

BlackHoodie

Introduction to x86-64 Reverse Engineering

Marion Marschalek | @pinkflawd | @blackhoodie_re

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/)



What is assembly?

Assembly is textual representation of machine code instructions

Assembler creates executable from assembly source file

Disassembler creates (dis)assembly from machine code bytes

Intel 64 and IA-32 Architectures Software Developer's Manuals

<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

x86 Opcode and Instruction Reference <http://ref.x86asm.net/coder32.html>

x86-64 Cheatsheet from Brown University

https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Hello World!

.LC0:

.string "Hello world!"

main:

```
push    rbp
mov     rbp, rsp
mov     edi, OFFSET FLAT:.LC0
call    puts
mov     eax, 0
pop     rbp
ret
```

Instruction Flow 101

Fetch instruction at program counter

Decode instruction

Increment program counter

Prepare components (ALU, memory, I/O)

Execute instruction

Store results

<https://techdecoded.intel.io/resources/understanding-the-instruction-pipeline/#gs.i0crl3>

Program counter, byte code, assembly, sry wot?

IDA View-RIP

```
ext:00007FF6D5B7119A CC CC CC CC CC CC align 20h
ext:00007FF6D5B711A0
ext:00007FF6D5B711A0 ; ===== S U B R O U T I N E =====
ext:00007FF6D5B711A0
ext:00007FF6D5B711A0 ; int __cdecl main(int argc, const char **argv, const char **envp)
ext:00007FF6D5B711A0 main proc near ; CODE XREF: __tmainCRTStartup+10A1p
ext:00007FF6D5B711A0 ; DATA XREF: .pdata:00007FF6D5B740181p
ext:00007FF6D5B711A0 48 83 EC 28 sub rsp, 28h
ext:00007FF6D5B711A4 E8 27 00 00 00 call sub_7FF6D5B711D0
ext:00007FF6D5B711A9 E8 42 00 00 00 call sub_7FF6D5B711F0
ext:00007FF6D5B711AE E8 4D FE FF FF call sub_7FF6D5B71000
ext:00007FF6D5B711B3 BA 0F 00 00 00 mov edx, 0Fh
ext:00007FF6D5B711B8 B9 05 00 00 00 mov ecx, 5
ext:00007FF6D5B711BD E8 6E FF FF FF call sub_7FF6D5B71130
ext:00007FF6D5B711C2 33 C0 xor eax, eax
ext:00007FF6D5B711C4 48 83 C4 28 add rsp, 28h
ext:00007FF6D5B711C8 C3 retn
ext:00007FF6D5B711C8 main endp
ext:00007FF6D5B711C8
ext:00007FF6D5B711C8 ; -----
ext:00007FF6D5B711C9 align_7FF6D5B711C9: ; DATA XREF: .pdata:00007FF6D5B740181p
ext:00007FF6D5B711C9 CC CC CC CC CC CC align 10h
ext:00007FF6D5B711D0
```

000005A0 00007FF6D5B711A0: main (Synchronized with RIP, Hex View-1)

General registers

RAX	00007FFA76745360	MSVCR110.dll:msvcr110__initenv	ID	0
RBX	0000000000000000		VIP	0
RCX	0000000000000001		VIF	0
RDY	0000014E683036A0	debug022:0000014E683036A0	AC	0
RSI	0000000000000000		VM	0
RDI	0000000000000000		RF	0
RBP	0000000000000000		NT	0
RSP	0000004CC85AF728	Stack[000044AC]:0000004CC85AF728	IOPL	0
RIP	00007FF6D5B711A0	main	OF	0
R8	0000014E683054B0	debug022:0000014E683054B0	DF	0
R9	0000000000000000		IF	1
R10	0000000000000000		TF	0
R11	0000004CC85AF228	Stack[000044AC]:0000004CC85AF228	SF	0
R12	0000000000000000		ZF	1
R13	0000000000000000		AF	0
R14	0000000000000000		PF	1
R15	0000000000000000		CF	0
EFL	00000246			

Hex View-1

```
00007FF6D5B71180 10 8B 4C 24 08 03 C8 8B C1 8B 4C 24 18 2B C8 8B .<L$.E<A<L$.+E<
00007FF6D5B71190 C1 89 44 24 18 8B 44 24 18 C3 CC CC CC CC CC CC A%D$.<D$.Äi1111i
00007FF6D5B711A0 48 83 EC 28 E8 27 00 00 00 E8 42 00 00 00 E8 4D Hfi(è'...èB...èM
00007FF6D5B711B0 FE FF FF BA 0F 00 00 00 B9 05 00 00 00 E8 6E FF bÿÿ°....¹....ènÿ
00007FF6D5B711C0 FF FF 33 C0 48 83 C4 28 C3 CC CC CC CC CC CC C3 ÿÿ3ÄHfÄ(Äi1111i
00007FF6D5B711D0 48 83 EC 28 48 8D 0D 25 1E 00 00 FF 15 0F 0F 00 Hfi(H..%...ÿ....
00007FF6D5B711E0 00 48 83 C4 28 C3 CC CC CC CC CC CC CC CC CC .HfÄ(Äi11111111i
```

000005A0 00007FF6D5B711A0: main (Synchronized with RIP, IDA View-RIP)

Stack view

```
0000004CC85AF728 00007FF6D5B7149F __tmainCRTStartup+10F
0000004CC85AF730 0000000000000000
0000004CC85AF738 0000000000000000
0000004CC85AF740 0000000000000000
0000004CC85AF748 0000000000000000
0000004CC85AF750 0000000000000000
```

UNKNOWN 0000004CC85AF740: Stack[000044AC]:0000 (Synchronized with RSP)

