Intro to x86 Assembly

- **ML instructions consist of 1’s and 0’s and are called ‘opcodes’**
- **Each instruction corresponds to a circuit which carries out the operation**
- **Give each opcode a name and build a syntax on top of it**
- **There, you have an Assembly Language..**

Intro to x86 Assembly

Address	Machine Language				Assembly Language
0000 0000	0000	0000	0000	0000	TOTAL .BLOCK 1
0000 0001	0000	0000	0000	0010	ABC .WORD 2
0000 0010	0000	0000	0000	0011	XYZ .WORD 3
0000 0011	0001	1101	0000	0001	LOAD REGD, ABC
0000 0100	0001	1110	0000	0010	LOAD REGE, XYZ
0000 0101	0101	1111	1101	1110	ADD REGF, REGD, REGE
0000 0110	0010	1111	0000	0000	STORE REGF, TOTAL
0000 0111	1111	0000	0000	0000	HALT

Registers

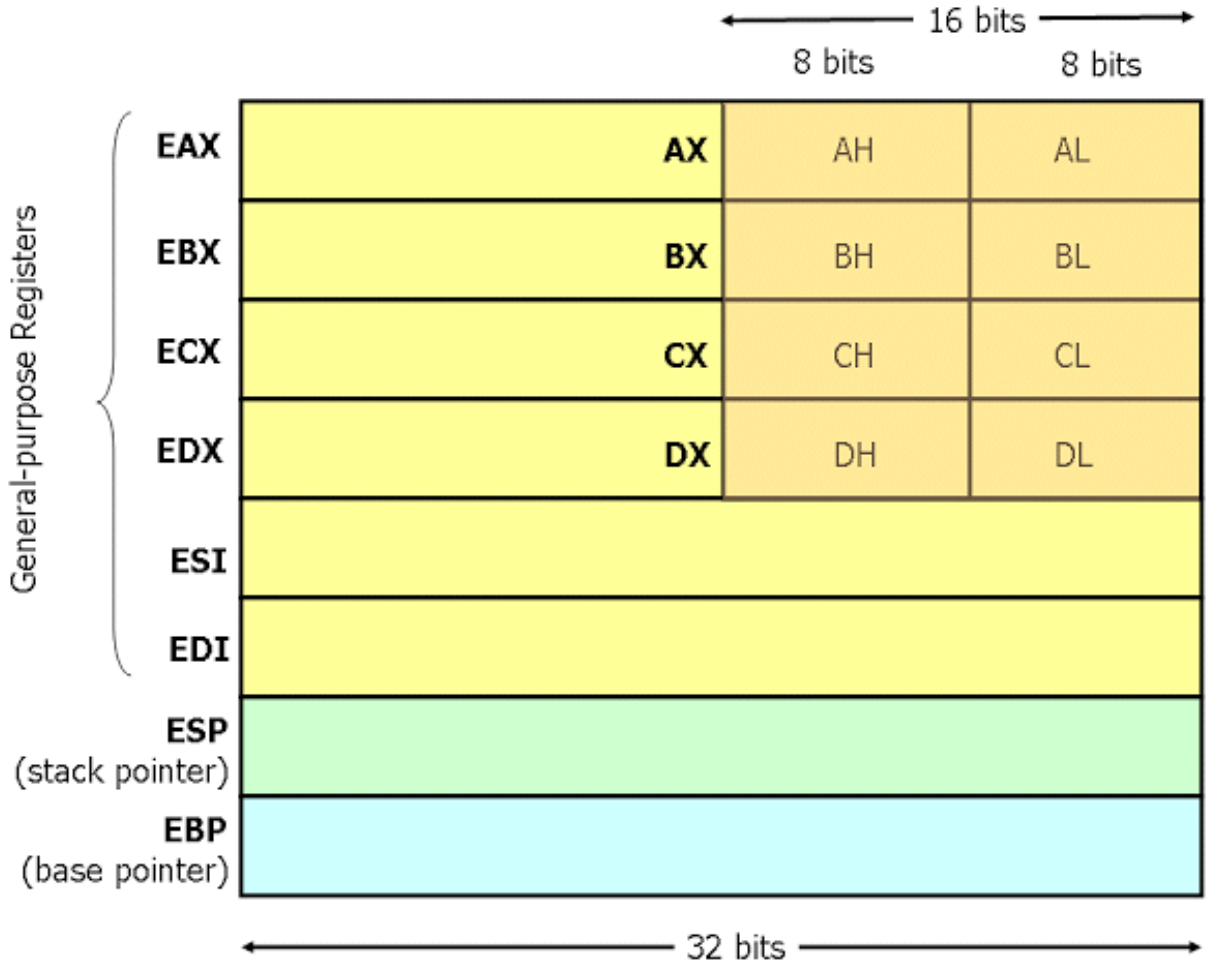
- **Small memory blocks on CPU, very fast!**
- **Size of a WORD**
- **Three types of registers;**
  - General Registers
  - FLAGS Register
  - Internal Registers (not our topic)

