Introduction to Reverse Engineering

01 - Assembly Primer

https://github.com/0x03c6/IRE/

What is Assembly?

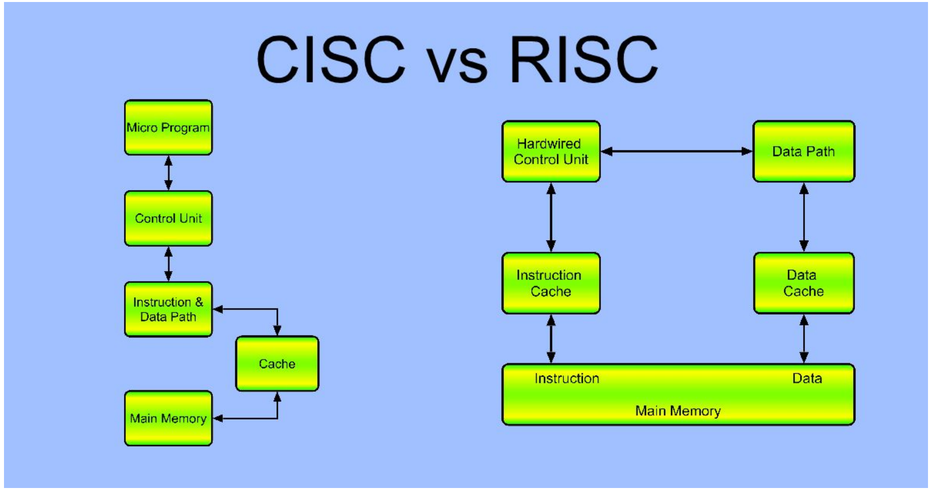
- The primitive instructions in which the **CPU** recognizes and executes.
- Assembly is an **imperative** / **procedural** programming language which doesn't have any inherent support for abstractions such as a **type system**.
- The language consists entirely of **instructions** and **keywords** which allow you to define and manipulate data at the **lowest level** (software).

```c
#include <stdlib.h>
int sub(int x, int y){
        return 2*x+y;
}

int main(int argc, char ** argv){
        int a;
        a = atoi(argv[1]);
        return sub(argc,a);
}
```

```
.text:00000000 _sub:     push   ebp
.text:00000001           mov    ebp, esp
.text:00000003           mov    eax, [ebp+8]
.text:00000006           mov    ecx, [ebp+0Ch]
.text:00000009           lea    eax, [ecx+eax*2]
.text:0000000C           pop    ebp
.text:0000000D           retn
.text:00000010 _main:    push   ebp
.text:00000011           mov    ebp, esp
.text:00000013           push   ecx
.text:00000014           mov    eax, [ebp+0Ch]
.text:00000017           mov    ecx, [eax+4]
.text:0000001A           push   ecx
.text:0000001B           call   dword ptr ds:__imp__atoi
.text:00000021           add    esp, 4
.text:00000024           mov    [ebp-4], eax
.text:00000027           mov    edx, [ebp-4]
.text:0000002A           push   edx
.text:0000002B           mov    eax, [ebp+8]
.text:0000002E           push   eax
.text:0000002F           call   _sub
.text:00000034           add    esp, 8
.text:00000037           mov    esp, ebp
.text:00000039           pop    ebp
.text:0000003A           retn
```

CISC & RISC

- **CISC** (Complex Instruction Set Computer) and **RISC** (Reduced Instruction Set Computer) are **CPU architectures** which are differentiated by the **complexity** of the instruction set.
- CISC architectures include a large quantity of **specific** instructions.
- RISC architectures contain a smaller variety of instructions which can achieve the same effects by composing operations.



Binary (base 2)

- **Binary** is a numerical representation which allows us to encode all **natural numbers** with only the two symbols: **0 and 1**.
- Each place value within base 2 represents **n^2**, as with **base 10**, each place value represents **n^10**. The symbols **(0, 1)** represent the values **(True, False)** respectively.
- Computers are built up entirely of boolean operations performed on True or False values. These operations or circuits are also referred to as **logic gates**.

Binary Base = 2

	Column 8	Column 7	Column 6	Column 5	Column 4	Column 3	Column 2	Column 1
Base^{exp}	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Weight	128	64	32	16	8	4	2	1

$2^0 = 1$

$2^1 = 2$

$2^2 = 2 * 2 = 4$

$2^3 = 2 * 2 * 2 = 8$

$2^4 = 2 * 2 * 2 * 2 = 16$

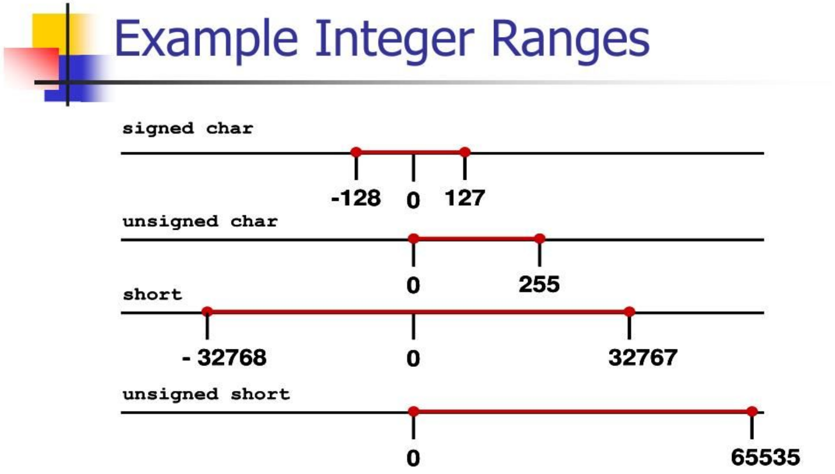
$2^5 = 2 * 2 * 2 * 2 * 2 = 32$

$2^6 = 2 * 2 * 2 * 2 * 2 * 2 = 64$

$2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128$

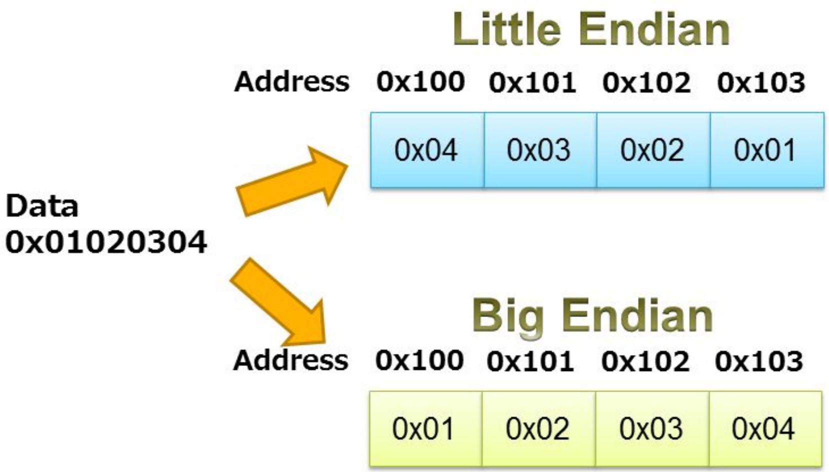
Signedness

- **Signedness** is a property of data which allows us to represent both positive and negative values.
- We can achieve this via **two's complement**, which essentially allows us to represent both negative and positive values by storing a **sign bit** at the **most significant bit**.
- For example, a **16 bit integer** is capable of storing **−32768 to 32767** as **signed**, and **0 to 65535 unsigned**.



