The Assembly Primer

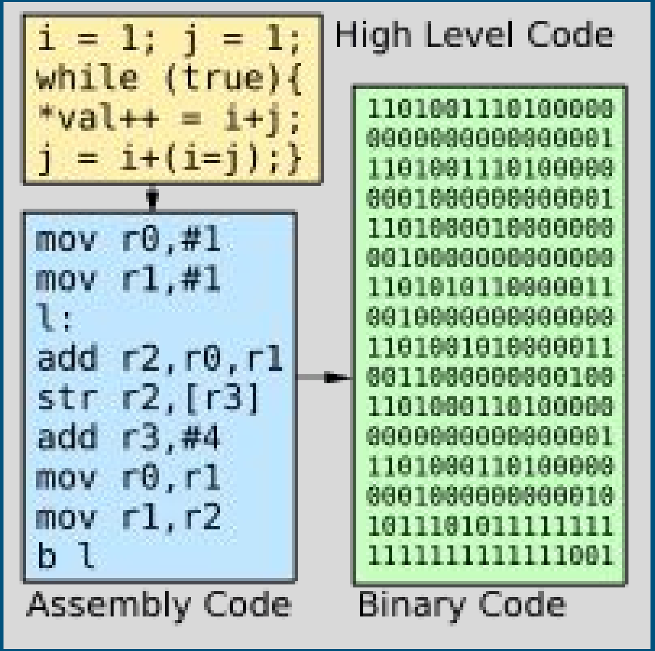
(Oh God)

What is Assembly?

We program in human readable code,
a high level language like python, C, haskell, etc.

The processor only understands binary, so we must compile our code into a ‘binary’

Assembly is a very low-level language that is in between our language and binary



High Level Code

```
i = 1; j = 1;
while (true){
*val++ = i+j;
j = i+(i=j);}
```

Assembly Code

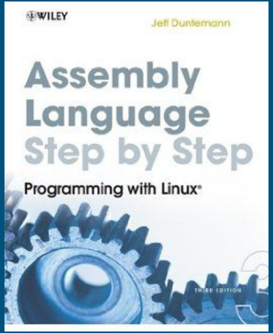
```
mov  r0,#1
mov  r1,#1
l:
add  r2,r0,r1
str  r2,[r3]
add  r3,#4
mov  r0,r1
mov  r1,r2
b l
```

Binary Code

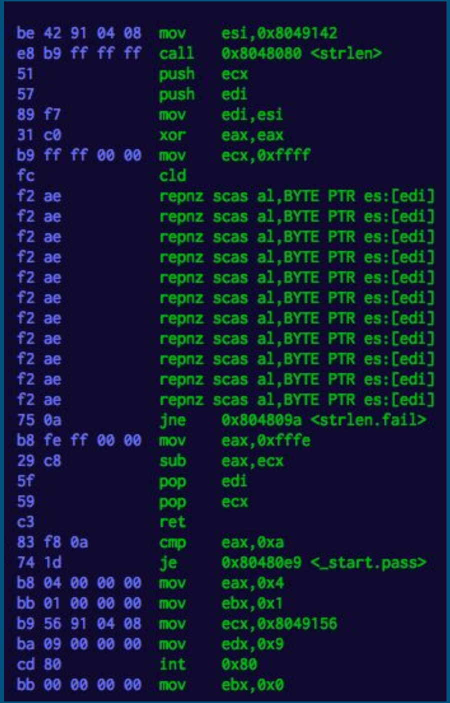
```
1101001110100000
0000000000000001
1101001110100000
0001000000000001
1101000010000000
0010000000000000
1101010110000011
0010000000000000
1101001010000011
0011000000000100
1101000110100000
0000000000000001
1101000110100000
0001000000000010
1011101011111111
1111111111111001
```

When do we see it?