Registers

- **General Registers**
  - Data Registers
  - Index Registers
  - Pointer Registers

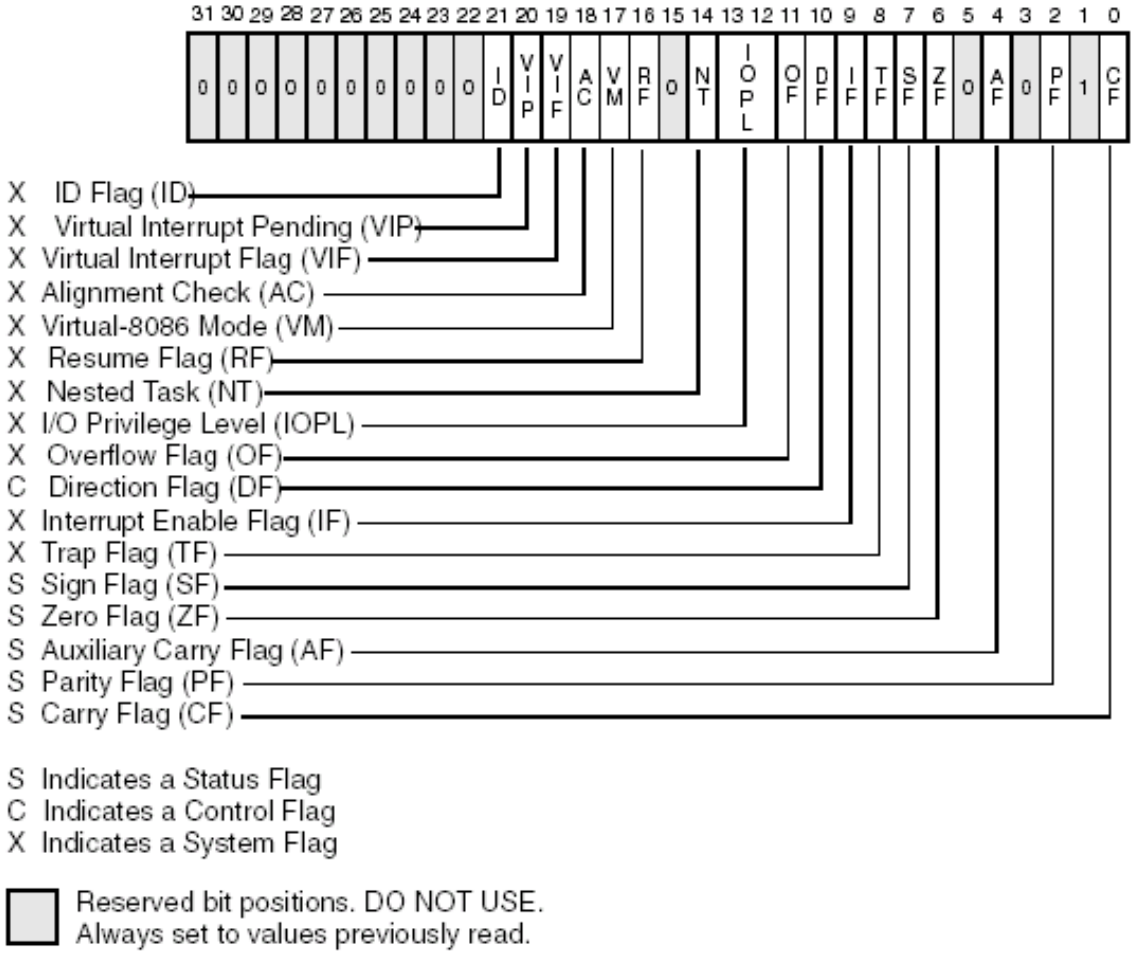
Registers

- **General Registers**



Registers

- **(E)FLAGS Register**



Arithmetic / Logic Instructions

- **inc, dec, add, sub, mul, div, imul, idiv…**
- **and, or, not, xor bla bla**
- **BORING!**
- **Syntax is available on the web.**
- **All you need to know is…**