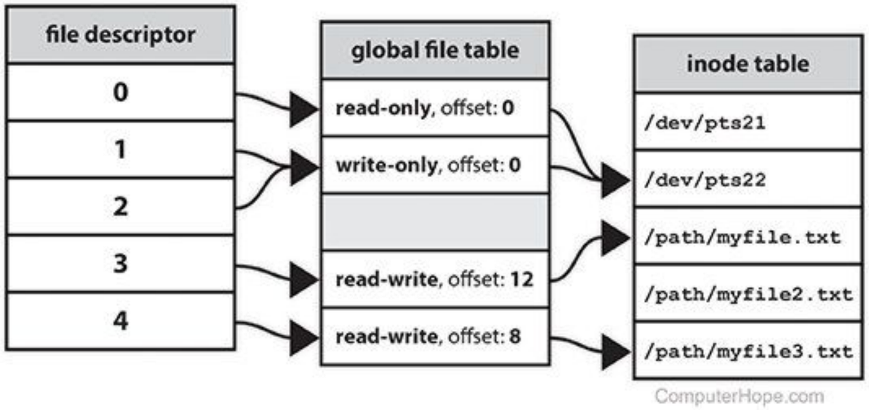
Endianness

- The term **endianness** refers to the **ordering** of bytes within memory.
- **Big endian** is a system which stores the **MSB** (Most Significant Byte) at the lowest and the **LSB** (Least Significant Byte) at the highest addresses.
- **Little Endian** is the opposite, storing the **lsb** at the lowest and the **msb** at the greatest addresses.
- Little endian is typically more commonly utilized as it is more **efficient** for a computer to read, as opposed to big endian, which is easier for a **human** to read.



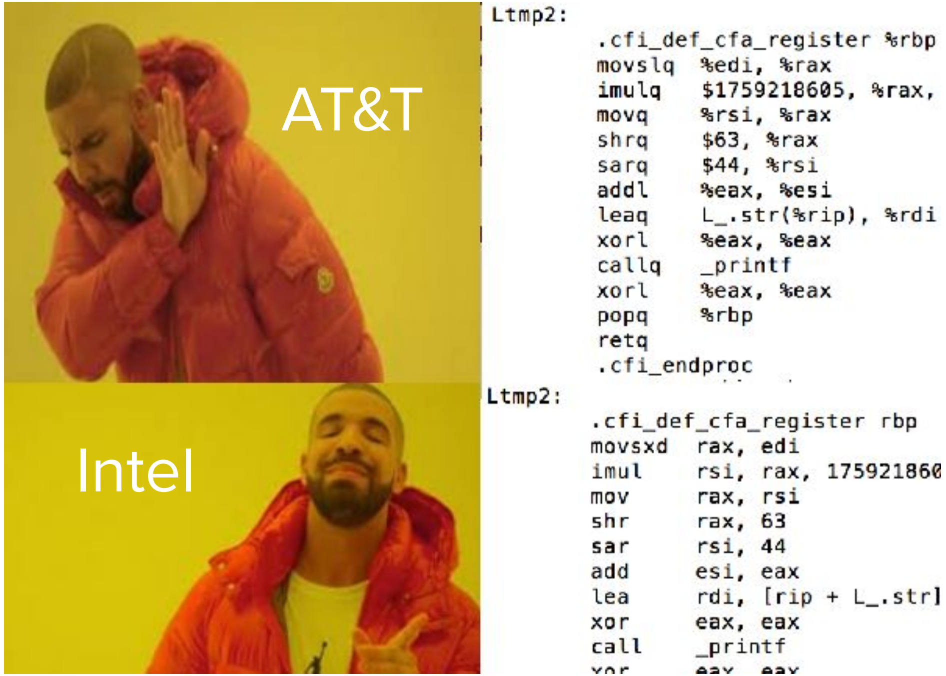
File Descriptors

- A unique integer **identifier** which represents a specific open **resource**.
- Everything within linux is a file, or at the very least behaves as one. This includes everything from **kernel modules** to **network connections** (sockets).
- File descriptors are an essential component of any **operating system**, windows (NT kernel) has **handles**.



Intel vs AT&T

- There are two canonical syntaxes for x86 which are named **Intel** and **AT&T**.
- The origin behind the AT&T is that it was created during the conception of **Unix** at **bell labs**; which was owned by AT&T at the time.
- We will be focusing primarily on Intel syntax as it is much simpler to understand and organize.



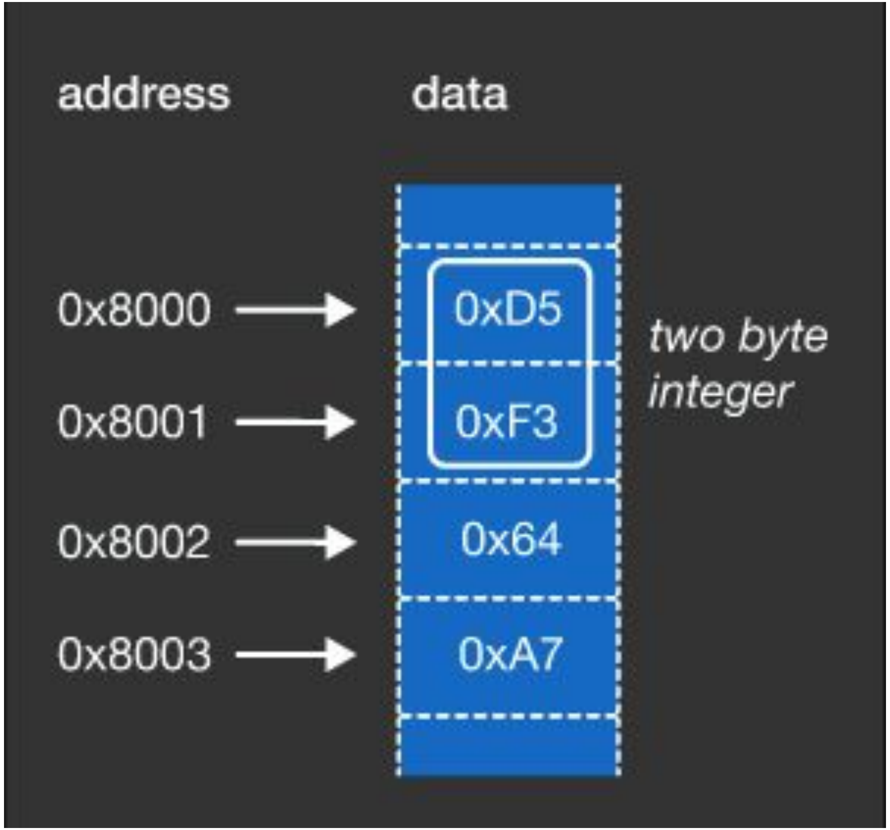
Registers

- x86 has 16 **general purpose registers** and a special register called the **instruction pointer**. The instruction pointer, as the name suggests, points to the current instruction being **executed**.
- The x86 architecture was introduced back in **1978**, where the first rendition only supported **8 bit addressing mode**.

64-bit register	Lower 32 bits	Lower 16 bits	Lower 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

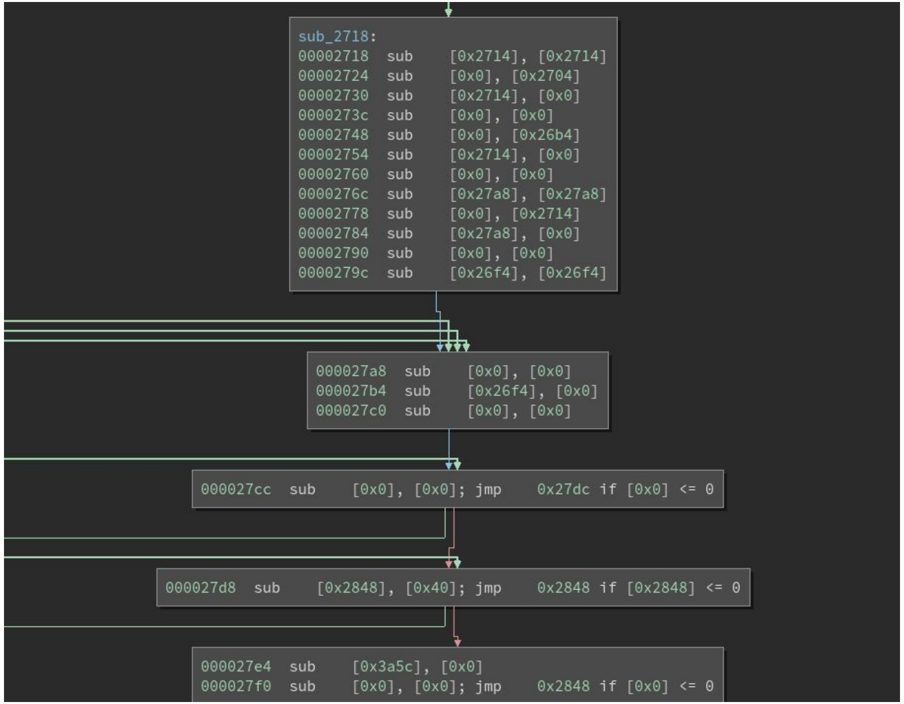
Memory

- We can think of memory as a **linear addressing model**, meaning that memory is simply a single contiguous address space.
- An easier representation of memory is to think of it like a very long strip of paper that has a specific width.
- Memory has **permissions**, which either allow or restrict the ability to **read**, **write** and **execute** on that specific **page** of memory.
- Our operating system has to handle and abstract our memory for performance reasons so its not this simple in practice.



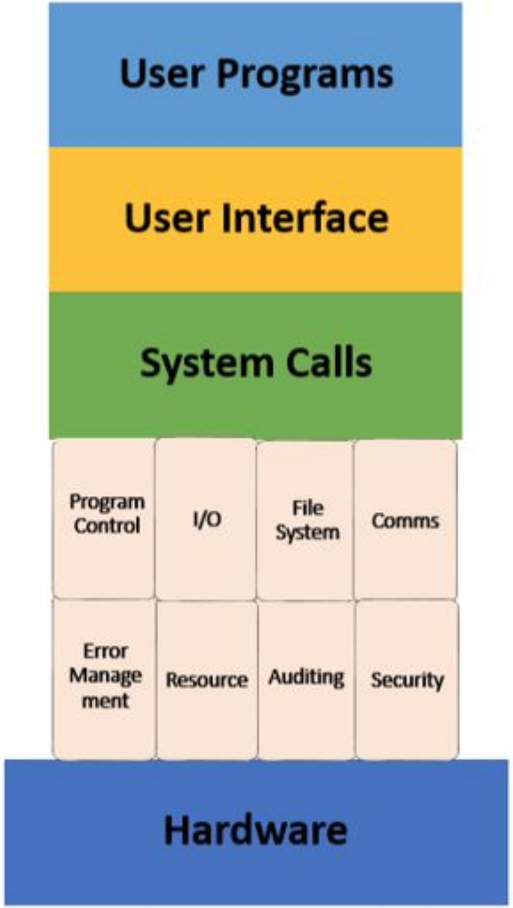
Control Flow

- **Control flow** is a simple concept which simply represents the order of **instructions** that are being **executed**.
- In the case of **assembly**, we have a few **instructions** which can allow us to dictate the control flow of our program. We can **arbitrary** and **conditional** jump instructions that allow us to **satisfy properties programmatically**.



System Calls

- But how do we interact with our **environment** (operating system)?
- We have **system calls**, which allow us to generate an **interrupt** and request an operation to be performed by the **kernel**.
- An example of how we can use system calls is the ability read and **write** from the **disk**, **network** and other resources.
- I have included a system call table reference named **unistd_32.h** / **unistd_64.h** within the repository.



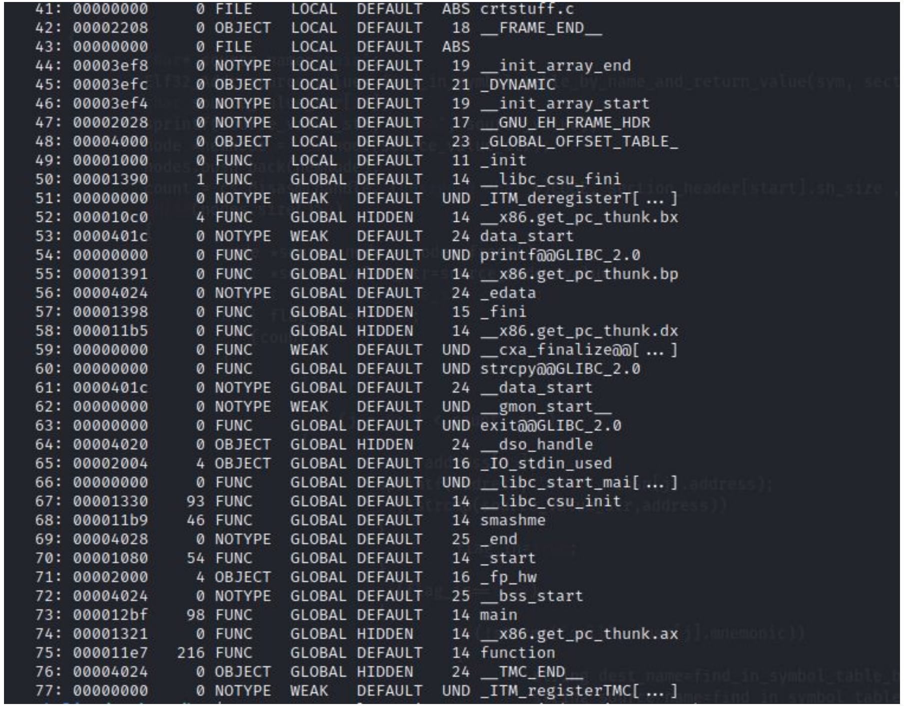
Data

- Everything is **bits** and **bytes** on a computer. It may not seem like it but it is all represented as binary when it comes down to the **CPU**.
- An example of this are **ASCII** characters. The standard ASCII format which display english symbols are represented by decimal **integers**.
- There are standard sizes in which our CPU reads and writes called **words**. A **byte** is 8 bits, a **word** is 2 bytes, a **dword** (double word) is 4 bytes and a **qword** (quad word) is 8 bytes.

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

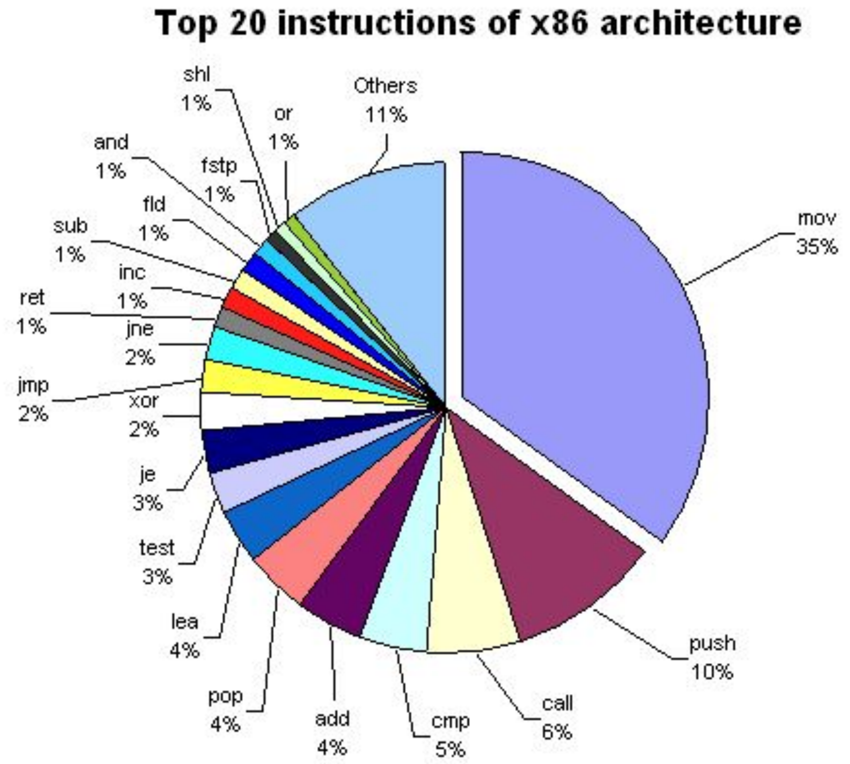
Symbols

- A **symbol**, otherwise known as **symbolic references**, are names which represent **important regions** of memory within our **binary**.
- An example of a symbol is something simple like a **global variable** or **function**. The function **printf** is a symbol.
- There are **import** and **export** symbols, which we will get into later when we dive deeper into the **ELF format** and the **linking process**.



