Introduction to x86-64 Reverse Engineering

Marion Marschalek | @pinkflawd | @blackhoodie_re

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

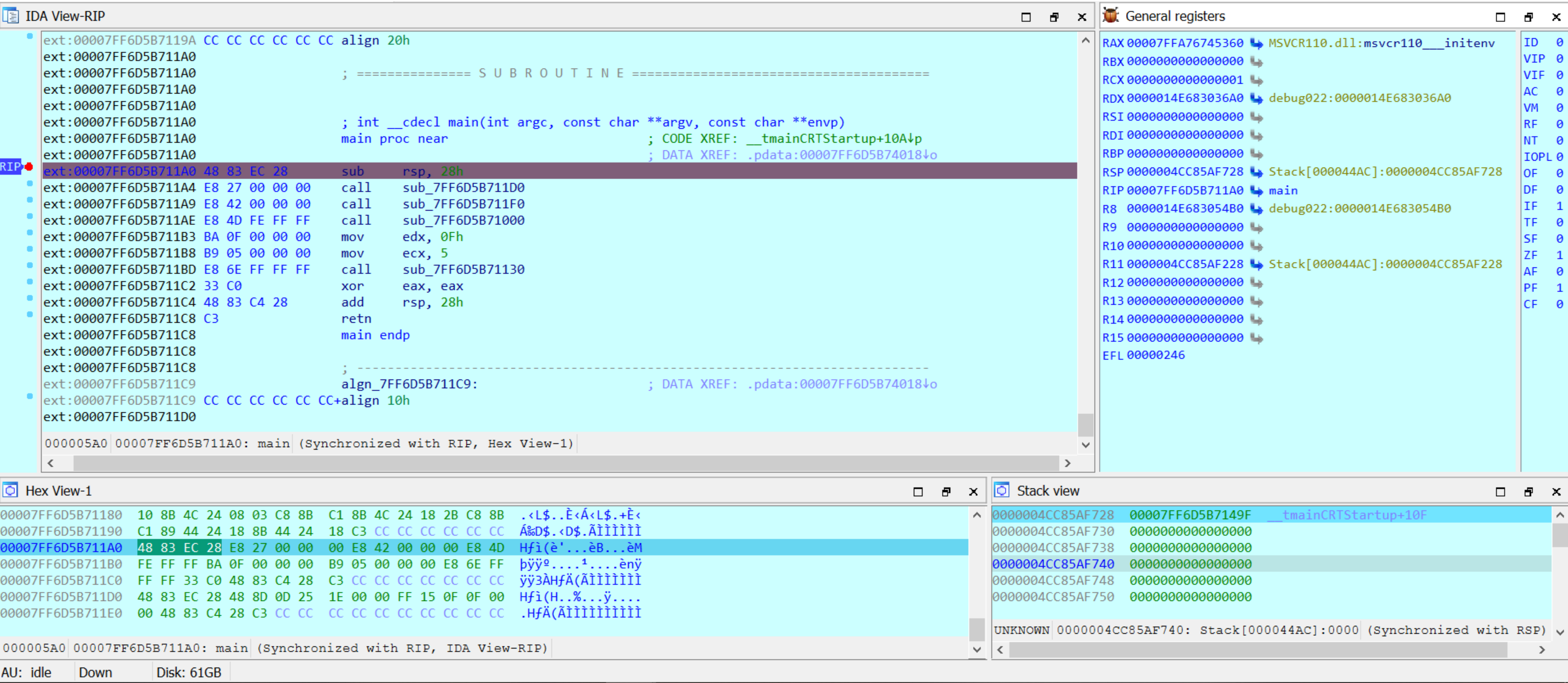
Prepare components (ALU, memory, I/O)

Execute instruction

Store results

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

Program counter, byte code, assembly, sry wot?



```
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
ext:00007FF6D5B711A0                    ; int __cdecl main(int argc, const char **argv, const char **envp)
ext:00007FF6D5B711A0                    main proc near                          ; CODE XREF: __tmainCRTStartup+10A↓p
ext:00007FF6D5B711A0                                                            ; DATA XREF: .pdata:00007FF6D5B74018↓o
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:00007FF6D5B711C8                    ; ---------------------------------------------------------------------------
ext:00007FF6D5B711C9                    algn_7FF6D5B711C9:                      ; DATA XREF: .pdata:00007FF6D5B74018↓o
ext:00007FF6D5B711C9 CC CC CC CC CC CC+align 10h
ext:00007FF6D5B711D0
000005A0 00007FF6D5B711A0: main (Synchronized with RIP, Hex View-1)
```

```
RAX 00007FFA76745360 ↳ MSVCR110.dll:msvcr110___initenv     ID   0
RBX 0000000000000000 ↳                                      VIP  0
RCX 0000000000000001 ↳                                      VIF  0
RDX 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
```

```
00007FF6D5B71180  10 8B 4C 24 08 03 C8 8B  C1 8B 4C 24 18 2B C8 8B  .‹L$..È‹Á‹L$.+È‹
00007FF6D5B71190  C1 89 44 24 18 8B 44 24  18 C3 CC CC CC CC CC CC  Á‰D$.‹D$.ÃÌÌÌÌÌÌ
00007FF6D5B711A0  48 83 EC 28 E8 27 00 00  00 E8 42 00 00 00 E8 4D  Hƒì(è'...èB...èM
00007FF6D5B711B0  FE FF FF BA 0F 00 00 00  B9 05 00 00 00 E8 6E FF  þÿÿº....¹....ènÿ
00007FF6D5B711C0  FF FF 33 C0 48 83 C4 28  C3 CC CC CC CC CC CC CC  ÿÿ3ÀHƒÄ(ÃÌÌÌÌÌÌÌ
00007FF6D5B711D0  48 83 EC 28 48 8D 0D 25  1E 00 00 FF 15 0F 0F 00  Hƒì(H..%...ÿ....
00007FF6D5B711E0  00 48 83 C4 28 C3 CC CC  CC CC CC CC CC CC CC CC  .HƒÄ(ÃÌÌÌÌÌÌÌÌÌÌ
000005A0 00007FF6D5B711A0: main (Synchronized with RIP, IDA View-RIP)
```

```
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
```

Fraunhofer

FKIE

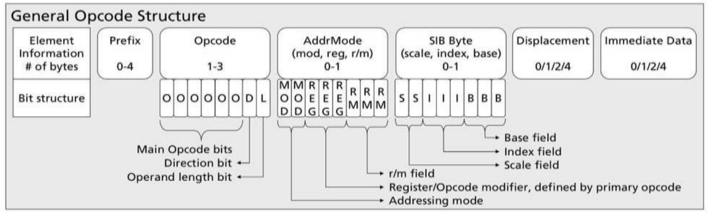
FRAUNHOFER-INSTITUT FÜR KOMMUNIKATION, INFORMATIONSVERARBEITUNG UND ERGONOMIE FKIE

x86 Opcode Structure and Instruction Overview

2nd 1st	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
1	ADC								SBB							POP DS
2	AND						ES SEGMENT OVERRIDE SS	DAA	SUB						CS SEGMENT OVERRIDE DS	DAS
3	XOR							AAA	CMP							AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS	GS	OPERAND SIZE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
					SEGMENT OVERRIDE		SIZE OVERRIDE									
7	JO	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
	Jcc															
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	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		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							
	ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR															
E	LOOPNZ	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
	CONDITIONAL LOOP															
F	LOCK	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
	EXCLUSIVE ACCESS		CONDITIONAL REPETITION													

2nd 1st	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	{L,S}LDT {L,S}TR 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	RDTSC	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	MOVSX	
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		



General Opcode Structure

Element Information # of bytes	Prefix 0-4	Opcode 1-3	AddrMode (mod, reg, r/m) 0-1	SIB Byte (scale, index, base) 0-1	Displacement 0/1/2/4	Immediate Data 0/1/2/4
Bit structure		O O O O O O D L	MOD MOD REG REG REG RM RM RM	S S I I I B B B		