More Interesting Instructions

- **mov R/M, o2**
- **lea R/M, o2** → **POINTERS!**
- **jmp addr**
- **test o1, o2**
- **cmp o1, o2**
- **jxx addr**

} **Decision making**

Decision Making

- 1: mov eax, 5
- 2: cmp eax, 16
- 3: jne 7
- …

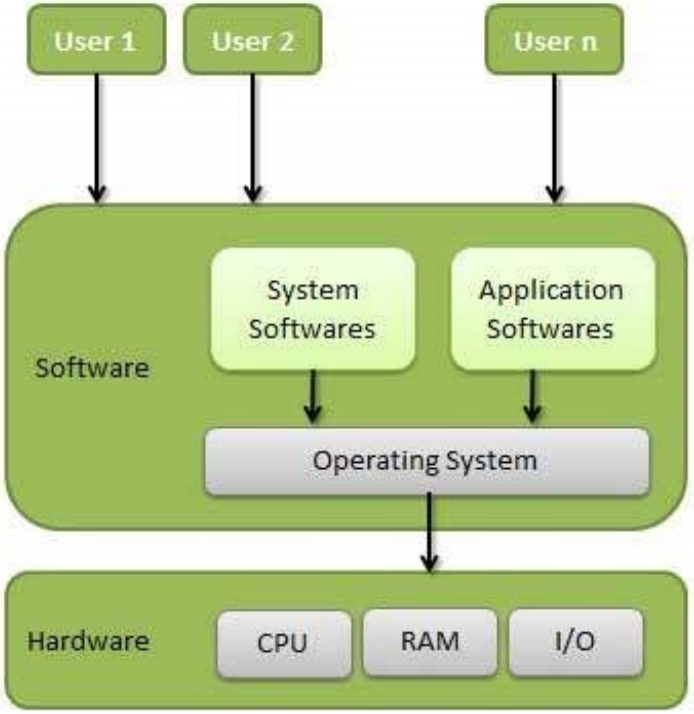
Decision Making

- 1: mov ecx, [100]
- 2: cmp ecx, 10
- 3: jg 7
- 4: inc ecx
- 5: mov [100], ecx
- 6: jmp 1
- 7: ...