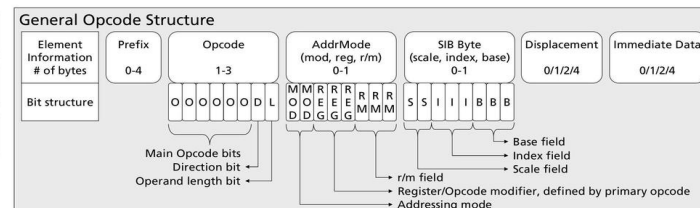
AU: idle Down Disk: 61GB

x86 Opcode Structure and Instruction Overview

1st	2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0		ADD						ES PUSH SS	ES POP SS	OR						CS PUSH DS	TWO BYTE POP DS	
1		ADC								SBB								
2		AND						ES SEGMENT OVERRIDE SS	DAA	SUB						CS SEGMENT OVERRIDE DS	DAS	
3		XOR						SS	AAA	CMP						DS	AAS	
4		INC								DEC								
5		PUSH								POP								
6		PUSHAD	POPAD	BOUND	ARPL	FS	GS	OPERAND SIZE OVERRIDE	ADDRESS SIZE OVERRIDE	PUSH	IMUL	PUSH	IMUL	INS	OUTS			
7		JO	JNO	JB	JNB	JE	JNE	JBE	JA	JCC	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
8		ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP	
9		NOP	XCHG EAX								CWD	CDQ	CALL/FWAIT	PUSHFD	POPFD	SAHF	LAHF	
A		MOV EAX		MOV		CMPS		TEST		STOS		LODS		SCAS				
B		MOV																
C		SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD	
D		SHIFT 1		SHIFT CL		AAM		AAD	SALC	XLAT	FPU							
E		LOOPNZ	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX		
F		LOCK EXCLUSIVE ACCESS	ICE BP	REPNE	REPE	HLT		CMC	TEST/NOT/NEG [i]MUL/[i]DIV	CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH	

1 st	2 nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		(L,S)LDT (L,S)STR VER(R,W)	(L,S)GDT (L,S)IDT (L,S)MSW	LAR	LSL			CLTS		INVD	WBINVD		UD2		NOP		
1		SSE{1,2,3}								Prefetch SSE1	HINT_NOP						
2		MOV CR/DR								SSE{1,2}							
3		WRMSR	RDTSR	RDMSR	RDPMC	SYSENTER	SYSEXIT			GETSEC SMX	MOVBE / THREE BYTE		THREE BYTE SSE4				
4		CMOV															
5		SSE{1,2}															
6		MMX, SSE2															
7		MMX, SSE{1,2,3}, VMX												MMX, SSE{2,3}			
8		JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
		Jcc SHORT															
9		SETO	SETNO	SETB	SETNB	SETE	SETNE	SETBE	SETA	SETS	SETNS	SETPE	SETPO	SETL	SETGE	SETLE	SETG
		SETcc															
A		PUSH FS	POP FS	CPUID	BT	SHLD				PUSH GS	POP GS	RSM	BTS	SHRD		*FENCE	IMUL
B		CMPXCHG	LSS	BTR	LFS	LGS	MOVZX		POPCNT	UD	BT BTS BTR BTC		BTC	BSF	BSR	MOVSB	
C		XADD	SSE{1,2}				CMPXCHG		BSWAP								
D		MMX, SSE{1,2,3}															
E		MMX, SSE{1,2}															
F		MMX, SSE{1,2,3}															

Arithmetic & Logic	Prefix
Memory	System & I/O
Stack	No Operation (NOP) / Multiple Instructions / Extended Instruction Set
Control Flow & Conditional	



Addressing Modes		00	01	10	11
r/m	16bit	32bit	16bit	32bit	32bit
000	[BX+SI]	[EAX]	[EAX]+disp8	[EAX]+disp16	[EAX]+disp32
001	[BX+DI]	[ECX]	[ECX]+disp8	[ECX]+disp16	[ECX]+disp32
010	[BP+SI]	[EDX]	[EDX]+disp8	[EDX]+disp16	[EDX]+disp32
011	[BP+DI]	[EBX]	[EBX]+disp8	[EBX]+disp16	[EBX]+disp32
100	[SI]	[SI]	[SI]+disp8	[SI]+disp16	[SI]+disp32
101	[DI]	[DI]	[DI]+disp8	[DI]+disp16	[DI]+disp32
110	[disp16]	[ESI]	[ESI]+disp8	[ESI]+disp16	[ESI]+disp32
111	[BX]	[EDI]	[EDI]+disp8	[EDI]+disp16	[EDI]+disp32

SIB Byte Structure			
encoding	scale (2bit)	Index (3bit)	Base (3bit)
000	2 ⁰ =1	[EAX]	EAX
001	2 ¹ =2	[ECX]	ECX
010	2 ² =4	[EDX]	EDX
011	2 ³ =8	[EBX]	EBX
100	--	none	ESP
101	--	[EBP]	disp32 / disp16 / [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI

SIB value = index * scale + base

INC—Increment by 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FE /0	INC <i>r/m8</i>	M	Valid	Valid	Increment <i>r/m</i> byte by 1.
REX + FE /0	INC <i>r/m8</i> *	M	Valid	N.E.	Increment <i>r/m</i> byte by 1.
FF /0	INC <i>r/m16</i>	M	Valid	Valid	Increment <i>r/m</i> word by 1.
FF /0	INC <i>r/m32</i>	M	Valid	Valid	Increment <i>r/m</i> doubleword by 1.
REX.W + FF /0	INC <i>r/m64</i>	M	Valid	N.E.	Increment <i>r/m</i> quadword by 1.
40+ <i>rw</i> **	INC <i>r16</i>	O	N.E.	Valid	Increment word register by 1.
40+ <i>rd</i>	INC <i>r32</i>	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> (<i>r, w</i>)	NA	NA	NA
O	opcode + <i>rd</i> (<i>r, w</i>)	NA	NA	NA

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC *r16* and INC *r32* are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

DEST ← DEST + 1;



Intel SDM

INC instruction



Static Analysis Tools on Linux

GNU Binutils

Binary toolsuite

building, modification, analysis

All time favorites:

File

Readelf

Objdump

Strings

Nm

strip

GNU Binutils

The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

But they also include:

- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.
- **windmc** - A Windows compatible message compiler.
- **windres** - A compiler for Windows resource files.

Registers

Memory

Instructions

General Purpose Registers

Eight 32-bit registers:

EAX: Accumulator

EBX: Base

ECX: Counter

EDX: Data

ESI: Source Index

EDI: Destination Index

EBP: Base pointer

ESP: Stack pointer

Register WORD and BYTE access

32-bit Registers can be accessed in the following ways:

31 0 32-bit (EAX)

15 0 16-bit (AX)

15 8 8-bit (AL)

7 0 8-bit (AH)

16-bit WORD access:

AX, BX, CX, DX, SI, DI, BP, SP

8-bit BYTE access:

AH, AL, BH, BL, CH, CL, DH, DL,

Byte, Word, DoubleWord, (64-bit) QuadWord

General Purpose Registers

Sixteen 64-bit
registers:

RAX	R8
RBX	R9
RCX	R10
RDY	R11
RSI	R12
RDI	R13
RBP	R14
RSP	R15

Access modes, eg.:

RAX - EAX - AX - AL

RBX - EBX - BX - BL

Etc. Etc.

R10 - R10d - R10w - R10b

Etc. Etc.

Register purpose decided by
system ABI, compiler convention
and instruction set convention

There's also 128bit SSE registers
etc. etc.

Instruction Pointer

EIP or RIP

- Points to the next instruction

- Cannot be accessed directly

- Can be modified by control flow instructions

- Can be read by function call and popping from stack

FLAGS, EFLAGS and RFLAGS

Status register

Respectively 16, 32 or 64 bits wide (although 64b RFLAGS' upper 32 bit aren't currently used)

Carry flag (CF, bit 0): unsigned overflow

Parity flag (PF, bit 2): LSByte contains even amount of 1s

Zero flag (ZF, bit 6): result is zero

Sign flag (SF, bit 7): MSBit is 1

Trap flag (TF, bit 8): single-step mode

Direction flag (DF, bit 10): increment or decrement on string instructions

Overflow flag (OF, bit 11): signedoverflow

Example

...

```
mov     eax, DWORD PTR [rbp-24]
movsx   rdx, eax
mov     rax, QWORD PTR [rbp-32]
add     rax, rdx
movzx   eax, BYTE PTR [rax]
cmp     al, 95
jne     .L5
add     DWORD PTR [rbp-20], 1
```

.L5:

```
add     DWORD PTR [rbp-24], 1
```

...

CMP instruction

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags [...]

=> Will set ZF (ZF=1) if result is 0

JNE instruction

Jump near if not equal (ZF=0)

Registers

Memory

Instructions

Data Endianness

Byte order

Memory: Little Endian (least significant byte first)

Register: Big Endian (most significant byte first)

Bit order: Big Endian (most significant bit first)

BIG-ENDIAN					Memory				
...	00	01	02	03	04	05	06	07	...
	a	$a+1$	$a+2$	$a+3$	$a+4$	$a+5$	$a+6$	$a+7$	
LITTLE-ENDIAN					Memory				
...	07	06	05	04	03	02	01	00	...
	a	$a+1$	$a+2$	$a+3$	$a+4$	$a+5$	$a+6$	$a+7$	

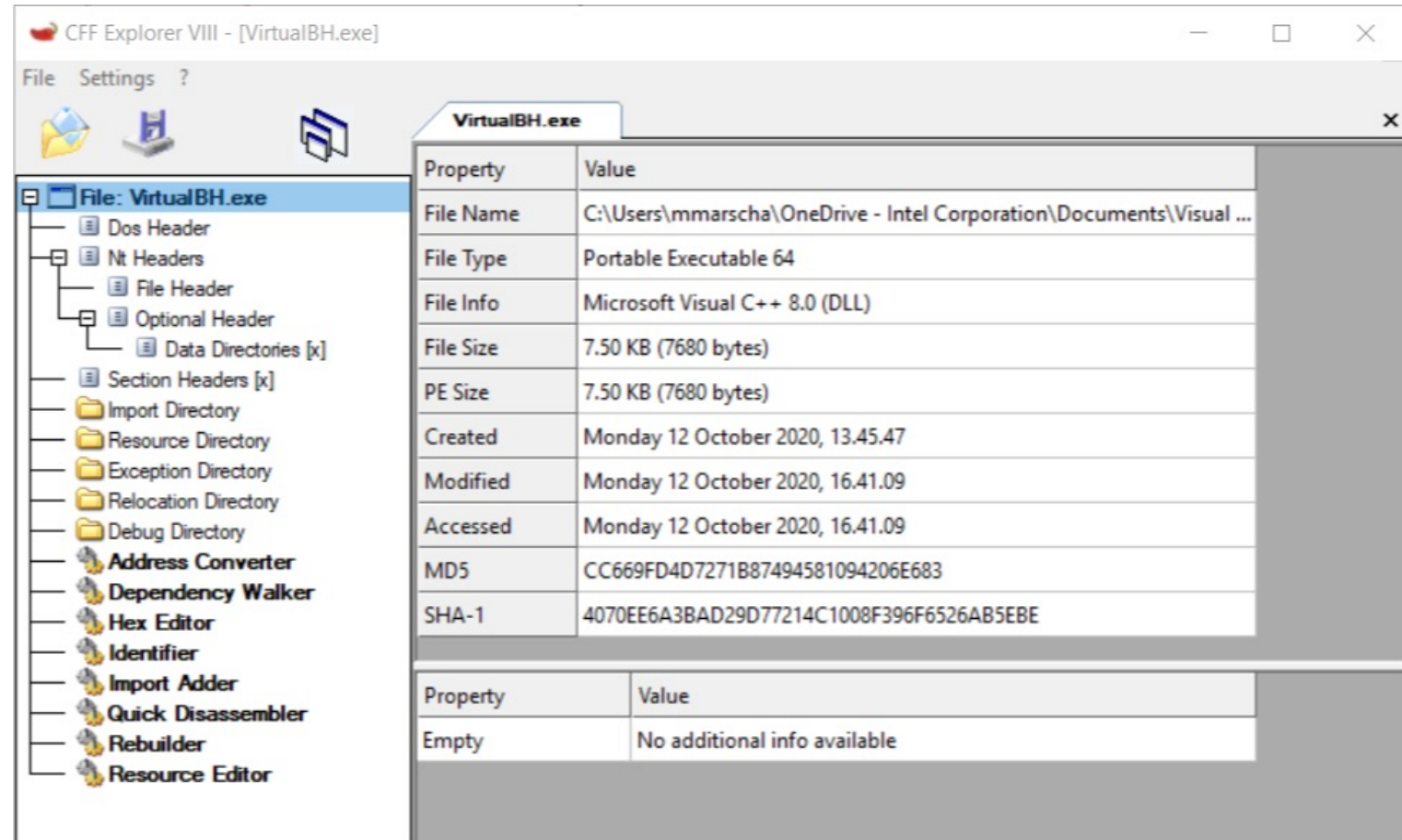
Executable File Formats

Portable Executable (PE)