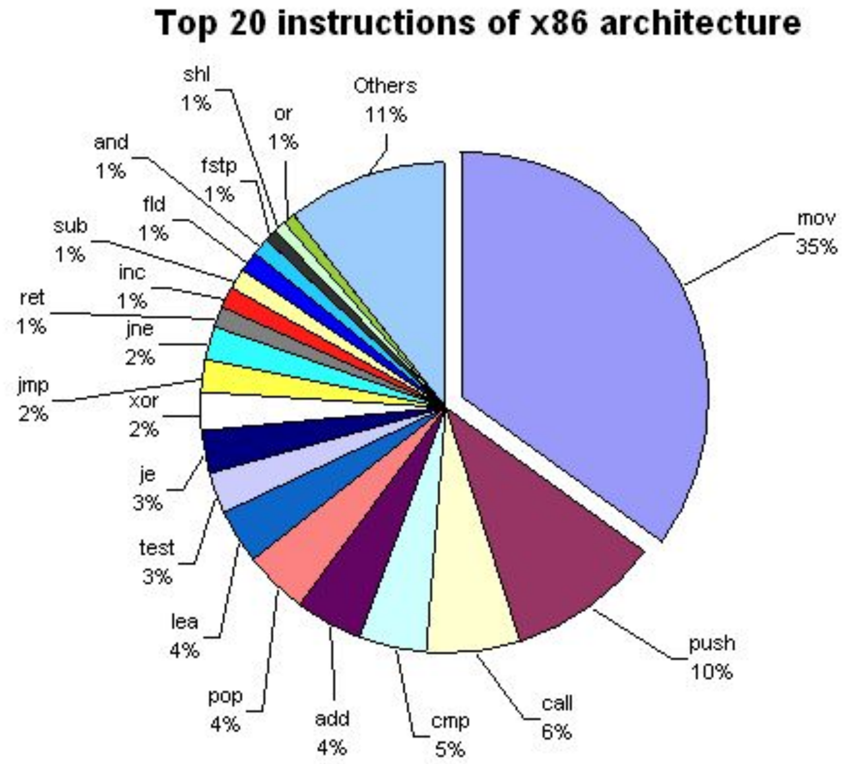
Instructions: MOV

- Move data from one **destination** into one **source**.
- The **arguments** or **parameters** that are passed to an instruction are commonly referred to as **operands**.
- **MOV AX, BX**: This instruction transfers the contents of the register **bx** into the register **ax**.
- The source and destination operands can vary from register to memory addresses to immediate values.



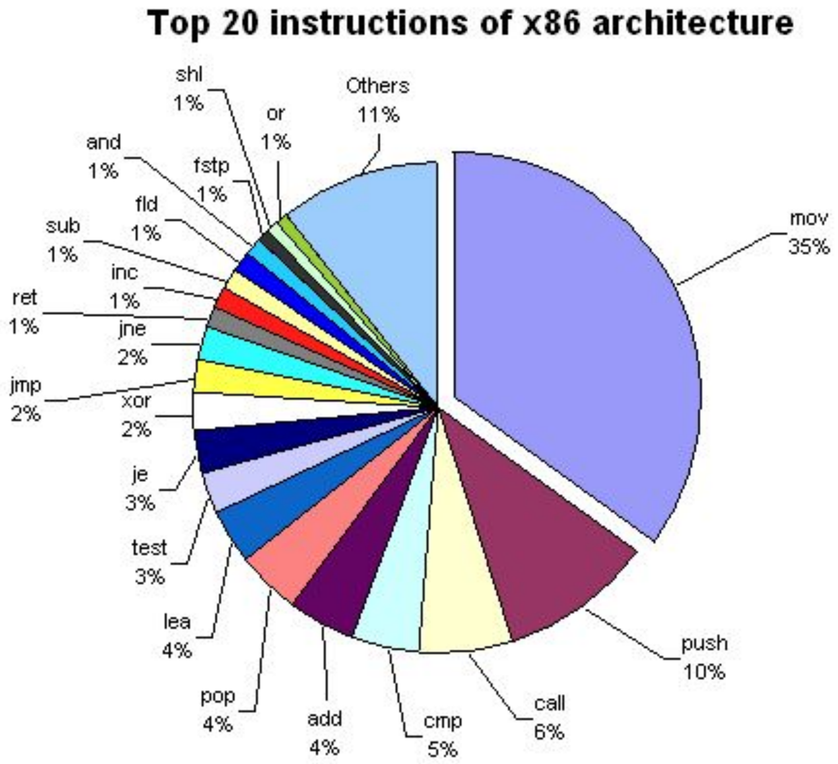
Instructions: LEA

- The **LEA** instruction which stands for **Load Effective Address**.
- It is very similar to the MOV instruction, except rather than directly moving the **value** of the source operand, it will load the address or reference of the source into the destination.
- You can think of this instruction as the **reference** (&) **operator** in C, which allows us to find the **address** of a **variable** within **memory**.



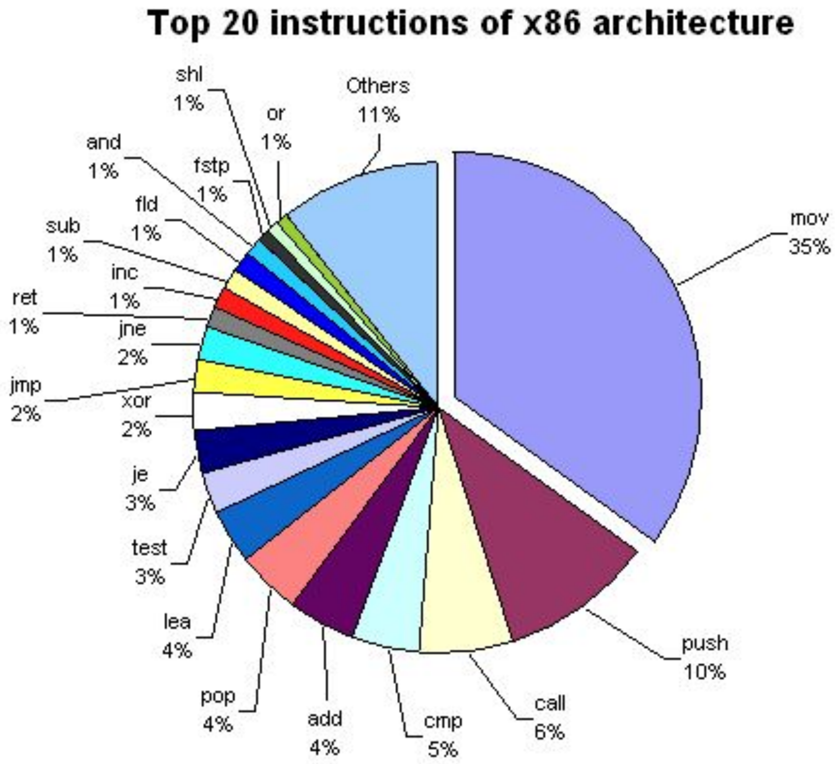
Instructions: ADD / SUB / MUL

- Allows us to perform **arithmetic operations**.
- The same format as the other **instructions** shown, it takes two **operands**, a **source** and a **destination**.
- The operation will be applied from the source to the destination, with the result being moved into the destination.
- **ADD AX, BX**: This instruction adds the values of ax with **BX**, then moves the result into the **AX** register.
- Just to keep in mind, there are **signed** and **unsigned** variants of these arithmetic operations.



