# Practical Malware Analysis

## Lecture 4 | Intro to x86 Disassembly

Disassembling x86 programs

- See how malware uses the functions it imports
- Dig into specialized inputs and outputs

# Abstraction

- Hardware (digital logic)
- Microcode (firmware)
- Machine Code
- Low-level Languages
- High-level Languages
- Interpreted Languages

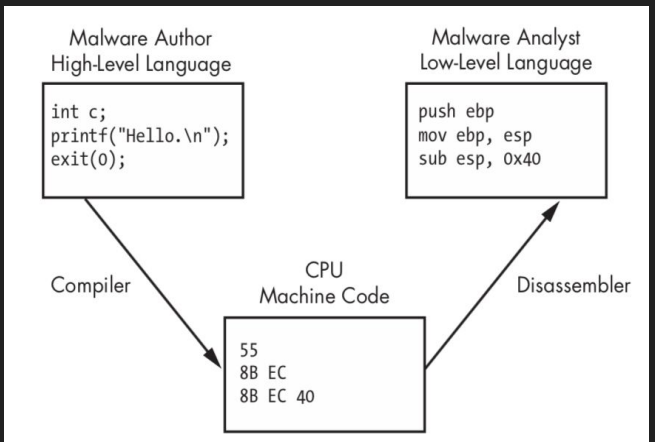


*Figure 4-1: Code level examples*

"In traditional computer architecture, a computer system can be represented as several levels of abstraction that create a way of hiding the implementation details. For example, you can run the Windows OS on many different types of hardware, because the underlying hardware is abstracted from the OS."

As shown in the diagram, malware authors typically write programs in a high-level language such as C, which is then compiled into machine code that the CPU can execute. In the event that we only have a binary sample (no source code), we must use a disassembler to turn this machine code into something more digestible for analysis (unless you really want to learn machine code). Due to compiler optimizations and the fact that most malware authors will compile their code without debugging symbols, we cannot perfectly decompile a binary into its original source code. "Assembly is the highest level language that can be reliably and consistently recovered from machine code when high-level language source code is not available."

When we discuss hardware, we're referring to the physical implementations of digital logic as electrical circuits (think AND, OR, XOR gates). Microcode refers to the code that is written specifically to interact with the hardware for which it was designed as an interface for higher levels of code. Machine code consists of opcodes (several microcode instructions for the CPU), and is created when a program is compiled. Low-level languages (such as assembly) represent a human-readable version of an architecture's instruction set and can be generated with a disassembler. High-level languages (like C) are what most authors will write code in and then compile it.

Interpreted languages (like Java) are compiled into bytecode (not machine code) and then executed in an interpreter which translates the bytecode into machine code at runtime, making it easier to develop a single binary that can be run on multiple operating systems.

# Assembly

- Comes in many flavors
  - x86, x64, SPARC, PowerPC, MIPS, ARM
- We will focus on x86 (Intel IA-32)
  - All 32-bit Windows OSes run on x86
  - Most x64 architectures running Windows support x86

Assembly
Comes in many flavors: x86, x64, SPARC, PowerPC, MIPS, ARM
We will focus on x86 (Intel IA-32) because…
All 32-bit Windows OSes run on x86
Most x64 architectures running Windows support x86

# Von Neumann Hardware Architecture

- CPU - executes code
  - Control Unit
  - Arithmetic Logic Unit
  - CPU registers
- RAM - stores data / code
- I/O device interface

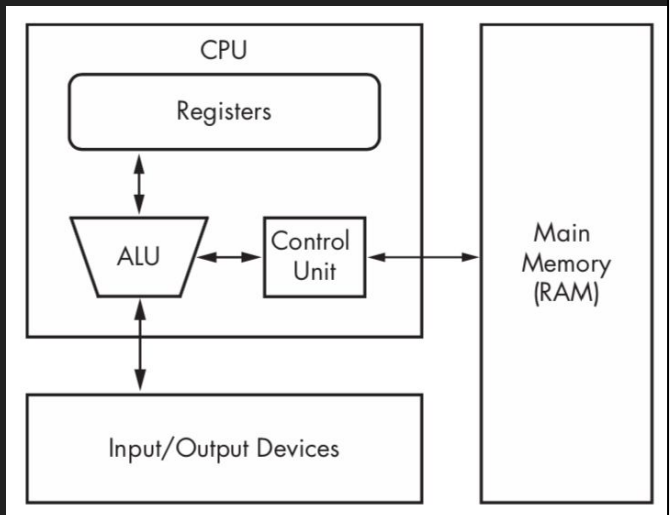


*Figure 4-2: Von Neumann architecture*

Most modern architectures follow the Von Neumann hardware architecture which has three main components: an input/output interface for devices such as keyboards, a set of main memory for storing quickly accessible data and code, and a central processing unit for executing code. In the CPU

1. The control unit fetches instructions for execution from RAM using a register (the instruction pointer) containing the address of the instruction to be executed.
2. The arithmetic logic unit executes the instruction and stores the results in registers or back into memory.
3. This process repeats as a program runs

Registers are simply units of memory that are quicker for the CPU to access than RAM.

# Main Memory (RAM)

- No fixed order ->
- Static/global values
- Instructions
- Dynamic memory
- Local variables & program flow

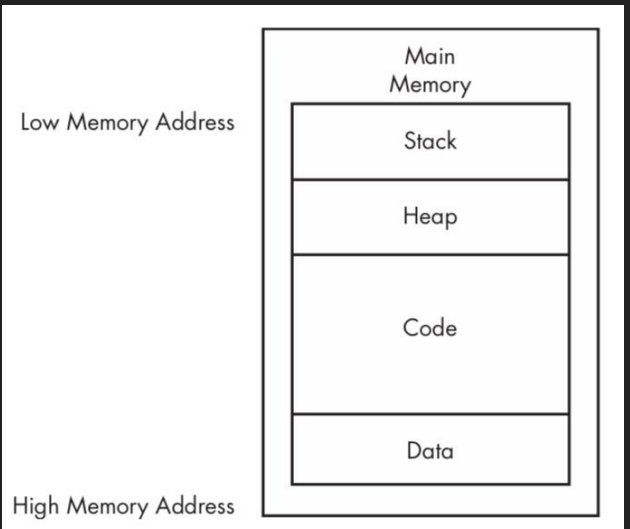


*Figure 4-3: Basic memory layout for a program*

There are four major sections that exist in volatile memory (RAM) though not necessarily in the order portrayed by the diagram.

The data section holds static values that are created when a program is loaded, also called global values - as they are accessible to any piece of the program being executed. The code section contains instructions to be fetched and executed by the CPU. The heap is used to dynamically allocate and free memory for values as they are needed by the program as it runs. The stack is used for local variables and controlling program flow during execution.

# Instructions and Opcodes

**Table 4-1:** Instruction Format

| Mnemonic | Destination operand | Source operand |
|---|---|---|
| mov | ecx | 0x42 |

**Table 4-2:** Instruction Opcodes

| | | |
|---|---|---|
| **Instruction** | mov ecx, | 0x42 |
| **Opcodes** | B9 | 42 00 00 00 |

Instructions are the components of programs in assembly, consisting of a single mnemonic representing the action to be taken by the CPU, and zero or more operands to use as source and destination values and locations. Each instruction in assembly represents an operation code (opcode), which is the actual machine instruction executed by the CPU. Since opcodes are not human-friendly, disassemblers are used to translate them into assembly instructions.

## Endianness

| Big Endian | Little Endian |
| --- | --- |
| Most significant bit first | Least significant bit first |
| Network data | Data in memory |
| 0x12345678 | 0x78563412 |

The endianness of data describes whether the most significant or least significant bit is at the smallest address within a larger unit of data. In x86 architecture, a program will have data that is little endian in memory, but network traffic will have data in big endian. Being aware of what endianness data is in is important during analysis, so you don't end up with indicators such as network addresses in a reversed order (e.g. 1.0.0.127, which is technically a valid IP address, versus 127.0.0.1, the loopback address).

## Operands

The data used by an instruction, of type:

| | |
|---|---|
| Immediate (fixed value) | 0x42 |
| Register | ecx |
| Memory address | [eax] or [0x12345678] |

There are three main type of data used by an instruction (called operands in the instruction format). Data may be an immediate operand - a fixed value, a register (quick-access storage on the CPU), or a memory address containing the value of interest, denoted by brackets around the numerical or named alias of the memory address.

# Registers

**Table 4-3:** The x86 Registers

| General registers | Segment registers | Status register | Instruction pointer |
|---|---|---|---|
| EAX (AX, AH, AL) | CS | EFLAGS | EIP |
| EBX (BX, BH, BL) | SS | | |
| ECX (CX, CH, CL) | DS | | |
| EDX (DX, DH, DL) | ES | | |
| EBP (BP) | FS | | |
| ESP (SP) | GS | | |
| ESI (SI) | | | |

General registers - used by the CPU during execution
Segment registers - used to keep track of sections in memory
Status registers - used to make decisions
Instruction pointer - used to track the next instruction to execute

Registers are 32 bits in size but can be referenced as 16-bit or in the case of EAX, EBX, ECX, EDC, 8-bit as well

# General Registers

*Conventions*

- Implemented by compilers
- Allows us to make assumptions
- Not always the case

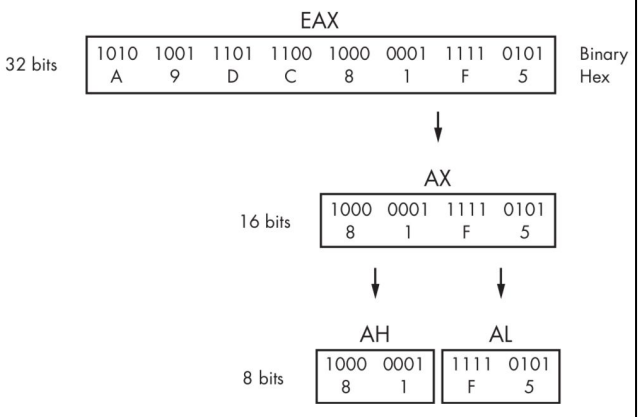


*Figure 4-4: x86 EAX register breakdown*

As you can see in the diagram, any reference to a 16-bit chunk of a general register references the lower 16 bits, which can be further broken down into two 8-bit registers (High and Low).

There are some common conventions used amongst x86 compilers that allow an analyst to make assumptions about how a general register is being used. For example, multiplication and division instructions always use EAX and EDX. EAX is assumed to contain the return value after a function call completes. However, just as malware authors don't have to follow naming conventions for Windows API functions, compilers aren't required to adhere to common register use conventions.

# Flags

- EFLAGS register

- Zero flag
- Carry flag
- Sign flag
- Trap flag

In x86 architecture, the EFLAGS register is 32-bits in size, with each bit set to a 1 (set) or a 0 (cleared). Each bit may be checked to control CPU operations or indicate the results of an operation. For now, we will focus on four commonly encountered flags.

The zero flag is set (1) when an operation results in a 0 value, otherwise it is cleared (0).
The carry flag is set when the result of an operation is too large or too little for the destination operand.
The sign flag is set when the result of an operation is negative (when the most significant bit is set) or cleared if positive.
The trap flag is set for debugging a program, the CPU will only execute one instruction at a time while set.

# EIP

- Instruction pointer / program counter
- Holds memory address of next instruction for execution

Attackers try to control EIP and point it to their code

EIP is a register containing the memory address of the next instruction to be executed by the CPU. If EIP does not point to code, a program will likely crash. If an attacker can gain control of EIP, they may point it to their own code in memory and gain local execution.