Addressing Modes

mod	00		01		10		11
r/m	16bit	32bit	16bit	32bit	16bit	32bit	r/m // REG
000	[BX+SI]	[EAX]	[BX+SI]+disp8	[EAX]+disp8	[BX+SI]+disp16	[EAX]+disp32	AL / AX / EAX
001	[BX+DI]	[ECX]	[BX+DI]+disp8	[ECX]+disp8	[BX+DI]+disp16	[ECX]+disp32	CL / CX / ECX
010	[BP+SI]	[EDX]	[BP+SI]+disp8	[EDX]+disp8	[BP+SI]+disp16	[EDX]+disp32	DL / DX / EDX
011	[BP+DI]	[EBX]	[BP+DI]+disp8	[EBX]+disp8	[BP+DI]+disp16	[EBX]+disp32	BL / BX / EBX
100	[SI]	*SIB*	[SI]+disp8	*SIB*+disp8	[SI]+disp16	*SIB*+disp32	AH / SP / ESP
101	[DI]	disp32	[DI]+disp8	[EBP]+disp8	[DI]+disp16	[EBP]+disp32	CH / BP / EBP
110	disp16	[ESI]	[BP]+disp8	[ESI]+disp8	[BP]+disp16	[ESI]+disp32	DH / SI / ESI
111	[BX]	[EDI]	[BX]+disp8	[EDI]+disp8	[BX]+disp16	[EDI]+disp32	BH / DI / EDI

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 / disp8+ [EBP] / disp32 + [EBP]
110	--	[ESI]	ESI
111	--	[EDI]	EDI
SIB value = index * scale + base			

v1.0 – 30.08.2011

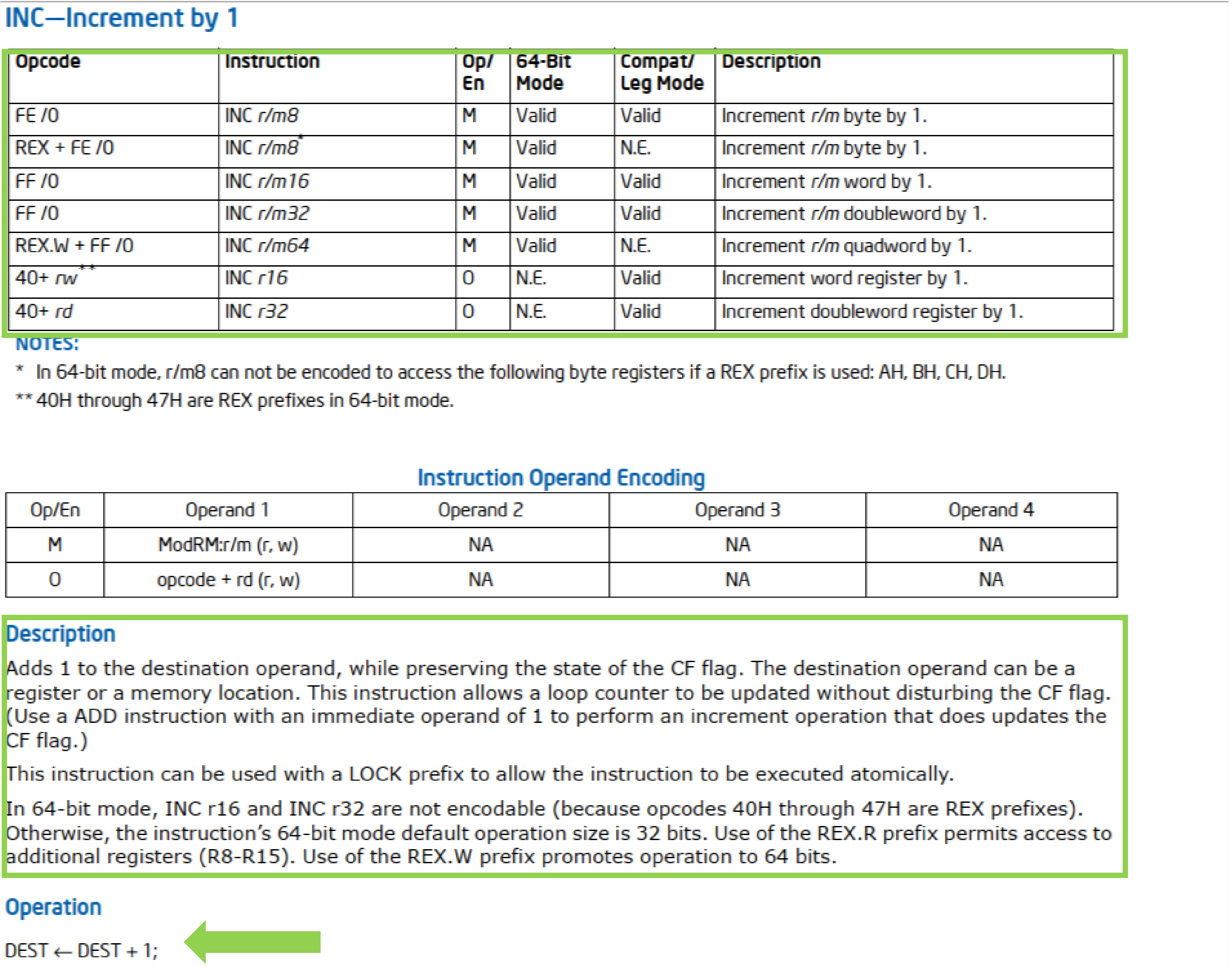
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference

Opcode table presentation inspired by work of Ange Albertini

Intel SDM

INC instruction



INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC *r/m8*	M	Valid	Valid	Increment *r/m* byte by 1.
REX + FE /0	INC r/m8*	M	Valid	N.E.	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	M	Valid	Valid	Increment *r/m* word by 1.
FF /0	INC *r/m32*	M	Valid	Valid	Increment *r/m* doubleword by 1.
REX.W + FF /0	INC *r/m64*	M	Valid	N.E.	Increment *r/m* quadword by 1.
40+ rw**	INC *r16*	O	N.E.	Valid	Increment word register by 1.
40+ *rd*	INC *r32*	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:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	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;
```

Registers

Memory

Instructions

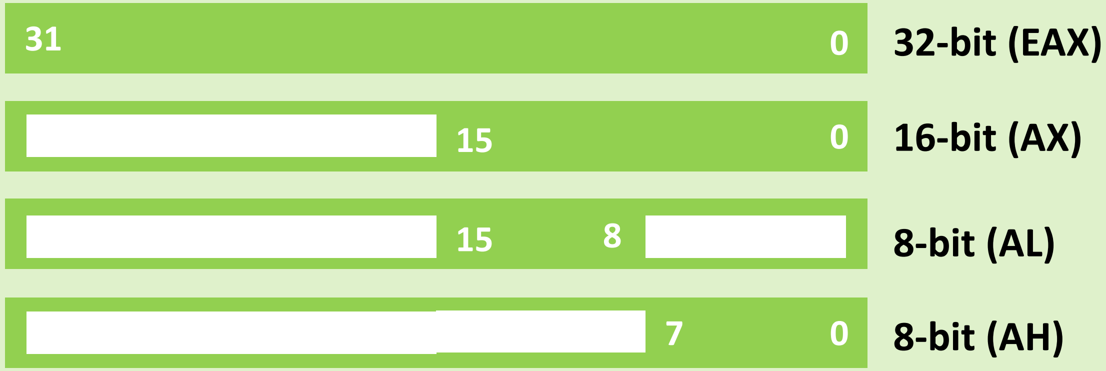
General Purpose Registers

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
RBX
RCX
RDX
RSI
RDI
RBP
RSP

R8
R9
R10
R11
R12
R13
R14
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

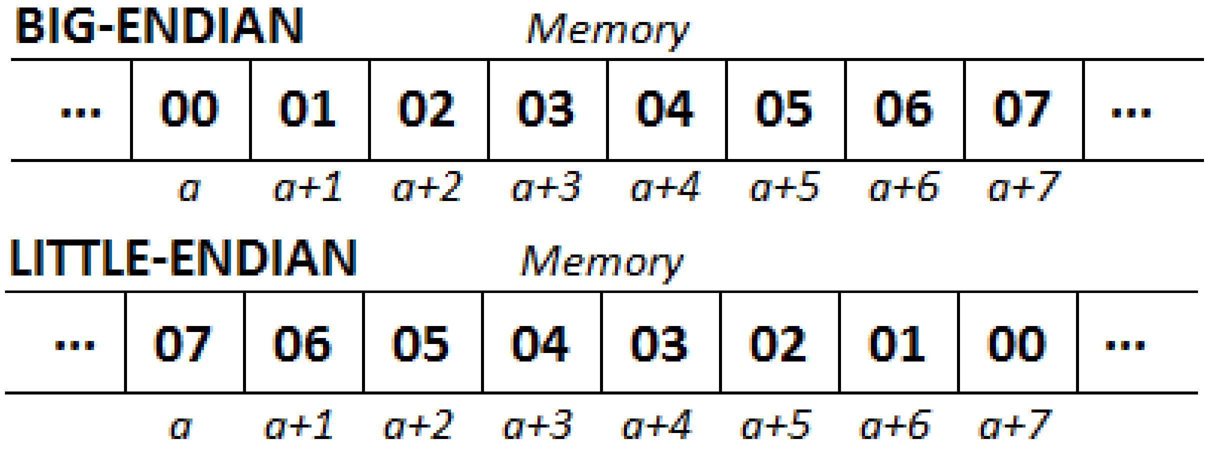
Instructions

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*	

https://thebittheories.com/little-endian-vs-big-endian-b4046c63e1f2

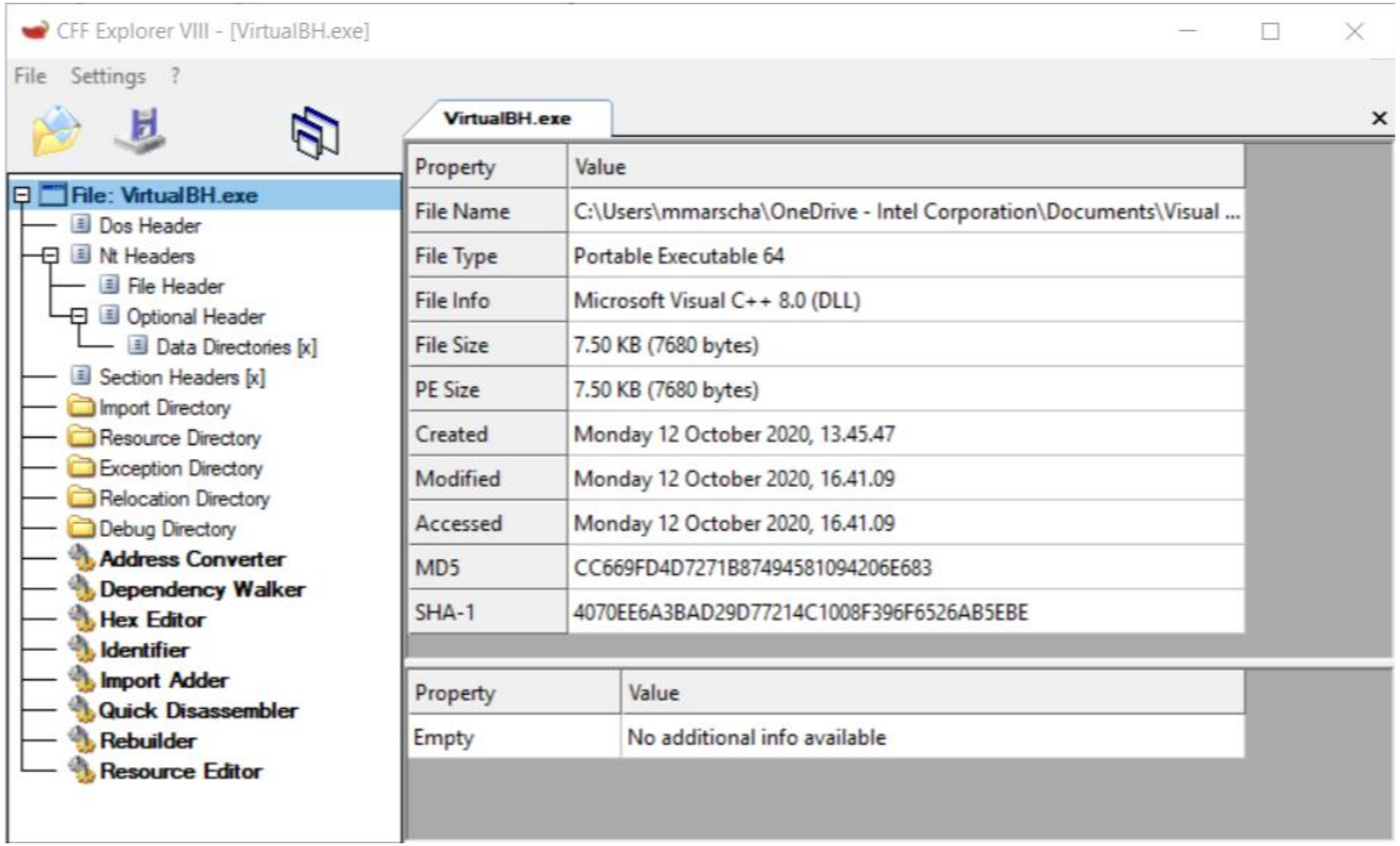
Executable File Formats

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**

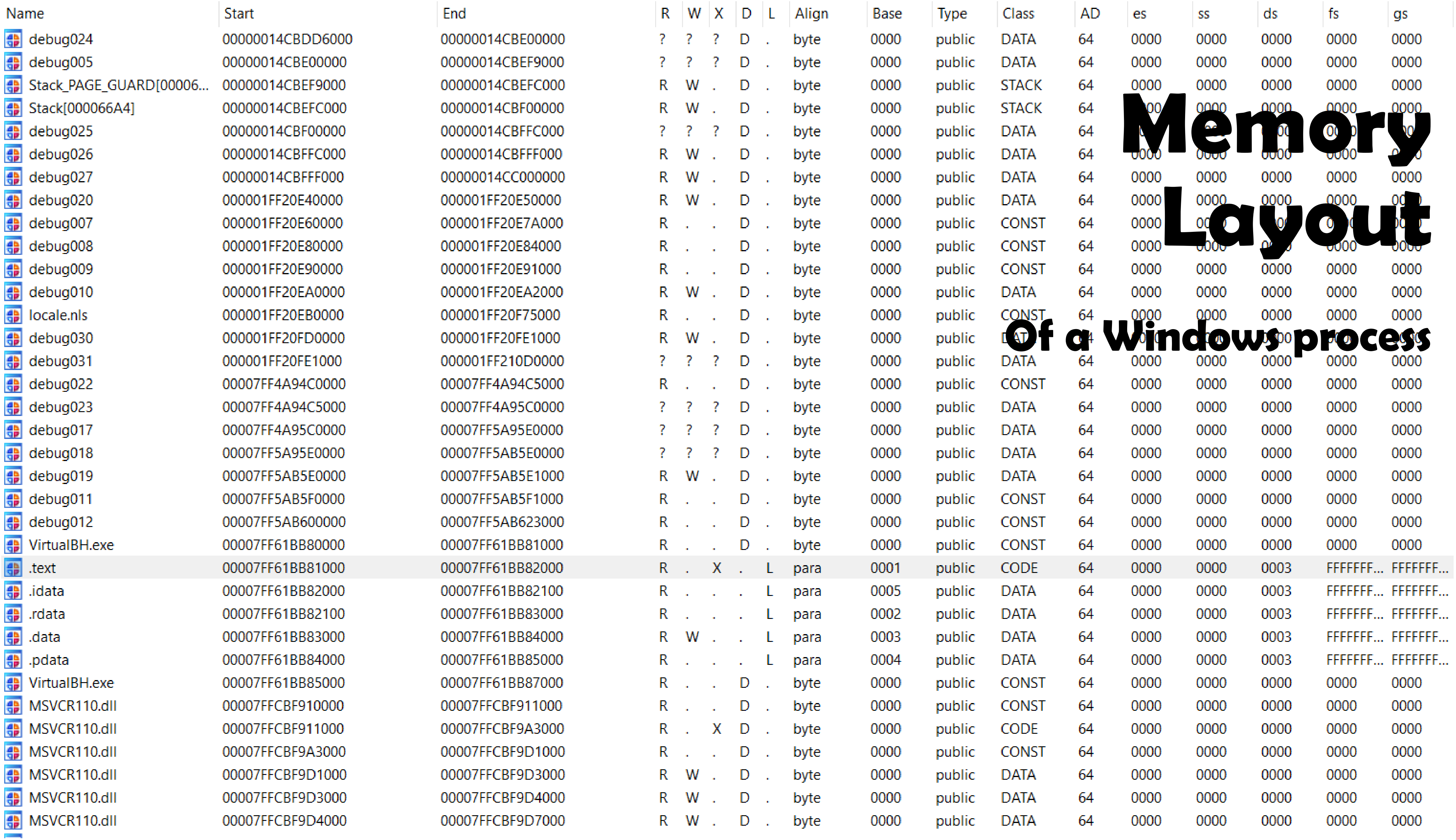


Property	Value
File Name	C:\Users\mmarscha\OneDrive - Intel Corporation\Documents\Visual ...
File Type	Portable Executable 64
File Info	Microsoft Visual C++ 8.0 (DLL)
File Size	7.50 KB (7680 bytes)
PE Size	7.50 KB (7680 bytes)
Created	Monday 12 October 2020, 13.45.47
Modified	Monday 12 October 2020, 16.41.09
Accessed	Monday 12 October 2020, 16.41.09
MD5	CC669FD4D7271B87494581094206E683
SHA-1	4070EE6A3BAD29D77214C1008F396F6526AB5EBE

