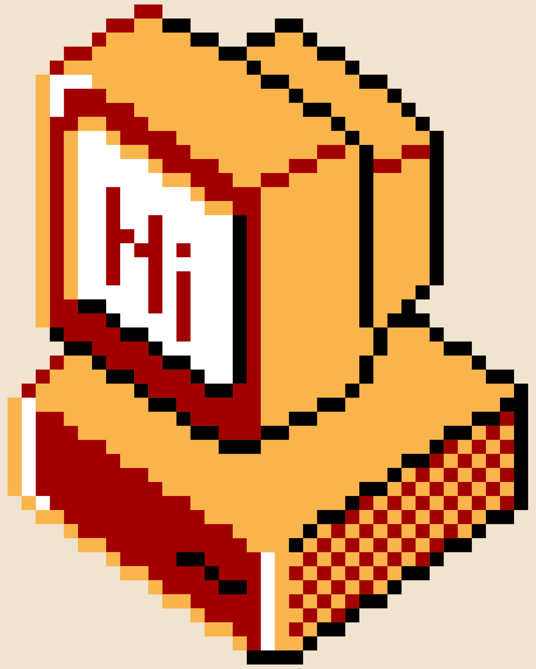


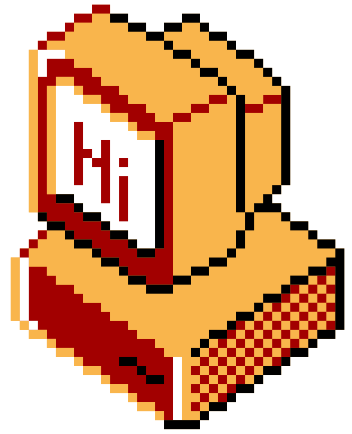
PERTH SOCIALWARE

0x02:

Reverse Engineering Workshop

Part 1

$ ~/: groups "socialware"



Welcome!

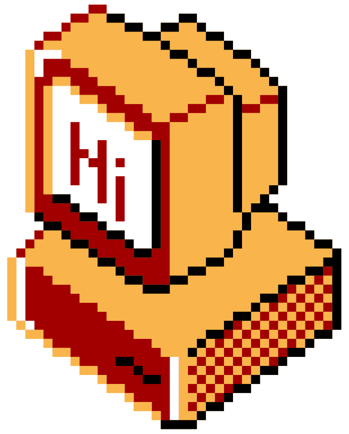
About & Aims

Enjoy!

$ ~/: groups "socialware"

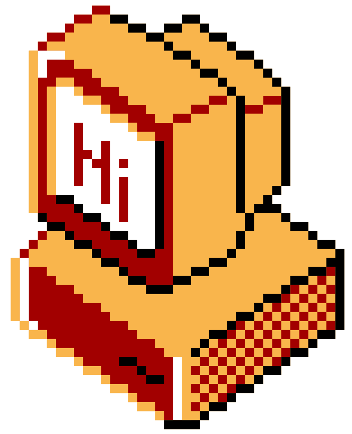
Thanks to **Rio Tinto** for the food and venue!

$ ~/: cat ./housekeeping



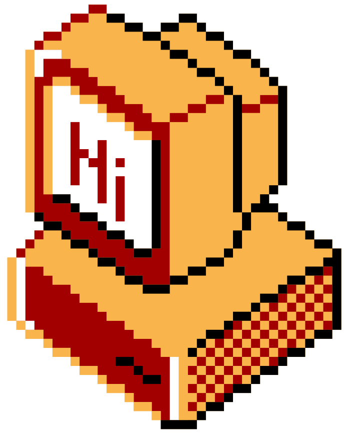
- Ensure induction is completed!

- Don't break stuff
- If you break stuff tell us
- Be respectful
- Have fun.



Acknowledgement of Country

$ ~/: whoami



Emu Exploit

- We are a competitive hacking team current rank #1 in Australia on CTFtime.org
- Founded in 2021, the team consists of many highschoolers as well as industry professionals

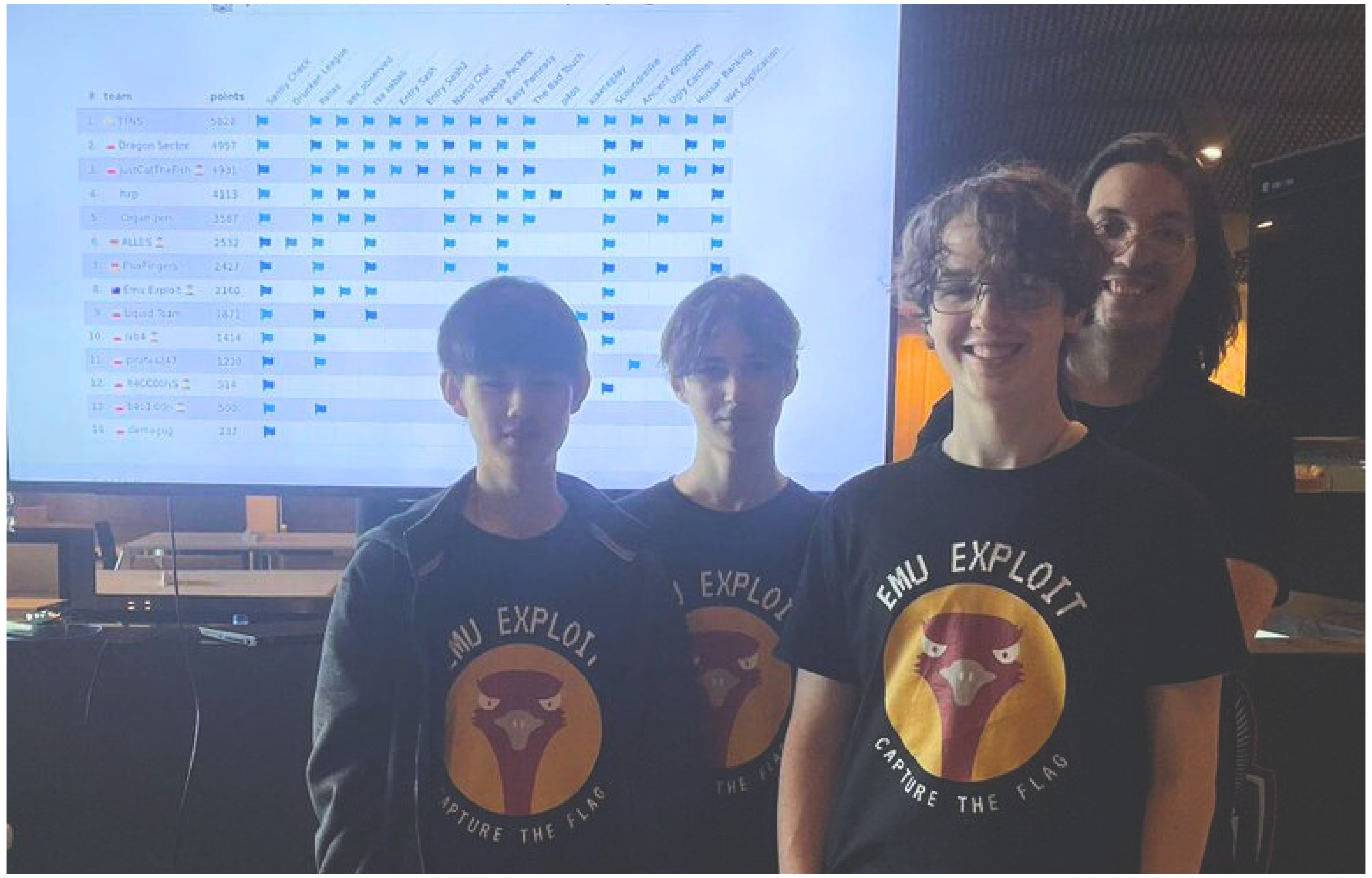
Today's Presenters

- Riley (toasterpwn) – Captain
- Rainier (teddy / TheSavageTeddy) – Vice Captain
- Torry (torry2)
- Orlando (q3st1on)
- Avery (nullableVoidPtr)

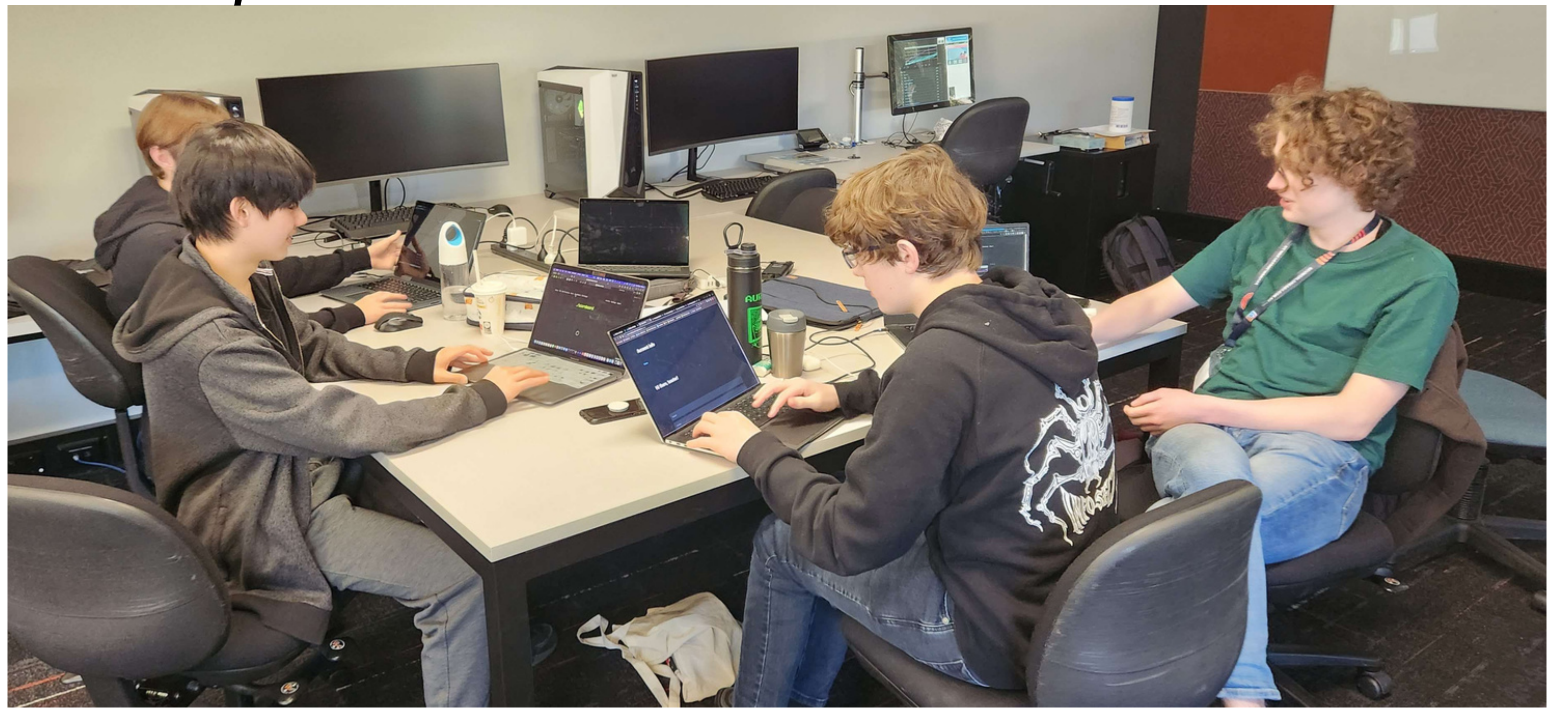
Emu Exploit at Pecan CTF 2023

$ ~/: whoami

Perth Socialware 0x01

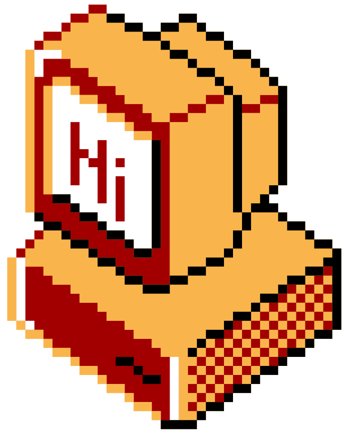


p4CTF in Katowice, Poland



Pecan CTF 2023

$ ~/: cat content



Timeline:

Presentation [6:00] -> Workshop [6:30] -> End [8:00]

- How does a CPU work? (Fetch Execute Cycle)
- What is memory?
- What is assembly (asm)
- Assembly Programs

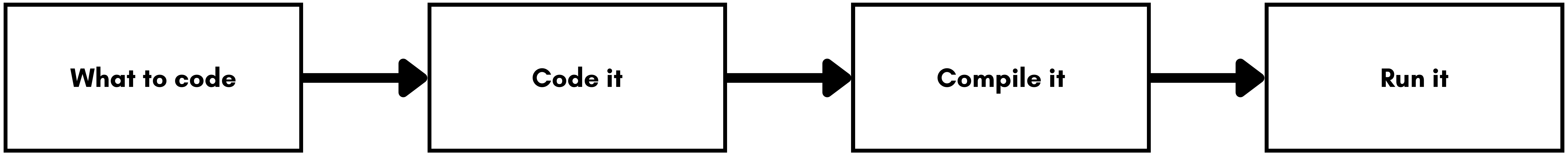
- Workshop Filedrop

$ ~/: Reverse Engineering

First of all, what is **reverse engineering**?

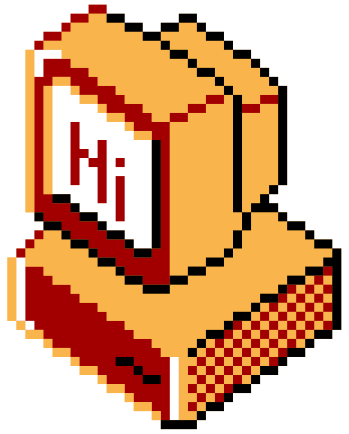
Consider the process of building a program:

- You figure out what you want to code
- You implement it in code
- You compile the code
- You run the code



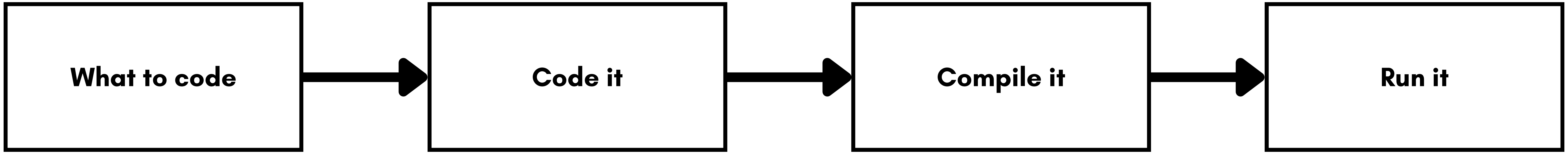
Information is lost at every stage!

$ ~/: Reverse Engineering

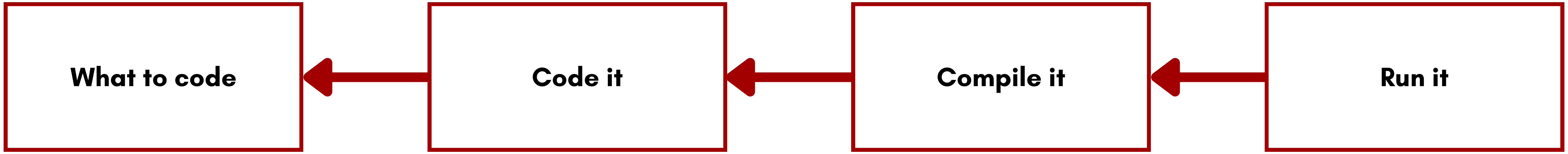


How can we get back the information that was lost?

This is what reverse engineering is!

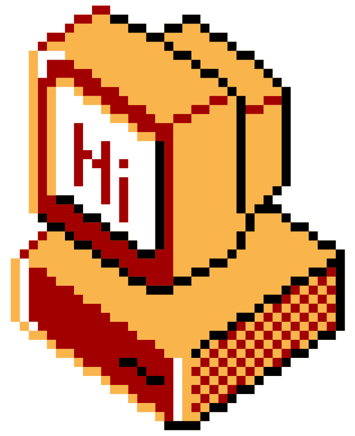


Information is lost at every stage!



How can we get it back?

$ ~/: Reverse Engineering



Very briefly – types of analysis

Static Analysis

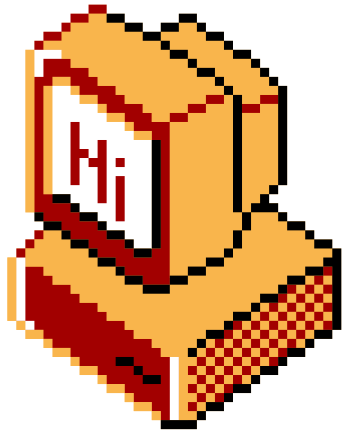
- Look at the compiled code, figure out what it's doing from there – no running the code

