Topics of Ch-3

- Overview of Intel Assembly Language
- Virtual Machines for Interpreted High-Level Languages
- Representation of Compiled High Level Language Structures in Assembly
- Operating Systems Background
- MS-DOS Internals Related to Malware Case Studies
  - Modern Windows Execution Environment
  - Executable File Formats
  - PE Files
  - Import Address Table

Overview of Intel Assembly Language

- The x86 instruction set architecture is at the heart of CPUs that power our home computers and remote servers for over two decades. Being able to read and write code in low-level assembly language is a powerful skill to have.
- It enables you to write faster code, use machine features unavailable in C, and reverse-engineer compiled code.
- An x86 CPU has eight 32-bit general-purpose registers. For historical reasons, the registers are named {eax, ecx, edx, ebx, esp, ebp, esi, edi}. (Other CPU architectures would simply name them r0, r1, ..., r7.) Each register can hold any 32-bit integer value.
- As a first approximation, a CPU executes a list of instructions sequentially, one by one, in the order listed in the source code.

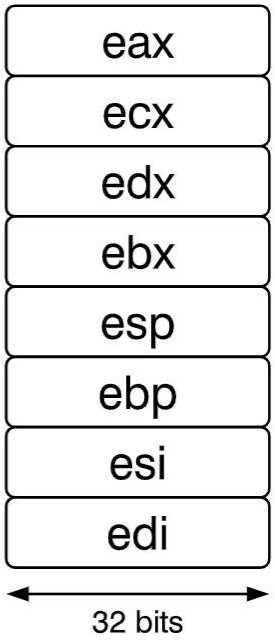
Overview of Intel Assembly Language

Simplified model of x86 CPU

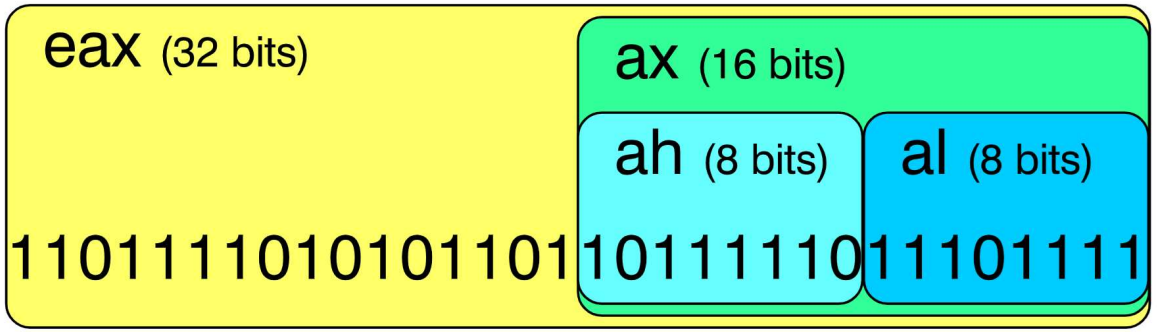
Instruction stream

```
    ...
    movl $0, %eax
    addl %eax, %ebx
    popl %eax
    looptop:
=>  imul %edx
    andl $0xFF, %eax
    cmpl $100, %eax
    jb looptop
    leal 4(%esp), %ebp
    movl %esi, %edi
    subl $8, %edi
    shrl %cl, %ebx
    movw %bx, -2(%ebp)
    ...
```

General-purpose registers



Register aliasing / sub-registers



Intel Assembly Language

- There are actually eight 16-bit and eight 8-bit registers that are subparts of the eight 32-bit general-purpose registers. These features come from the 16-bit era of x86 CPUs, but still have some occasional use in 32-bit mode.
- The 16-bit registers are named {ax, cx, dx, bx, sp, bp, si, di} and represent the bottom 16 bits of the corresponding 32-bit registers {eax, ecx, ..., edi} (the prefix “e” stands for “extended”).
- The 8-bit registers are named {al, cl, dl, bl, ah, ch, dh, bh} and represent the low and high 8 bits of the registers {ax, cx, dx, bx}. Whenever the value of a 16-bit or 8-bit register is modified, the upper bits belonging to the full 32-bit register will remain unchanged.

Intel Assembly Language

The **Many** Assembly Languages

- Most **microprocessors** are created to understand a **binary machine language**
- Machine Language provides means to manipulate internal memory, perform arithmetic, etc.
- The Machine Language of one processor is **not understood** by other processors

MOS Technology 6502

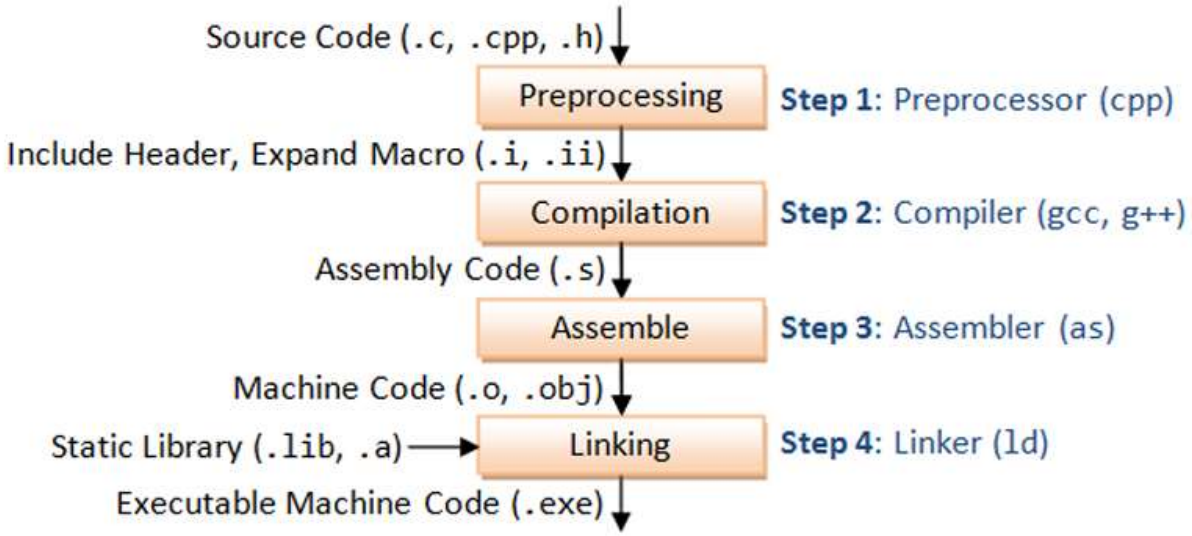
- 8-bit operations, limited addressable memory, **1 general purpose register**, powered notable gaming systems in the 1980s
- Apple IIe, Atari 2600, Commodore
- Nintendo Entertainment System / Famicom

IBM Cell Microprocessor

- Developed in early 2000s, many cores (execution elements), many registers (32 on the PPE), large addressable space, fast multimedia performance, is a **pain** to program
- Playstation 3 and Blue Gene Supercomputer

Intel Assembly Language

Assemblers and Compilers



- **Compiler**: chain of tools that translate high level languages to lower ones, may perform optimizations
- **Assembler**: translates text description of the machine code to binary, formats for execution by processor, late compiler stage
- Consequence: The compiler can **generate assembly code**
- Generated assembly is a pain to read but is often quite fast
- Consequence: A compiler on an Intel chip can generate assembly code for a different processor, **cross compiling**

Intel Assembly Language

- x86-64 Targets Intel/AMD chips with 64-bit word size
  Reminder: 64-bit "word size" ≈ size of pointers/addresses
- Descended from IA32: Intel Architecture 32-bit systems
- IA32 descended from earlier 16-bit systems like Intel 8086
- There is a **LOT** of cruft in x86-64 for backwards compatibility
  - Can run compiled code from the 70's / 80's on modern processors without much trouble
  - x86-64 is not the assembly language you would design from scratch today, it's the assembly you have to code against
  - RISC-V is a new assembly language that is "clean" as it has no history to support (and CPUs run it)
- Will touch on evolution of Intel Assembly as we move forward
- Warning: Lots of information available on the web for Intel assembly programming **BUT** some of it is dated, IA32 info which may not work on 64-bit systems

Intel Assembly Language

x86-64 Assembly Language Syntax(es)

- Different assemblers understand different syntaxes for the same assembly language
- GCC use the GNU Assembler (GAS, command `'as file.s'`)
- GAS and Textbook favor AT&T syntax so **we will too**
- NASM assembler favors Intel, may see this online

AT&T Syntax (Our Focus)

```
multstore:
        pushq   %rbx
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        ret
```

- Use of % to indicate registers
- Use of q/l/w/b to indicate 64 / 32 / 16 / 8-bit operands

Intel Syntax

```
multstore:
        push    rbx
        mov     rbx, rdx
        call    mult2@PLT
        mov     QWORD PTR [rbx], rax
        pop     rbx
        ret
```

- Register names are bare
- Use of QWORD etc. to indicate operand size

Every Programming Language have below listed things

Comments
Statements/Expressions
Variable Types
Assignment
Basic Input/Output
Function Declarations
Conditionals (if-else)
Iteration (loops)
Aggregate data (arrays, structs, objects, etc)
Library System

VM for Interpreted High-Level Languages

- **Interpreter**
- In computer science, an **interpreter** is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:
- Parse the source code and perform its behavior directly;
- Translate source code into some efficient intermediate representation or object code and immediately execute that;
- Explicitly execute stored precompiled bytecode made by a compiler and matched with the interpreter Virtual Machine.

Types of Interpreter

- **Text-based interpreters** directly interpret the textual source of the program. They are very seldom used, except for trivial languages where every expression is evaluated at most once — i.e. languages without loops or functions. Plausible example: a calculator program, which evaluates arithmetic expressions while parsing them.
- **Tree-based interpreters** walk over the abstract syntax tree of the program to interpret it. Their advantage compared to string-based interpreters is that parsing — and name/type analysis, if applicable — is done only once. Plausible example: a graphing program, which has to repeatedly evaluate a function supplied by the user to plot it. All the interpreters included in the L3 compiler are also tree-based.

Virtual machines

- Virtual machines resemble real processors, but are implemented in software.
- They accept as input a program composed of a sequence of instructions. Virtual machines often provide more than the simple interpretation of programs
- They also abstract the underlying system by managing memory, threads, and sometimes I/O. Perhaps surprisingly, virtual machines are a very old concept, dating back to ~1950.
- They have been — and still are — used in the implementation of many important languages, like SmallTalk, Lisp, Forth, Pascal, and more recently Java and C#.
- Since the compiler has to generate code for some machine, why prefer a virtual over a real one?
  - for portability: compiled VM code can be run on many actual machines,
  - for simplicity: a VM is usually more high-level than a real machine, which simplifies the task of the compiler, also a VM is easier to monitor and profile, which eases debugging.

Virtual machines drawbacks & VM Types

- The only drawback of virtual machines compared to real ones is that the former tend to be slower than the latter.
- This is due to the overhead associated with interpretation: fetching and decoding instructions, executing them, etc. Moreover, the high number of indirect jumps in interpreters causes pipeline stalls in modern processors.
- To a (sometimes large) degree, this is mitigated by the tendency of modern VMs to compile the program being executed, and to perform optimizations based on its
- There are two kinds of virtual machines:
  - stack-based VMs, which use a stack to store intermediate results, variables, etc.
  - register-based VMs, which use a limited set of registers for that purpose, like a real CPU.
- For a compiler writer, it is usually easier to target a stackbased VM than a register-based VM, as the complex task of register allocation can be avoided. Most widely-used virtual machines today are stack-based (e.g. the JVM, .NET's CLR, etc.) but a few recent ones are register-based (e.g. Lua 5.0, Parrot, etc.). behavior.

VM Input & Implementation

- Virtual machines take as input a program expressed as a sequence of instructions. Each instruction is identified by its opcode (operation code), a simple number. Often, opcodes occupy one byte, hence the name byte code. Some instructions have additional arguments, which appear after the opcode in the instruction stream.
- Virtual machines are implemented in much the same way as a real processor:
  - the next instruction to execute is fetched from memory and decoded,
  - the operands are fetched, the result computed, and the state updated,
  - the process is repeated.
- Many VMs today are written in C or C++, because these languages are at the right abstraction level for the task, fast and relatively portable.
- The Gnu C compiler (GCC) has an extension that makes it possible to use labels as normal values. This extension can be used to write very efficient VMs, and for that reason, several of them are written specifically for GCC or compatible compilers.

Interpreter & Compiler

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Debug & Compile

- Debugging is the process of removing bugs from the code, typically by stepping through code to identify the bug. A tool that helps one step through code is called a debugger. A debug build is one that has symbols to allow the developer to step through lines of source code while executing.
- Compiling is the process of turning code into machine instructions (or some kind of intermediate language, or bytecode, etc). A tool that does this is called a compiler.

Representation of Compiled High Level Language Structures in Assembly

- When a high-level programming language like C, Java, or Python is compiled into assembly language, it undergoes a transformation where each high-level language construct is translated into one or more assembly language instructions. Here's a brief overview of how some common high-level language structures are represented in assembly:
- **Variables and Data Types**:
  - Variables are typically allocated memory addresses in assembly language.
  - Different data types (integers, floating-point numbers, characters, etc.) are represented using appropriate memory allocation and storage conventions.
- **Control Structures**:
  - **Conditional Statements**: Such as if-else statements, are translated into assembly code using conditional jump instructions (e.g., JMP, JE, JNE).
  - **Loops**: Such as for loops, while loops, and do-while loops, are translated into assembly using conditional jumps and loop control instructions (e.g., JMP, CMP, JZ, JL).
  - Direct memory access may involve instructions like MOV, LOAD, or STORE.

Representation of Compiled High Level Language Structures in Assembly

- **Function Calls**:
  - Function calls involve pushing arguments onto the stack, jumping to the function's entry point, executing the function's code, and then returning control back to the caller.
  - Registers like EBP (base pointer) and ESP (stack pointer) are often used to manage the function's stack frame.
- **Memory Access**:
  - Reading and writing to memory involves loading/storing values from/to memory addresses specified by registers or immediate values.
- **Arithmetic and Logical Operations**:
  - Arithmetic operations (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT) are performed using instructions specific to the CPU architecture.
- **Arrays and Pointers**:
  - Arrays and pointers are implemented using memory addresses and offset calculations.
  - Pointer arithmetic involves adding an offset to a base address to access specific elements of an array or structure.

Representation of Compiled High Level Language Structures in Assembly

- **Structures and Objects**:
  - Structures and objects are typically represented as blocks of memory with specific offsets for each member variable.
  - Accessing structure members involves adding the appropriate offset to the base address of the structure.
- **Error Handling**:
  - Exception handling and error checking may involve specific assembly language constructs and interrupt handling mechanisms provided by the CPU architecture.
- **Optimization**:
  - Compiler optimizations may reorder instructions, eliminate redundant code, and utilize processor-specific features to improve performance.
- Overall, the translation from high-level language constructs to assembly language involves a combination of code generation, memory management, and CPU-specific operations to produce efficient and executable machine code.

Operating Systems Background

- Operating systems (OS) serve as the intermediary between computer hardware and software applications, managing resources and providing a user interface for interaction.
- Here's an overview of operating systems and their key components:
- **Kernel**: At the heart of every operating system lies the kernel, which is responsible for low-level tasks such as managing memory, handling input/output requests, managing processes and threads, and enforcing security policies.
- **Process Management**: The OS manages processes, which are instances of executing programs. This involves creating, scheduling, pausing, resuming, and terminating processes. Process management ensures that each process gets its fair share of CPU time and memory resources.
- **Memory Management**: Memory management involves allocating and deallocating memory for processes, ensuring that each process has enough memory to execute without interfering with others. This includes virtual memory management, which allows processes to use more memory than physically available by swapping data between RAM and disk.

Operating Systems Background

- **File System**: The file system provides a structured way to store, retrieve, and organize data on storage devices such as hard drives, SSDs, and flash drives. It manages files, directories, and metadata, and provides mechanisms for file access, storage, and retrieval.
- **Device Management**: The OS interacts with hardware devices such as keyboards, mice, printers, network cards, and storage devices. Device drivers translate high-level OS commands into low-level instructions that hardware devices understand, enabling communication between software and hardware.
- **User Interface**: Operating systems provide user interfaces through which users interact with the computer. This can include command-line interfaces (CLI), graphical user interfaces (GUI), or a combination of both. The user interface facilitates tasks such as launching applications, managing files, and configuring system settings.
- **Security**: OSes implement security measures to protect the system and user data from unauthorized access, viruses, malware, and other threats. This includes user authentication, access control, encryption, firewall management, and antivirus software integration.

Operating Systems Background

- **Networking**: Operating systems facilitate network communication by providing networking protocols, APIs, and services. This enables processes to communicate with each other across local networks or the internet, facilitating tasks such as file sharing, remote access, and communication between distributed systems.
- **Concurrency and Synchronization**: OSes manage concurrency by allowing multiple processes or threads to execute simultaneously on multicore processors. Synchronization mechanisms such as locks, semaphores, and mutexes are used to coordinate access to shared resources and prevent race conditions.
- **Fault Tolerance and Reliability**: Operating systems incorporate mechanisms to detect, recover from, and mitigate errors and failures. This includes features such as fault tolerance, error handling, and system recovery mechanisms to ensure system stability and reliability.

MS-DOS Internals Related to Malware Case Studies

- MS-DOS (Microsoft Disk Operating System) was one of the earliest operating systems for IBM-compatible personal computers. It played a significant role in the evolution of computer systems and software development. Due to its simplicity and widespread use during the 1980s and early 1990s, MS-DOS became a target for malware authors seeking to exploit its vulnerabilities.
- Here's a brief overview of MS-DOS internals related to malware case studies:
- **File System Vulnerabilities**: MS-DOS primarily used the FAT (File Allocation Table) file system, which had vulnerabilities that malware could exploit. For example, malware could infect executable files by inserting its code into unused sections of the file, altering the file's entry in the directory structure, and modifying the file's execution path.
- **Boot Sector Viruses**: One of the earliest forms of malware targeting MS-DOS was boot sector viruses. These viruses infected the boot sector of storage devices (e.g., floppy disks, hard drives) and executed when the infected device was booted. Boot sector viruses could spread rapidly by infecting other storage devices accessed by the infected computer.

MS-DOS Internals Related to Malware Case Studies

- **Resident Viruses**: MS-DOS allowed programs to stay resident in memory even after their execution completed. Resident viruses exploited this capability to load themselves into memory and remain active, infecting other executable files as they were executed or accessed by the user.
- **Interrupts and Hooks**: MS-DOS provided interrupt vectors and hooks that allowed programs to intercept and handle system calls and hardware interrupts. Malware could use these mechanisms to hijack system functions, gain control over the system, and perform malicious actions without detection.
- **System Configuration**: MS-DOS relied on configuration files such as CONFIG.SYS and AUTOEXEC.BAT to initialize system settings and execute startup commands. Malware could modify these files to load itself into memory during system bootup, ensuring persistence across system reboots.
- **Social Engineering**: Many MS-DOS malware strains relied on social engineering tactics to deceive users into executing infected files. For example, malware authors disguised their programs as legitimate applications or enticing content to trick users into running them, thereby infecting their systems.

MS-DOS Internals Related to Malware Case Studies

- **Polymorphic Code**: Some MS-DOS malware employed polymorphic code techniques to evade detection by antivirus software. Polymorphic viruses modified their code each time they infected a new file or system, making it challenging for antivirus programs to recognize and remove them.
- **Payloads and Damage**: MS-DOS malware payloads varied widely, ranging from harmless pranks to destructive actions such as data corruption, system crashes, and unauthorized access to resources. Some malware displayed messages or animations, while others encrypted or deleted user data.
- While MS-DOS is no longer widely used, studying historical malware case studies targeting MS-DOS provides valuable insights into the evolution of malware techniques, cybersecurity practices

Modern Windows Execution Environment

- The modern Windows execution environment refers to the software and hardware environment in which Windows-based applications run.
- **Windows Kernel**: The core of the modern Windows operating system is the Windows Kernel, which manages system resources, provides hardware abstraction, and facilitates communication between software components. It includes features such as process and memory management, device drivers, and security mechanisms.
- **User Mode and Kernel Mode**: Windows utilizes a protected memory architecture with two distinct privilege levels: user mode and kernel mode. User-mode processes run with limited privileges and access to system resources, while the kernel has unrestricted access to hardware and system resources. This separation enhances system stability and security.
- **NTFS File System**: Windows primarily uses the New Technology File System (NTFS) for disk storage. NTFS supports features such as file and folder permissions, encryption, compression, and journaling, providing reliability, security, and performance for file operations.

Modern Windows Execution Environment

- **Executable Formats**: Windows supports multiple executable file formats, including Portable Executable (PE) for executables (.exe), dynamic link libraries (.dll), and drivers (.sys). These files contain metadata, resources, and code sections, and they adhere to the PE file format specification.
- **Windows Subsystem for Linux (WSL)**: Introduced in Windows 10, WSL allows users to run native Linux command-line tools and applications directly within Windows. WSL provides a Linux-compatible kernel interface and translates Linux system calls to their Windows equivalents, enabling seamless integration between Windows and Linux environments.
- **Virtualization**: Windows supports hardware virtualization technologies such as Hyper-V, which enables users to run multiple isolated virtual machines (VMs) on a single physical host. Hyper-V provides features such as snapshotting, live migration, and integration services for enhanced VM performance.
- **User Interface**: The Windows graphical user interface (GUI) includes features such as the Start menu, taskbar, window management, and desktop environment. Windows provides APIs and frameworks such as Win32, .NET, and Universal Windows Platform (UWP) for developing GUI applications.

Modern Windows Execution Environment

- **Security Features**: Windows includes various security features to protect against malware, unauthorized access, and data breaches. These features include Windows Defender antivirus, Windows Firewall, User Account Control (UAC), Windows Hello biometric authentication, and Secure Boot.
- **Networking**: Windows supports a wide range of networking protocols and services for communication over local area networks (LANs), wide area networks (WANs), and the internet. This includes features such as TCP/IP networking, Wi-Fi connectivity, domain join for enterprise networks, and remote desktop services.
- **Development Tools**: Windows provides a comprehensive set of development tools and SDKs for building applications, including Visual Studio, .NET Framework, .NET Core, and the Windows Software Development Kit (SDK). These tools support various programming languages and frameworks for desktop, web, and mobile development.
- Overall, the modern Windows execution environment offers a robust platform for developing, deploying, and running a wide range of applications, from traditional desktop software to cloud-native and containerized solutions. It continues to evolve with new features, security enhancements, and compatibility improvements to meet the needs of developers, businesses, and end users.

Executable File Formats

- Executable file formats are specific structures that dictate how a program's code, data, and resources are stored in a file and how the operating system should load and execute it. Here are some common executable file formats used across different operating systems:
- **Windows Executable Format (PE/PE32+)**:
  - **Portable Executable (PE)**: This is the standard executable file format used in Windows operating systems. It supports 32-bit and 64-bit architectures.
  - **PE32+**: An extension of the PE format, also known as PE32+ or PE64, supports 64-bit architectures and larger address spaces.
- **Linux Executable Formats**:
  - **ELF (Executable and Linkable Format)**: The ELF format is used by most Unix-like operating systems, including Linux. It supports various architectures such as x86, x86-64, ARM, MIPS, etc.
  - **a.out**: Historically used by some Unix systems, but less common now.

Executable File Formats

- **Mac OS X Executable Formats**:
  - **Mach-O (Mach Object)**: This is the executable file format used by macOS, iOS, watchOS, and tvOS. It supports both 32-bit and 64-bit architectures.
  - **Universal Binary**: A format that allows executable files to contain code for multiple architectures, enabling them to run on different CPU types.
- **Android Executable Format**:
  - **DEX (Dalvik Executable)**: Android apps are compiled into DEX files, which are executed by the Dalvik Virtual Machine or the Android Runtime (ART). DEX files are stored within APK (Android Package) files.
- **Java Executable Format**:
  - **JAR (Java ARchive)**: Java applications are often distributed as JAR files, which contain compiled Java classes and resources. These files are executed by the Java Virtual Machine (JVM).

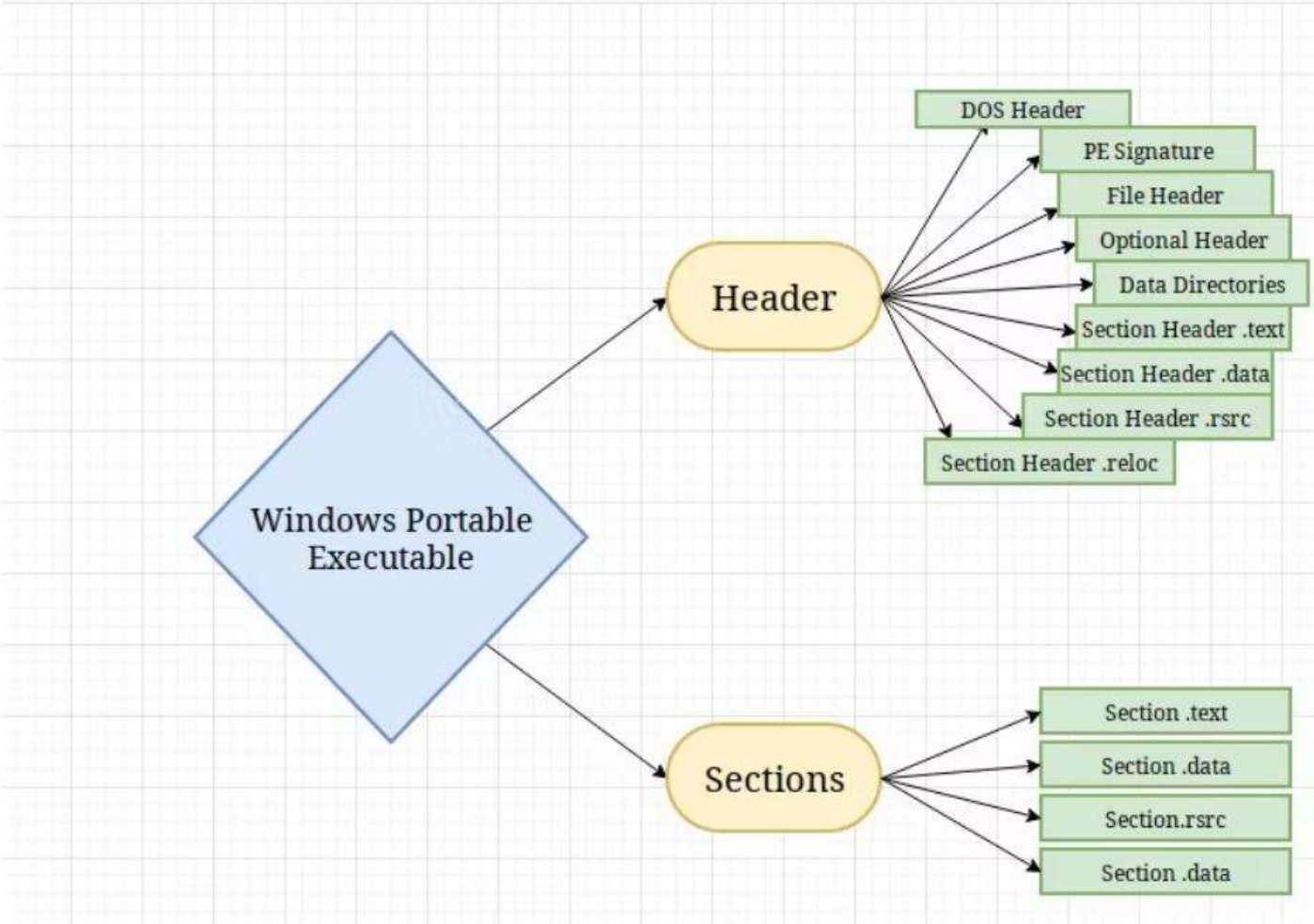
Executable File Formats

- **Web Executable Formats**:
  - **HTML/JavaScript**: Web applications are typically executed within web browsers and are written in HTML, CSS, and JavaScript.
  - **WebAssembly (Wasm)**: A binary instruction format for a stack-based virtual machine, designed to be efficient for both web browsers and standalone environments.
- **Others**:
  - **COFF (Common Object File Format)**: Used by some Unix-like systems for object files and libraries.
  - **COM (DOS Executable)**: The old executable format used by MS-DOS and early Windows versions. It has been largely replaced by the PE format.
  - **XCOFF (Extended COFF)**: Used by IBM AIX operating system for executable files.
- These executable file formats encapsulate various elements required for program execution, including machine code, symbols, relocation information, resources, and metadata. Each format is tailored to the requirements and conventions of the respective operating system or runtime environment.

PE Files

- When you compile the source code to a program, the compiler generates an object file (.obj). This object file contains instructions for the computer in binary format.
- COFF or Common Object File Format is a standardized set of conventions for representing binary instructions. COFF helps in maintaining cross-platform compatibility as all COFF file formats follow the same set of rules and conventions for organizing code and data. Although COFF was originally developed for use on *NIX systems, it is now ubiquitous across all platforms.
- The Windows Portable Executable (PE) file format is a modification of COFF and has been developed to be exclusively used on 32-bit and 64-bit Windows systems.
- It contains sections and headers which provide information about the executable in question and helps the system loader manage data related to the executable. The headers in a PE file help the system loader map the file onto the memory, resolve dependencies such as API exports/imports, manage resources and prepare the file for execution.

The Structure of a Windows Portable Executable



PE Files

- The Portable Executable file format consists of several components, each with a specific purpose. These components include:
  - Section headers, which describe the layout and characteristics of each section of the file The sections themselves, which contain executable code, data, and resources.
  - The PE header, which provides information about the file's overall structure and requirements.
  - The DOS header, which includes a small program that runs when the file is executed on a DOS system.
  - And finally, the PE section headers, which describe each section's location and attributes within the file.
- Overall, these components work together to create a structured format that allows the operating system to properly load, execute, and manage the executable code contained in the file

PE Files

- **DOS Header**
- The first part of a PE file is called the DOS Header. A small amount of executable code is stored in the DOS header which can also be run on a DOS machine.
- This code is also called the MS-DOS stub and is used to throw an error message on systems that don't support the PE file.
- **PE Header**
- The Portable Executable header gives information about the executable, like how big the file is, where the different parts are located, and what resources the executable needs. The PE header also has information about the type of executable, whether it’s a Windows .DLL file or an .EXE.
- **Section Headers**
- Sections are implemented to organize the many components of an executable such as code, data, and resources like text strings, images, etc. The section headers include information regarding the size and location of each section, as well as any associated flags.
- The flags associated with each section header can indicate various attributes of the section, such as whether it is executable, writable, or readable. These flags help the operating system to properly load and manage the contents of each section during program execution.

PE Files

- **Sections**
- The sections themselves comprise the executable's real code, data, and resources. Each segment is aligned to a certain memory boundary and has its own set of attributes that affect how the operating system handles it.

- **Tool:**
- https://petoolse.github.io/petools/

Import Address Table

- To understand how PE files handle their imports, we'll go over some of the Data Directories present in the
  - Import Data section (.idata)
  - The Import Directory Table
  - The Import Lookup Table (ILT) or also referred to as the Import Name Table (INT)
  - The Import Address Table (IAT)
- The Import Address Table (IAT) is a crucial component of the Portable Executable (PE) file format used in Windows operating systems. It is a data structure that stores addresses of functions and procedures imported from external dynamic link libraries (DLLs) or executables.
- The IAT enables the dynamic linking process, allowing a program to call functions located in external modules at runtime.

Import Address Table

- When the application was first compiled, it was designed so that all API calls will not use direct hardcoded addresses but rather work through a function pointer. Conventially this pointer table can be accessed in several ways. Either directly by a call[pointer address] or via a jmp thunk table. Below are examples of each
- By using the pointer table, the loader does not need to fixup all of the places in the code that want to use the api call, all it has to do is add the pointer to a single place in a table and its work is done.

Jmp Thunk Table:

```
...inline app code...
00401002  |. E8 7B0D0000    CALL 00401D82              ; \GetModuleHandleA
...thunk table...
00401D82   $-FF25 4C204000 , JMP DWORD PTR DS:[40204C]  ;  KERNEL32.GetModuleHandleA
...memory address value of pointer...
40204C > FC 3D 57 7C   ;little endian pointer value
```

JMP DWORD PTR DS:[40204C] is the same as jmp 7C573DFC which is GetModuleHandleA fx address. that was filled into this address by the windows loader.

The pointer values can also be used directly in the compiled code without the use of a thunk table:

```
0040103A  |. FF15 7A204000  CALL DWORD PTR DS:[40207A]              ; \MessageBoxA
```