Property	Value
Empty	No additional info available

Memory Layout

Of a Windows process



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	FFFFFFF...	FFFFFFF...
.idata	00007FF61BB82000	00007FF61BB82100	R	.	.	.	L	para	0005	public	DATA	64	0000	0000	0003	FFFFFFF...	FFFFFFF...
.rdata	00007FF61BB82100	00007FF61BB83000	R	.	.	.	L	para	0002	public	DATA	64	0000	0000	0003	FFFFFFF...	FFFFFFF...
.data	00007FF61BB83000	00007FF61BB84000	R	W	.	.	L	para	0003	public	DATA	64	0000	0000	0003	FFFFFFF...	FFFFFFF...
.pdata	00007FF61BB84000	00007FF61BB85000	R	.	.	.	L	para	0004	public	DATA	64	0000	0000	0003	FFFFFFF...	FFFFFFF...
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 Manipulation in Assembly

Instruction (+ Operand)

Immediate values	e.g. mov r8d, 6Eh
Registers	e.g. mov rcx, rbx
Memory addresses	
Virtual address	e.g. mov rax, [0x7FF6AB373000]
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

stack variable

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

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

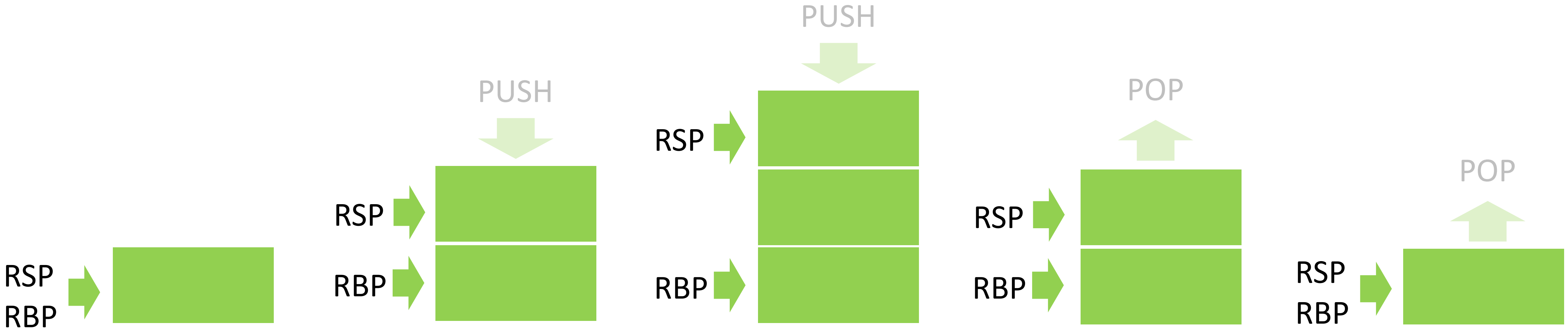
https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

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

https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019

Little Less Abstract Depiction of Stack

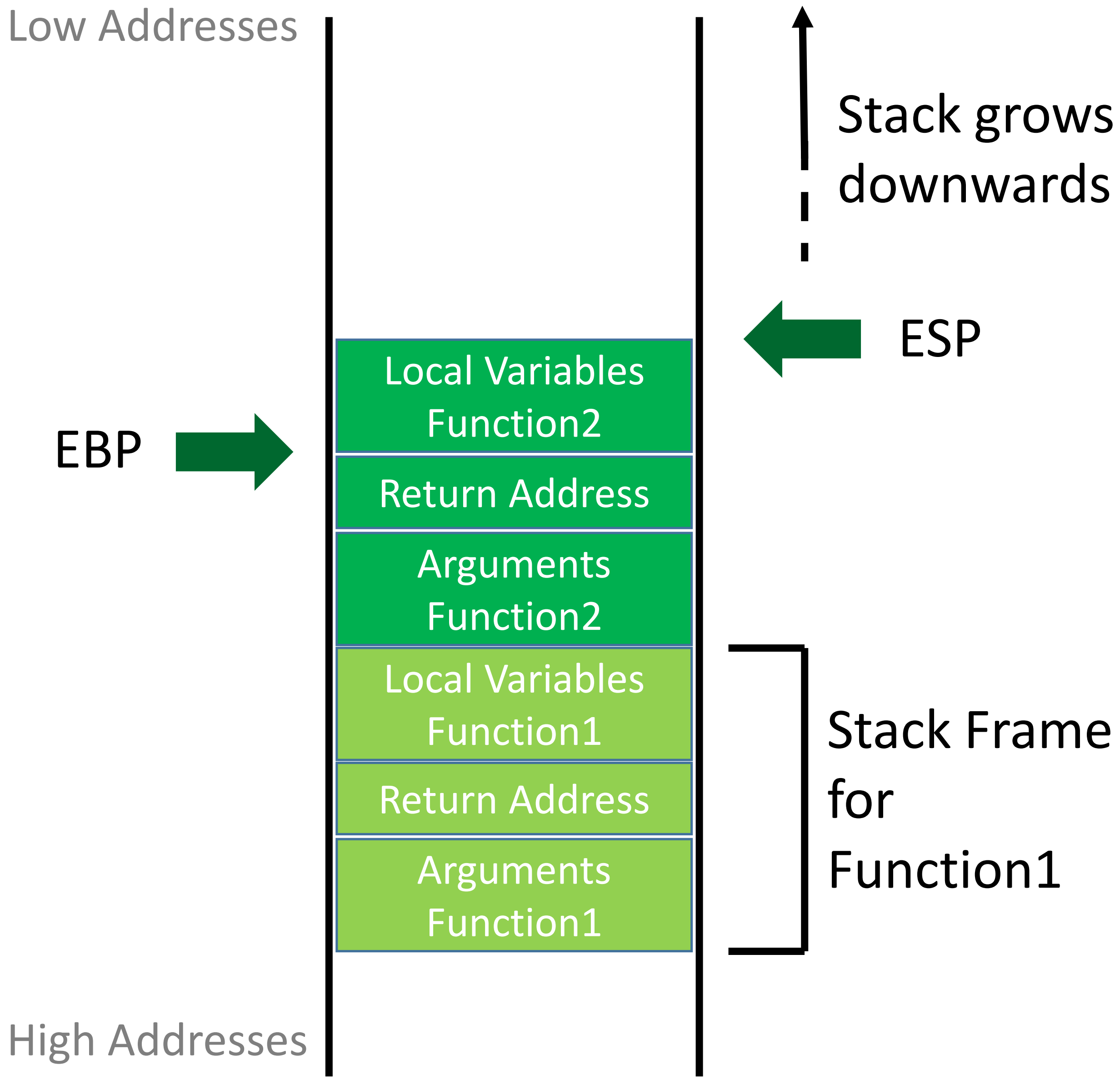
The classic: **32-bit cdecl** calling convention

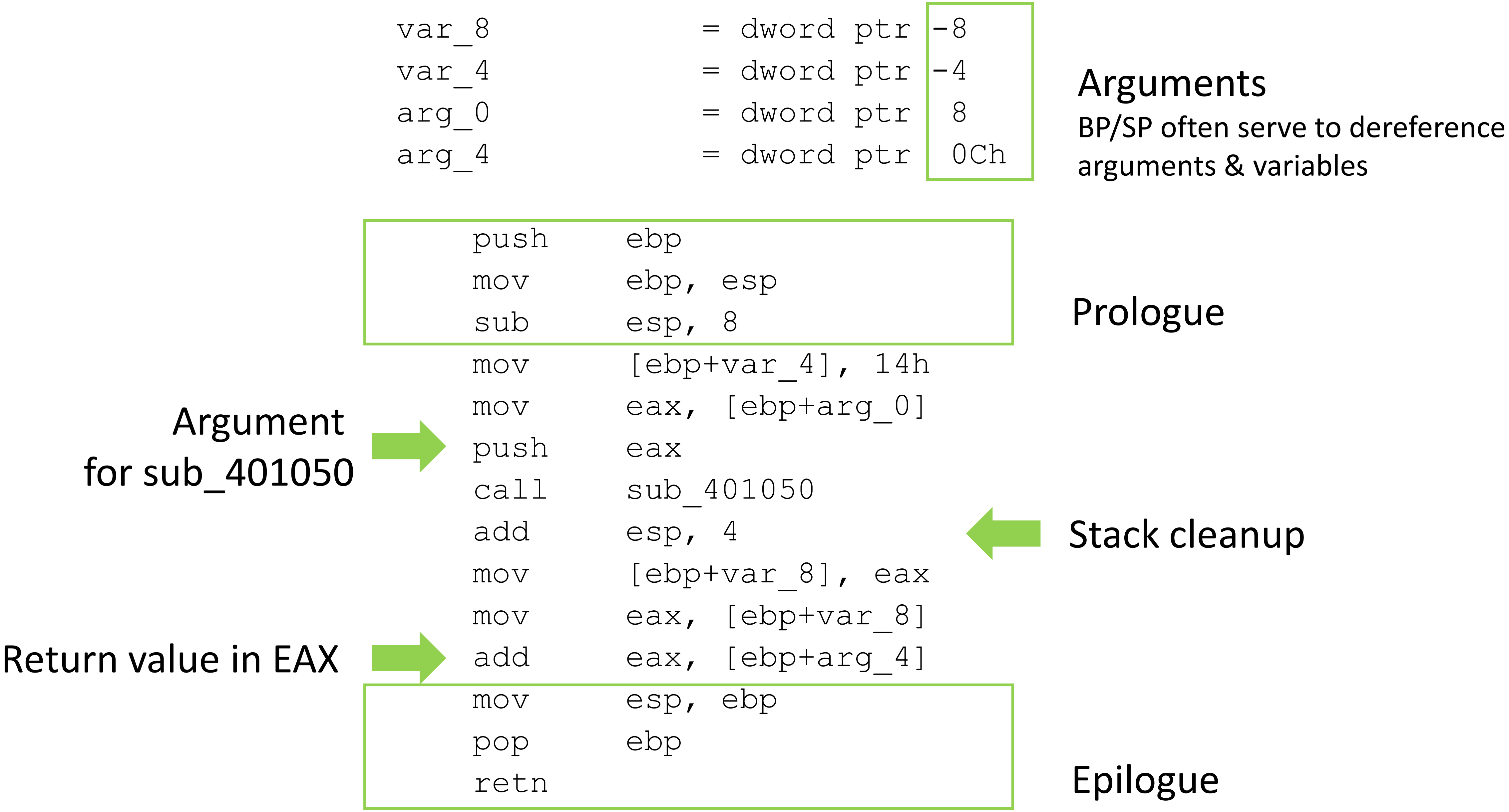
Caller pushes function arguments, in reverse order

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

        push    ebp
        mov     ebp, esp
        sub     esp, 8
        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]
        mov     esp, ebp
        pop     ebp
        retn
```