# mov

(move)

*mov <dst>, <src>* (intel syntax)

**Table 4-4:** `mov` Instruction Examples

| Instruction | Description |
|---|---|
| `mov eax, ebx` | Copies the contents of EBX into the EAX register |
| `mov eax, 0x42` | Copies the value 0x42 into the EAX register |
| `mov eax, [0x4037C4]` | Copies the 4 bytes at the memory location 0x4037C4 into the EAX register |
| `mov eax, [ebx]` | Copies the 4 bytes at the memory location specified by the EBX register into the EAX register |
| `mov eax, [ebx+esi*4]` | Copies the 4 bytes at the memory location specified by the result of the equation `ebx+esi*4` into the EAX register |

The most common instruction you will come across is mov - used for moving data from one location to another. In Intel syntax, the format for the mov instruction is mov <dst>, <src>. In the examples below you can see mov being called with the EAX general register as the destination and various types of operands as the source.

# lea

(load effective address)

*lea <dst>, <src>*

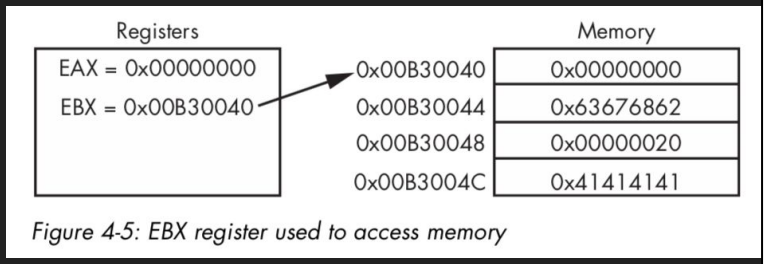

*Figure 4-5: EBX register used to access memory*

mov eax, [ebx+8] sets eax = 20

lea eax, [ebx+8] sets eax = 0x00B30048

lea eax, [ebx*2 + ebx] = ebx*3 = 0xB30040*3 = 0x021900C0

The lea (which stands for load effective address) instruction is used to put a memory address into the destination. lea is also used in compiler optimizations to perform arithmetic operations (instead of using mul, add, etc which require more operations by the CPU). When used in this manner, it does not dereference memory, but computes the memory address specified by the arithmetic and stores that value into the source register. In this case, the brackets around the source operand do not actually mean “what is stored at this address”. Additionally, arithmetic done using lea will not affect any of the previously discussed flags.

# add, sub, inc, dec

- sub modifies ZF and CF

**Table 4-5:** Addition and Subtraction Instruction Examples

| Instruction | Description |
|---|---|
| `sub eax, 0x10` | Subtracts 0x10 from EAX |
| `add eax, ebx` | Adds EBX to EAX and stores the result in EAX |
| `inc edx` | Increments EDX by 1 |
| `dec ecx` | Decrements ECX by 1 |

The addition and subtraction instructions are of the format *add/sub <dst>, value*. The subtraction instruction may modify the zero flag, if an operation results in a 0 value, or the carry flag, if the value supplied is greater than the destination specified. The increment and decrement instructions either increment or decrement the specified register by one.

## mul and div

*mul* <*value*> multiplies EAX by <value>

- Result is stored in EDX, EAX

*div* <*value*> divides the 64-bit value of EDX, EAX by <value>

- Result -> EAX, remainder -> EDX

*imul* and *idiv* for signed

mul <value> multiplies EAX by <value>
	Result is stored in EDX, EAX
div <value> divides the 64-bit value of EDX, EAX by <value>
	Result -> EAX, remainder -> EDX
imul and idiv for signed integers

## and, or, xor, shl, shr, ror, rol

- xor eax, eax -> compiler optimization for setting eax to 0
- *shl <dst>, count* shifts bits in <dst> left by <count>
  - Bits shifted beyond the bounds of <dst> are moved into CF
  - *shl eax, 1* gives the same result as *mul 2* (*shl* by $n$ = *mul* $2^n$)
- *ror <dst>, count* rotates bits in <dst> right by <count>
  - Bits shifted beyond the bounds of <dst> are rotated to the MSB (right rotation) or LSB (left rotation)

The and, or, and xor instructions perform bitwise operations using the <dst> and <src> operands provided. The xor instruction is also commonly used as a compiler optimization to clear a register by xoring the register against itself, since the opcode for this instruction requires less bytes than an opcode for moving 0 into the register with mov.

