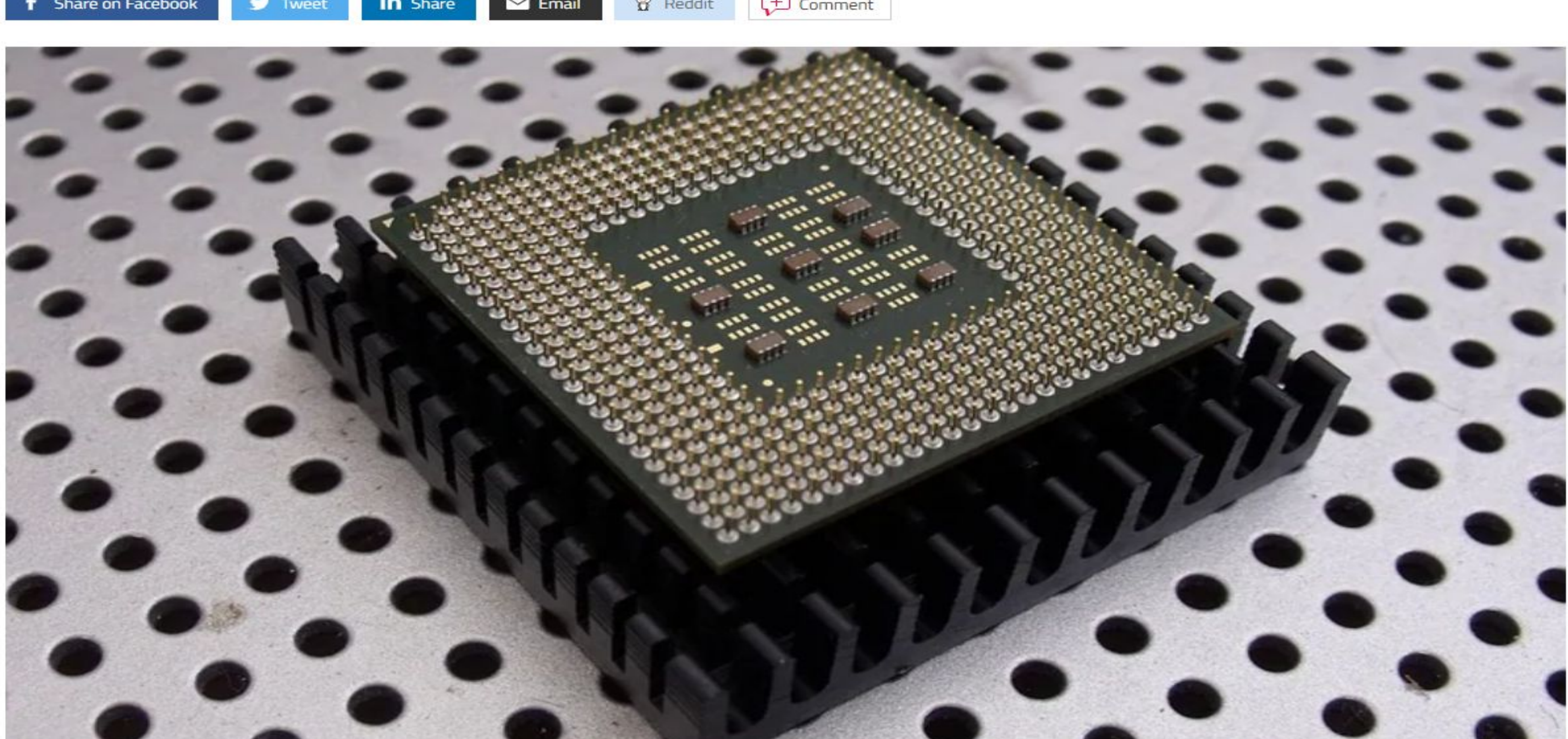
Prominent ISAs

ARM

IIT-Madras Develops 'India's First Microprocessor', Shakti

By Indo-Asian News Service | Updated: 2 November 2018 16:03 IST

Share on Facebook | Tweet | Share | Email | Reddit | Comment



An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

→

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

Some Basics

- % - indicates register names. Example : %rbp

- $ - indicates constants Example : $100

- Accessing register values:
    - %rbp   : Access value stored in register rbp

    - (%rbp)   : Treat value stored in register rbp as a pointer. Access the value stored at address pointed by the pointer. Basically *rbp