Executable and Linkable
Format (ELF)

TL;DR Loader fetches
executable, maps it into
virtual memory space

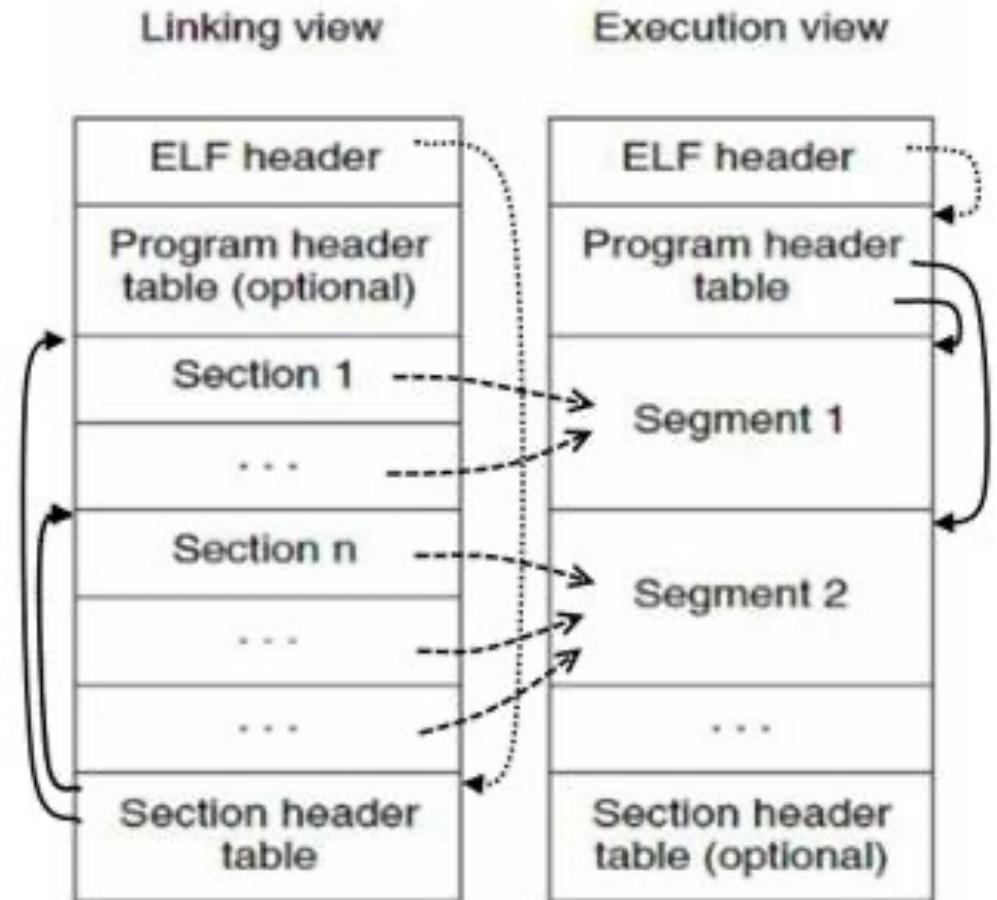
Entry point, sections,
imports/exports, many
many many other things =>
executable file header






































ELF – The Executable & Linkable Format

Standard file format for executable files, object code and shared libraries
Flexible, extensible, cross-platform
=> has grown complex over time

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
 debug024	00000014CBDD6000	00000014CBE00000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug005	00000014CBE00000	00000014CBEF9000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 Stack_PAGE_GUARD[00006...	00000014CBEF9000	00000014CBEFC000	R	W	.	D	.	byte	0000	public	STACK	64	0000	0000	0000	0000	0000
 Stack[000066A4]	00000014CBEFC000	00000014CBF00000	R	W	.	D	.	byte	0000	public	STACK	64	0000	0000	0000	0000	0000
 debug025	00000014CBF00000	00000014CBFFC000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug026	00000014CBFFC000	00000014CBFFF000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug027	00000014CBFFF000	00000014CC000000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug020	000001FF20E40000	000001FF20E50000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug007	000001FF20E60000	000001FF20E7A000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug008	000001FF20E80000	000001FF20E84000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug009	000001FF20E90000	000001FF20E91000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug010	000001FF20EA0000	000001FF20EA2000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 locale.nls	000001FF20EB0000	000001FF20F75000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug030	000001FF20FD0000	000001FF20FE1000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug031	000001FF20FE1000	000001FF210D0000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug022	00007FF4A94C0000	00007FF4A94C5000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug023	00007FF4A94C5000	00007FF4A95C0000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug017	00007FF4A95C0000	00007FF5A95E0000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug018	00007FF5A95E0000	00007FF5AB5E0000	?	?	?	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug019	00007FF5AB5E0000	00007FF5AB5E1000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 debug011	00007FF5AB5F0000	00007FF5AB5F1000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 debug012	00007FF5AB600000	00007FF5AB623000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 VirtualBH.exe	00007FF61BB80000	00007FF61BB81000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 .text	00007FF61BB81000	00007FF61BB82000	R	.	X	.	L	para	0001	public	CODE	64	0000	0000	0003	FFFFFFFF...	FFFFFFFF...
 .idata	00007FF61BB82000	00007FF61BB82100	R	.	.	.	L	para	0005	public	DATA	64	0000	0000	0003	FFFFFFFF...	FFFFFFFF...
 .rdata	00007FF61BB82100	00007FF61BB83000	R	.	.	.	L	para	0002	public	DATA	64	0000	0000	0003	FFFFFFFF...	FFFFFFFF...
 .data	00007FF61BB83000	00007FF61BB84000	R	W	.	.	L	para	0003	public	DATA	64	0000	0000	0003	FFFFFFFF...	FFFFFFFF...
 .pdata	00007FF61BB84000	00007FF61BB85000	R	.	.	.	L	para	0004	public	DATA	64	0000	0000	0003	FFFFFFFF...	FFFFFFFF...
 VirtualBH.exe	00007FF61BB85000	00007FF61BB87000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF910000	00007FFCBF911000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF911000	00007FFCBF9A3000	R	.	X	D	.	byte	0000	public	CODE	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF9A3000	00007FFCBF9D1000	R	.	.	D	.	byte	0000	public	CONST	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF9D1000	00007FFCBF9D3000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF9D3000	00007FFCBF9D4000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000
 MSVCR110.dll	00007FFCBF9D4000	00007FFCBF9D7000	R	W	.	D	.	byte	0000	public	DATA	64	0000	0000	0000	0000	0000