The shift left instruction shifts bits in <dst> left by <count>

Bits shifted beyond the bounds of <dst> are moved into CF, so after the instruction executes, CF will contain the last bit to be shifted out of <dst>. The shift right instruction functions the same way, in the opposite direction.

Shifting is also taken advantage of in compiler optimizations as a shortcut for multiplication.

shl eax, 1 gives the same result as mul 2 (shl by n = mul 2n)

The right rotation instruction rotates bits in <dst> right by <count>.

Bits shifted beyond the bounds of <dst> are rotated to the MSB (or LSB in left rotation).

**Table 4-7:** Common Logical and Shifting Arithmetic Instructions

| Instruction | Description |
|---|---|
| `xor eax, eax` | Clears the EAX register |
| `or eax, 0x7575` | Performs the logical or operation on EAX with 0x7575 |
| `mov eax, 0xA`<br>`shl eax, 2` | Shifts the EAX register to the left 2 bits; these two instructions result in EAX = 0x28, because 1010 (0xA in binary) shifted 2 bits left is 101000 (0x28) |
| `mov bl, 0xA`<br>`ror bl, 2` | Rotates the BL register to the right 2 bits; these two instructions result in BL = 10000010, because 1010 rotated 2 bits right is 10000010 |

- All of these instructions together, randomly and repeatedly may indicate compression or encryption

All of these instructions together, randomly and repeatedly may indicate compression or encryption

# nop

No operation

- *xchg eax, eax* (does nothing)
- Opcode 0x90
- Many together may indicate a NOP sled (buffer overflow attack)

No operation (nop instruction) is a pseudonym for xchg eax, eax (which does nothing).
The opcode is 0x90. Seeing many of these back-to-back may indicate a NOP sled for buffer overflow attacks.

## The Stack

0x00000000 (low)

- FILO data structure
- *push* things on
- *pop* things off
- Extended Stack Pointer (ESP) contains memory address of top
- Extended Base Pointer (EBP) contains memory address of base

0xFFFFFFFF (high)

The stack is an important data structure which represents how space is allocated and used in RAM for short-term storage of function local variables, parameters, and the return address. The stack stores information used for flow control, managing data that is passed between function calls.

A stack is a first-in-last-out data structure, meaning that the first thing to be pushed onto a stack will be the last thing to be popped from a stack. Two general purpose registers, ESP and EBP are used to track the upper and lower bounds of the stack, respectively. ESP will typically change as things are added to the stack, while EBP remains static and is used to reference the locations of local variables and parameters (the first things to be pushed onto a stack when a function is called).

The stack grows towards lower memory addresses (those closer to 0x00000000).

## Function Calls

Function: a unit of (independent) code within a program that performs a specific task.

- **Main** calls and transfers execution to *function*
- *Function* executes and passes control back to **main**
- Prologue - prepares stack and registers for use
- Epilogue - restores the original state of stack/registers

*How the stack is used by a program is consistent throughout a binary.

Function: a unit of (independent) code within a program that performs a specific task.

Main calls and transfers execution to function
Function executes and passes control back to main

Functions contain a prologue—a few lines of code at the start of the function that prepare the stack and registers for use within the function. An epilogue at the end of a function restores the stack and registers to the state they were in before the function was called.

How the stack is used by a program is consistent throughout a binary.

# Calling Convention - cdecl

1. *push* arguments onto the stack
2. *call* function using *call <memory_location>*
   a. *push* address of current instruction (EIP) onto stack
   b. Set EIP to <memory_location>
3. Allocate space for local variables, *push* EBP
4. Function executes
5. *leave* sets ESP = EBP, *pops* EBP
6. Function calls *ret*, *pops* return address -> EIP to continue caller execution
7. Adjust stack to remove arguments, unless they are used again later

The C declaration (cdecl) calling convention is the most common calling convention for x86 architecture.