Dynamic Analysis

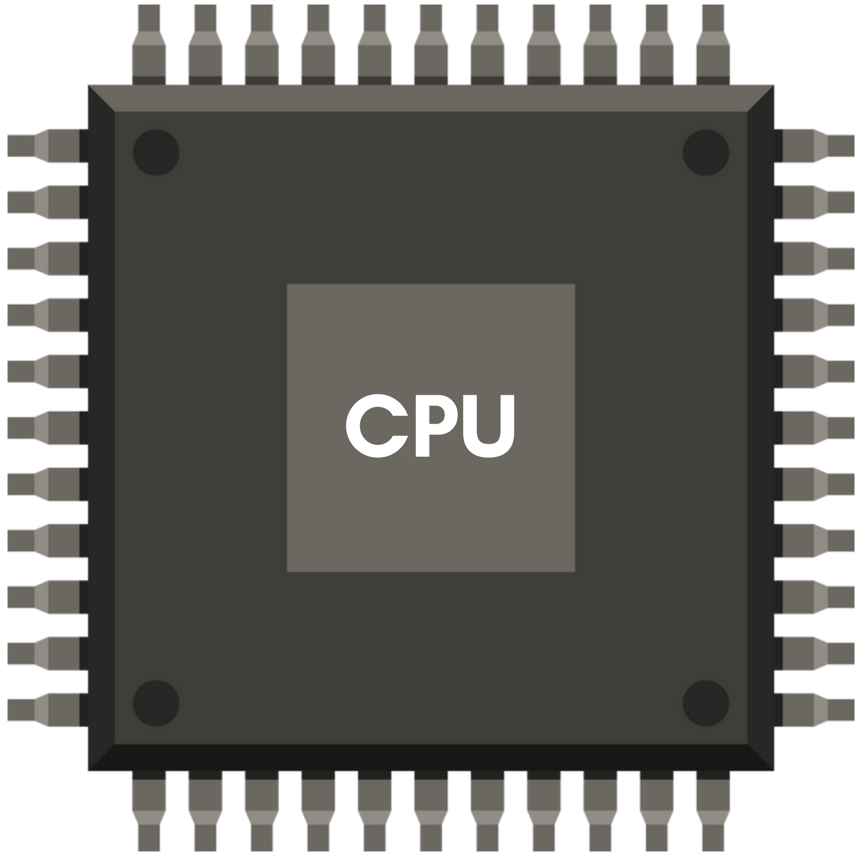
- Run the code to see what it does

We will only be doing **static analysis** today, but the knowledge also translates over to dynamic analysis!

$ ~/: The Fetch-Execute Cycle



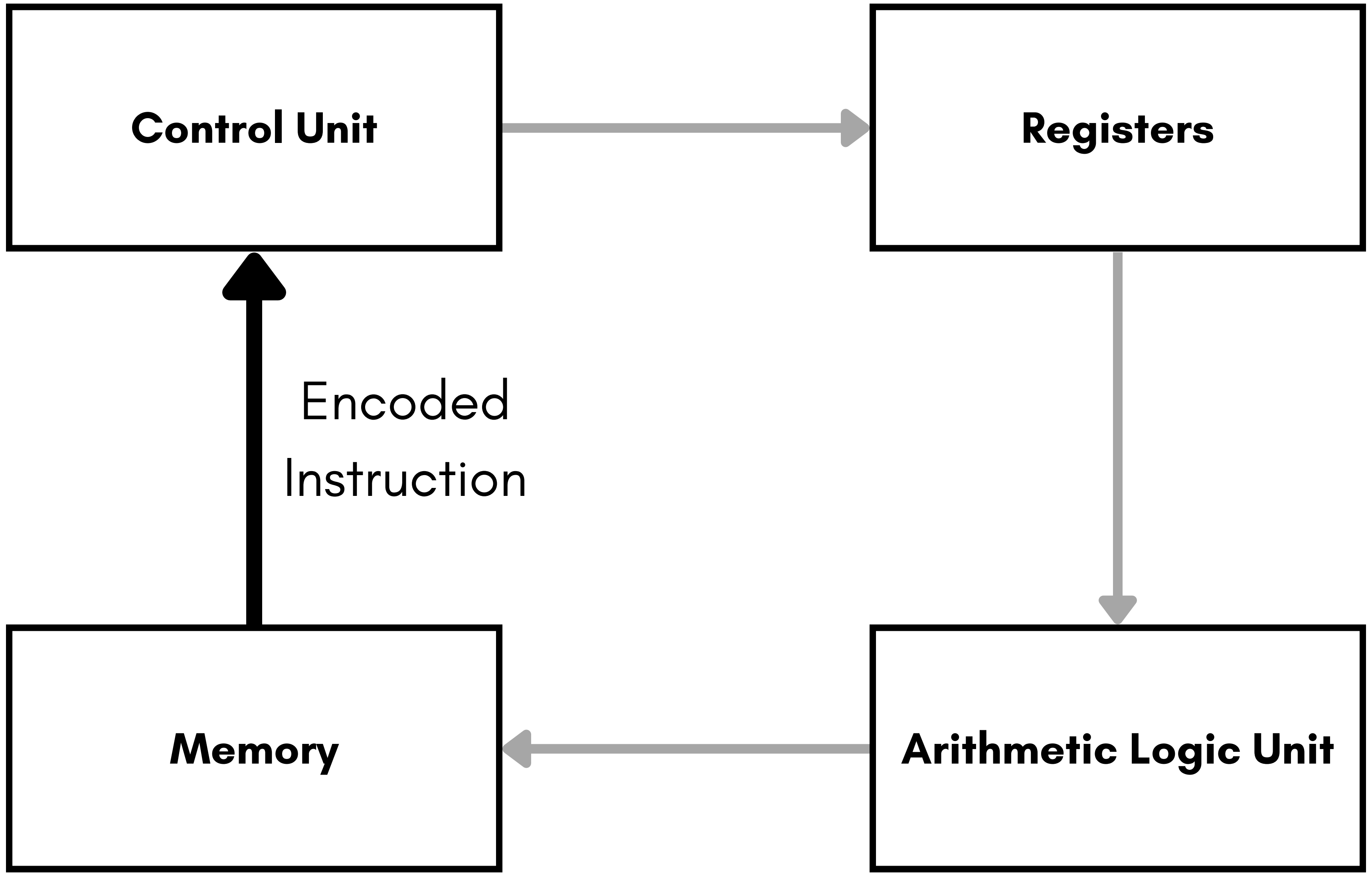
The fetch execute outlines what the **CPU** (*Central Processing Unit*) does.



The CPU's job is to carry out given instructions, thus it follows a "cycle" where it retrieves an instruction from memory, executes the instruction, and repeats.

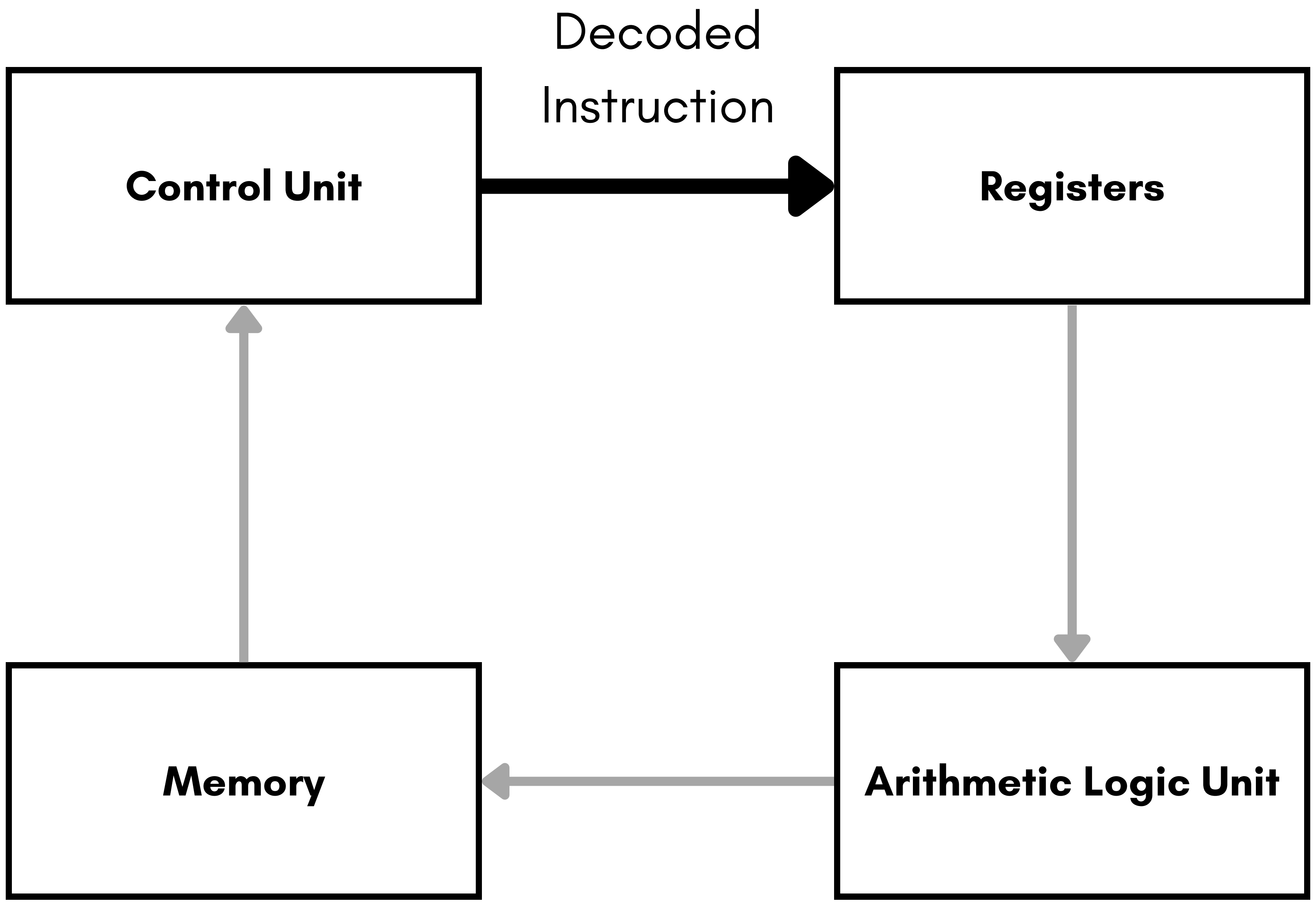
This cycle is known as the "fetch-execute cycle", or "fetch-decode-execute cycle"