Arguments

BP/SP often serve to dereference arguments & variables

Prologue

Argument for sub_401050

Stack cleanup

Return value in EAX

Epilogue

Windows x64 Calling Convention & Stack

One convention to rule them all!

First 4 function arguments are handed over via registers

- RCX, RDX, R8, R9 (if integers, in that order!)
- XMM0L, XMM1L, XMM2L, XMM3L (if floating point)
- Arguments 5+ are pushed to stack (in reverse order.. as usual)
- And yet: also "home space" reserved for register arguments on stack

RAX is return register (or XMM0)

RCX will hold "this" pointer for member functions

R10, R11 volatile (as are RAX, RCX, RDX, R8, R9)

R12-R15, RDI, RSI, RBX, RBP, RSP nonvolatile

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

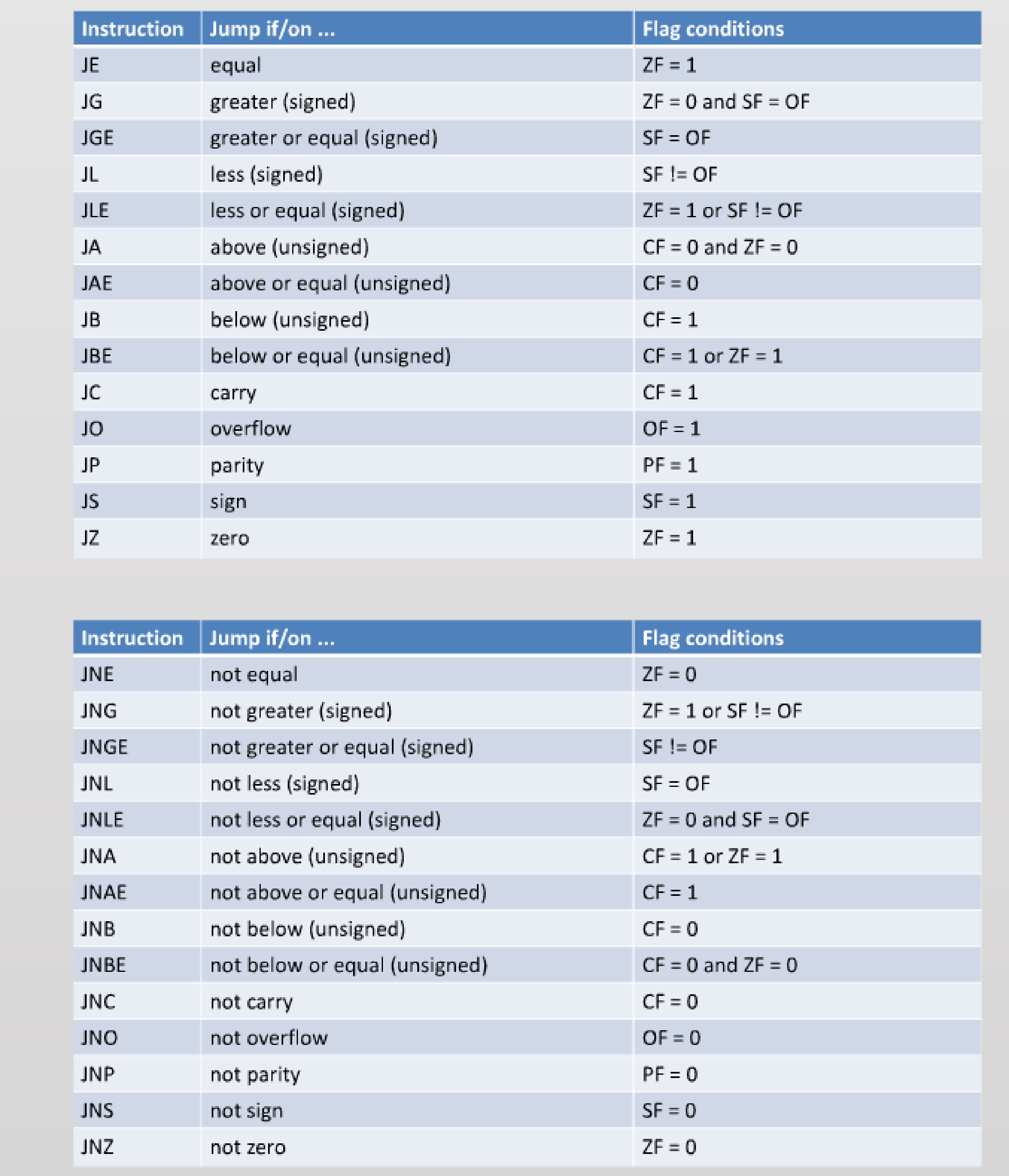
Memory

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 ;)