The most common flow for function calls is as follows:
“1. Arguments are placed on the stack using push instructions.
2. A function is called using call memory_location. This causes the current instruction address (that is, the contents of the EIP register) to be pushed onto the stack. This address will be used to return to the main code when the function is finished. When the function begins, EIP is set to memory_location (the start of the function).
3. Through the use of a function prologue, space is allocated on the stack for local variables and EBP (the base pointer) is pushed onto the stack. This is done to save EBP for the calling function.
4. The function performs its work.
5. Through the use of a function epilogue, the stack is restored. ESP is adjusted to free the local variables, and EBP is restored so that the calling function can address its variables properly. The leave instruction can be used as an epilogue because it sets ESP to equal EBP and pops EBP off the stack.
6. The function returns by calling the ret instruction. This pops the return address off the stack and into EIP, so that the program will continue executing from where the original call was made.
7. The stack is adjusted to remove the arguments that were sent, unless they’ll be used again later.”

# Stack Layout

Each new function call generates a new stack frame

*pusha/popa (16-bit)*
*pushad/popad (32-bit)*

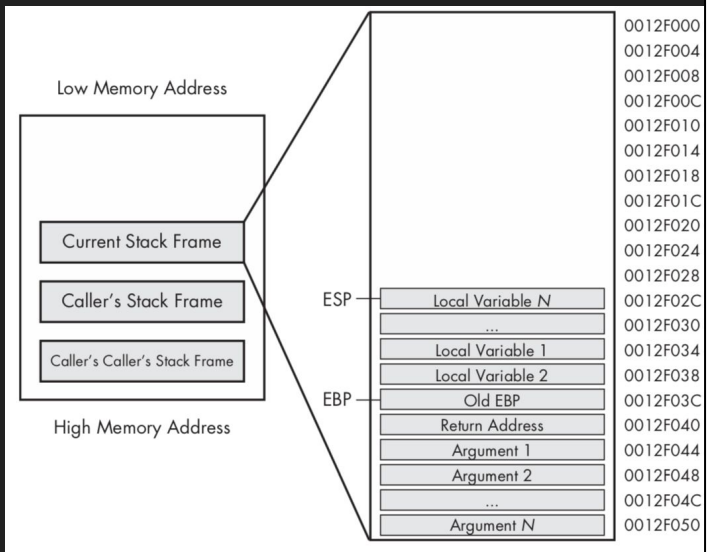

*Figure 4-8: Individual stack frame*

We can see here that each function call generates a new stack frame, which is maintained by the function until it returns. Memory within a stack frame is set up according to the cdecl calling convention, with arguments pushed first, followed by the return address (EIP), the caller's EBP, and space for local variables. ESP is decreased when variables are pushed onto the stack and incremented when variables are popped from the stack. For example, "if the instruction push eax were executed, ESP would be decremented by four and would contain 0x12F028, and the data contained in EAX would be copied to 0x12F028. If the instruction pop ebx were executed, the data at 0x12F028 would be moved into the EBX register, and then ESP would be incremented by four."

Additionally, other instructions for modifying the stack include pusha/popa which push or pop each 16-bit register, and pushad/popad which push or pop every 32-bit register. These instructions may occur in shellcode, if an author is saving the state of all registers in order to restore them after malicious code executes, and rarely occur in non-malicious compiled code.

# Conditionals

## Make comparisons and decisions

- *test* = *and* but only sets flags (ZF)
- *cmp* = *sub* but only sets flags (ZF, CF)

**Table 4-8:** `cmp` Instruction and Flags

| cmp dst, src | ZF | CF |
|---|---|---|
| dst = src | 1 | 0 |
| dst < src | 0 | 1 |
| dst > src | 0 | 0 |

Conditional instructions are used to make comparisons between values, usually in order to use the results to make a decision such as whether to jump to another piece of code. *test* and *cmp* are identical to the *and* and *sub* instructions, but they do not modify their operands and are only used to set the flags. A test of something against itself is usually used to check for a null value (will set the ZF). The table shows how the cmp instruction affects the ZF and CF depending on the outcome.