$ ~/: Fetch



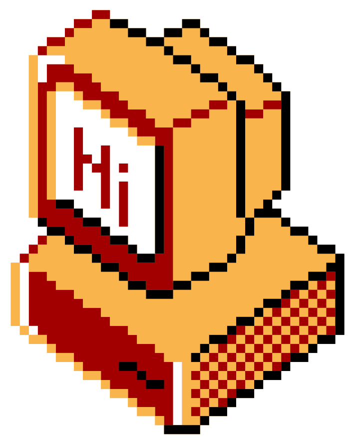
The next instruction is **fetched** from memory

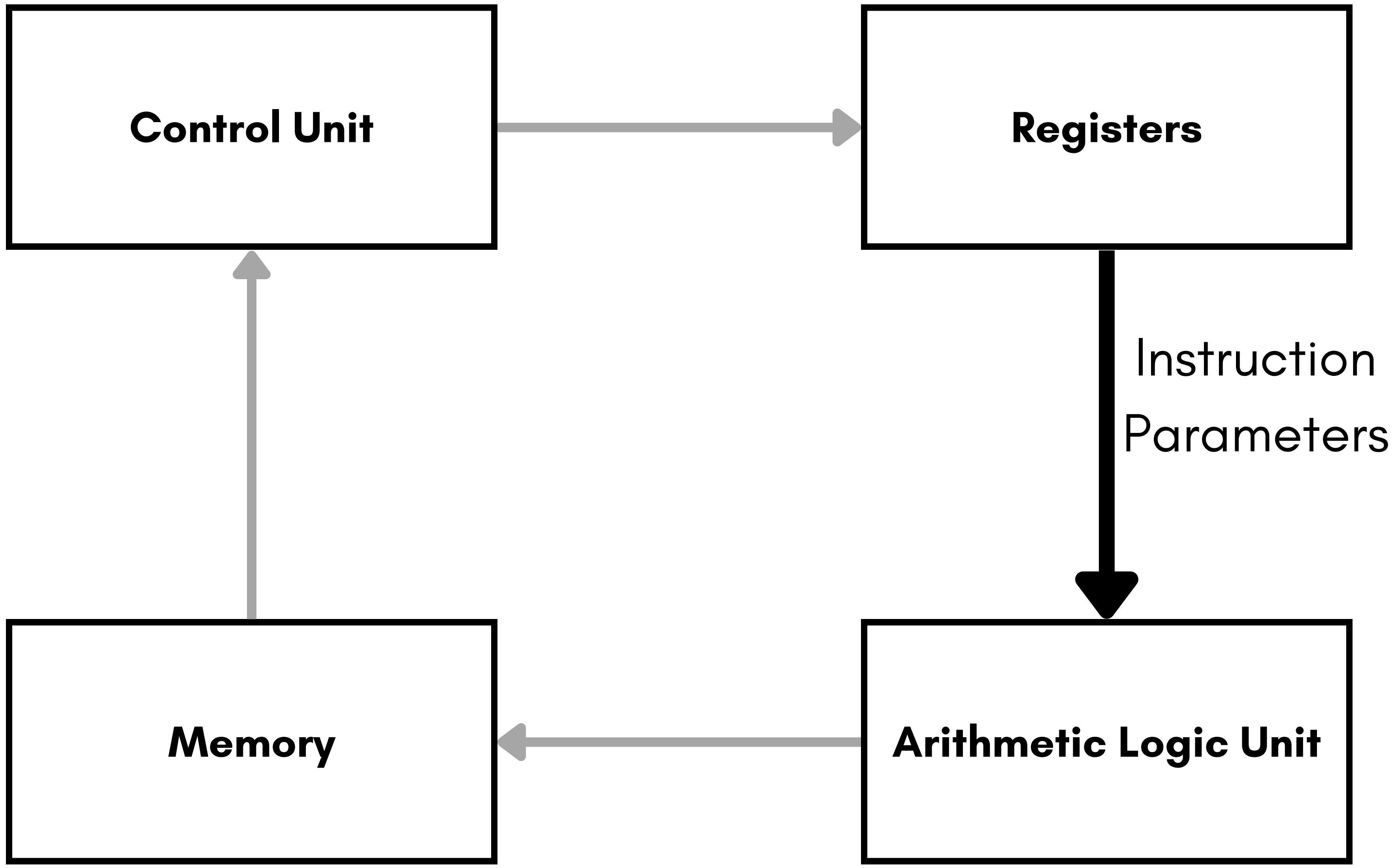
$ ~/: Decode



The instruction is **decoded** into basic operations and memory addresses.