    - 4(%rbp) : Access value stored at address which is 4 bytes after the address stored in rbp. Basically *(rbp + 4)

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

For each function call, new space is created on the stack to store local variables and other data. This is known as a stack frame. To accomplish this, you will need to write some code at the beginning and end of each function to create and destroy the stack frame

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

rbp is the frame pointer. In our code, it gets a snapshot of the stack pointer (**rsp**) so that when rsp is changed, local variables and function parameters are still accessible from a constant offset from **rbp**.

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

move immediate value 3000 to (%rbp-8)

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

add immediate value 3 to (%rbp-8)

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

Move immediate value 100 to (%rbp-4)

An intriguing Example!

```c
#include<stdio.h>

int main(){
	int x = 3000,y;
	x = x + 3;
	y = 100;
	return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
  ⇒     movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

Move (%rbp-4) to auxiliary register

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
  ===>  popq    %rbp
        ret
```

Pop the base pointer to restore state

An intriguing Example!

```c
#include<stdio.h>

int main(){
        int x = 3000,y;
        x = x + 3;
        y = 100;
        return y;
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $3000, -8(%rbp)
        addl    $3, -8(%rbp)
        movl    $100, -4(%rbp)
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

The calling convention dictates that a function’s return value is stored in %eax, so the above instruction sets us up to return y at the end of our function.

Operation Suffixes

- b = byte (8 bit)
- s = single (32-bit floating point)
- w = word (16 bit)
- l = long (32 bit integer or 64-bit floating point)
- q = quad (64 bit)
- t = ten bytes (80-bit floating point)

How to get assembly code?

Two ways:

- While Compiling
    - Use **-S flag with gcc**. WIll create a .s file containing assembly

- Using Binary
    - Use **objdump**. Will show the assembly in terminal.

Understanding the output

- The output will have assembly, but there is more information!
- You will see lots of Directives like:
  - .file
  - .text
  - .global name

Understanding the output

- The output will have assembly, but there is more information also!.
- You will see lots of Directives like:
  - .file
  - .text
  - .global name

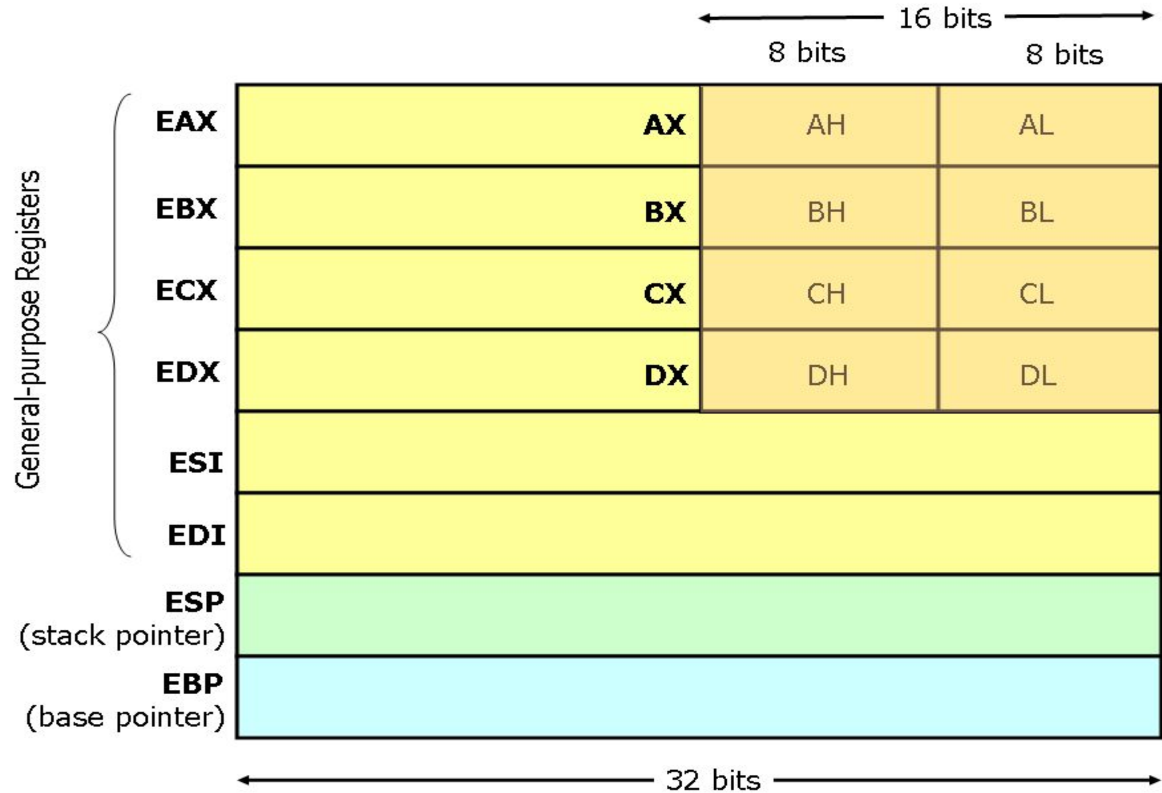
To disable these, use the gcc option

137