# Branching

## Conditional execution

### (IF statements)

*jmp*

*Jcc*

**Table 4-9:** Conditional Jumps

| Instruction | Description |
|---|---|
| `jz loc` | Jump to specified location if ZF = 1. |
| `jnz loc` | Jump to specified location if ZF = 0. |
| `je loc` | Same as `jz`, but commonly used after a `cmp` instruction. Jump will occur if the destination operand equals the source operand. |
| `jne loc` | Same as `jnz`, but commonly used after a `cmp`. Jump will occur if the destination operand is not equal to the source operand. |
| `jg loc` | Performs signed comparison jump after a `cmp` if the destination operand is greater than the source operand. |
| `jge loc` | Performs signed comparison jump after a `cmp` if the destination operand is greater than or equal to the source operand. |
| `ja loc` | Same as `jg`, but an unsigned comparison is performed. |
| `jae loc` | Same as `jge`, but an unsigned comparison is performed. |
| `jl loc` | Performs signed comparison jump after a `cmp` if the destination operand is less than the source operand. |
| `jle loc` | Performs signed comparison jump after a `cmp` if the destination operand is less than or equal to the source operand. |
| `jb loc` | Same as `jl`, but an unsigned comparison is performed. |
| `jbe loc` | Same as `jle`, but an unsigned comparison is performed. |
| `jo loc` | Jump if the previous instruction set the overflow flag (OF = 1). |
| `js loc` | Jump if the sign flag is set (SF = 1). |
| `jecxz loc` | Jump to location if ECX = 0. |

A branch is a series of instructions that is optionally executed, based on the flow of the program (think IF statements). Branching is most commonly implemented with *jmp* instructions (an unconditional jump that simply specifies the next instruction to be executed) or conditional jumps as shown in the table. There are more than 30 different types of conditional jumps, denoted generally by *Jcc*.

## rep (Repeat)

- Manipulate data buffers
- Prefixes manipulation instructions
  - *movs*x, *cmps*x, *stos*x, and *scas*x where x is:
  - Bytes (b), words (w), or double words (d)
- Uses ESI, EDI, and ECX
  - Increments indexes, decrements counter

**Table 4-10:** rep Instruction Termination Requirements

| Instruction | Description |
|---|---|
| rep | Repeat until ECX = 0 |
| repe, repz | Repeat until ECX = 0 or ZF = 0 |
| repne, repnz | Repeat until ECX = 0 or ZF = 1 |

Rep (repeat) instruction

Manipulate data buffers (usually in the form of an array of bytes)

Prefixes manipulation instructions that would otherwise only act on a single unit (byte) to create multibyte operations

movsx (move sequence), cmpsx (compare sequence), stosx (store sequence), and scasx (scan sequence) where x is:

Bytes (b), words (w), or double words (d)

Uses ESI (source index), EDI (dest index), and ECX (counter)

Increments indexes, decrements counter until a requirement is met as picture in the table

## *movsb*

Move a sequence of [bytes] from one location to another

- *rep movsb* ~ memcpy in C
  a. Check if ECX = 0
  b. Grab byte at ESI
  c. Move to EDI
  d. *Increment (DF = 0) or decrement ESI and EDI
  e. Decrement ECX

  * DF sometimes flipped in shellcode

Move a sequence of [bytes] from one location to another
rep movsb ~ memcpy in C
	Check if ECX = 0
	Grab byte at ESI
	Move to EDI
	*Increment (DF = 0) or decrement ESI and EDI
	Decrement ECX
* DF sometimes flipped in shellcode to store data in the reverse direction

## *cmpsb*

- Compare two sequences of [bytes]
  - sub EDI from ESI
  - Update flags
  - Increment ESI, EDI

*repe cmpsb* ~ memcmp in C

- compare until different or end
  - ECX = 0 or ZF = 0

Compare two sequences of [bytes]
sub EDI from ESI
Update flags
Increment ESI, EDI
repe cmpsb ~ memcmp in C
compare until different or end
ECX = 0 or ZF = 0

## *scasb*

Search for a value in a sequence of [bytes]

- Value -> AL
- *cmpsb* ESI to AL
  - *repne scasb* until byte is found (ESI stores) OR ECX = 0