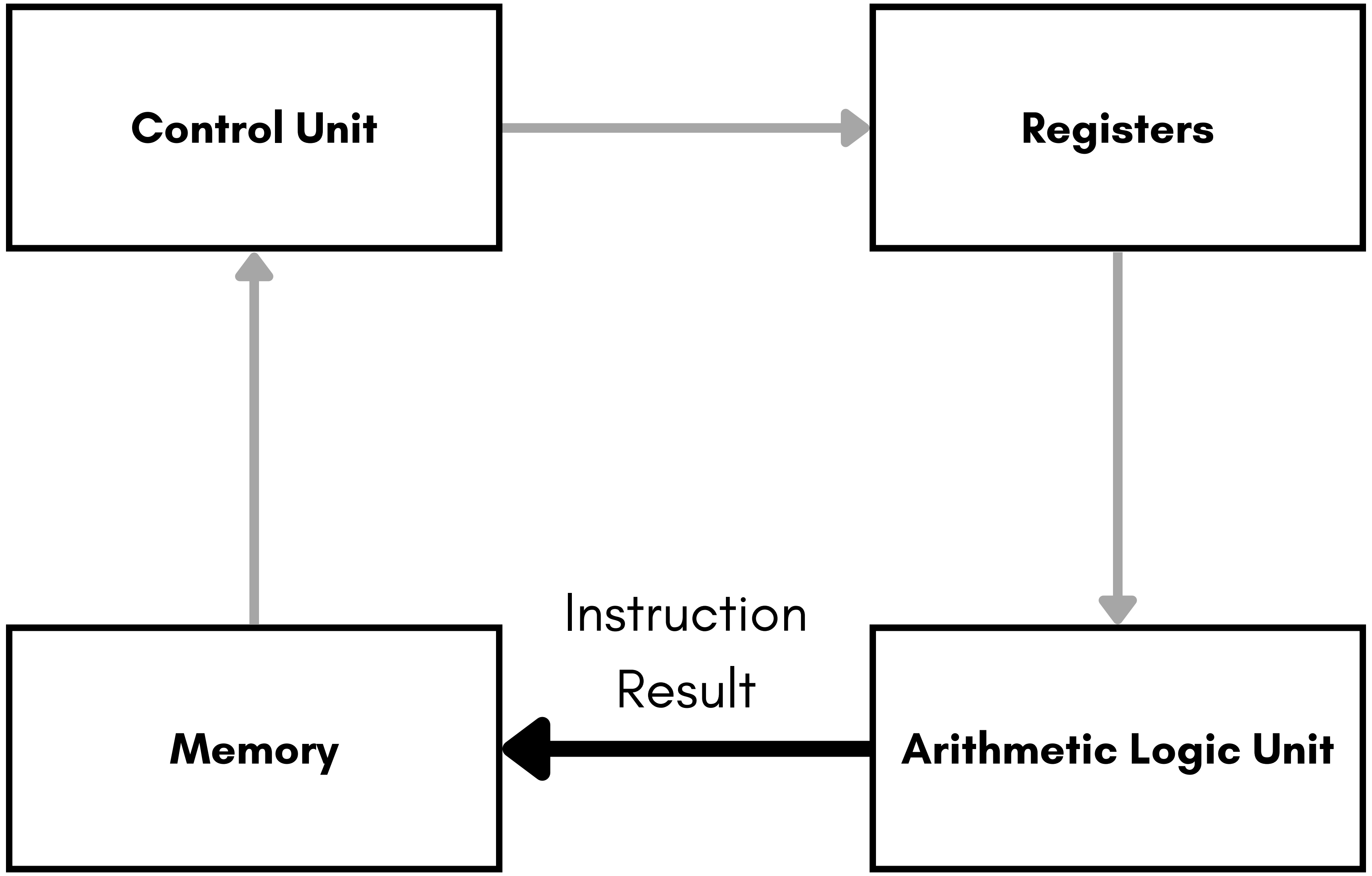
$ ~/: Execute





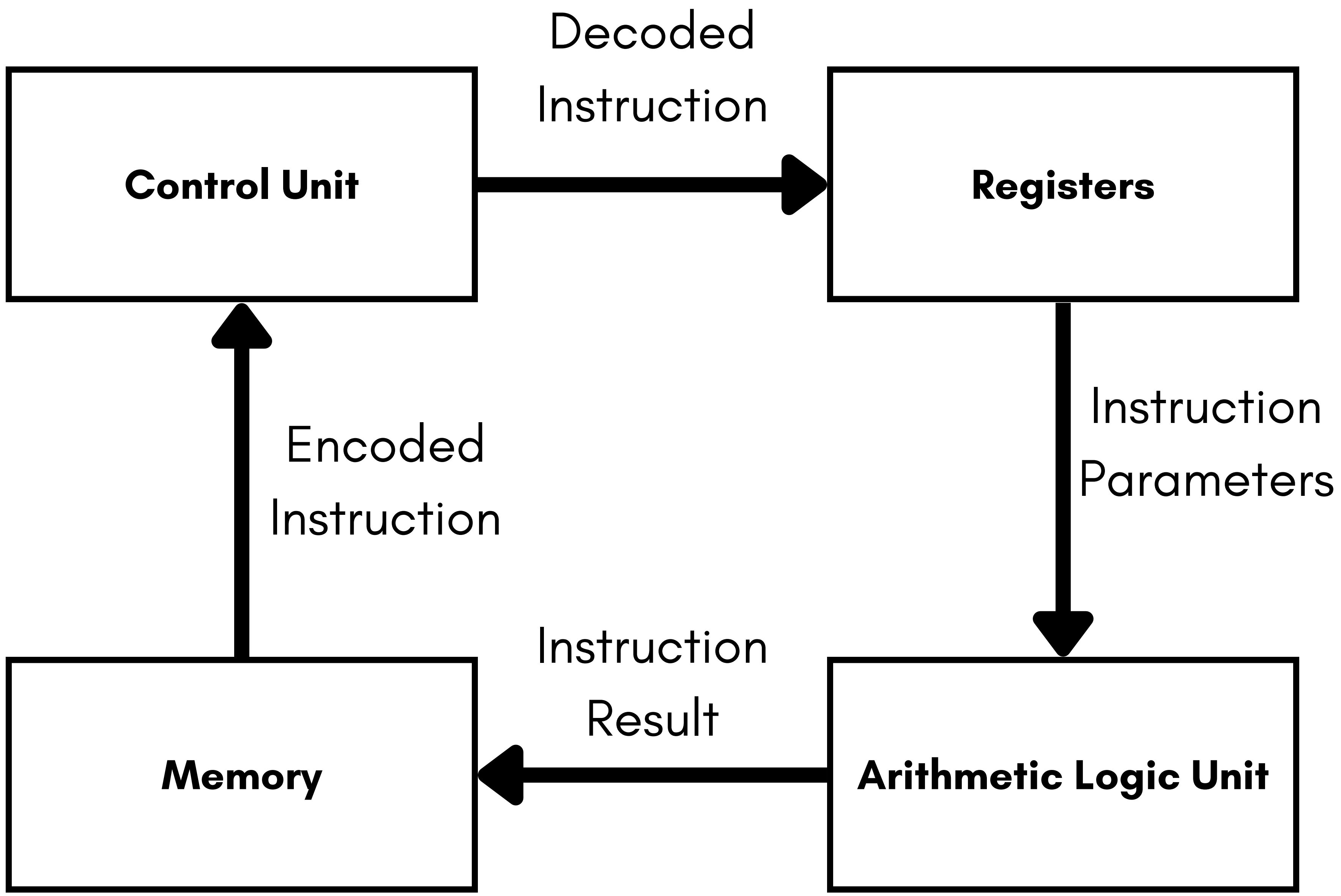
The information in registers tells the ALU (Arithmetic Logic Unit) to **execute** an action

$ ~/: Store



The result of that action is **stored** in memory according to the decoded instruction.

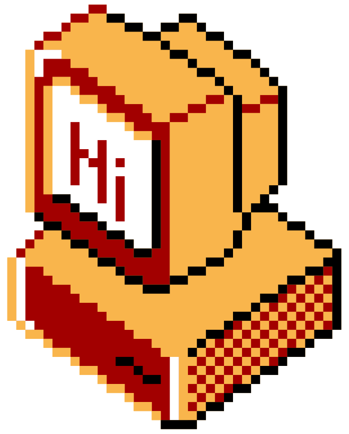
$ ~/: Instructions & Archs



Instructions are encoded based on the type of CPU (*architecture*) in a computer.

- **ARM**
  - Thumb
  - aarch32
  - aarch64
- **x86(-64)**
- **Power ISA**

$ ~/: Hexadecimal



We normally represent numbers like 123, or 1337 – this is known as **base 10**

- We use 10 characters: 0123456789

You may know computers work in **binary**, also known as **base 2**

- There are 2 characters: 0 and 1

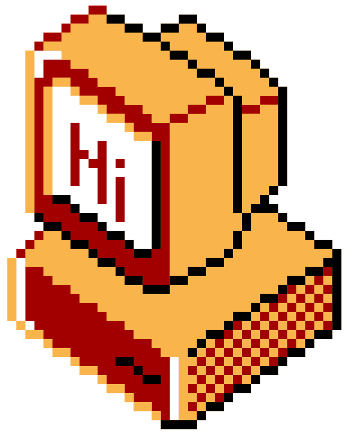
Hexadecimal, or **base 16** is simply another way to represent numbers

- We use it to better view values the computer uses, which are closely linked to powers of two
- Hexadecimal numbers are often prefixed with **0x**
- **Characters are 0-9, then A through F**

```
1337
0x0539
10100111001
```

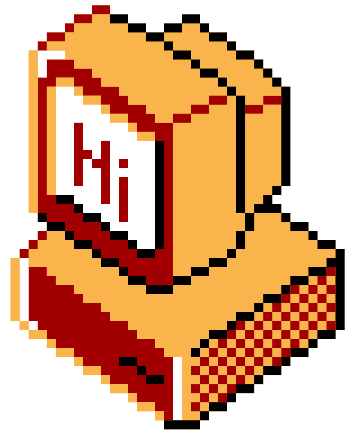
These are all the **same number**

$ ~/: Registers



- Small stores of data that can be quickly accessible to instructions
- Specifics vary between architectures
- Some are reserved by convention
    - Function calls within the program
    - System calls to the OS
- Some are “special” to the processor
    - **I**nstruction **P**ointer (*IP*) or **P**rogram **C**ounter (*PC*)
    - Address registers like Stack Pointer

$ ~/: Syscalls



- A **syscall** is an **instruction** that communicates with the **operating system** to do something
- Parameters for a system call are set up in the CPU's registers, then a syscall instruction is called
- Some examples for syscalls include **read**, **write**, **open** and **exit**

- Website containing syscalls & calling conventions
- https://syscall.sh

$ ~/: Assembly

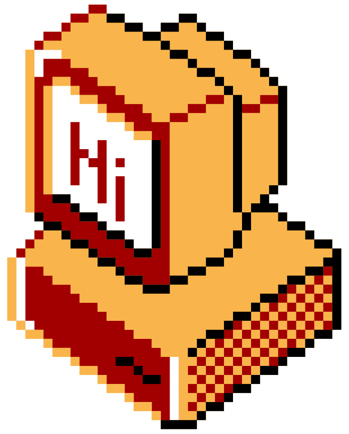
Instructions are written in Assembly Language.