Memory Layout

Of a Windows process

Memory Manipulation in Assembly

Instruction (+ Operand)

Immediate values

e.g. `mov r8d, 6Eh`

Registers

e.g. `mov rcx, rbx`

Memory addresses

Virtual address

`[0x7FF6AB373000]`

e.g. `mov rax,`

Pointer dereferentiation

e.g. `mov rax, [rax]`

Calculation

e.g. `mov rax, [rbx+rdx+10h]`

Heap

Memory that can be dynamically allocated (eg. malloc)

One heap per process

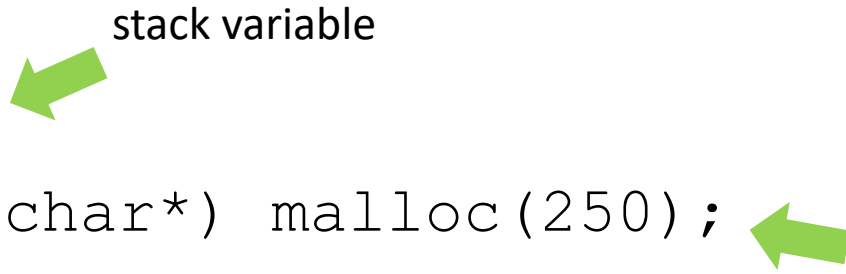
Operating system provides functions for memory de-/allocation

Threads need to synchronize memory de-/allocation

```
...  
int a = 25;  
char* b = (char*) malloc(250);  
...
```

stack variable

heap buffer



Stack

One stack per thread

PUSH and POP instructions interact with the stack, direct access of stack variables common too

Operating system provides stack memory block, and guard page

It's a 64 bit wide LIFO buffer (64b systems.., 32 on 32 bit)

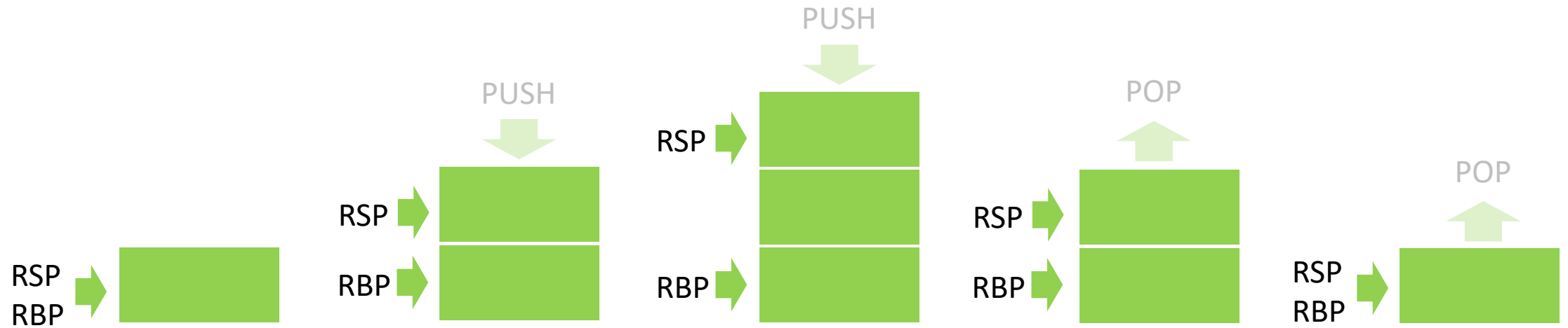
RSP and RBP manage function frame on stack

Abstract Depiction of Stack

RBP = Base Pointer

RSP = Stack Pointer

PUSH/POP increment/decrement RSP



Calling Conventions and ABIs

Application Binary Interface – defines how functions are called (parameter handling, stack usage, special register designation)

History: “convention madness” on Windows x86

cdecl, stdcall, fastcall, thiscall, safecall,....

Calling conventions define:

- How arguments are passed to the callee

- Which registers may be used, which must be saved before use

- How function stack is set up and torn down after callee is done

Little Less Abstract Depiction of Stack

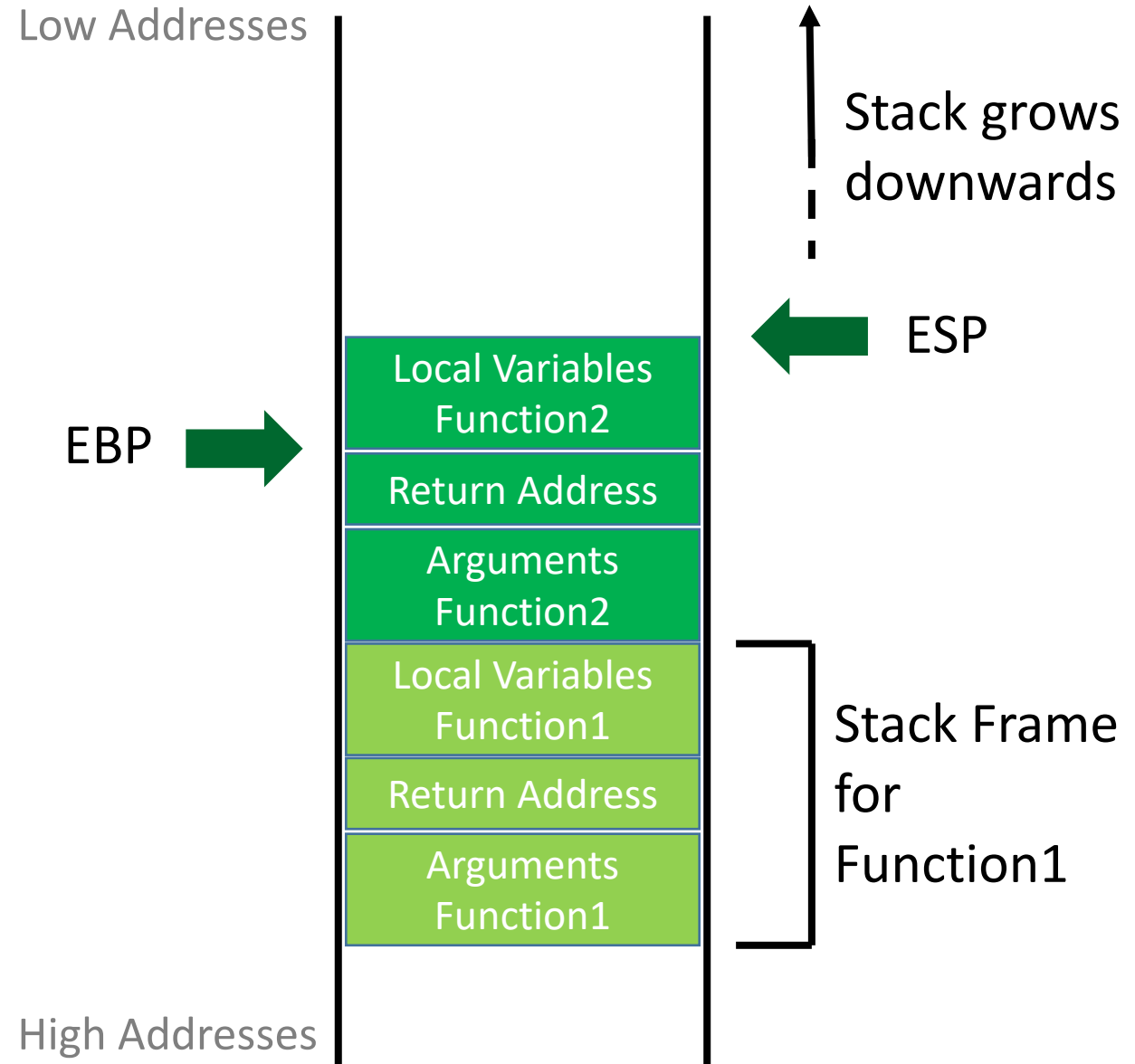
The classic: 32-bit cdecl calling convention

Caller pushes function arguments, in reverse order

CALL instruction pushes return address

Callee sets up local stack variables

Return: Stack cleanup & jump to stored EIP



var_8	= dword ptr	-8
var_4	= dword ptr	-4
arg_0	= dword ptr	8
arg_4	= dword ptr	0Ch

Arguments

BP/SP often serve to dereference arguments & variables

```
push    ebp
mov     ebp, esp
sub     esp, 8
```

Prologue

Argument
for sub_401050



```
mov     [ebp+var_4], 14h
mov     eax, [ebp+arg_0]
push    eax
call    sub_401050
```

```
add     esp, 4
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
add     eax, [ebp+arg_4]
```

Return value in EAX



```
mov     esp, ebp
pop     ebp
retn
```

Stack cleanup



Epilogue

System V 64bit Calling Convention

Function Parameters:

RDI, RSI, RDX, RCX, R8, R9, or stack in reverse order

RAX will hold return value, or RAX and RDX if 128-bit value

https://wiki.osdev.org/System_V_ABI

Other x64 “Eastereggs”

Vararg functions

Tailcall optimization

Creative home space utilization

RIP-relative addressing

Struct arguments

Fancy processor extensions

Alignment ftw...

Etc. etc. etc. lots of things I’m not aware of myself ;)