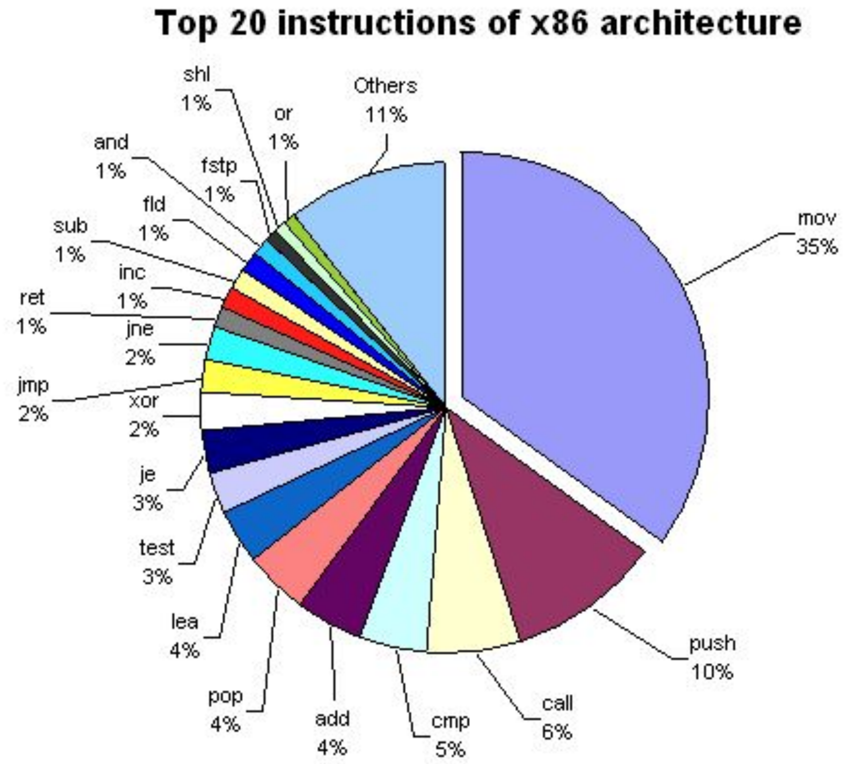
Instructions: DIV

- The **DIV** (divide) instructions resembles the other arithmetic operations with a few caveats.
- When performing a division operation, there are more things to be aware of, specifically things like division by zero errors and such. There are also two outputs for a division operation so we cannot store the output conventionally like the other arithmetic instructions.
- Not to get too specific, but the DIV instruction will store the quotient and remainder in two separate general purpose registers.



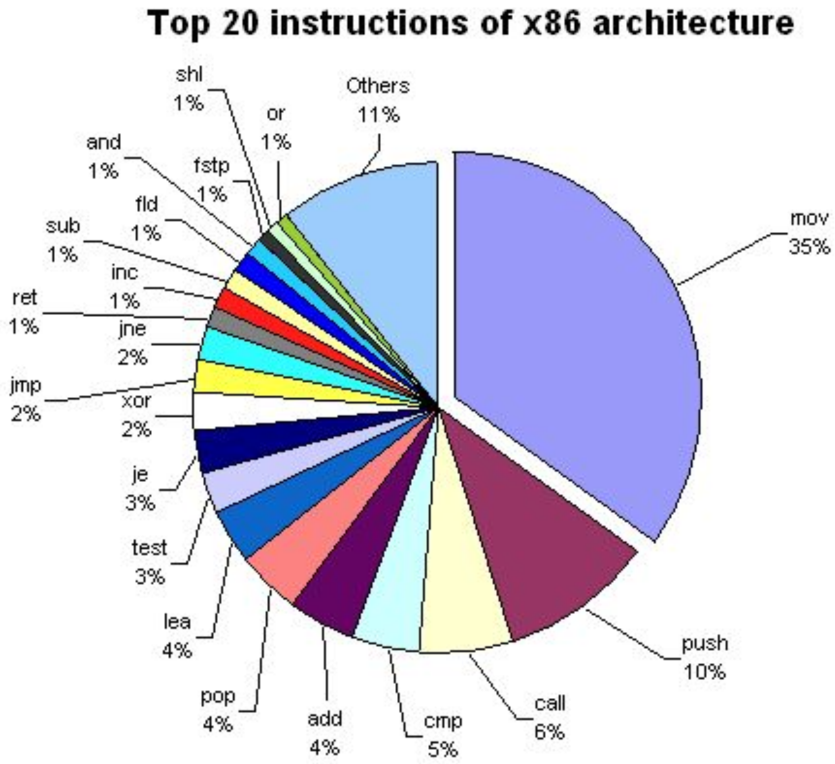
Instructions: CMP

- The **CMP** (compare) instruction allow us to **compare** the value of two **operands**.
- This instruction tests if the two operands are equal, less than, or greater than and will set the CPU flags accordingly.
- Comparison instructions set CPU **flags**, which allows the CPU to remember the state of a specific condition. This is not particularly relevant.



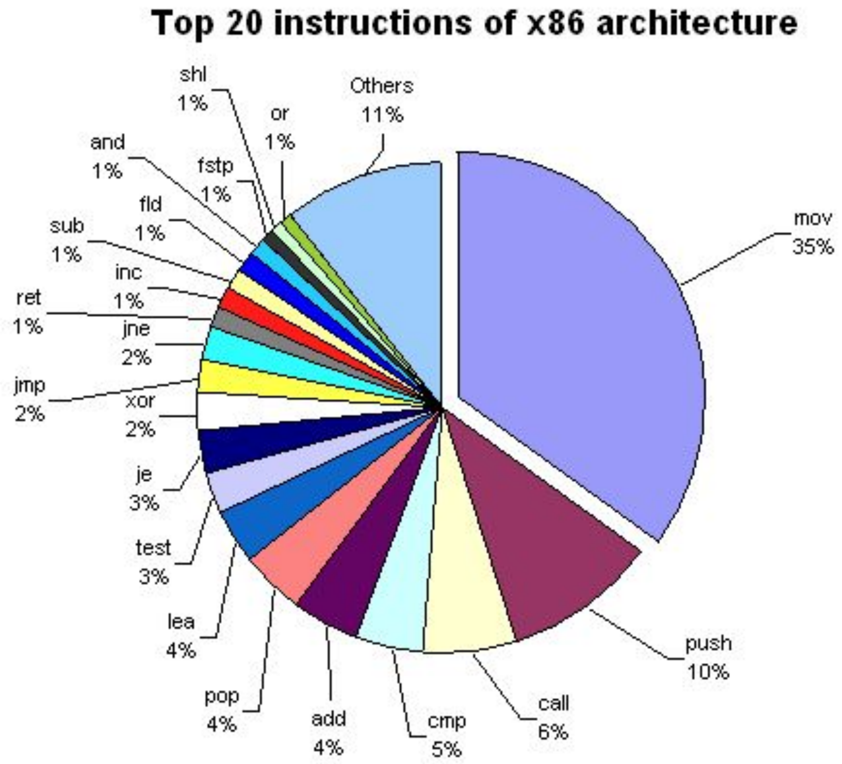
Instructions: TEST

- **TEST** is another **conditional** instruction which allows us to test **properties** of two **operands**.
- In the case of the TEST instruction, it will perform a simple **bitwise AND** on the two operands.
- The result of the bitwise AND will set the **SF** (Sign Flag), **ZF** (Zero Flag) and **PF** (Parity) flags.
- The important flag to remember is the ZF flag, as it is what generally dictates **control flow**.



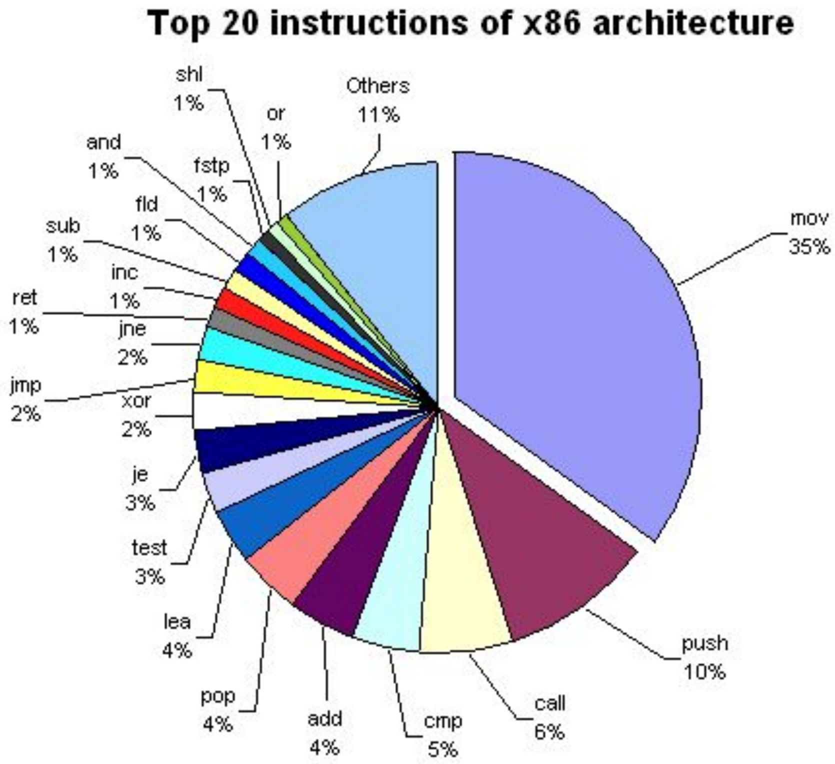
Instructions: JMP

- The **JMP** (Jump) instruction allows us to dictate **control flow** to any arbitrary **instruction**.
- There are **conditional** variants of the JMP instruction which allow us to jump **if and only if** a condition is met (or if a cpu flag is set).
- As there is no **loop structure** present within any assembler, we will have to rely on the use of conditional jumps to perform the same operations.