This language, both in syntax and functionality, varies between architectures

Additionally, programs on the same architecture will vary as syscalls differ between operating systems

Windows

```
extern GetStdHandle
extern WriteFile
extern ExitProcess

section .rodata

msg db "Hello World!", 0x0d, 0x0a

msg_len equ $-msg
stdout_query equ -11

section .data

stdout dw 0
bytes_written dw 0

section .text

global start

start:
    mov rcx, stdout_query
    call GetStdHandle
    mov [rel stdout], rax

    mov  rcx, [rel stdout]
    mov  rdx, msg
    mov  r8, msg_len
    mov  r9, bytes_written
    push qword 0
    call WriteFile

    xor rcx, rcx
    call ExitProcess
```



Linux

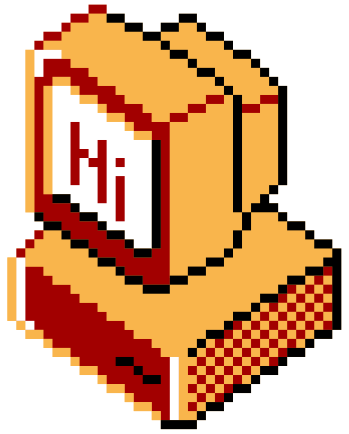
```
global _start

section .text

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, msg
  mov rdx, msglen
  syscall

  mov rax, 60
  mov rdi, 0
  syscall

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

$ ~/: Syntax

- Line deliminated
- [Instruction] [x], [y]

mov	rax	rbx
Instruction	value/register	value/register

$ ~/: Common Instructions



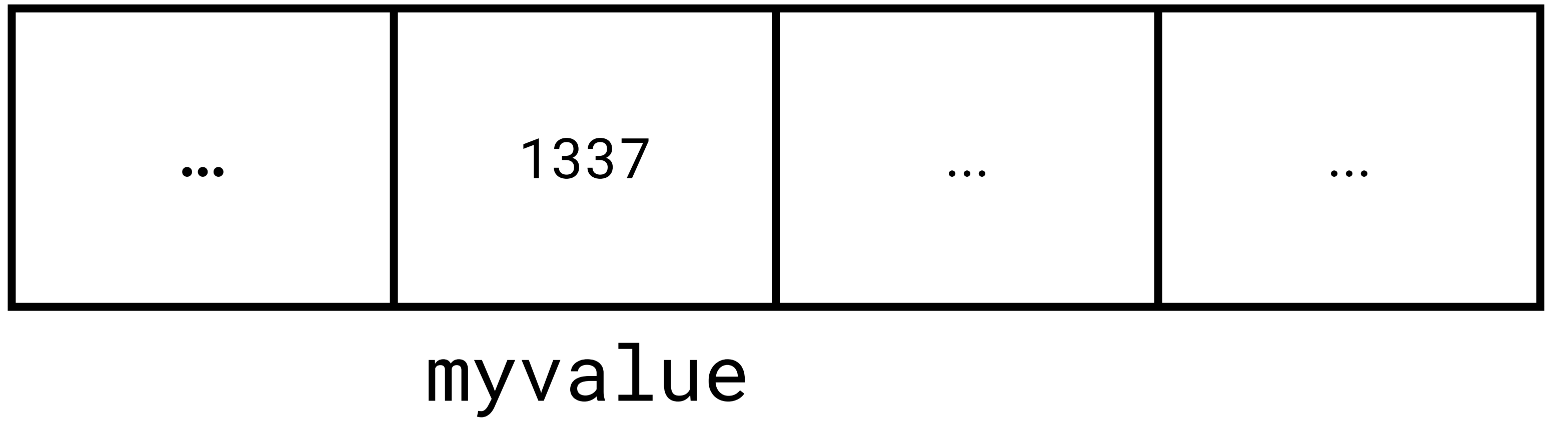
- Common Instructions (there is HUNDREDS)

Data Movement	Arithmetic	Control Flow
mov	add	cmp
push	sub	jmp
pop	mul	je
xchg	div	jne
lea	shl	jle
	shr	jge
	xor	jnae

$ ~/: Memory & Addresses

- Suppose the following:

```
unsigned int myvalue = 1337;
```

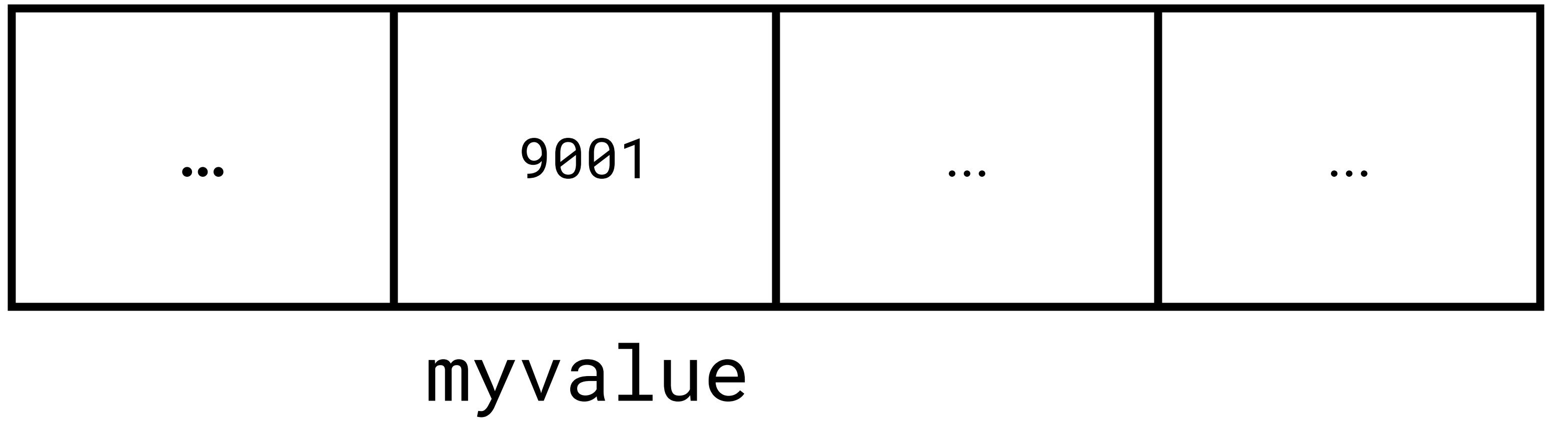


- Compiling the code, this variable is stored at a known and fixed location (generally)
- You can access it when writing your code, but what does it look like to the CPU?

$ ~/: Memory & Addresses

- When modifying that variable:

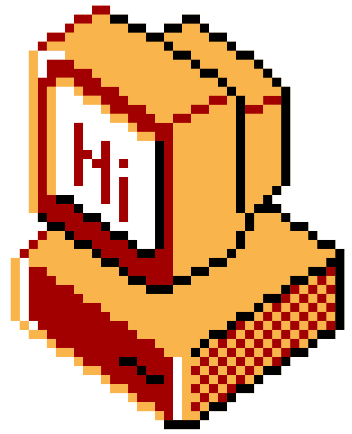
```
unsigned int myvalue = 1337;
myvalue = 9001;
```



- In assembly, it would probably look like:

```
mov myvalue, 9001
```

$ ~/: Memory & Addresses

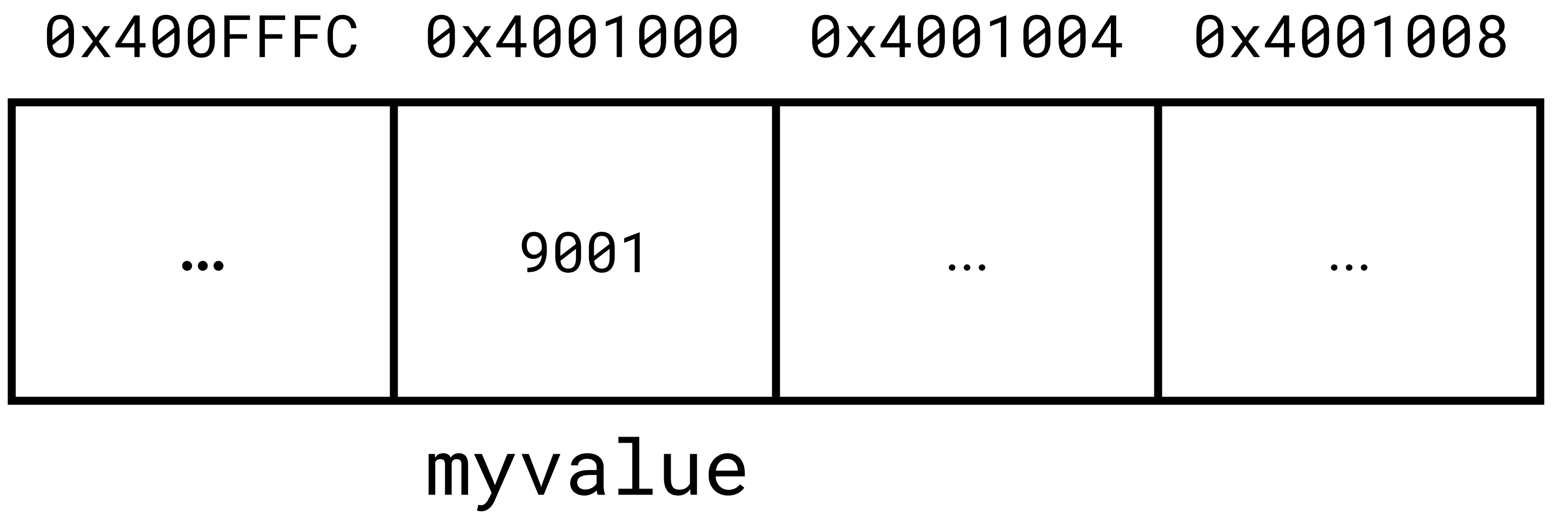


- CPUs don't "name" variables in memory like you would in C or Python.
- Really,

`mov myvalue, 9001`

is encoded as something like:

`mov [0x4001000], 9001`

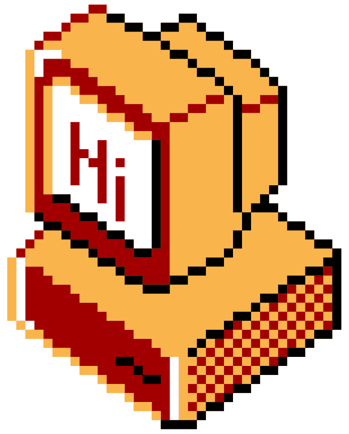


- In this context, the number 0x4001000 is an *address* to our myvalue variable.

$ ~/: Memory & Addresses

- When a program executes, it stores everything in the memory:
  - variables
  - library functions
  - its own code
- Within a compiled program, an address can refer to many things:
  - Functions
    - Blocks *within* functions
  - Other addresses(!)
    - e.g. an address which points to an address, which in turn points to an address...

~/: pause



Workshop/Networking will now commence!

Filedrop! Find the exercise and challenge files here:

- https://emu.team/filedrop_0x02
- 3 Exercises +crackme challenge ! (solutions soon)

Download "Binary Ninja": (cross platform)

- https://binary.ninja/demo/

$ ~/: Exercise 0x01

