X86/WIN32 REVERSE ENGINEERING CHEAT-SHEET

Registers

GENERAL PURPOSE 32-BIT REGISTERS

EAX	Contains the return value of a function call.
ECX	Used as a loop counter. "this" pointer in C++.
EBX	General Purpose
EDX	General Purpose
ESI	Source index pointer
EDI	Destination index pointer
ESP	Stack pointer
EBP	Stack base pointer

SEGMENT REGISTERS

CS	Code segment
SS	Stack segment
DS	Data segment
ES	Extra data segment
FS	Points to Thread Information Block (TIB)
GS	Extra data segment

MISC. REGISTERS

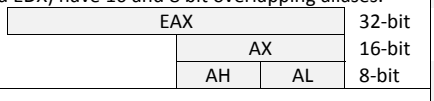
EIP	Instruction pointer
EFLAGS	Processor status flags.

STATUS FLAGS

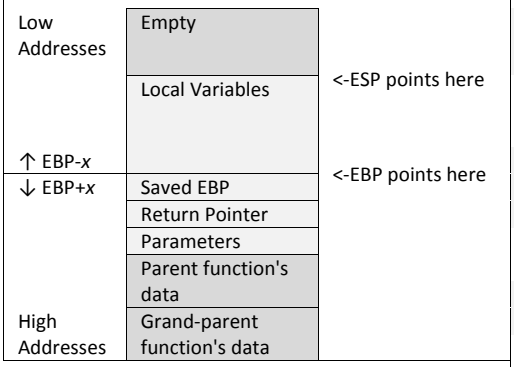
ZF	Zero: Operation resulted in Zero
CF	Carry: source > destination in subtract
SF	Sign: Operation resulted in a negative #
OF	Overflow: result too large for destination

16-BIT AND 8-BIT REGISTERS

The four primary general purpose registers (EAX, EBX, ECX and EDX) have 16 and 8 bit overlapping aliases.



The Stack



Assembly Language

Instruction listings contain at least a mnemonic, which is the operation to be performed. Many instructions will take operands. Instructions with multiple operands list the destination operand first and the source operand second *(<dest>*, <source>). Assembler directives may also be listed which appear similar to instructions.

ASSEMBLER DIRECTIVES

DB *<byte>*	Define Byte. Reserves an explicit byte of memory at the current location. Initialized to <byte> value.
DW *<word>*	Define Word. 2-Bytes
DD *<dword>*	Define DWord. 4-Bytes

OPERAND TYPES

Immediate	A numeric operand, hard coded
Register	A general purpose register
Memory	Memory address w/ brackets [ ]