Decision Making

System Calls

- **Linux at your service.**
- **Wait.. what is Linux?**
- **What is an OS?**
- **What does it do really?**



System Calls

- **Kernel’s API for userland applications**
- **Going places and doing stuff…**
- **Arguments sent over registers and calls are triggered by kernel interrupts**
- **Call ID is stored in EAX**

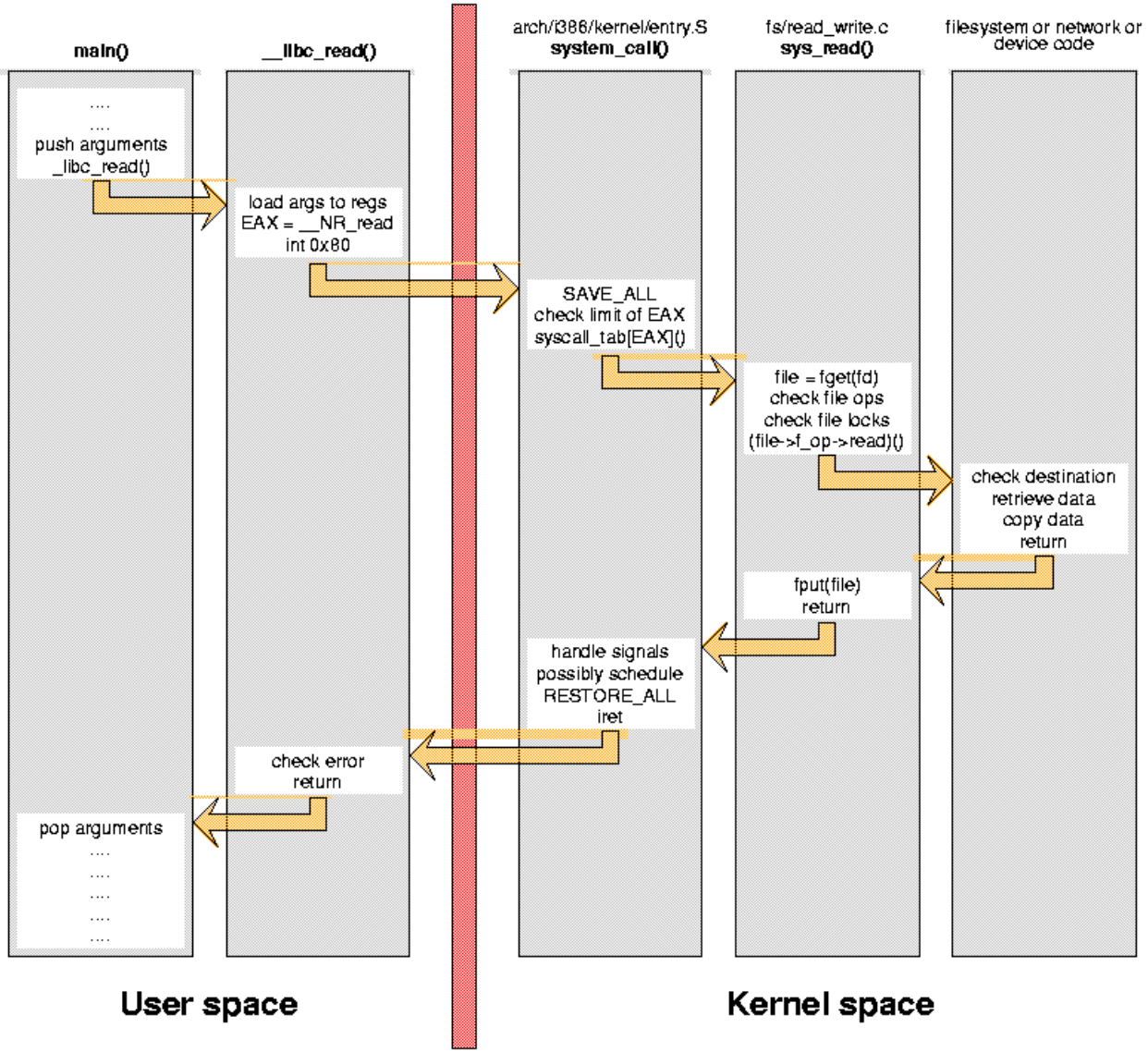
System Calls

- 1: mov ebx, 0
- 2: mov eax, 1
- 3: int 0x80

Glibc Functions

- **System call → API for kernel functions**
- **GNU Library C → Wrappers for System calls**

Glibc Functions



Baby Steps

- **Let’s write a C wrapper for printing text to console.**
- **Let’s write a simple pincode authentication program with assembly.**
- **We’ve got all we need: Decision making structures and System calls!**

Exercises

- **Write a multiplier application in Assembly**
  - Take two inputs and multiply them then writeback the result
- **Write a shell application in Assembly**
  - Hint: execve(“/bin/sh”)
- **Now make it Password Protected**

Code Patterns

- **Data types**
- **Local and global variables**
- **Statics and (un)initialized constants**
- **Arrays and pointers**
- **Conditions and Loops**
- **Functions and External Procedures**

Code Patterns

- **mul/div → Unsigned number**
- **imul/idiv → Signed number**

Code Patterns

- **Constants lay in .rodata section (RO)**
- **Global variables, if initialized are in .data (RW) if not are in .bss (RW)**

Code Patterns

- **Fickling with pointers**
  - *LEA* → Load Efficient Address
- **Array items**
  - base_addr + sizeof(data_type)*index
    - lea eax, [esp+0x4]
    - add eax, 0x5

Code Patterns

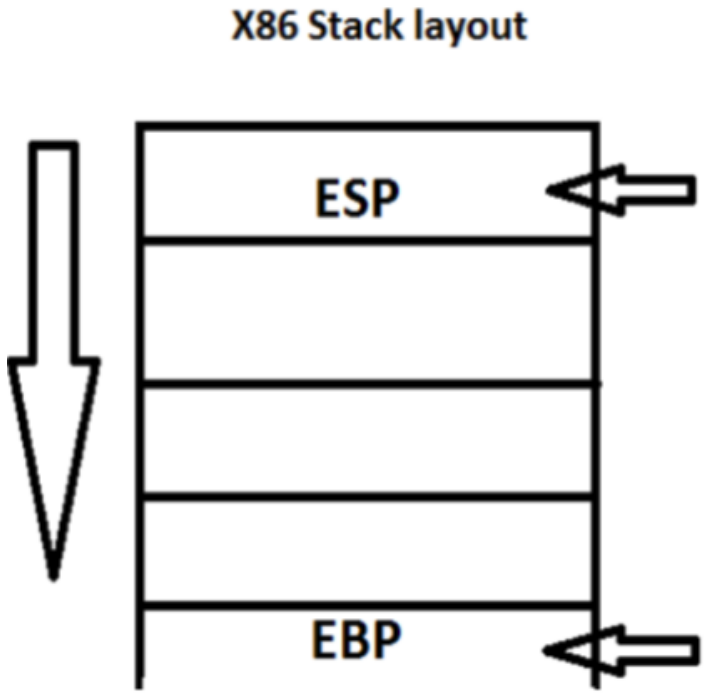
- **Get character from a char array**
  - lea eax, [ebp-0x10]
  - inc eax
  - movzx eax, BYTE PTR [eax]
  - movsx eax, al

Code Patterns

- **Condition Structures**
  - cmp/test a, b
  - jxx address
- **Loops**
  - Back jumps may indicate loops

Stack Memory

- **Stack data type, nothing exciting**
- **Two related instructions;**
  - push
  - pop
- **Two related registers;**
  - ESP
  - EBP