```
global _start

section .text

_start:
          mov rax, 1; SYS_write syscall number
          mov rdi, X; FIX THIS
          mov rsi, msg; Set the output buffer to our message
          mov rdx, XYZ; FIX THIS
          syscall;

          mov rax, 60; SYS_exit syscall number
          mov rdi, 0; EXIT_SUCCESS exit status
          syscall;

section .data
          msg db "Hello, World!", 0xa; our message string, plus a 0xa (newline character)
          msglen equ $ - msg
```

You may want to check out

https://x64.syscall.sh/

$ ~/: Exercise 0x01 - Solution

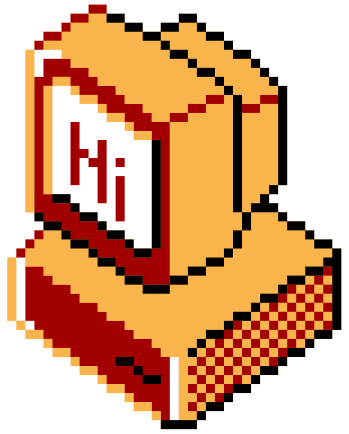
```
global _start

section .text

_start:
        mov rax, 1; SYS_write syscall number
        mov rdi, 1; Set FD to stdout
        mov rsi, msg; Set the output buffer to our message
        mov rdx, msglen; Set rdx to msglen
        syscall;

        mov rax, 60; SYS_exit syscall number
        mov rdi, 0; EXIT_SUCCESS exit status
        syscall;

section .data
        msg db "Hello, World!", 0xa; our message string, plus a 0xa (newline character)
        msglen equ $ - msg
```

NR	SYSCALL NAME	references	RAX	ARG0 (rdi)	ARG1 (rsi)	ARG2 (rdx)
0	read	man/ cs/	0	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	1	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	2	const char *filename	int flags	umode_t mode
3	close	man/ cs/	3	unsigned int fd	-	-

(from https://x64.syscall.sh/)

fd (file descriptor) number	name
0	stdin
1	stdout
2	stderr

$ ~/: Exercise 0x01 - Solution




```
global _start

section .text

_start:
        mov rax, 1; SYS_write syscall number
        mov rdi, 1; Set FD to stdout
        mov rsi, msg; Set the output buffer to our message
        mov rdx, msglen; Set rdx to msglen
        syscall;

        mov rax, 60; SYS_exit syscall number
        mov rdi, 0; EXIT_SUCCESS exit status
        syscall;

section .data
        msg db "Hello, World!", 0xa; our message string, plus a 0xa (newline character)
        msglen equ $ - msg
```

we set output to stdout (mov rdi, 1)

NR	SYSCALL NAME	references	RAX	ARG0 (rdi)	ARG1 (rsi)	ARG2 (rdx)
0	read	man/ cs/	0	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	1	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	2	const char *filename	int flags	umode_t mode
3	close	man/ cs/	3	unsigned int fd	-	-

(from https://x64.syscall.sh/)

fd (file descriptor) number	name
0	stdin
1	stdout
2	stderr

$ ~/: Exercise 0x01 - Solution

```
global _start

section .text

_start:
        mov rax, 1; SYS_write syscall number
        mov rdi, 1; Set FD to stdout
        mov rsi, msg; Set the output buffer to our message
        mov rdx, msglen; Set rdx to msglen
        syscall;

        mov rax, 60; SYS_exit syscall number
        mov rdi, 0; EXIT_SUCCESS exit status
        syscall;

section .data
        msg db "Hello, World!", 0xa; our message string, plus a 0xa (newline character)
        msglen equ $ - msg
```

we set output to stdout (mov rdi, 1)

we specify our message length

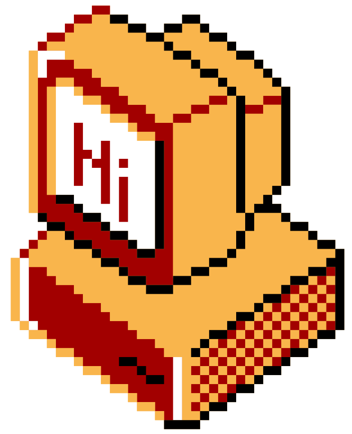
NR	SYSCALL NAME	references	RAX	ARG0 (rdi)	ARG1 (rsi)	ARG2 (rdx)
0	read	man/ cs/	0	unsigned int fd	char *buf	size_t count
1	write	man/ cs/	1	unsigned int fd	const char *buf	size_t count
2	open	man/ cs/	2	const char *filename	int flags	umode_t mode
3	close	man/ cs/	3	unsigned int fd	-	-

(from https://x64.syscall.sh/)

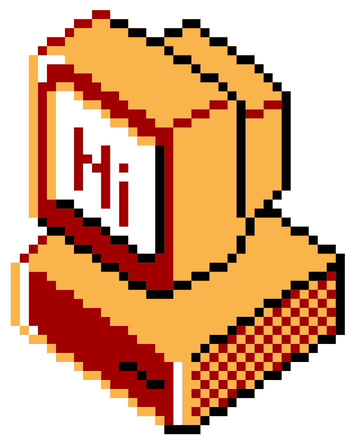
fd (file descriptor) number	name
0	stdin
1	stdout
2	stderr

$ ~/: Exercise 0x02

```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  ; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle XYZ;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  ; There should be an instruction here... FIX THIS
```

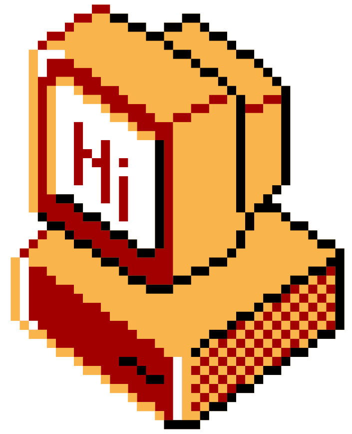


$ ~/: Exercise 0x02 - Solution




```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  inc rbx; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle .loop;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  syscall; There should be an instruction here... FIX THIS
```

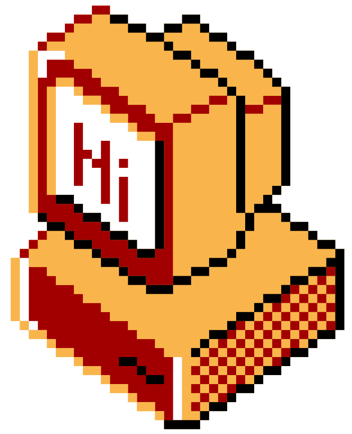
$ ~/: Exercise 0x02 - Solution




```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  inc rbx; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle .loop;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  syscall; There should be an instruction here... FIX THIS
```

we need to increment rbx (inc rbx)

$ ~/: Exercise 0x02 - Solution



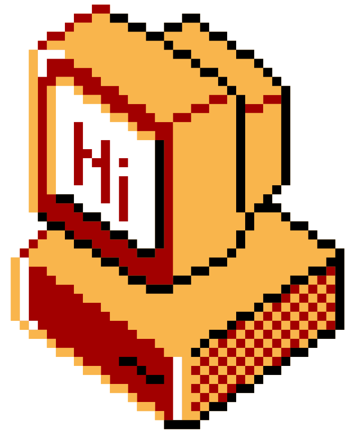

```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  inc rbx; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle .loop;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  syscall; There should be an instruction here... FIX THIS
```

we need to increment rbx (inc rbx)

since the value of rsi = rbx

and rsi is printed

$ ~/: Exercise 0x02 - Solution