```
-fno-asynchronous-unwind-tables
```

Note, I know this is a really old thread, but this is the top result on google for cfi_startproc, so many people probably come here to disable that output.

x86 Register Set



x86 Register Set : A few more

- Registers starting with “r”
  - Same as “e” registers but 64 bits wide
- EIP : The Instruction Pointer or the Program Counter

An Example with Loops!

```c
#include<stdio.h>

int main(){
        int x = 0;
        for(int i=0;i<10;i++){
                x = x + 1;
        }
        return x;
}
```

→

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        addl    $1, -8(%rbp)
        addl    $1, -4(%rbp)
.L2:
        cmpl    $9, -4(%rbp)
        jle     .L3
        movl    -8(%rbp), %eax
        popq    %rbp
        ret
```

System Calls in Assembly

```
kernel:
        int     80h   ; //Call kernel
        ret

open:
        push  dword mode
        push  dword flags
        push  dword path
        mov   eax, 5
        call  kernel
        add   esp, byte 12
        ret
```

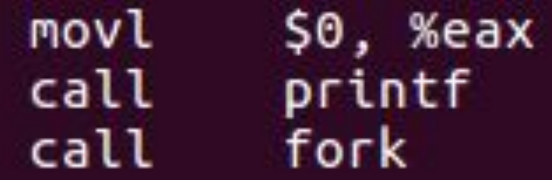
System Calls in Assembly

```
kernel:
        int     80h    ; //Call kernel
        ret

open:
        push  dword mode
        push  dword flags
        push  dword path
        mov   eax, 5          → Syscall Number
        call  kernel
        add   esp, byte 12
        ret
```

A bit different!

A simple fork program



Embedding Assembly in C

__asm__(“instruction 1”, “instruction 2”, …)

Example:

```
__asm__(
        "movl %edx, %eax\n\t"
        "addl $2, %eax\n\t"
       );
```

Embedding Assembly in C

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  int x = 5;

  printf("x = %d\n", x);

  __asm__("add $10, %0":"=m"(x));

  printf("x = %d\n", x);

  return 0;
}
```

```
x = 5
x = 15
```

Where will I use assembly?

Where will I use assembly?

- To write Compilers and Device Drivers
- To write viruses and for malware analysis
- Used while programming Real Time Embedded systems
- Implementing Locks for Concurrency.
  We will cover this in the third module of the course!

References

- Chapter 11. x86 Assembly Language Programming, FreeBSD, https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/x86.html
- Easy x86-64, http://ian.seyler.me/easy_x86-64/
- Introduction to the GNU/Linux assembler and linker for Intel Pentium processors, https://www.cs.usfca.edu/~cruse/cs210s07/lesson01.ppt
- Is there a way to insert assembly code into C?, https://stackoverflow.com/questions/61341/is-there-a-way-to-insert-assembly-code-into-c