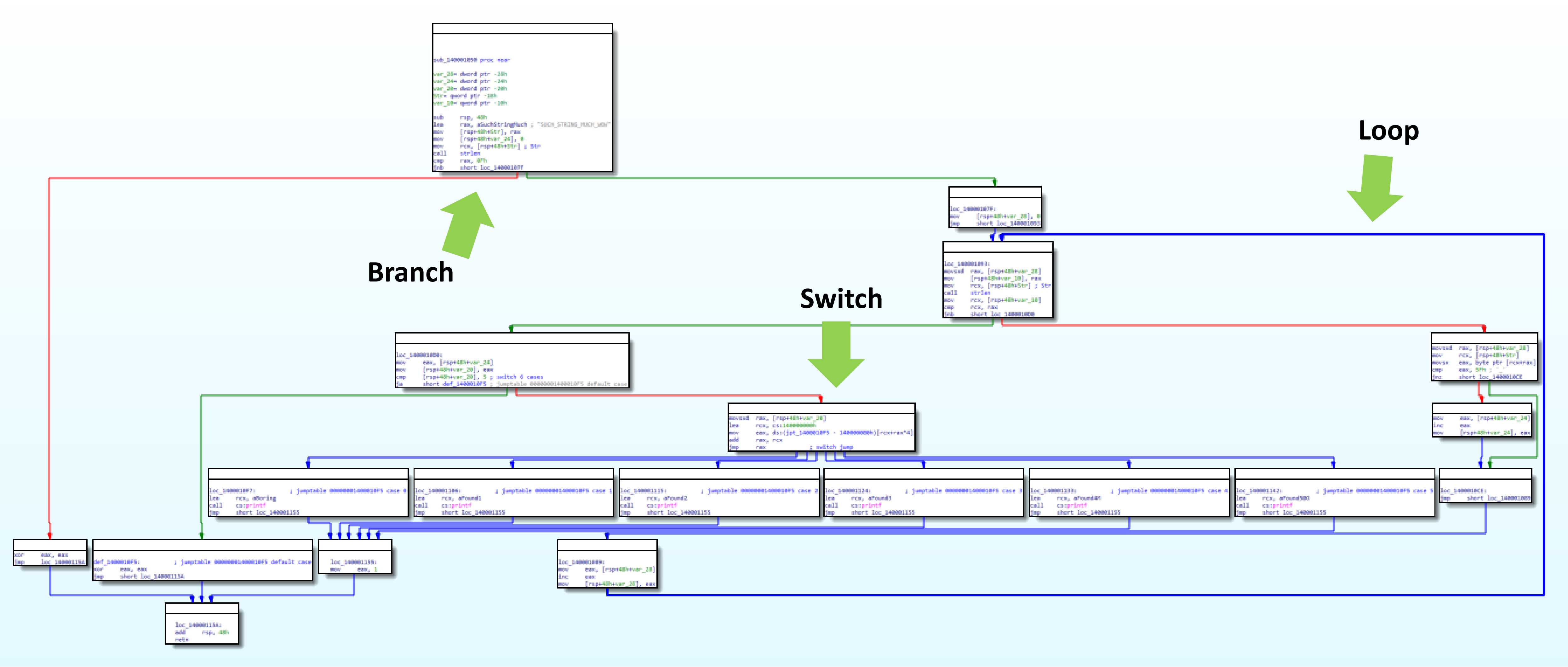
Jump variations



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
JAE	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



Branch

Switch

Loop

```
sub_140001050 proc near

var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
Str= qword ptr -18h
var_10= qword ptr -10h

sub     rsp, 48h
lea     rax, aSuchStringMuch ; "SUCH_STRING_MUCH_WOW"
mov     [rsp+48h+Str], rax
mov     [rsp+48h+var_24], 0
mov     rcx, [rsp+48h+Str] ; Str
call    strlen
cmp     rax, 0Fh
jnb     short loc_14000107F
```

```
loc_14000107F:
mov     [rsp+48h+var_28], 0
jmp     short loc_140001093
```

```
loc_140001093:
movsxd  rax, [rsp+48h+var_28]
mov     [rsp+48h+var_10], rax
mov     rcx, [rsp+48h+Str] ; Str
call    strlen
mov     rcx, [rsp+48h+var_10]
cmp     rcx, rax
jnb     short loc_1400010D0
```

```
movsxd  rax, [rsp+48h+var_28]
mov     rcx, [rsp+48h+Str]
movsx   eax, byte ptr [rcx+rax]
cmp     eax, 5Fh ; '_'
jnz     short loc_1400010CE
```

```
mov     eax, [rsp+48h+var_24]
inc     eax
mov     [rsp+48h+var_24], eax
```

```
loc_1400010CE:
jmp     short loc_1400010B9
```

```
loc_1400010D0:
mov     eax, [rsp+48h+var_24]
mov     [rsp+48h+var_20], eax
cmp     [rsp+48h+var_20], 5 ; switch 6 cases
ja      short def_1400010F5 ; jumptable 00000001400010F5 default case
```

```
movsxd  rax, [rsp+48h+var_20]
lea     rcx, cs:140000000h
mov     eax, ds:(jpt_1400010F5 - 140000000h)[rcx+rax*4]
add     rax, rcx
jmp     rax             ; switch jump
```

```
loc_1400010F7:          ; jumptable 00000001400010F5 case 0
lea     rcx, aBoring
call    cs:printf
jmp     short loc_140001155
```

```
loc_140001106:          ; jumptable 00000001400010F5 case 1
lea     rcx, aFound1
call    cs:printf
jmp     short loc_140001155
```

```
loc_140001115:          ; jumptable 00000001400010F5 case 2
lea     rcx, aFound2
call    cs:printf
jmp     short loc_140001155
```

```
loc_140001124:          ; jumptable 00000001400010F5 case 3
lea     rcx, aFound3
call    cs:printf
jmp     short loc_140001155
```

```
loc_140001133:          ; jumptable 00000001400010F5 case 4
lea     rcx, aFound4M
call    cs:printf
jmp     short loc_140001155
```

```
loc_140001142:          ; jumptable 00000001400010F5 case 5
lea     rcx, aFound5OO
call    cs:printf
jmp     short loc_140001155
```

```
xor     eax, eax
jmp     loc_14000115A
```

```
def_1400010F5:          ; jumptable 00000001400010F5 default case
xor     eax, eax
jmp     short loc_14000115A
```

```
loc_140001155:
mov     eax, 1
```

```
loc_1400010B9:
mov     eax, [rsp+48h+var_28]
inc     eax
mov     [rsp+48h+var_28], eax
```