```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  inc rbx; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle .loop;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  syscall; There should be an instruction here... FIX THIS
```

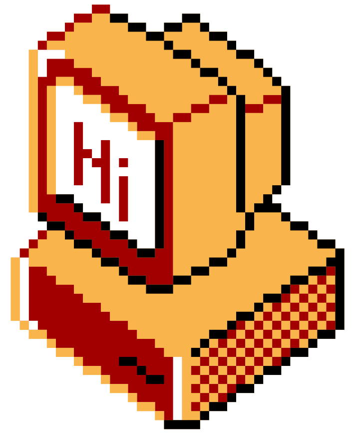
we need to increment rbx (inc rbx)

since the value of rsi = rbx

and rsi is printed

add “jle .loop”, to jump back to the .loop label

$ ~/: Exercise 0x02 - Solution



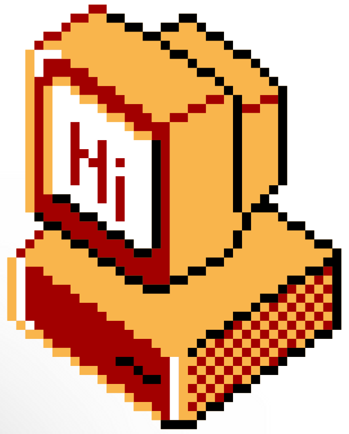

```
global _start

section .text

_start:
  mov rbx, 0; set the counter to 0 to start
  .loop:;   mark this position with `loop` so we can jump to it
  inc rbx; increment the number we are printing.... FIX ME
  mov rsi, rbx; move it into rsi to be the buffer we print
  add rsi, 48; convert number from decimal to it's ascii code
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdi, 1; set fd to stdout
  mov rdx, 1; we are writing one byte
  mov rax, 1; set syscall number to SYS_write
  syscall;

  mov rax, 1; set syscall number to SYS_write
  mov rdi, 1; set fd to stdout
  mov rsi, 0xa; newline character
  push rsi;  put it on the stack so we can get the address
  mov rsi, rsp; get the address of the first item of the stack, so we can print it
  mov rdx, 1; we are writing one byte
  syscall;

  cmp rbx, 8; compare to the max number we will print, minus 1
  jle .loop;  if less than, jump back to ... FIX THIS

  mov rax, 60; set syscall number to SYS_exit
  mov rdi, 0; set code to EXIT_SUCCESS
  syscall; There should be an instruction here... FIX THIS
```

we need to increment rbx (inc rbx) since the value of rsi = rbx and rsi is printed

add "jle .loop", to jump back to the .loop label

add syscall instruction to actually initiate the exit

$ ~/: Exercise 0x03




```
Exercise 3 Instructions:
     Fix the file
     It should print every even number (between 1 and 9)
     Compile the program by running make
     If something screws up, run make clean to start again from the source file
```

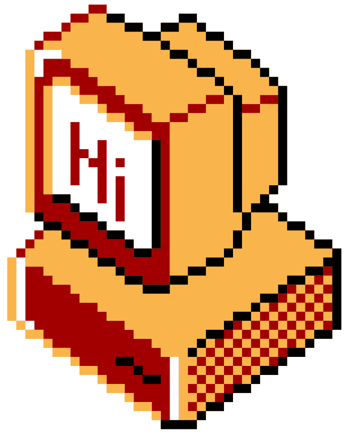
Take a look at the previous exercises

They might be helpful for this one...

$ ~/: Exercise 0x03 - Solution

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `.loop` so we can jump to it
    add rbx, 2;   increment rbx by 2
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 7;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to `.loop`

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall;
```

$ ~/: Exercise 0x03 - Solution




```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `.loop` so we can jump to it
    add rbx, 2;   increment rbx by 2
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 7;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to `.loop`

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall;
```

We increment rbx by 2 (add rbx, 2)

$ ~/: Exercise 0x03 - Solution

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `.loop` so we can jump to it
    add rbx, 2;   increment rbx by 2
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 7;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to `.loop`

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall;
```

We increment rbx by 2 (add rbx, 2)

rbx is copied to rsi and printed

$ ~/: Exercise 0x03 - Solution

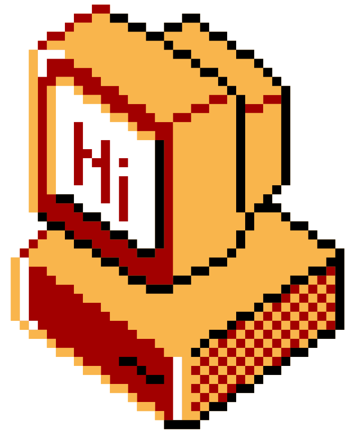
```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `.loop` so we can jump to it
    add rbx, 2;   increment rbx by 2
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 7;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to `.loop`

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall;
```

We increment rbx by 2 (add rbx, 2)

rbx is copied to rsi and printed

if rbx ⩽ 7 then we loop again

$ ~/: Exercise 0x03 - Solution




```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `.loop` so we can jump to it
    add rbx, 2;   increment rbx by 2
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 7;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to `.loop`

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall;
```

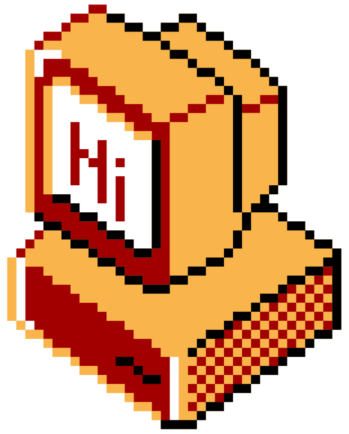
We increment rbx by 2 (add rbx, 2)

rbx is copied to rsi and printed

if rbx ⩽ 7 then we loop again

Otherwise we exit

$ ~/: "crackme" Challenge

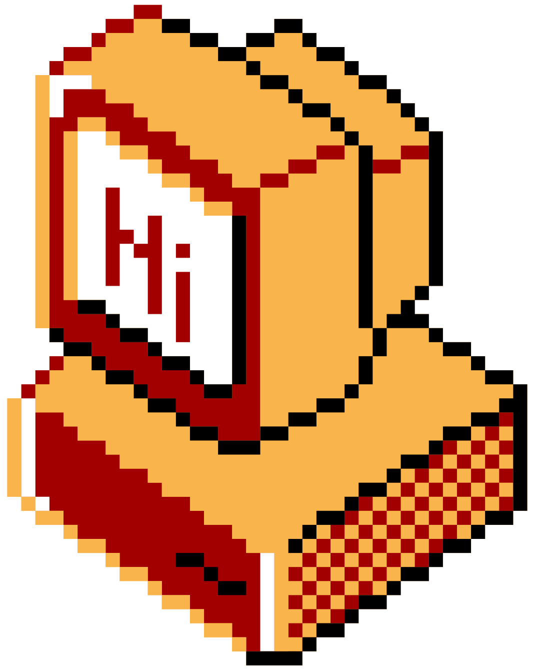


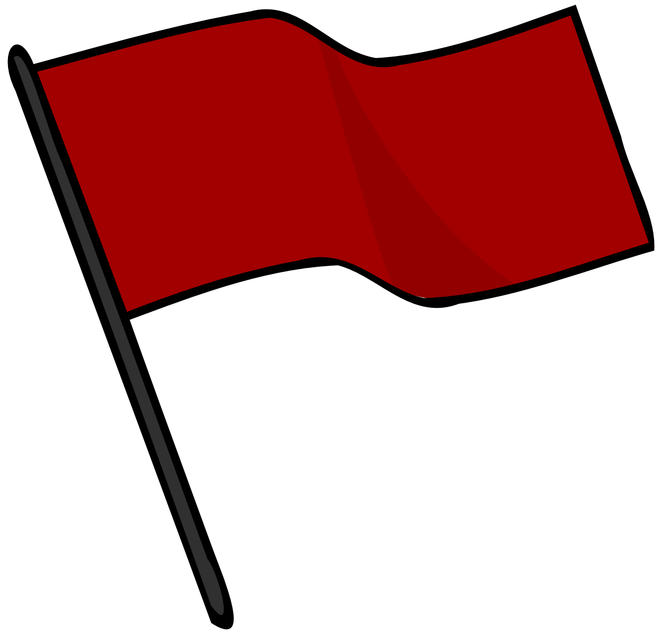
- Blood Prize
- Hak5 Rubber Ducky

- Exercise 3 (Best Solution)
- ESP32 + Accessories

- More to win! We're looking for those taking on the exercises.

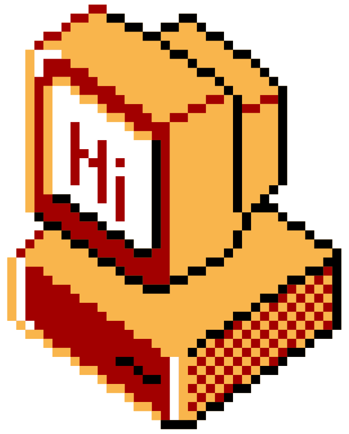
$ ~/: questions





Questions!

~/: shutdown



Thank you!