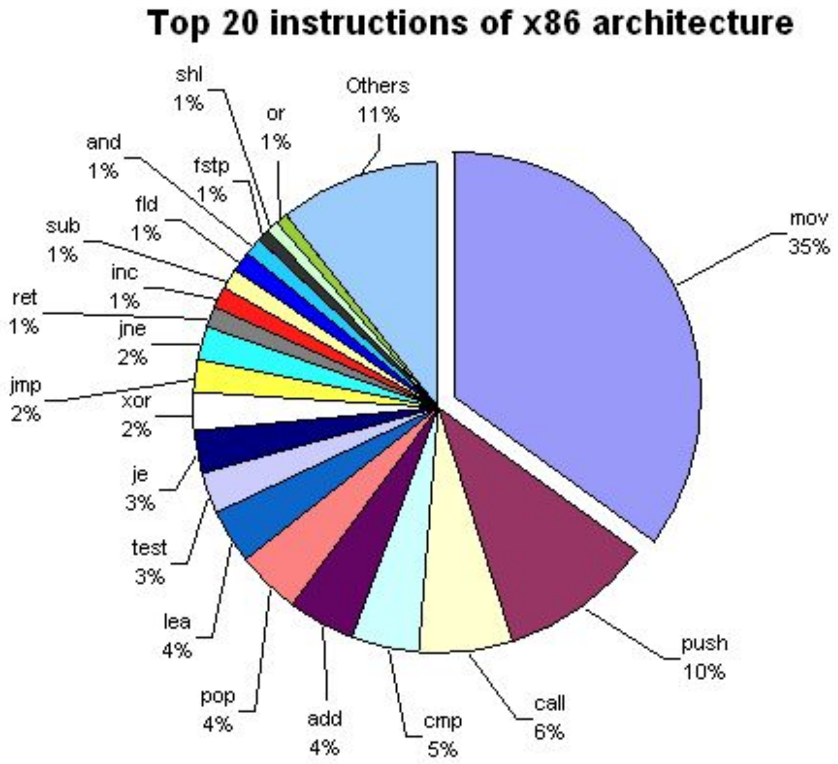
Instructions: PUSH / POP

- These **instructions** allow us to operate on our **process stack**.
- The **PUSH** instruction will simply write the operands value to the top of the stack and **decrement** the stack pointer (rsp).
- The **POP** instruction does the opposite, writing the value at the top of the stack into the operand and **incrementing** the stack pointer (rsp).



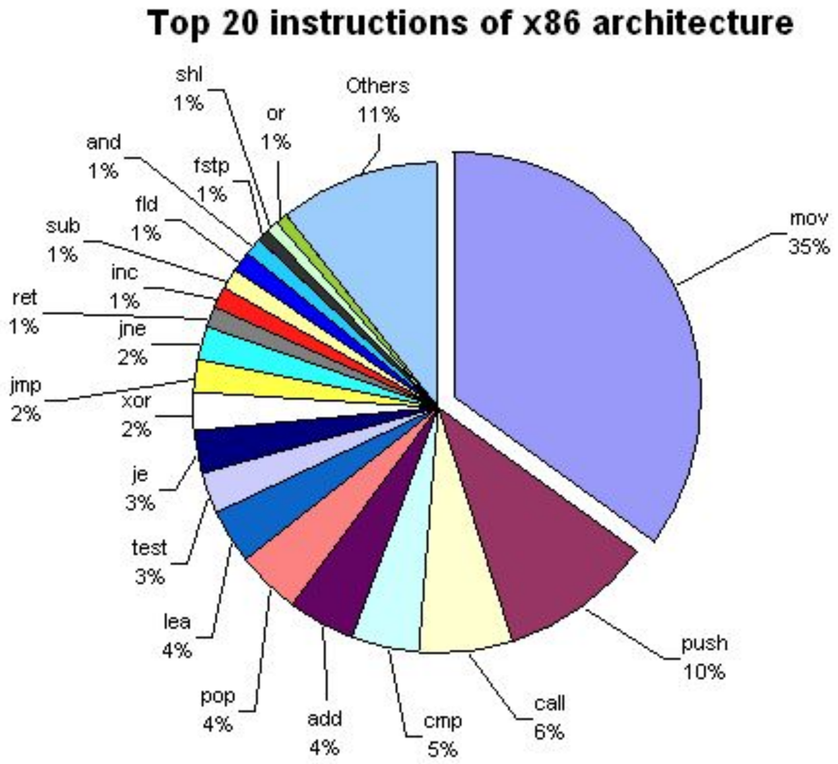
Instructions: CALL

- The **CALL** instruction is similar to the **JMP** with a very minute difference.
- This instruction will first **push** the **return address** onto the **stack** before jumping to the address. The return address is simply an address within memory that the program can now use to return to the **original point** of **execution**.
- The CALL instruction is typically used for calling **functions** or **procedures**. We use the CALL instruction when we expect to return to the same point after the **execution** of the **subroutine**.
- Just like **calling** any **function** in any programming language, it always returns to the same point after the function completes.



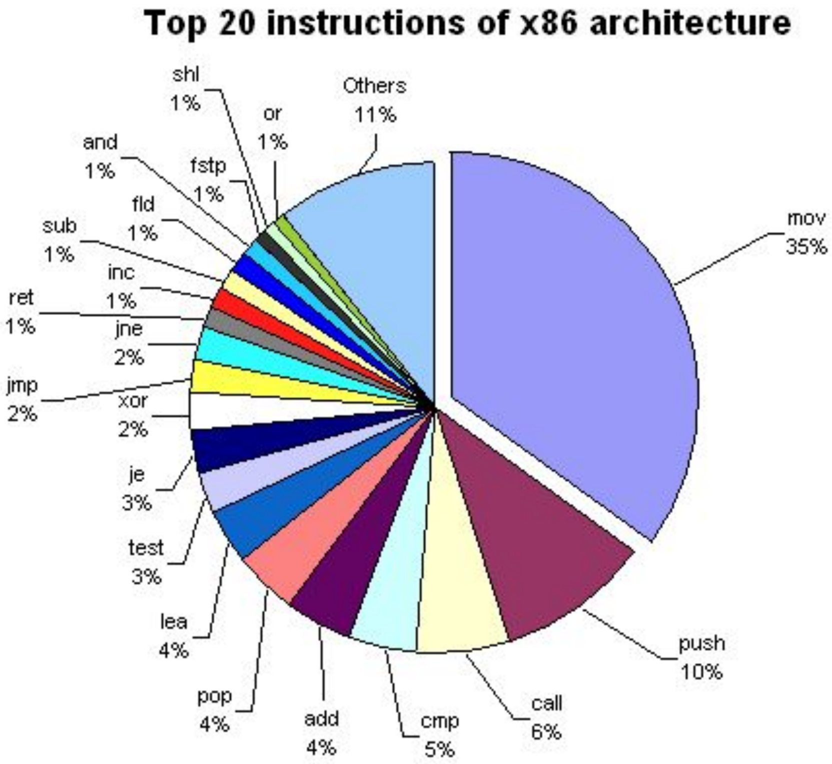
Instructions: RET

- The **RET** (return) instruction works in tandem with the **CALL** instruction. This instruction simply **pops** the value at the top of the **stack** into the **instruction pointer**.
- This essentially has the program **jump** back to the original **return address** which is stored on the **stack**.
- The RET instruction is also fairly **vulnerable** as it leads to a technique known as **ROP** (Return Oriented Programming), which abuses these mechanisms.



Instructions: SYSCALL

- The **SYSCALL** instruction essentially allows us to **communicate** with the **kernel** that we want a specific task completed.
- In x86_64, the **RAX** register is used to hold the syscall we want to execute. **RDI** (Destination), **RSI** (Source) and **RDX** are used to pass parameters to our system call.