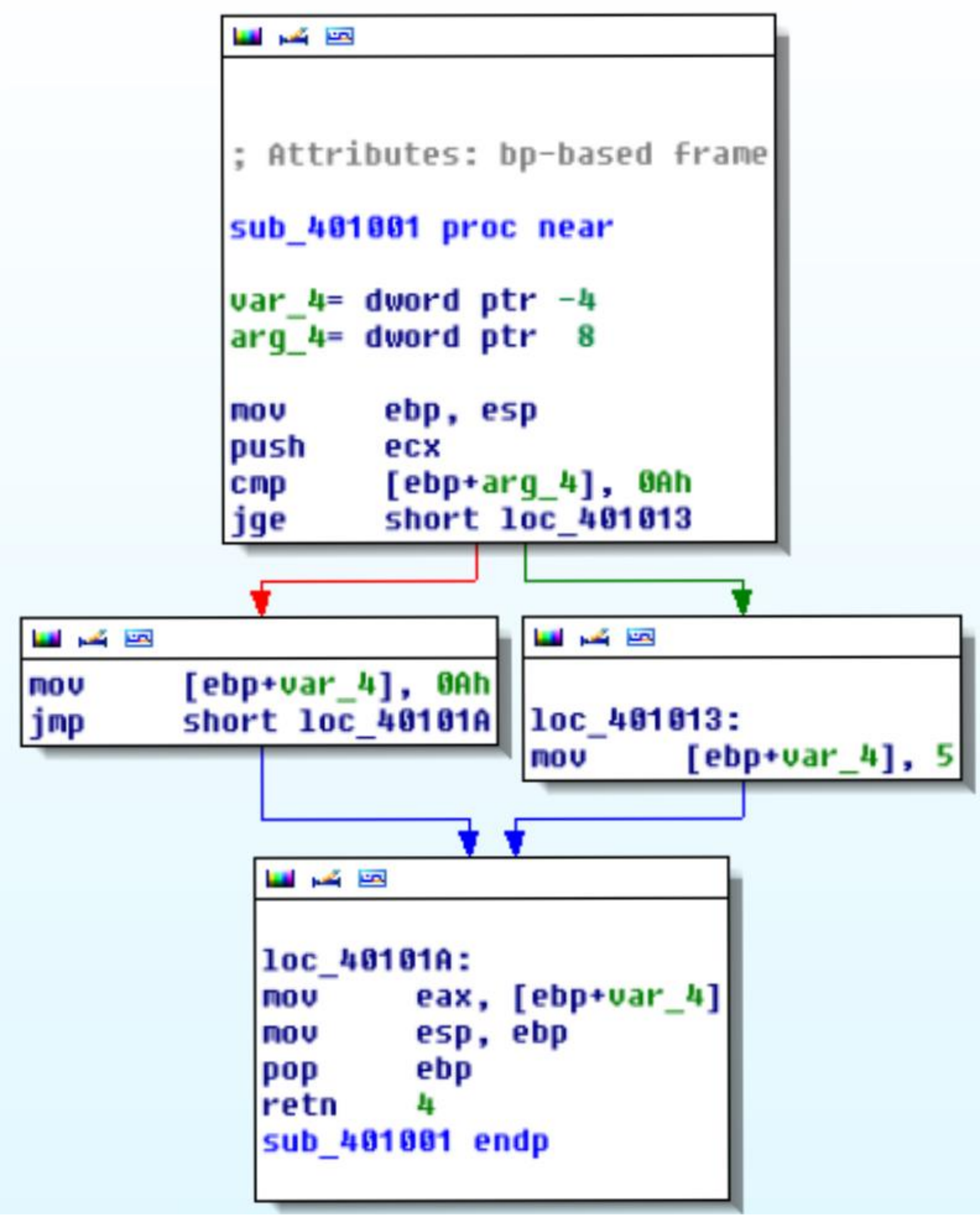
```
loc_14000115A:
add     rsp, 48h
retn
```

Branches

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

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

    return ret;
}
```



```
; Attributes: bp-based frame

sub_401001 proc near

var_4= dword ptr -4
arg_4= dword ptr  8

mov     ebp, esp
push    ecx
cmp     [ebp+arg_4], 0Ah
jge     short loc_401013
```

```
mov     [ebp+var_4], 0Ah
jmp     short loc_40101A
```

```
loc_401013:
mov     [ebp+var_4], 5
```

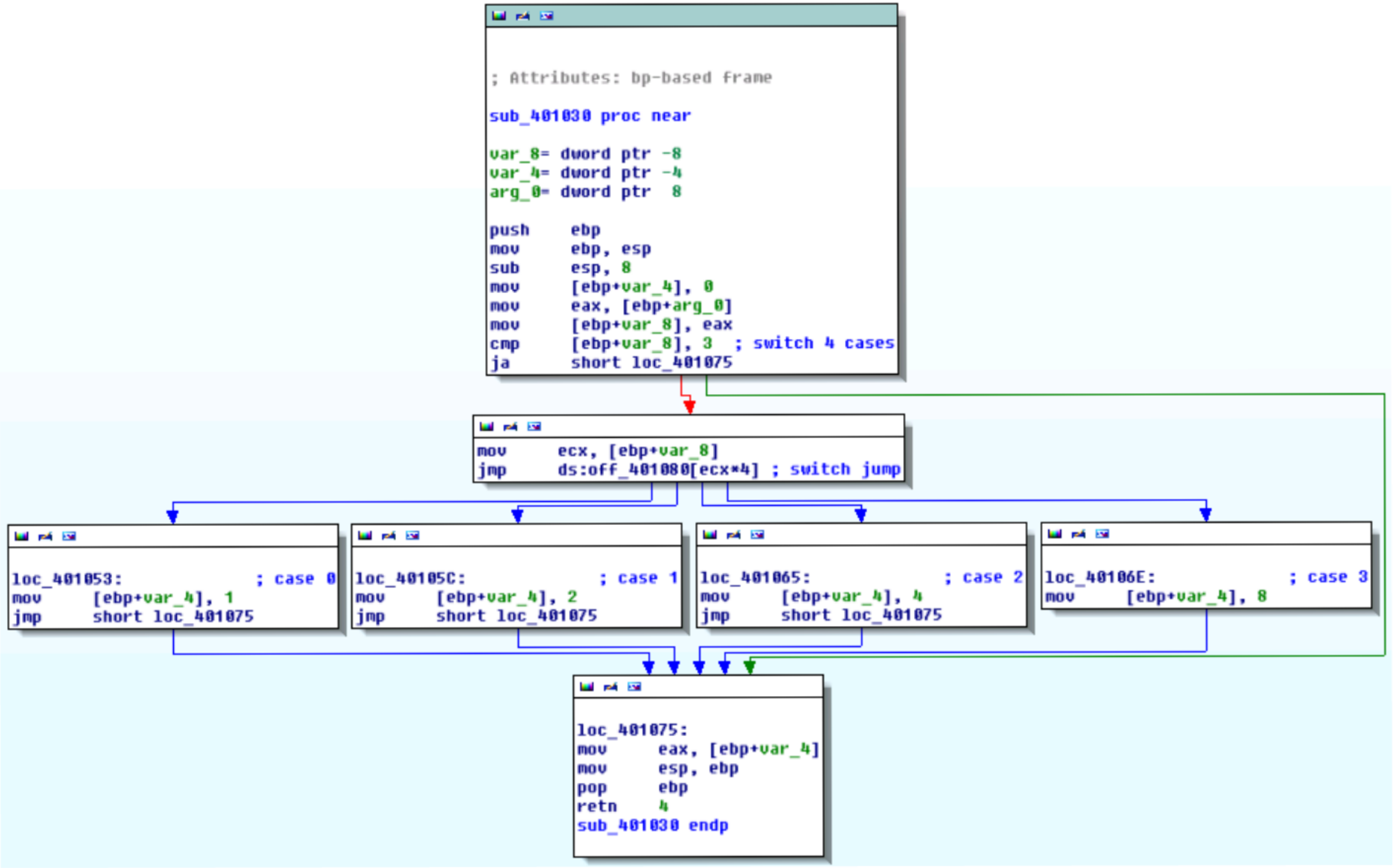
```
loc_40101A:
mov     eax, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn    4
sub_401001 endp
```

Switch

```c
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;
}
```



```
; Attributes: bp-based frame

sub_401030 proc near

var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
mov     [ebp+var_4], 0
mov     eax, [ebp+arg_0]
mov     [ebp+var_8], eax
cmp     [ebp+var_8], 3  ; switch 4 cases
ja      short loc_401075
```