You probably aren't going to write anything in assembly anytime soon (or ever)



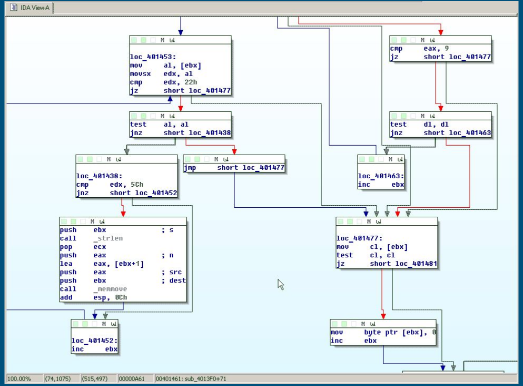
(Doesn't this look like a fun read!)



```
be 42 91 04 08  mov    esi,0x8049142
e8 b9 ff ff ff  call   0x8048080 <strlen>
51              push   ecx
57              push   edi
89 f7           mov    edi,esi
31 c0           xor    eax,eax
b9 ff ff 00 00  mov    ecx,0xffff
fc              cld
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
f2 ae           repnz scas al,BYTE PTR es:[edi]
75 0a           jne    0x804809a <strlen.fail>
b8 fe ff 00 00  mov    eax,0xfffe
29 c8           sub    eax,ecx
5f              pop    edi
59              pop    ecx
c3              ret
83 f8 0a        cmp    eax,0xa
74 1d           je     0x80480e9 <_start.pass>
b8 04 00 00 00  mov    eax,0x4
bb 01 00 00 00  mov    ebx,0x1
b9 56 91 04 08  mov    ecx,0x8049156
ba 09 00 00 00  mov    edx,0x9
cd 80           int    0x80
bb 00 00 00 00  mov    ebx,0x0
```

But you will be seeing it a lot during

BINARY ANALYSIS



```
loc_401453:
mov     al, [ebx]
movsx   edx, al
cmp     edx, 22h
jz      short loc_401477
```

```
test    al, al
jnz     short loc_401438
```

```
loc_401438:
cmp     edx, 5Ch
jnz     short loc_401452
```

```
jmp     short loc_401477
```

```
push    ebx             ; s
call    _strlen
pop     ecx
push    eax             ; n
lea     eax, [ebx+1]
push    eax             ; src
push    ebx             ; dest
call    _memmove
add     esp, 0Ch
```

```
loc_401452:
inc     ebx
```

```
cmp     eax, 9
jz      short loc_401477
```

```
test    dl, dl
jnz     short loc_401463
```

```
loc_401463:
inc     ebx
```

```
loc_401477:
mov     cl, [ebx]
test    cl, cl
jz      short loc_401481
```

```
mov     byte ptr [ebx], 0
inc     ebx
```

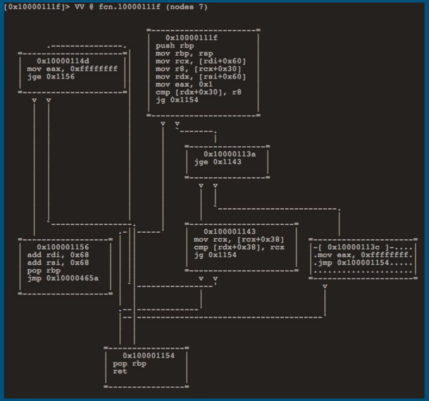
Binary Analysis

Often you will find yourself with a compiled executable that has information you want or has a behavior you want to know

- Malware
- Reversing
- Binary exploitation

Static Analysis	Dynamic Debugging
IDA Pro Hopper Binary Ninja strings	gdb Radare2 Ollydbg (windows) strace