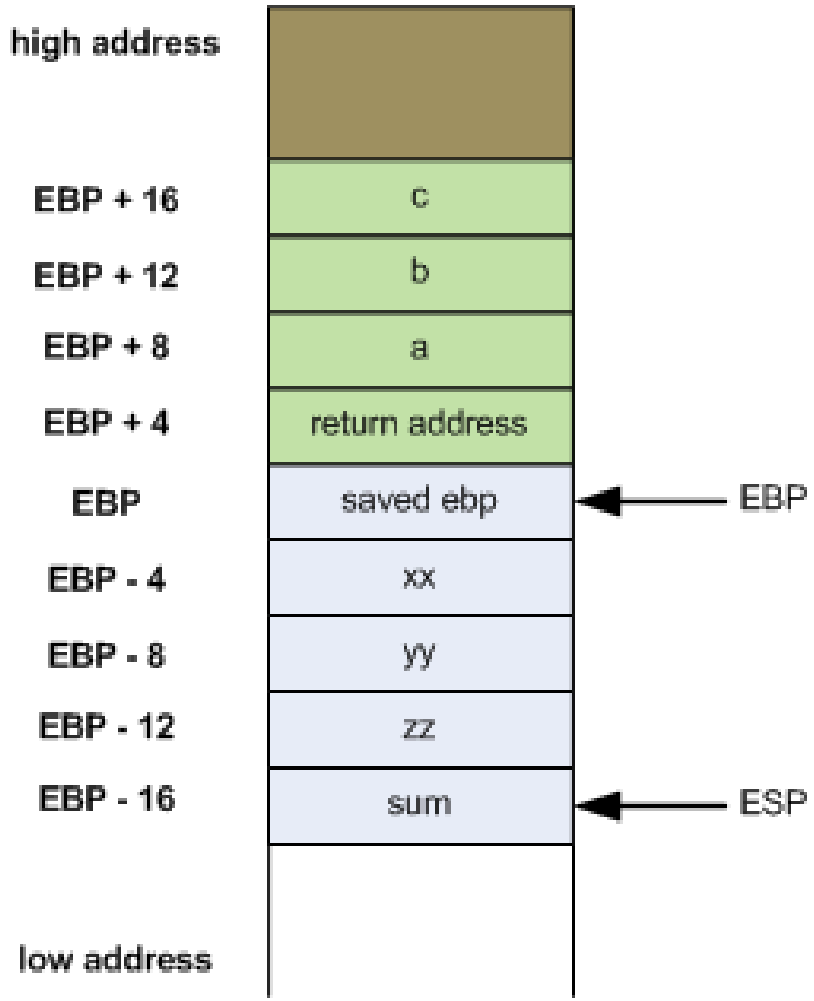
Functions a.k.a “Procedures”

- **Stack Frame**
  - Only a fragment of Stack Memory
  - Generally used for storing local variables and some internal values
  - Every function has their own
  - Size controlled by EBP and ESP

Functions a.k.a “Procedures”

- **Stack Frame**

```
void func1(int a, int b, int c) {
    int xx, yy, zz;
    int sum;
}
int main() {
    func1(1, 2, 3);
    return 0;
}
```



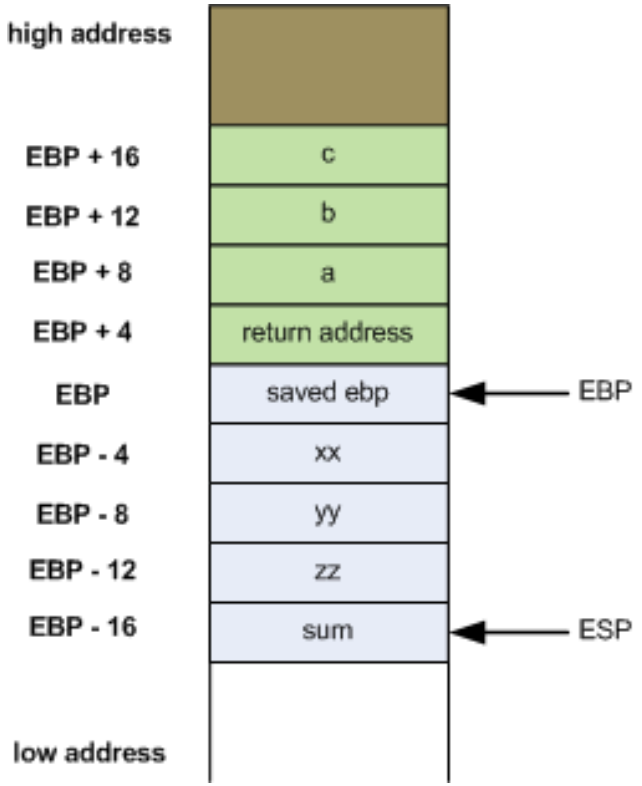
Calling Conventions

- **Syntax for functions in high level languages;**
  - var result = function_name(arg1, arg2);
- **How functions are called in Assembly ?**
- **How it can return a value ??**
- **How it can continue executing from where it left off ???**