```
mov     ecx, [ebp+var_8]
jmp     ds:off_401080[ecx*4] ; switch jump
```

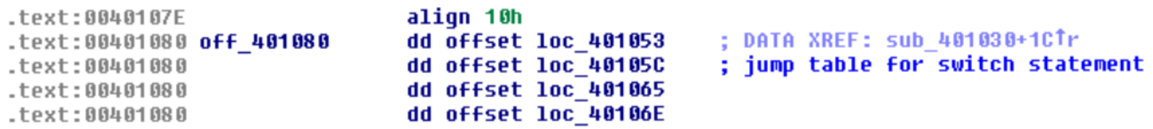
```
loc_401053:                 ; case 0
mov     [ebp+var_4], 1
jmp     short loc_401075
```

```
loc_40105C:                 ; case 1
mov     [ebp+var_4], 2
jmp     short loc_401075
```

```
loc_401065:                 ; case 2
mov     [ebp+var_4], 4
jmp     short loc_401075
```

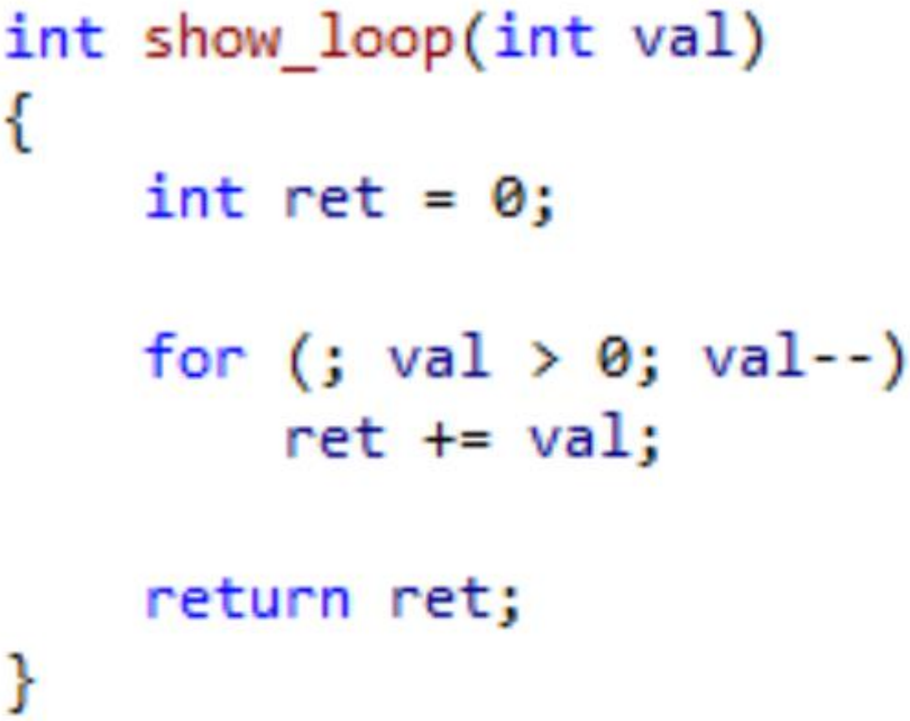
```
loc_40106E:                 ; case 3
mov     [ebp+var_4], 8
```

```
loc_401075:
mov     eax, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn    4
sub_401030 endp
```



```
.text:0040107E                  align 10h
.text:00401080 off_401080       dd offset loc_401053    ; DATA XREF: sub_401030+1C↑r
.text:00401080                  dd offset loc_40105C    ; jump table for switch statement
.text:00401080                  dd offset loc_401065
.text:00401080                  dd offset loc_40106E
```

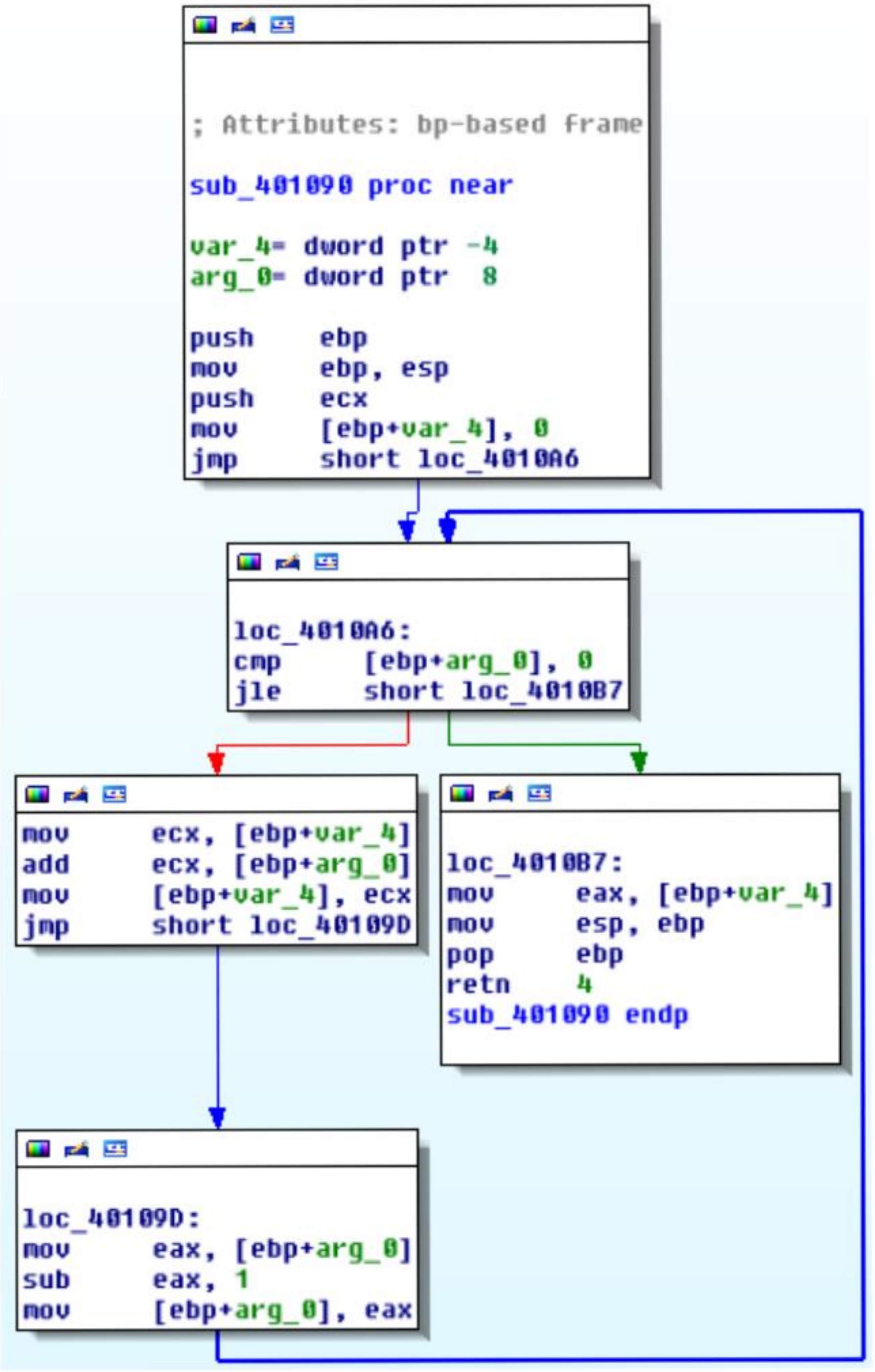
Loops



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

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

    return ret;
}
```



```
; Attributes: bp-based frame

sub_401090 proc near

var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], 0
jmp     short loc_4010A6
```

```
loc_4010A6:
cmp     [ebp+arg_0], 0
jle     short loc_4010B7
```

```
mov     ecx, [ebp+var_4]
add     ecx, [ebp+arg_0]
mov     [ebp+var_4], ecx
jmp     short loc_40109D
```

```
loc_4010B7:
mov     eax, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn    4
sub_401090 endp
```

```
loc_40109D:
mov     eax, [ebp+arg_0]
sub     eax, 1
mov     [ebp+arg_0], eax
```