Instructions

ADD *<dest>, <source>*	Adds *<source>* to *<dest>*. *<dest>* may be a register or memory. *<source>* may Be a register, memory or immediate value.
CALL *<loc>*	Call a function and return to the next instruction when finished. *<proc>* may be a relative offset from the current location, a register or memory addr.
CMP *<dest>, <source>*	Compare *<source>* with *<dest>*. Similar to SUB instruction but does not Modify the *<dest>* operand with the result of the subtraction.
DEC *<dest>*	Subtract 1 from <dest>. <dest> may be a register or memory.
DIV *<divisor>*	Divide the EDX:EAX registers (64-bit combo) by *<divisor>*. *<divisor>* may be a register or memory.
INC *<dest>*	Add 1 to *<dest>*. *<dest>* may be a register or memory.
JE *<loc>*	Jump if Equal (ZF=1) to *<loc>*.
JG *<loc>*	Jump if Greater (ZF=0 and SF=OF) to *<loc>*.
JGE *<loc>*	Jump if Greater or Equal (SF=OF) to *<loc>*.
JLE *<loc>*	Jump is Less or Equal (SF<>OF) to *<loc>*.
JMP *<loc>*	Jump to *<loc>*. Unconditional.
JNE *<loc>*	Jump if Not Equal (ZF=0) to *<loc>*.
JNZ *<loc>*	Jump if Not Zero (ZF=0) to *<loc>*.
JZ *<loc>*	Jump if Zero (ZF=1) to *<loc>*.
LEA *<dest>, <source>*	Load Effective Address. Gets a pointer to the memory expression *<source>* and stores it in *<dest>*.
MOV *<dest>, <source>*	Move data from *<source>* to *<dest>*. *<source>* may be an immediate value, register, or a memory address. Dest may be either a memory address or a register. Both *<source>* and *<dest>* may not be memory addresses.
MUL *<source>*	Multiply the EDX:EAX registers (64-bit combo) by *<source>*. *<source>* may be a register or memory.
POP *<dest>*	Take a 32-bit value from the stack and store it in *<dest>*. ESP is incremented by 4. *<dest>* may be a register, including segment registers, or memory.
PUSH *<value>*	Adds a 32-bit value to the top of the stack. Decrements ESP by 4. *<value>* may be a register, segment register, memory or immediate value.
ROL *<dest>, <count>*	Bitwise Rotate Left the value in *<dest>* by *<count>* bits. *<dest>* may be a register or memory address. *<count>* may be immediate or CL register.
ROR *<dest>, <count>*	Bitwise Rotate Right the value in *<dest>* by *<count>* bits. *<dest>* may be a register or memory address. *<count>* may be immediate or CL register.
SHL *<dest>, <count>*	Bitwise Shift Left the value in *<dest>* by *<count>* bits. Zero bits added to the least significant bits. *<dest>* may be reg. or mem. *<count>* is imm. or CL.
SHR *<dest>, <count>*	Bitwise Shift Left the value in *<dest>* by *<count>* bits. Zero bits added to the least significant bits. *<dest>* may be reg. or mem. *<count>* is imm. or CL.
SUB *<dest>, <source>*	Subtract *<source>* from *<dest>*. *<source>* may be immediate, memory or a register. *<dest>* may be memory or a register. (source = dest)->ZF=1, (source > dest)->CF=1, (source < dest)->CF=0 and ZF=0
TEST *<dest>, <source>*	Performs a logical OR operation but does not modify the value in the *<dest>* operand. (source = dest)->ZF=1, (source <> dest)->ZF=0.
XCHG *<dest, <source>*	Exchange the contents of *<source>* and *<dest>*. Operands may be register or memory. Both operands may not be memory.
XOR *<dest>, <source>*	Bitwise XOR the value in *<source>* with the value in *<dest>*, storing the result in *<dest>*. *<dest>* may be reg or mem and *<source>* may be reg, mem or imm.

Terminology and Formulas

Pointer to Raw Data	Offset of section data within the executable file.
Size of Raw Data	Amount of section data within the executable file.
RVA	Relative Virtual Address. Memory offset from the beginning of the executable.
Virtual Address (VA)	Absolute Memory Address (RVA + Base). The PE Header fields named VirtualAddress actually contain Relative Virtual Addresses.
Virtual Size	Amount of section data in memory.
Base Address	Offset in memory that the executable module is loaded.
ImageBase	Base Address requested in the PE header of a module.
Module	An PE formatted file loaded into memory. Typically EXE or DLL.
Pointer	A memory address
Entry Point	The address of the first instruction to be executed when the module is loaded.
Import	DLL functions required for use by an executable module.
Export	Functions provided by a DLL which may be Imported by another module.
RVA->Raw Conversion	Raw = (RVA - SectionStartRVA) + (SectionStartRVA - SectionStartPtrToRaw)
RVA->VA Conversion	VA = RVA + BaseAddress
VA->RVA Conversion	RVA = VA - BaseAddress
Raw->VA Conversion	VA = (Raw - SectionStartPtrToRaw) + (SectionStartRVA + ImageBase)

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed $\geq$)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed $\leq$)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned $\geq$)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned $\leq$)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed $\geq$)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed $\leq$)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest $*$ Src
`xorq` Src, Dest	Dest = Dest ^ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = $\sim$ Dest
`salq` k, Dest	Dest = Dest $\ll k$
`sarq` k, Dest	Dest = Dest $\gg k$ (arithmetic)
`shrq` k, Dest	Dest = Dest $\gg k$ (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved