Architecture

In the modern era, most of our application are made on high level languages and then compiled to assembly language. This is because at the physical level our hardware essentially only understands binary (1's and 0's) and assembly provides direct instructions the processor can understand, since it would be very difficult for people to interact with the processors without refering to a manual to know the hex code that runs that instruction. This is why low-level assembly languages were built. By using Assembly devs can write human readable machine instructions which are then "assembled" into their machine code equal, as an example i will put a code snippet and show its equals

Assembly: add rax, 1
Shellcode: 4883C001
Binary: 01001000 10000011 11000000 00000001

All 3 of those items are synonymous with eachother but only 1 is easy to read, machine code is frequently represented as shellcode, this is a hex representation of machine code bytes

High-Level and Low-Level

As more and more different processor designs came to existence therefore came multiple assembly instructions which meant no uniformity in making applications, this was like building a pool for a property that you do not know the dimensions of. Then, in the 1970's high languages like C were developed to make it possible to write a single and easy code that would work on any processor without rewriting it for each processor. This was made possible by creating compilers for each language.

When high-level code is compiled it is eventually converted into assembly instructions for the processor and then assembled into machine code

Now we have applications like python, PHP, Bash, JS, and many others which are not compiled but are instead interpreted at run time via the use of pre-built libraries to run the instructions. These libs are often written and compiled in other High Level Langs like C or C++. This means that the way these runtime languages work is by using compiled libs to run the command which uses the assembly code/ machine code to perform all the instructions necessary to run this command on the processor

Compilation Stages

Using the classic example of "Hello World!" we will demonstrate how python eventually end up as Binary machine code:

```
print("Hello World!")
```

This is a simple example that will, when run, print "Hello World!" to our screen. As it is run with the respective library it would look like this in linux

```
#include <unistd.h>

int main(){
    write(1,"Hello World!",12);
    _exit(0);
}
```

This is still quite easy to read, this is how it would look when it is compiled at runtime, next it is put into Assembly instructions

```
mov rax, 1
mov rdi, 1
mov rsi, message
mov rdx, 12
syscall

mov rax, 60
mov rdi, 0
syscall
```

now it becomes a little awkward to read but with the proper knowledge it becomes very easy to understand, now we will see this gets converted into shellcode which is very odd to try to read

```
48 c7 c0 01
48 c7 c7 01
48 8b 34 25
48 c7 c2 0d
0f 05
```

```
48 c7 c0 3c
48 c7 c7 00
0f 05
```

and then lastly it will be converted to the binary instructions

```
01001000 11000111 11000000 00000001
01001000 11000111 11000111 00000001
01001000 10001011 00110100 00100101
01001000 11000111 11000010 00001101
00001111 00000101

01001000 11000111 11000000 00111100
01001000 11000111 11000111 00000000
00001111 00000101
```

Computer Architecture

Most computers today are built on something know as the Von Neumann Architecture which was developed back in 1945 by Von Neumann to allow for the creation of "General-Purpose Computers" according to the image that Alan Turing put forth. Alan Turing based his ideas on Charles Babbage's mid-19th century "Programmable Computer" concept. All of these individuals were mathmaticians

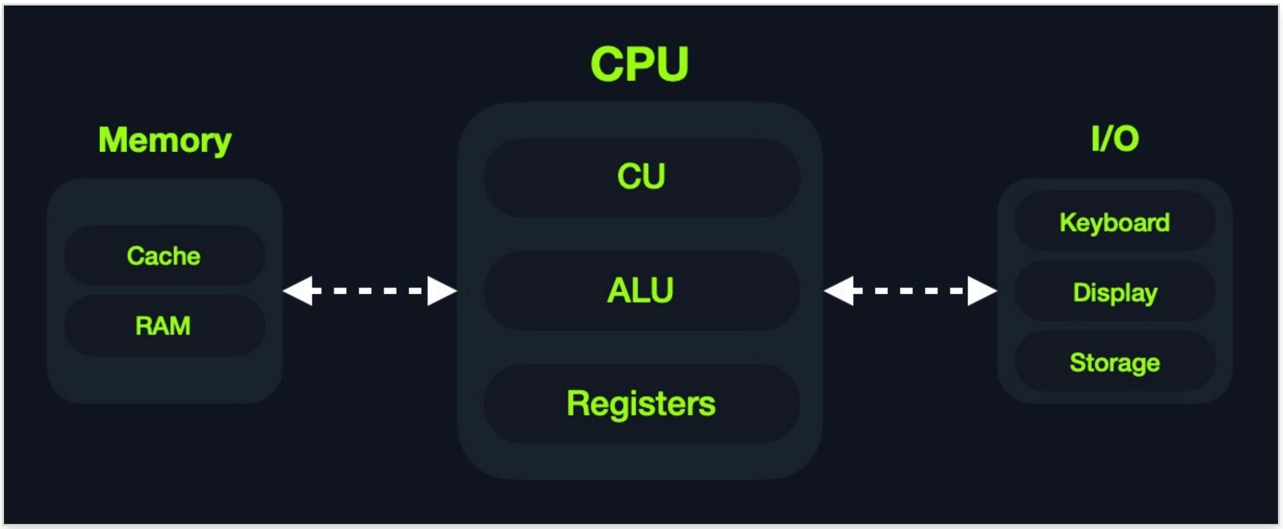
This arch executes machine code to perform specific algorithms and is mainly comprised of the following elements:

- Central processing unit(CPU)
- Memory Unit
- Input/Output devices
    - mass storage unit
    - keyboard
    - display

on top of this the CPU is made of 3 main parts

- Control unit (CU)
- Arithmetic/Logic Unit(ALU)

- Registers



This is a very old and basic description it is the basis of most modern computers

Assembly mainly works with CPU and memory and in order to learn this it will be of the utmost importance to understand how the computer operates. This is because we need know how fast and expensive each function is. On top of this the more advanced the binary exploitation, the more it requires proper understanding of the PC arch. This all becomes especially true with ROP and Heap exploits

PC Components

Memory

The computer memory is the area in which temporary data and instructions of currently running programs are located. The computer memory is also refered to as the primary memory as it is the primary location the CPU uses to retrieve and process data. It does this frequently (billions of times a second) so the memory must be fast in storing and retrieving data and instructions. There are 2 main types of memory:

- cache
- Random access memory (RAM)

Cache

Cache memory is usually within the CPU itself and is therefore extremly fast in comparison to RAM because it runs at the same clock speed as the CPU. However, it is very limited in size and very sophisticated, and very expensive to make due to it being so close to the CPU core.

Since RAM clock speed is much slower than CPU cores, in addition to it being far from the CPU, if the CPU had to wait for the RAM to recieve instructions it would run at way lower clock speeds, this is why we have cache memory because it allows the CPU to access upcoming instructions much faster.

There are usually 3 levels of cache memory depending to the closeness to the CPU core:

- Level 1 Cache (L1)
    - Usually in kilobytes and the fastest memory available second only to the registers
- Level 2 Cache (L2)
    - Usually in megabytes, extremely fast (slower than L1) and shared among all CPU cores
- Level 3 Cache (L3)
    - Usually in megabytes (larger than L2), faster than RAM but slower than L1 and L2. (not all CPU's use L3)

RAM

RAM is much larger than caches going anywhere from gigabytes to terabytes and is quite far from the CPU and is therefore much slower than cache mem and accessing data from RAM requires many more instructions

As an example getting instructions from a register takes one clock cycle, a few clock cycles for cache, and around 200 clock cycles for RAM. When we consider the billions of times this is done in a second it is quite significant

In the past with 32-bit systems we were limited with the memory addresses from 0x00000000 to 0xffffffff which meant that we were limited to a RAM size of 2^32 which is only 4GB, at which point we run out of unique memory addresses. With 64-bit addresses, the range is now up to 0xffffffffffffffff, with a theoretical max RAM size of 2^64 bytes which is around 18.5 exabytes or 18.5 million terabytes so we will not run out of mem addresses anytime soon

When a program is run all of its data and instructions are moved from storage to RAM to be accessed when needed by the CPU. This is because accessing them from the storage device is much slower and requires more data processing items. When the program is closed its data is removed or made available to reuse from the RAM

The 4 main segments are as follows:

- Stack
    - Has a last in first out design and is fixed in size, data in it can only be accessed in a specific order by pushing and popping data
- Heap
    - Has a hierarchal design which makes it much larger and more versatile in storing data as data can be stored and retrieved in any order. This does make the heap slower than the stack
- Data
    - has 2 parts: Data, which is used to store variable and .bss, which holds unassigned variables (ie. buffer mem for later allocation)
- Text
  - main assembly instructions are loaded into this segment to be fetched and executed by the CPU
  Even though this segmentation applies to the entire RAM, each app is allocated its virtual memory when it is run. This means that each app has its own stack, heap, data, and text

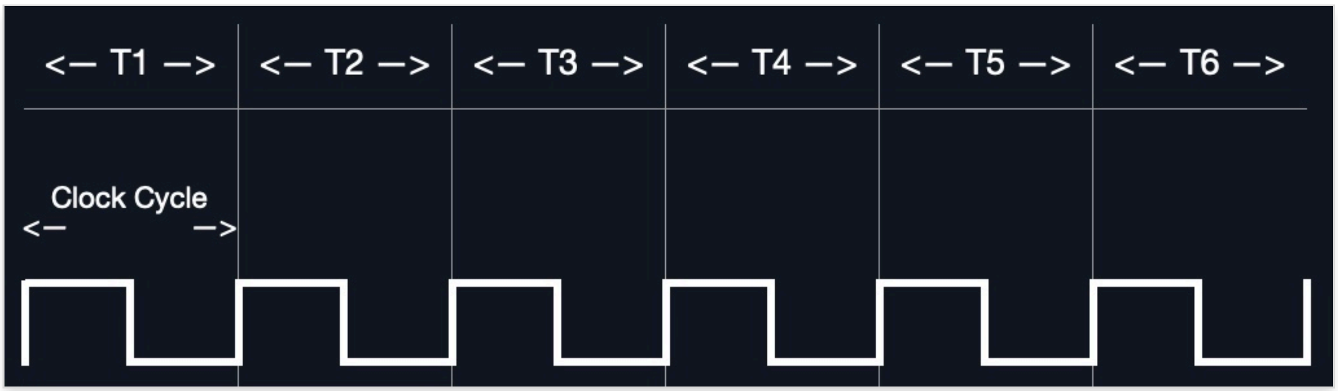
CPU Arch

In the CPU the CU (control unit) is responsible for moving and controlling data and the ALU (arithmetic/logic unit) performs arithmetic and logical calculations based on the program instructions. The efficiency of CPU's depends on the Instruction Set Architecture (ISA), there are many ISAs with each having its own method of processing data. RISC arch is based on processing more simple instruction which takes more cycles but each cycle is shorter and takes less power. On the contrary CISC arch is based on fewer but more complex instructions which can finish items in fewer cycles but each cycle is longer and requires more power.

Clock Speed and Clock Cycle

Each CPU has a clock speed that indicates its overall speed, with each "tick" of this clock representing a basic instruction being processed and the frequency indicating its temporal value, the cycles per second unit of measurement is Hertz, so if we had a CPU of a speed of

3.0GHz it would then run 3 billion cycles per second per core. Here is a basic visualization

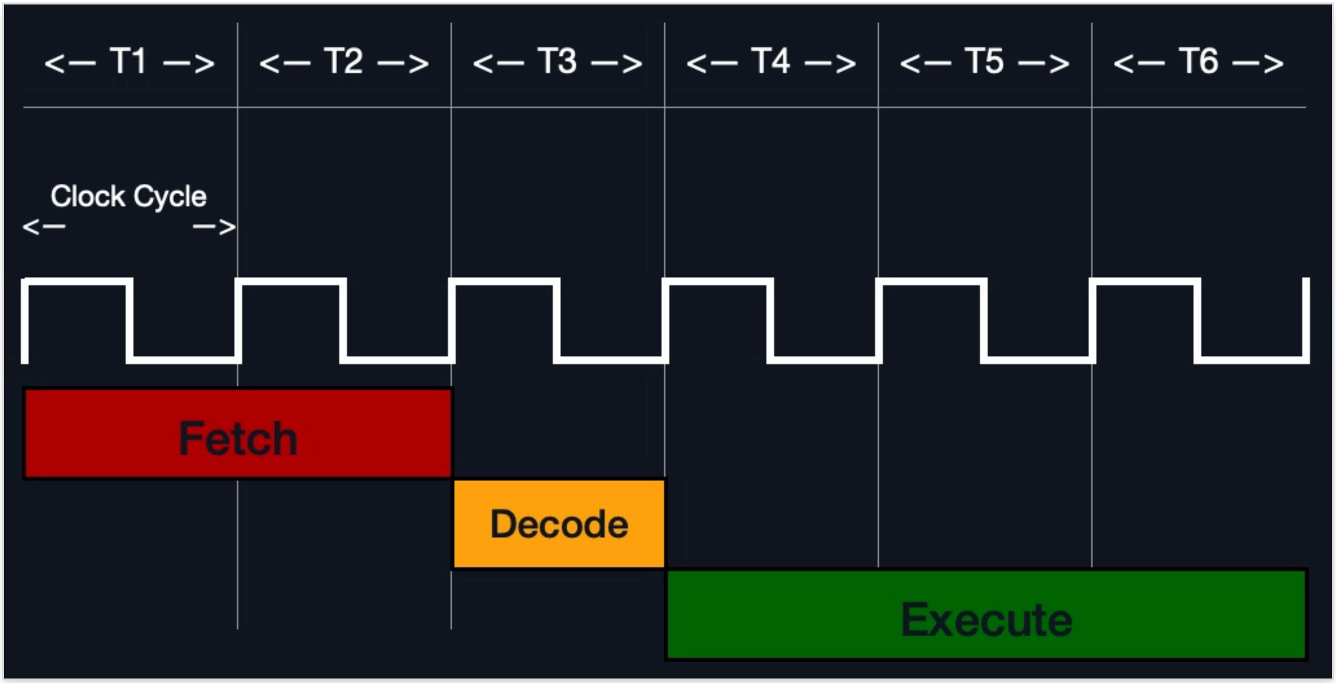


Instruction Cycle

An instruction cycle is not the same as clock cycle, a clock cycle is the process it takes a CPU to process a single machine instruction. It contains the following 4 stages:

- Fetch
    - Takes the next instruction address from the **Instruction Address Register**(IAR), which tells it where the next instruction is located
- Decode
    - Takes instruction from the IAR and decodes it from binary to see what is required to be executed
- Execute
    - Fetch instruction operands from register/memory and process instruction in the ALU or CU
- Store
  - Store new value in the destination operand
  All of the stages in the instruction cycle are carried out by the Control Unit, except when arithmetic instructions need to be executed "add, sub, ..etc", which are executed by the ALU.

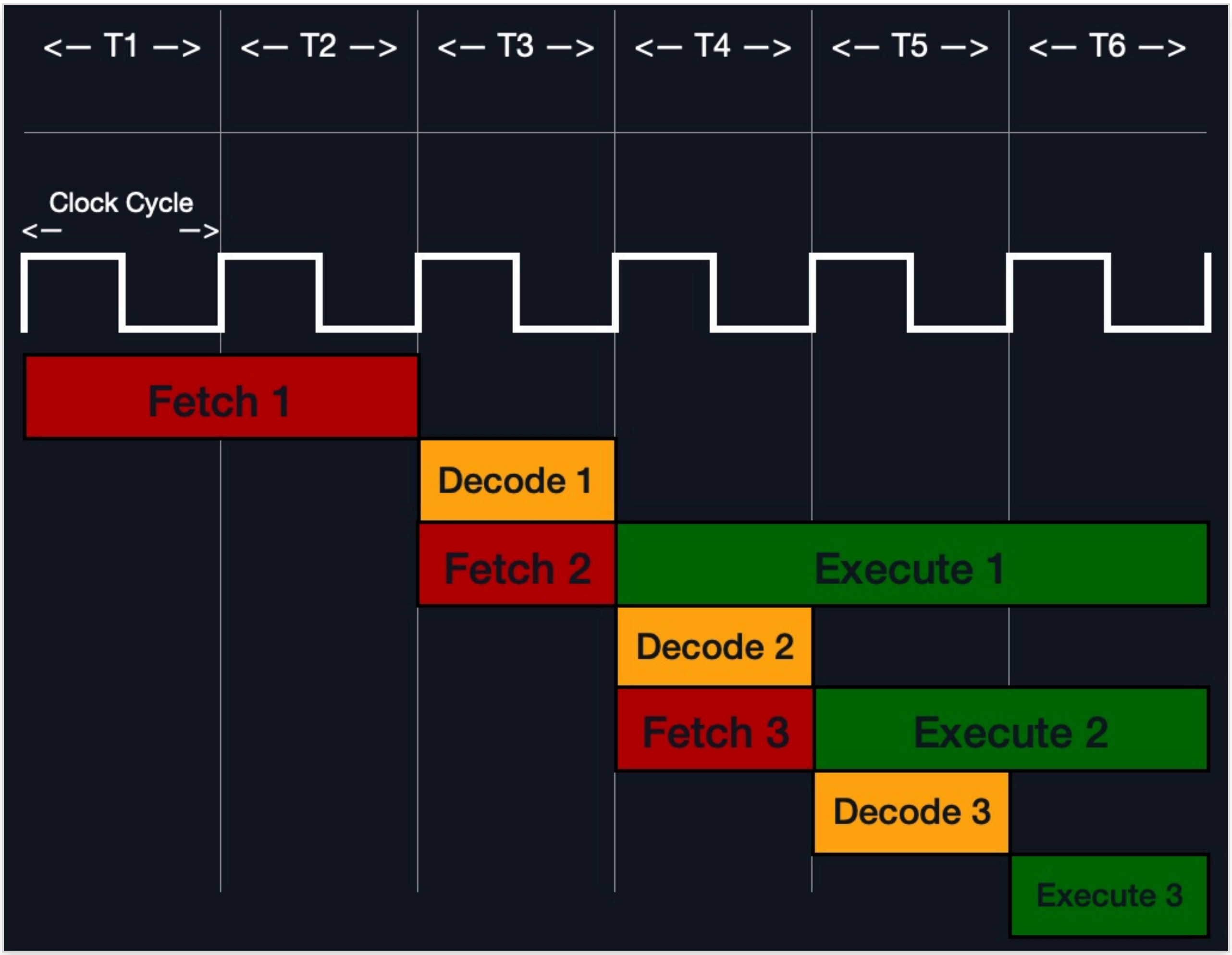
Each instruction cycle takes multiple clock cycles to finish, dependent upon the CPU arch and the complexity of the instruction. Once an instruction cycle completes the CU increments to the next instruction and runs the same cycle on it and this repeats



As an example of all of this, if we had the assembly instruction `add rax, 1` this is how it would run in pseudo-code:

1. Fetch the instruction from the `rip` register, `48 83 C0 01` (in binary)
2. Decode '`48 83 C0 01`' to know it need to perform an `add` of `1` to the value of `rax`
3. Get the current value at `rax` (by the CU), add 1 to it (via the ALU)
4. Store the new value back to `rax`
   In the more archaic processors this would have been ran sequentially which means that they would have to finish one instruction in order to begin the next. Now, processors can handle multiple instructions in parallel by having multiple instruction/clock cycles running

at the same time which is done by having a multicore and multithread design



Processor Specific

As we spoke about earlier, each processor handles a different set of instructions. For example, while intel processor based on the 64-bit x86 architecture may interpret `4883C001` as `add rax, 1` an ARM processor would translate it to `biceq r8, r0, r8, asr #6` instruction. This shows that the same machine code performs an entirely different instruction on each processor.

This is because each processor has a different assembly structure (ISA). For example the add instruction we have been using `add rax, 1` is for intel x86 64-bit processors, the same instruction if it was for an ARM processor would be `add r1, r1, 1`

Building upon this, a single ISA may have several syntax interpretations for the same assembly code. For example, the `add` instruction is based on x86 arch, which is supported by processors like Intel, AMD, and legacy AT&T processors. The instruction is written as `add rax, 1` with intel syntax and as `addb $0x1,%rax` for AT&T syntax

Instruction Set Architecture(ISA)

An ISA specifies the syntax and semantics of the assembly language on each arch. It is not just a different syntax but is built in the core design of a processor, as it affects the way and order instructions are executed and their lvl of complexity. ISA mainly consists of the following components

Component	Description	Example
Instruction	The instruction to be processed in the opcode operand_list format. There are usually 1,2, or 3 comma separated operands	add rax, 1; mov rsp, rax; push rax
Registers	Used to store operands, addresses, or instructions temporarily	rax; rsp; rip
Memory Addresses	The address in which data or instructions are stored. May to memory or registers	0xffffffffaa8a25ff; 0x44d0; $rax
Data Types	The type of stored data	byte; word; double word

These are the main components that distinguish different ISAs and assembly languages. The 2 main instruction sets that are widely used

1. Complex Instruction Set Computer (CISC)- Used in Intel and AMD processors in most computers and servers
2. Reduced Instruction Set Computer (RISC)- Used in ARM and Apple processor, in most smartphones, and some laptops

CISC

Was one of the earliest ISAs ever developed and as the name suggests it favors more complex instructions over the amount of overall instructions. This is done by combining minor instructions into complex instructions for example, if we add 2 registers with `add rax, rbx` instruction a CISC processor can do this in a single instruction cycle (Fetch-Decode-Execute-Store) without a need to split it into multiple instructions to fetch `rax` and then fetch `rbx`, then add them, and store them in `rax`. Each of those steps would take its own instruction cycle which is not how CISC favors to run things. The reasons that this design of ISA was created has to do with

1. To enable more instructions to be executed at once by designing the processor to handle more advanced instructions
2. In the past components like memory and transistors were limited so shorter programs meant more complex programs

Another item to speak on is that CISC processors are more complex than RISC because of its design to execute a vast array of different complex instructions with their own unit dedicated to executing it. As the instructions get more complex, the instruction cycle takes more clock cycles to complete

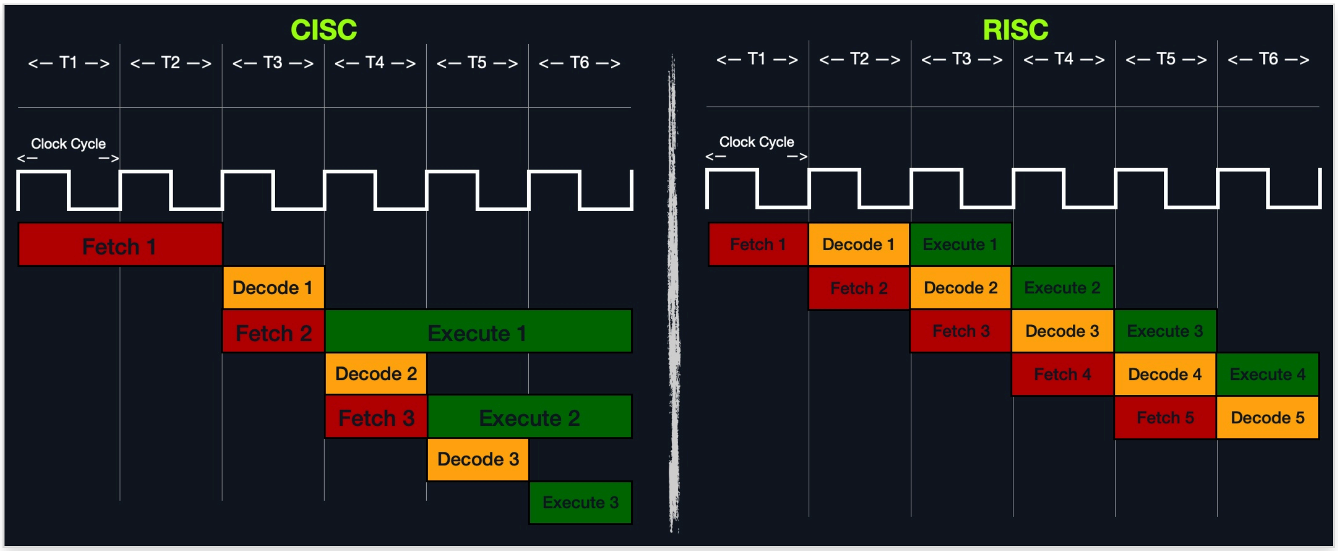
RISC

RISC approaches instructions by splitting instructions into minor ones meaning the CPU is only designed to handle the simple instruction, an example of this would be something like `add r1, r2, r3` which would then fetch `r2` then `r3`, then add them, then store them in `r1` . With each of these events taking an entire instruction cycle to complete which means that every program run takes a large amount of instructions and therefore longer assembly code.

The crux of this is that RISC processors support a very limited amount of instructions with the ability to support only about 200 instructions vs. CISC which supports about 1500. Putting this all together RISC takes complex instructions and chunks them into simple instructions

To add to this insanity that is the processor, it is possible to have a general use computer that can run off of only one instruction, with that one instruction being a very complex instruction.

The advantage to RISC is the fact that every instruction in the instruction cycle takes exactly one clock cycle



Registers, Addresses, and Data Types

Before we dive into the world of assembly language we have to know a few key elements which are *Registers, Memory Addresses, Address Endianness, and Data Types*

Registers

Each CPU core has a set of registers which are the fastest components in the computer and they are built in the CPU core. These registers are limited in size and can only hold few bytes of data at a given time. For the purposes of learning assembly language we will only focus on the necessary registers and the ones needed for binary exploitation. There are 2 main types of registers that I will go through: Data registers and pointer registers

Data Registers	Pointer Registers
rax	rbp
rbx	rsp
rcx	rip
rdx	
r8	
r9	
r10	

Data Registers

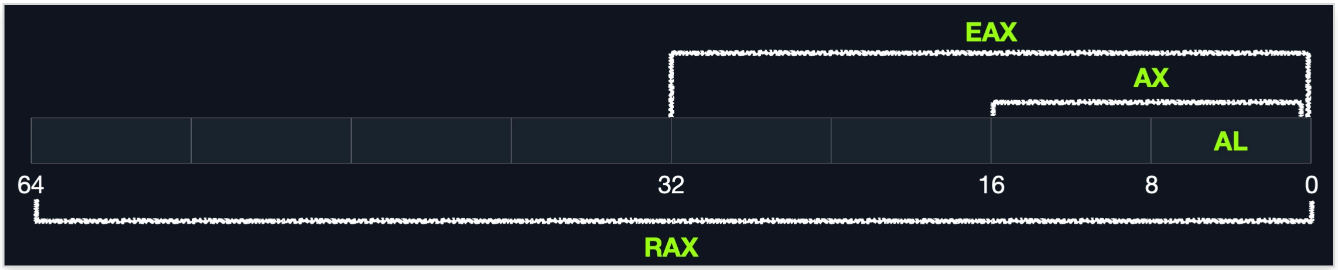
Used for storing instructions/syscall arguments, we see above the primary data registers that will be covered. On top of these there is also the `rdi` and `rsi` which are usually used for the instruction destination and source operands. The `r8` `r9` and `r10` registers are secondary registers that we can use when the aforementioned ones are in use.

Pointer Registers

Store specific important address pointers. The main pointer registers are the base stack pointer(`rbp`) which points to the beginning of the stack, then there is the current stack pointer(`rsp`) which points to the current location within the stack(which is the top of the stack), and lastly the instruction pointer(`rip`) which holds the address of the next instruction

Sub-Registers

Each 64-bit register can be further broken down into smaller sub-registers containing the lower bits, at one byte(8 bits), 2 bytes(16 bits), and 4 bytes(32 bits). This is important to note because each sub-register can be used and accessed on their own meaning we do not have to use the full 64-bits if we have a smaller amount of data



Sub-registers can be accessed as:

Size in bits	Size in bytes	Name	Example
16-bit	2 bytes	the base name	ax
8-bit	1 byte	base name and/or ends with l	al
32-bit	4 bytes	base name + starts with an e prefix	eax
64-bit	8 bytes	base name + starts with an r prefix	rax

The essential sub-registers that we need to know for the x86_64 architecture are as follows

Description	64-bit Register	32-bit Register	16-bit Register	8-bit Register
DATA/ARGUMENT REGISTERS				

Description	64-bit Register	32-bit Register	16-bit Register	8-bit Register
Syscall Number/Return Value	`rax`	`eax`	`ax`	`al`
Callee Saved	`rbx`	`ebx`	`bx`	`bl`
1st arg-Destination Operand	`rdi`	`edi`	`di`	`dil`
2nd arg-Source Operand	`rsi`	`esi`	`si`	`sil`
3rd arg	`rdx`	`edx`	`dx`	`dl`
4th arg-Loop counter	`rcx`	`ecx`	`cx`	`cl`
5th arg	`r8`	`r8d`	`r8w`	`r8b`
6th arg	`r9`	`r9d`	`r9w`	`r9b`
POINTER REGISTERS				
Base Stack Pointer	`rbp`	`ebp`	`bp`	`bpl`
Current/Top Stack Pointer	`rsp`	`esp`	`sp`	`spl`
Instruction Pointer 'call only'	`rip`	`eip`	`ip`	`ipl`

These are by no means all the registers located in the CPU, these are just the ones that are of interest to use. Just as an example there is the `RFLAGS` which maintains various flags used by the CPU like the zero flag `ZF` which is used for conditional instructions

Memory Addresses

As we spoke of earlier x86 64-bit processors have 64-bit wide addresses which means they range from 0x0 to 0xffffffffffffffff, this means that a program's addresses must fall within this range. With this in mind it is important to note that RAM is partitioned into various regions like the Stack, the Heap, as well as other program and kernel specific areas. Each of the regions have specific read, write, execute permissions that determine whether we can read from it, write to it, or even call an address in it.

Whenever we have an instruction that goes through the instruction cycle to be executed, the first step is fetch the instruction from the address it is located at. There are many types of address fetching (aka addressing modes) in the x86 arch:

Addressing Mode	Description	Example
Immediate	The value given within the instruction	`add 2`
Register	The register name that holds the value is given in the instruction	`add rax`
Direct	The direct full address is given in the instruction	`call 0xffffffffaa8a25ff`
Indirect	A reference pointer is given in the instruction	`call 0x44d000` or `call [rax]`
Stack	Address is on the top of the stack	`add rsp`

The thing to note on this table is that the lower it comes in the order, the slower it is to be fetched hence, immediate is faster than register

Even though speed is not of the utmost concern to us we should still know where and how each address is located which will help us with binary exploitation like buffer overflows, ROP or heap exploitation

Address Endianness

This section gets a bit complex so it may be long winded

Address Endianness is the order of its bytes in which thar are stored or retrieved from memory. There are 2 main types of endianness: `Little-Endian` and `Big-Endian`. With Little-Endian processors, the little-end byte of the address is filled/retrieved first right-to-left, while Big-Endian processors the big-end byte is filled/retrived first left-to-right

As an example of this, if we had the address `0x0011223344556677` to be stored in memory, little endian processors would store the `0x00` byte on the right most bytes, and then the `0x11` byte would be filled after it, so it then becomes `0x1100`, then the `0x22`byte which makes it become `0x221100` and so on so forth. This means that by the end of this process we would have `0x7766554433221100` which we can see is the reverse of the original value, which seems like it would cause confusion but in reality the processor uses the same process to retrieve this address meaning that it would return to the original value. To put this into an easy to understand example, if we had a 2 byte integer like 426 its binary representation would be `00000001 10101010` , when this value is stored in the little endian method its value would change to 43521 or `10101010 00000001` .

With the Big-Endian processors it would store these bytes as `00000001 10101010` left to right vs the little endian processors which would store this as`10101010 00000001` left to right. The processor has to use the same endianness as the storage or else the wrong value is put forth, this means that to order of byte storage makes a massive difference.

It is of the utmost importance that know that our bytes are stored in memory from right to left because if we were to push an address or string in assembly we would have to reverse it. If we wanted to store the string `Hello` we would have to push its bytes in reverse: o,l,l,e, and finally H

This may seem unnecessary but this has many advantages when processing data, like being able to retrieve a subregister without having to go through the entire register or being able to perform arithmetic in the correct order right-to-left

Data Types

Lastly, in the x86 arch, there are many types of data sizes which can be used with a multitude of instructions, these are the most common data types that will be used

Component	Length	Example
byte	8 bits	0xab
word	16 bits-2bytes	0xabcd
double word(dword)	32 bits-4bytes	0xabcdef12
quad word(qword)	64 bits-8bytes	0xabcdef1234567890

It is important that we use a variable with a certain data type or use a data type with an instruction, both of the operands should be of the same size, as an example, we cant use a variable defined as byte with `rax` because `rax` is 8 bytes while byte is only 1 byte. This means that we should use `al` with the byte operand as they are of the same size, here is a table that explains this hierarchy

Sub-Register	Data Type
al	byte
ax	word

Sub-Register	Data Type
eax	dword
rax	qword