Calling Conventions

- **Calling conventions are a part of ABI**
- **Lets do an example!**
  - callconv.c

Remember this?? ->



Function Prologue/Epilogue

- **Prologue: When entering a function**
  - Save the “state” of program
  - Create memory for function’s use
- **Epilogue: When leaving a function**
  - Return a value (if any)
  - Restore the “state”

Code Patterns

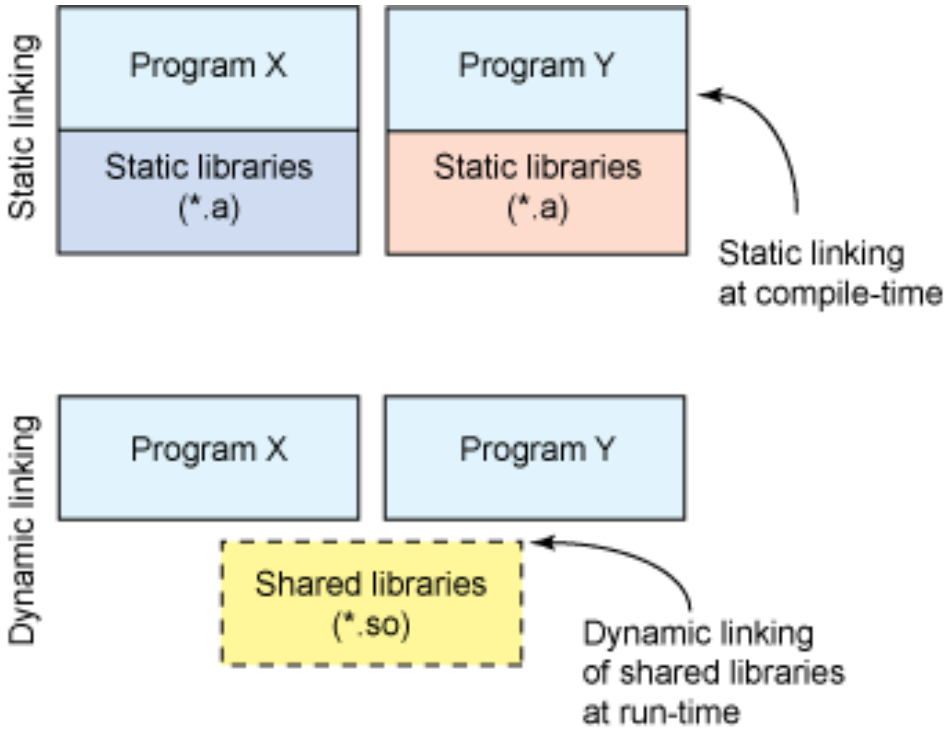
- **Local variables lay in the stack frame**
  - [ebp-0xN] or [esp+0xN]
- **Function arguments**
  - [ebp+0x8 + 4*index]
    - Argv[0] = [ebp+0x8]
    - Argv[1] = [ebp+0xc]
    - Argv[2] = [ebp+0x10]
  - Different conventions exist

Code Patterns

- **Return value**
  - mov eax, value

External Procedures

- **Function addresses are resolved at runtime**
- **Keywords: Relocations, PLT/GOT**



Code Patterns

- **Lets inspect :)**
  - -ggdb and readelf are helpful

Code Patterns

Code Patterns

Code Patterns

Code Patterns