Current cybears strategy



```
[0x10000111f]> VV @ fcn.10000111f (nodes 7)

  0x10000111f
 push rbp
 mov rbp, rsp
 mov rcx, [rdi+0x60]
 mov r8, [rcx+0x30]
 mov rdx, [rsi+0x60]
 mov eax, 0x1
 cmp [rdx+0x30], r8
 jg 0x1154

  0x10000114d
 mov eax, 0xffffffff
 jge 0x1156

  0x10000113a
 jge 0x1143

  0x100001156
 add rdi, 0x68
 add rsi, 0x68
 pop rbp
 jmp 0x10000465a

  0x100001143
 mov rcx, [rcx+0x38]
 cmp [rdx+0x38], rcx
 jg 0x1154

-[ 0x10000113c ]-....
.mov eax, 0xffffffff.
.jmp 0x100001154.....
.....................

  0x100001154
 pop rbp
 ret
```

What is this garbage?

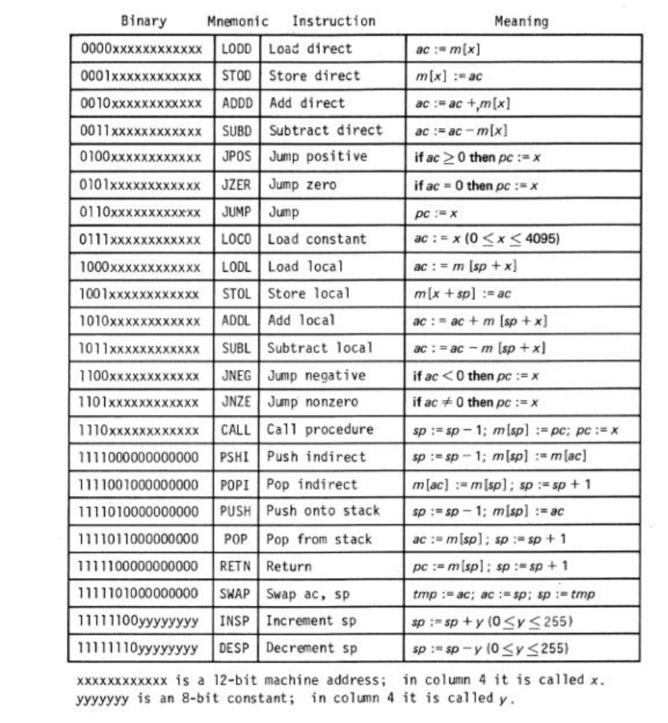
```
1.  2.   3.
MOV EAX, 0x6543
```

1.	2.	3.
Operation	operand	operand
This is the actual command. It’’s like a function in a program. This one moves a value into a register.	This is a register. It is being used as an argument to the function MOV. in this case, it is being used to store the value 0x6543.	This is a hexidecimal number. Almost all the data you see during analysis will be in hexidecimal. In decimal, this number is 25923. It is being stored in the register EAX.

There’s a lot of operations...

Google dat

Shiznit!



Binary	Mnemonic	Instruction	Meaning
0000xxxxxxxxxxxx	LODD	Load direct	$ac := m[x]$
0001xxxxxxxxxxxx	STOD	Store direct	$m[x] := ac$
0010xxxxxxxxxxxx	ADDD	Add direct	$ac := ac + m[x]$
0011xxxxxxxxxxxx	SUBD	Subtract direct	$ac := ac - m[x]$
0100xxxxxxxxxxxx	JPOS	Jump positive	if $ac \geq 0$ then $pc := x$
0101xxxxxxxxxxxx	JZER	Jump zero	if $ac = 0$ then $pc := x$
0110xxxxxxxxxxxx	JUMP	Jump	$pc := x$
0111xxxxxxxxxxxx	LOCO	Load constant	$ac := x \ (0 \leq x \leq 4095)$
1000xxxxxxxxxxxx	LODL	Load local	$ac := m[sp + x]$
1001xxxxxxxxxxxx	STOL	Store local	$m[x + sp] := ac$
1010xxxxxxxxxxxx	ADDL	Add local	$ac := ac + m[sp + x]$
1011xxxxxxxxxxxx	SUBL	Subtract local	$ac := ac - m[sp + x]$
1100xxxxxxxxxxxx	JNEG	Jump negative	if $ac < 0$ then $pc := x$
1101xxxxxxxxxxxx	JNZE	Jump nonzero	if $ac \neq 0$ then $pc := x$
1110xxxxxxxxxxxx	CALL	Call procedure	$sp := sp - 1;\ m[sp] := pc;\ pc := x$
1111000000000000	PSHI	Push indirect	$sp := sp - 1;\ m[sp] := m[ac]$
1111001000000000	POPI	Pop indirect	$m[ac] := m[sp];\ sp := sp + 1$
1111010000000000	PUSH	Push onto stack	$sp := sp - 1;\ m[sp] := ac$
1111011000000000	POP	Pop from stack	$ac := m[sp];\ sp := sp + 1$
1111100000000000	RETN	Return	$pc := m[sp];\ sp := sp + 1$
1111101000000000	SWAP	Swap ac, sp	$tmp := ac;\ ac := sp;\ sp := tmp$
11111100yyyyyyyy	INSP	Increment sp	$sp := sp + y \ (0 \leq y \leq 255)$
11111110yyyyyyyy	DESP	Decrement sp	$sp := sp - y \ (0 \leq y \leq 255)$

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.
yyyyyyy is an 8-bit constant; in column 4 it is called y.

Registers

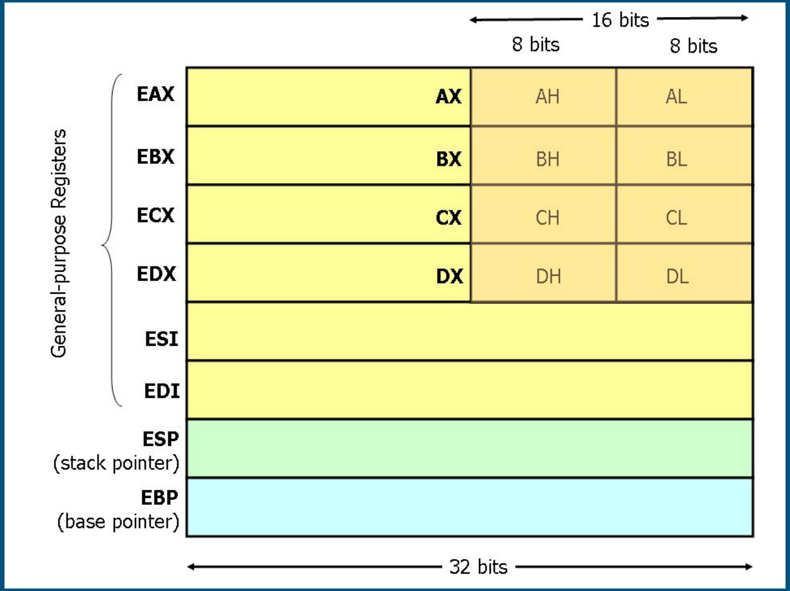
A teeny tiny little storage space on the CPU that can send and receive information very quickly.

Special Registers

EBP- Base Pointer. A reference point for variables (call variable stored 4 slots above EBP)

ESP- Stack Pointer. Shows where the top of the stack is.

FLAG- contains certain status flags for the processor, such as if a comparative operation returned true or false.



The Stack (oh god I hope I get this right)

The stack is a representation of the memory of the computer. Think about it like a continuous strip of magnetic tape in a cassette. You can write functions, variables, and other data to anywhere on this big, continuous strip.

The Stack

Locations on this strip of memory are referenced using hexidecimal addresses. Higher on the stack addresses are lower, and increase as the stack grows downward.

Functions will refer to information stored on the stack by their hexidecimal address. This information is often moved into registries to be used, but can be used straight off the stack.

adress	data
0x000000	earlier
0x000004	stuff
...	...
0xfffffb	later
0xffffff	stuff

Let’s see it