Registers

Memory

Instructions

– Data transfer

- MOV
- PUSH/POP
- PUSHA/POPA
- PUSHF/POPF
- XCHG

– Control transfer

- JMP
- Jxx
- LOOP
- CALL
- LEAVE
- RET

– Logical instructions

- AND
- TEST
- OR
- XOR
- NOT

– Arithmetic instructions

- ADD/SUB
- CMP
- INC/DEC
- MUL/DIV
- NEG

– Other instructions

- NOP
- LEA
- RDTSC
- SYSCALL/SYSENTER
- SYSRET/SYSEXIT
- IN/OUT
- ROL/ROR
- SHL/SHR
- SETxx
- LODS
- STOS
- MOVS

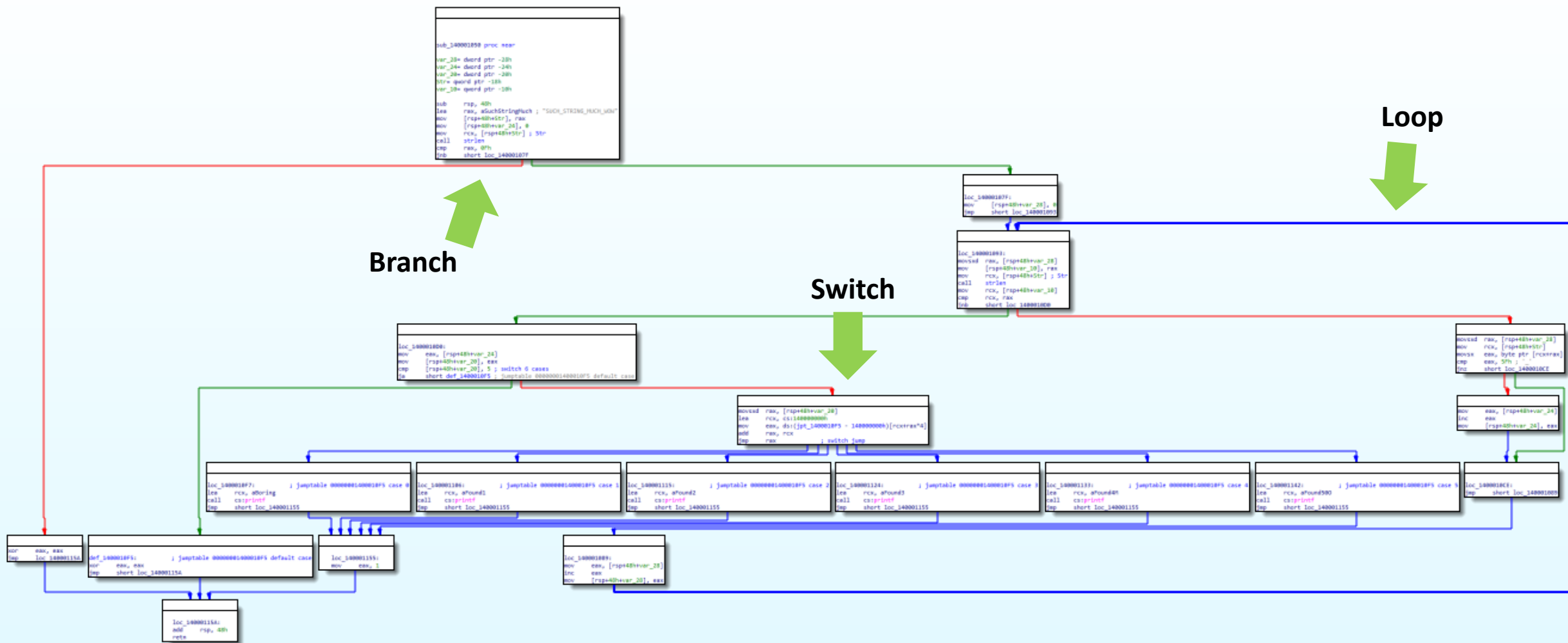
**And many MANY more
Intel SDM, Volume 2 ;)**

Jump variations

Instruction	Jump if/on ...	Flag conditions
JE	equal	ZF = 1
JG	greater (signed)	ZF = 0 and SF = OF
JGE	greater or equal (signed)	SF = OF
JL	less (signed)	SF != OF
JLE	less or equal (signed)	ZF = 1 or SF != OF
JA	above (unsigned)	CF = 0 and ZF = 0
JAЕ	above or equal (unsigned)	CF = 0
JB	below (unsigned)	CF = 1
JBE	below or equal (unsigned)	CF = 1 or ZF = 1
JC	carry	CF = 1
JO	overflow	OF = 1
JP	parity	PF = 1
JS	sign	SF = 1
JZ	zero	ZF = 1

Instruction	Jump if/on ...	Flag conditions
JNE	not equal	ZF = 0
JNG	not greater (signed)	ZF = 1 or SF != OF
JNGE	not greater or equal (signed)	SF != OF
JNL	not less (signed)	SF = OF
JNLE	not less or equal (signed)	ZF = 0 and SF = OF
JNA	not above (unsigned)	CF = 1 or ZF = 1
JNAE	not above or equal (unsigned)	CF = 1
JNB	not below (unsigned)	CF = 0
JNBE	not below or equal (unsigned)	CF = 0 and ZF = 0
JNC	not carry	CF = 0
JNO	not overflow	OF = 0
JNP	not parity	PF = 0
JNS	not sign	SF = 0
JNZ	not zero	ZF = 0

Control Flow

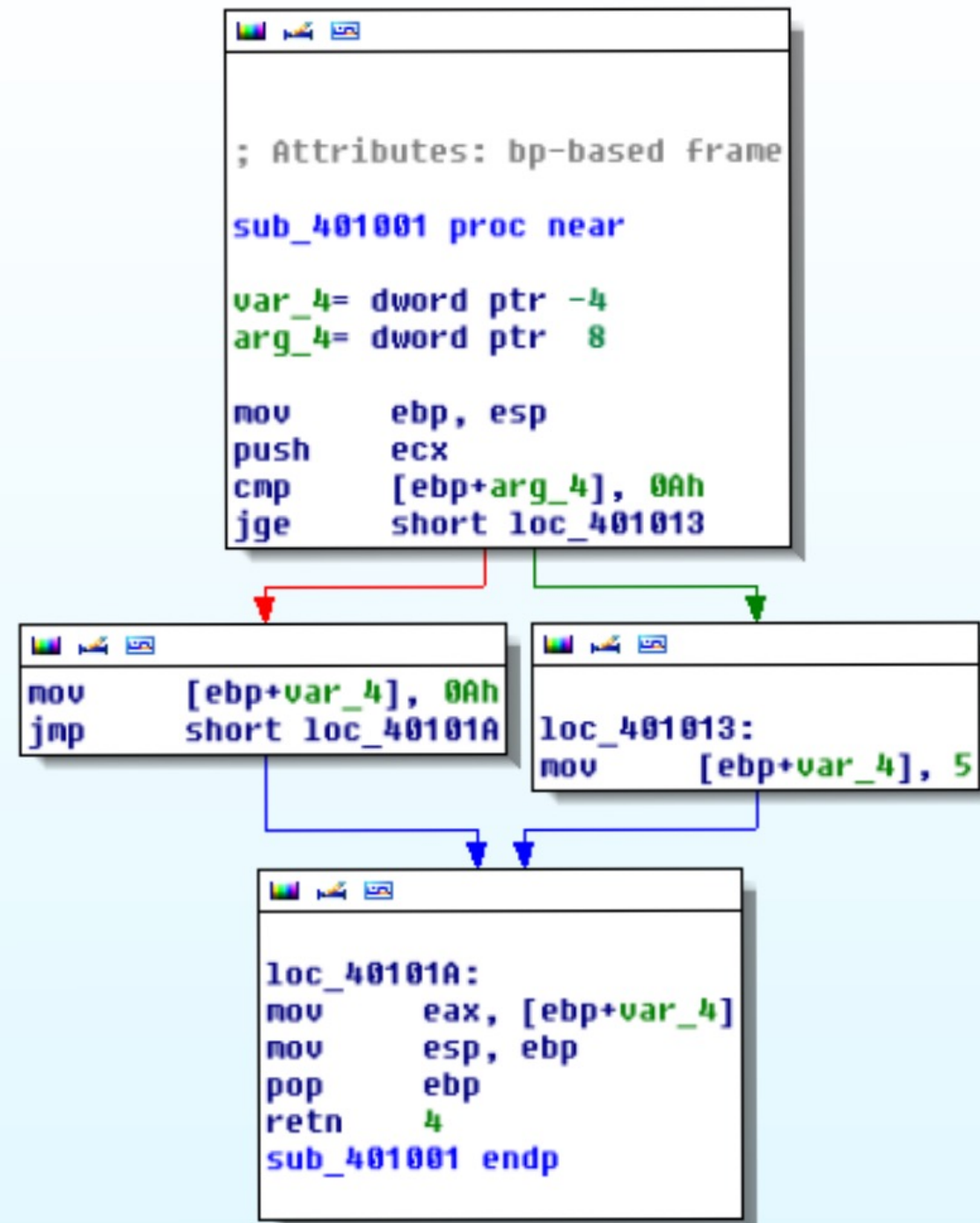


Branches

```
int show_branch(int val)
{
    int ret;

    if (val < 10)
        ret = 10;
    else
        ret = 5;

    return ret;
}
```

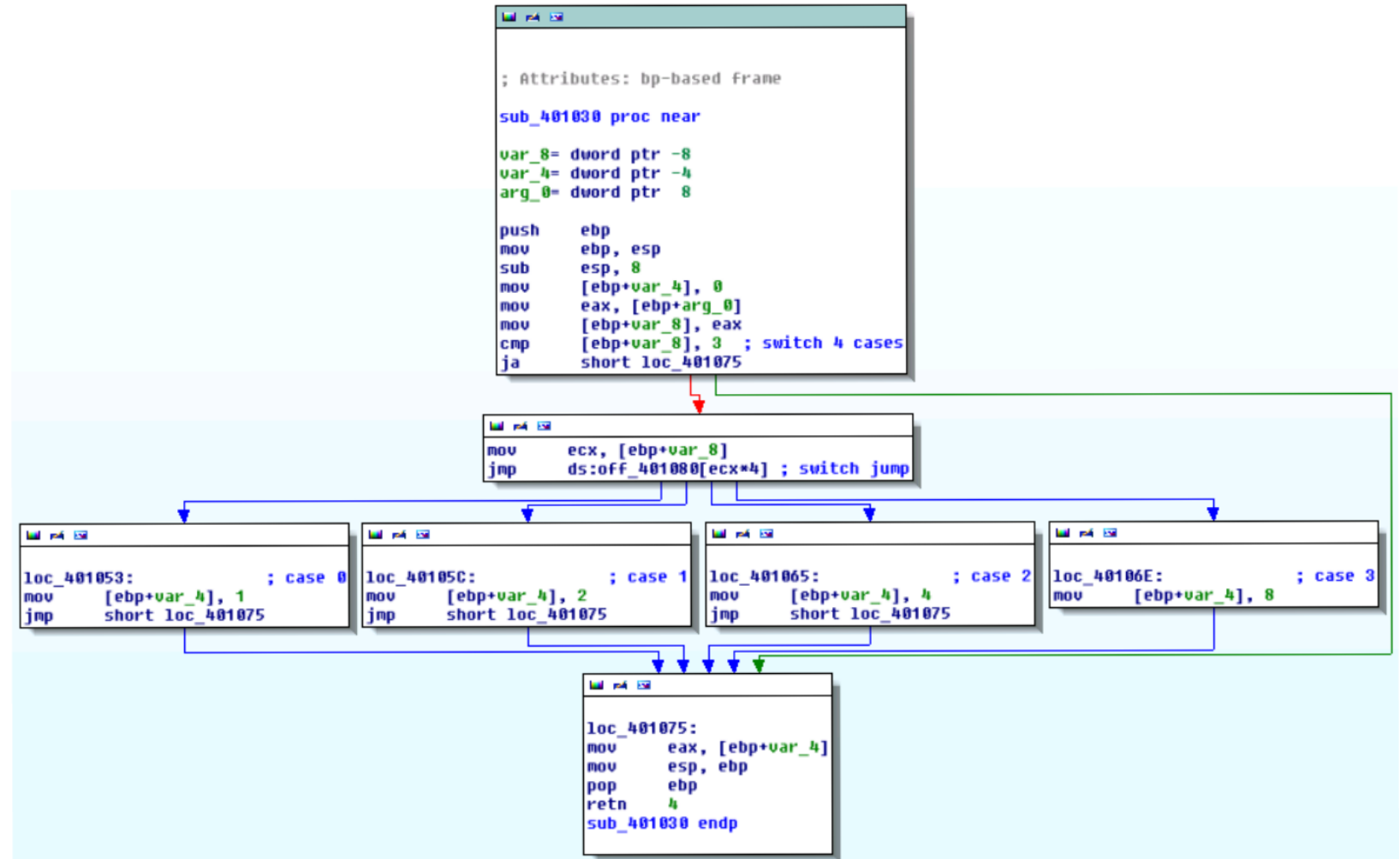


Switch

```
int multibranch(int val)
{
    int ret = 0;

    switch (val)
    {
        case 0:
            ret = 1; break;
        case 1:
            ret = 2; break;
        case 2:
            ret = 4; break;
        case 3:
            ret = 8; break;
    }

    return ret;
}
```



```
.text:0040107E
.text:00401080 off_401080
.text:00401080
.text:00401080
.text:00401080
```

```
align 10h
dd offset loc_401053 ; DATA XREF: sub_401030+1C↑r
dd offset loc_40105C ; jump table for switch statement
dd offset loc_401065
dd offset loc_40106E
```

Loops

```
int show_loop(int val)
{
    int ret = 0;

    for (; val > 0; val--)
        ret += val;

    return ret;
}
```