Search for a value in a sequence of [bytes]
Value come from the AL register
cmpsb ESI to AL
repe scasb until byte is found (ESI stores) OR ECX = 0

## *stosb*

Store a sequence of [bytes]

- Location specified by EDI

*rep stosb* ~ memset in C

- Initialize a buffer where all bytes = AL

Store a sequence of [bytes]
Location specified by EDI
rep stosb ~ memset in C
Initialize a buffer where all bytes = AL

**Table 4-11:** `rep` Instruction Examples

| Instruction | Description |
|---|---|
| `repe cmpsb` (memcmp) | Used to compare two data buffers. EDI and ESI must be set to the two buffer locations, and ECX must be set to the buffer length. The comparison will continue until ECX = 0 or the buffers are not equal. |
| `rep stosb` (memset) | Used to initialize all bytes of a buffer to a certain value. EDI will contain the buffer location, and AL must contain the initialization value. This instruction is often seen used with `xor eax, eax`. |
| `rep movsb` (memcpy) | Typically used to copy a buffer of bytes. ESI must be set to the source buffer address, EDI must be set to the destination buffer address, and ECX must contain the length to copy. Byte-by-byte copy will continue until ECX = 0. |
| `repne scasb` | Used for searching a data buffer for a single byte. EDI must contain the address of the buffer, AL must contain the byte you are looking for, and ECX must be set to the buffer length. The comparison will continue until ECX = 0 or until the byte is found. |

# C Main Method and Offsets

- int main(int argc, char ** argv)
  - argc = # of args on the command line
  - argv = pointer to an array of strings, the arguments

```
filetestprogram.exe -r filename.txt

argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

A main method in C usually has two arguments whose values are determined at runtime

argc - an integer that contains the number of arguments on the command line (including the program name)

argv - a pointer to an array of strings that contain the command-line arguments

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

*Listing 4-1: C code, main method example*

```
004113CE    cmp     [ebp+argc], 3 ❶
004113D2    jz      short loc_4113D8
004113D4    xor     eax, eax
004113D6    jmp     short loc_411414
004113D8    mov     esi, esp
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
004113FC    mov     esi, esp ❸
004113FE    mov     eax, [ebp+argv]
00411401    mov     ecx, [eax+8]
00411404    push    ecx             ; lpFileNam
00411405    call    DeleteFileA
```

*Listing 4-2: Assembly code, C main method parameters*

argc is compared to 3 at (1)
argv[1] is compared to -r at (2) through the use of a strncmp
Notice how argv[1] is accessed:
First the location of the beginning of the array is loaded into eax
Then 4 (the offset) is added to eax to get argv[1].
The number 4 is used because each entry in the argv array is an address to a string, and each address is 4 bytes in size on a 32-bit system.
If -r is provided on the command line, the code starting at (3) will be executed
argv[2] is accessed at offset 8 relative to argv and provided as an argument to the DeleteFileA function.

# Resources

The Art of Assembly Language, 2nd Edition (No Starch Press, 2010)

Intel x86 Architecture Manuals

**“Volume 1: Basic Architecture**

This manual describes the architecture and programming environment. It is useful for helping you understand how memory works, including registers, memory layout, addressing, and the stack. This manual also contains details about general instruction groups.

**Volume 2A: Instruction Set Reference, A–M, and Volume 2B: Instruction Set Reference, N–Z**

These are the most useful manuals for the malware analyst. They alphabetize the entire instruction set and discuss every aspect of each instruction, including the format of the instruction, opcode information, and how the instruction impacts the system.

**Volume 3A: System Programming Guide, Part 1, and Volume 3B: System Programming Guide, Part 2**

In addition to general-purpose registers, x86 has many special-purpose registers and instructions that impact execution and support the OS, including debugging, memory management, protection, task management, interrupt and exception handling, multiprocessor support, and more. If you encounter special-purpose registers, refer to the System Programming Guide to see how they impact execution.

**Optimization Reference Manual**

This manual describes code-optimization techniques for applications. It offers additional insight into the code generated by compilers and has many good examples of how instructions can be used in unconventional ways.”