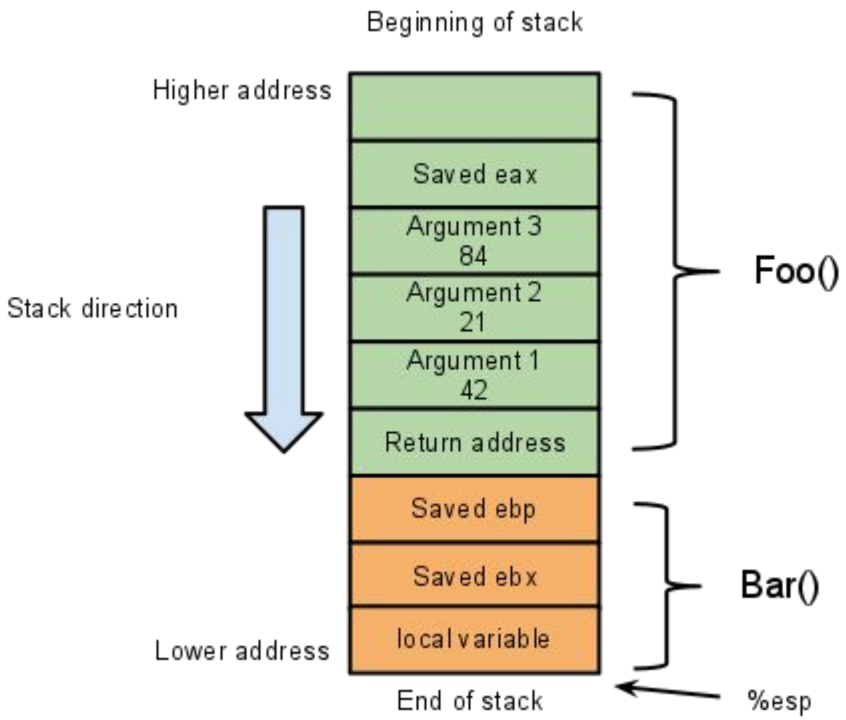
Calling Conventions

- **Calling conventions** vary drastically from **architecture** to **platform** to even individual pieces of **software**.
- The term calling convention simply refers to the convention which dictates how **functions** are called, how **parameters** are passed and how **stack frames** are managed and cleaned.
- **cdecl**, **clrcall**, **fastcall**, **stdcall**, **thiscall** and more.

Keyword	Stack cleanup	Parameter passing
__cdecl	Caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	n/a	Load parameters onto CLR expression stack in order (left to right).
__stdcall	Callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	Callee	Stored in registers, then pushed on stack
__thiscall	Callee	Pushed on stack; **this** pointer stored in ECX

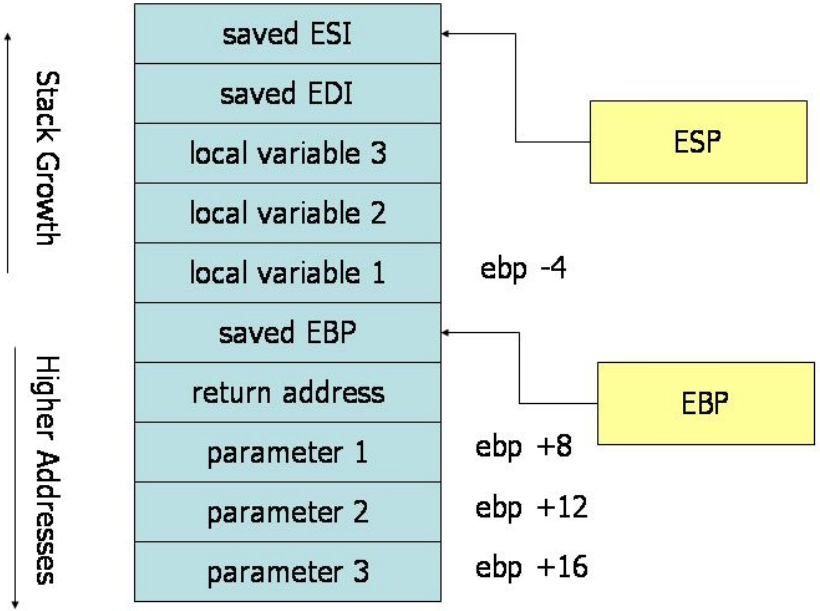
Calling Conventions: cdecl

- **Cdecl** (C declaration) is a calling convention which relies on passing **parameters** via the **registers** and the **stack**.
- On x86_64, **arguments** are passed through either registers, or the stack. The **return value** is passed through the **ax** register (or **rax** on x86_64).



The Stack

- The stack is a **large region** of **memory** mapped to the **process** by the **loader** which stores information about the **execution** of the program, as well as **local variables**.
- **LIFO** (Last in First out)
- The stack grows **downwards** in memory.
- The stack is just memory, it is just **registers** which essentially dictate its structure.



The Stack: cdecl

rip

call main

main:

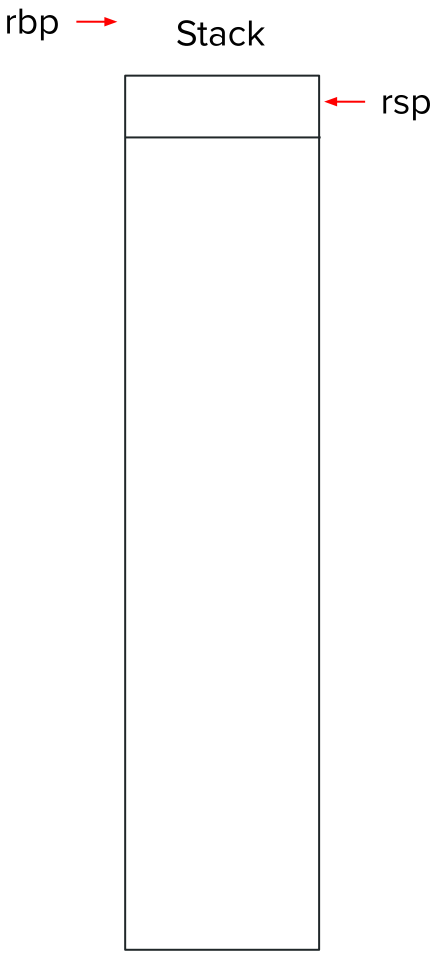
push rbp

mov rbp, rsp

; code goes here

pop rbp

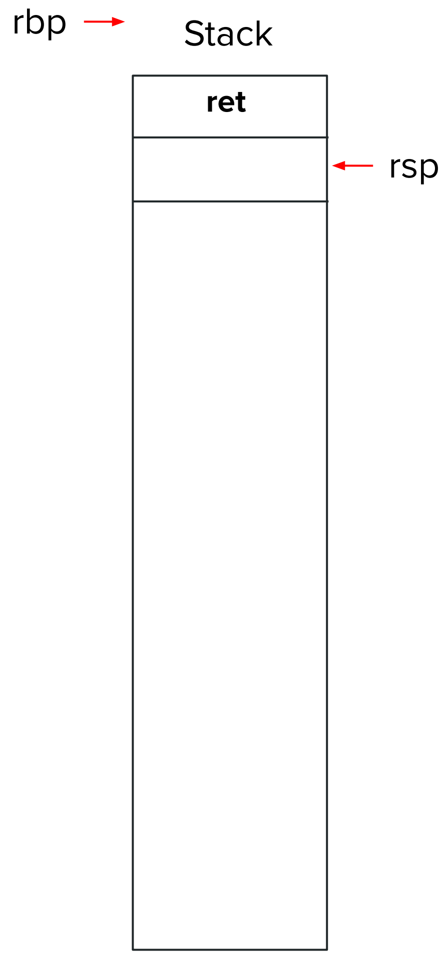
ret



The Stack: cdecl

```
call main

main:

push rbp

mov rbp, rsp

; code goes here

pop rbp

ret
```

rip



The Stack: cdecl

call main

main:

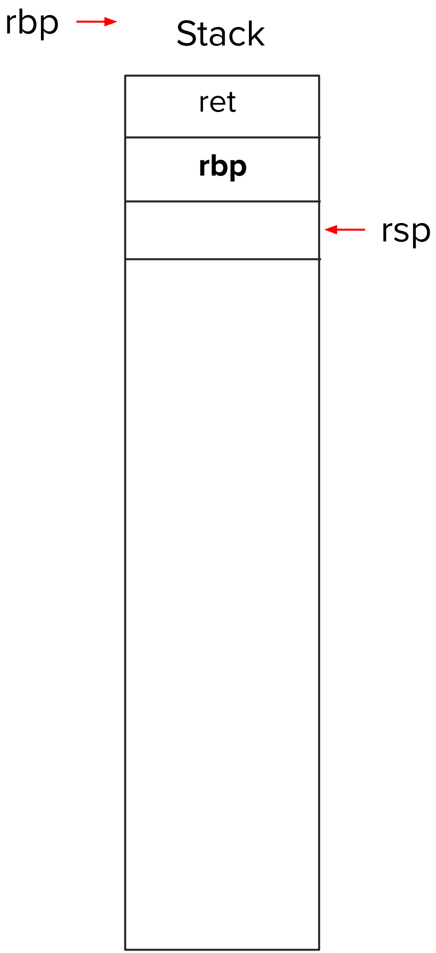
push rbp ← rip

mov rbp, rsp

; code goes here

pop rbp

ret



The Stack: cdecl

call main

main:

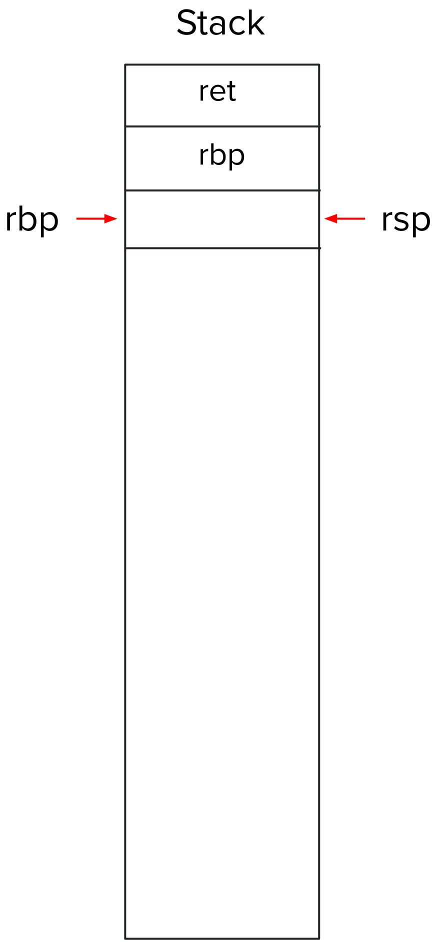
push rbp

mov rbp, rsp ← rip

; code goes here

pop rbp

ret



The Stack: cdecl

call main

main:

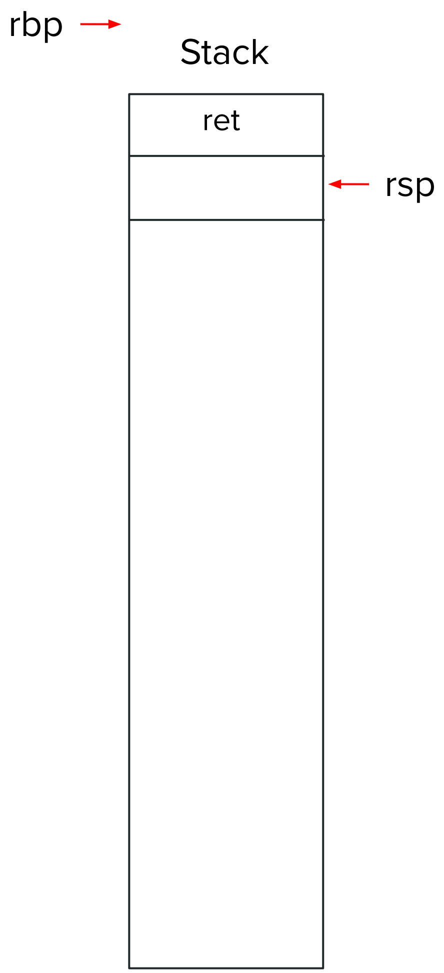
push rbp

mov rbp, rsp

; code goes here

pop rbp ← rip

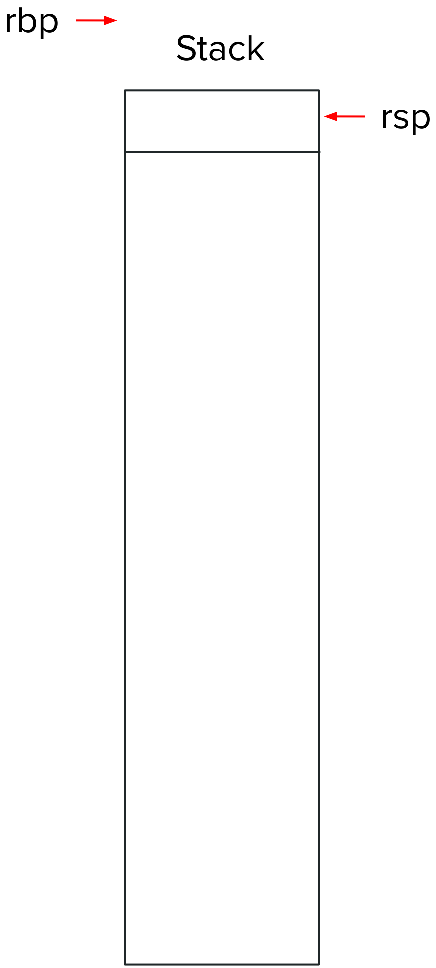
ret



The Stack: cdecl

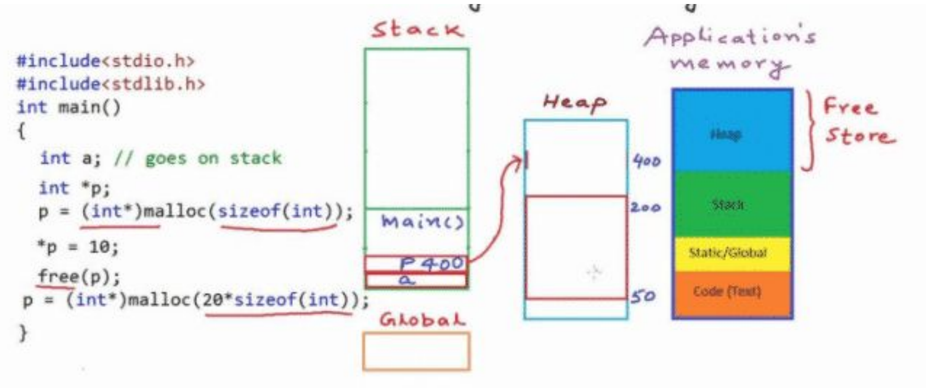
```
call main
main:
push rbp
mov rbp, rsp
; code goes here
pop rbp
ret
```

rip



The Heap

- The **heap** is a **large pool of memory** which allows the user to efficiently and **dynamically** allocate **chunks** of memory at **runtime**.
- Efficient as we don't have to constantly map **pages** from the **kernel** to **userspace**.
- There are many implementations of memory **allocator** algorithms and models that primarily serve specific purposes.



Program: safe_exit.s

- Safely exit the process using system calls.

Program: hello_world.s

- Print “Hello World” to standard output.

Program: user_input.s

- Write an application which reads a name from standard input onto the stack buffer and responds with a greeting.

Challenge: menu.s (optional)

Write your own menu application with assembly!

No strict prompt, do whatever topic you like but it must satisfy the following conditions:

1. It must store the initial input buffer on the stack.
2. The menu application must contain at least 3 options and 3 subroutines. Each subroutine must effectively abide by the cdecl calling convention. You can use the examples provided within the git repo as reference.
3. Have fun!

Closing Thoughts

Remember to practice, everything comes with time you just have to be persistent. This is a vital skill that will be crucial for the coming presentations.

Next, we will dive into the ELF binary format, as well as process internals. We will be writing a simple ELF parser in C.

Any questions?

Additional Resources

https://asmtutor.com

https://www.youtube.com/@OpenSecurityTraining/

https://en.wikibooks.org/wiki/X86_Assembly

https://godbolt.org/