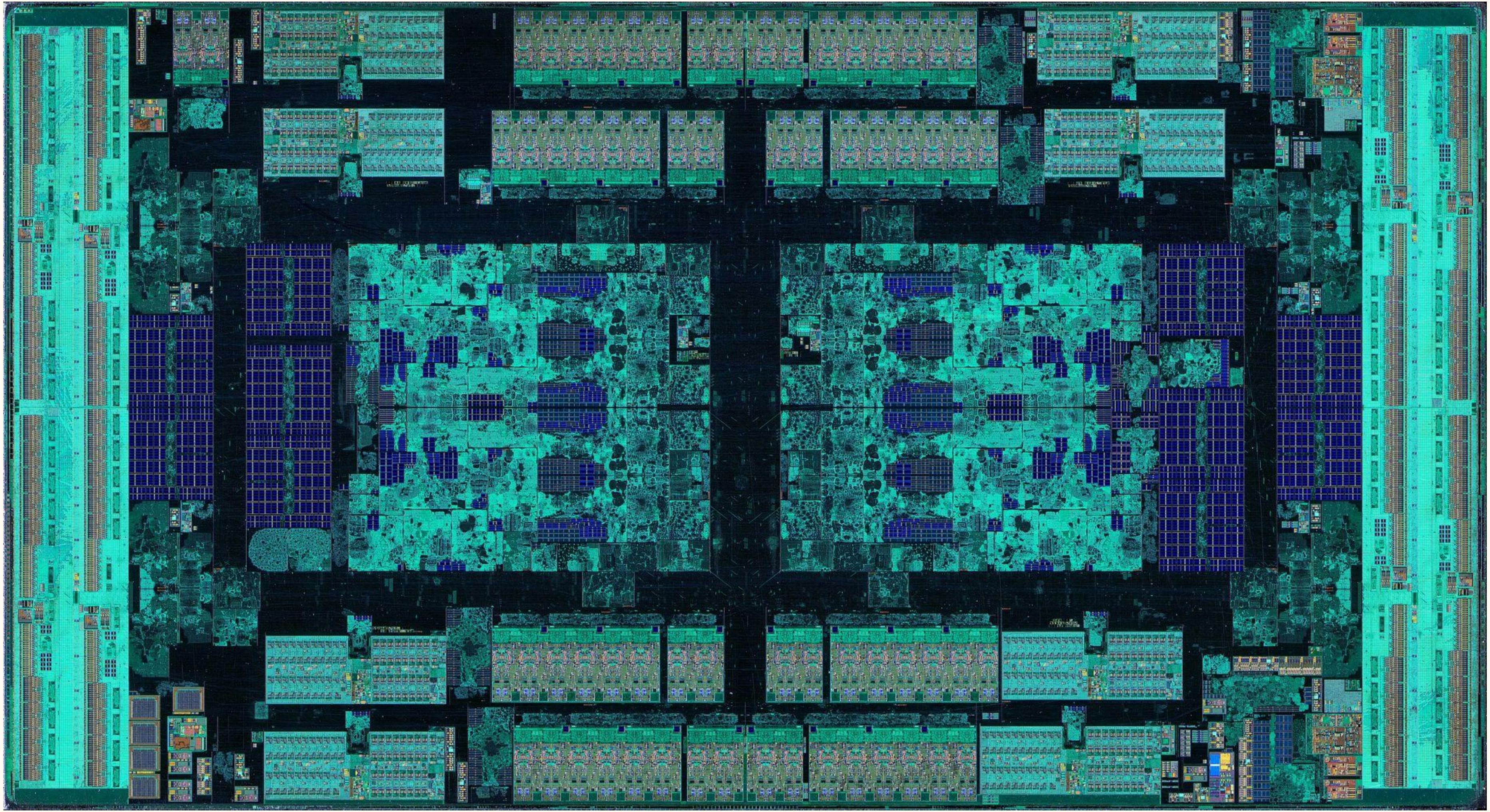
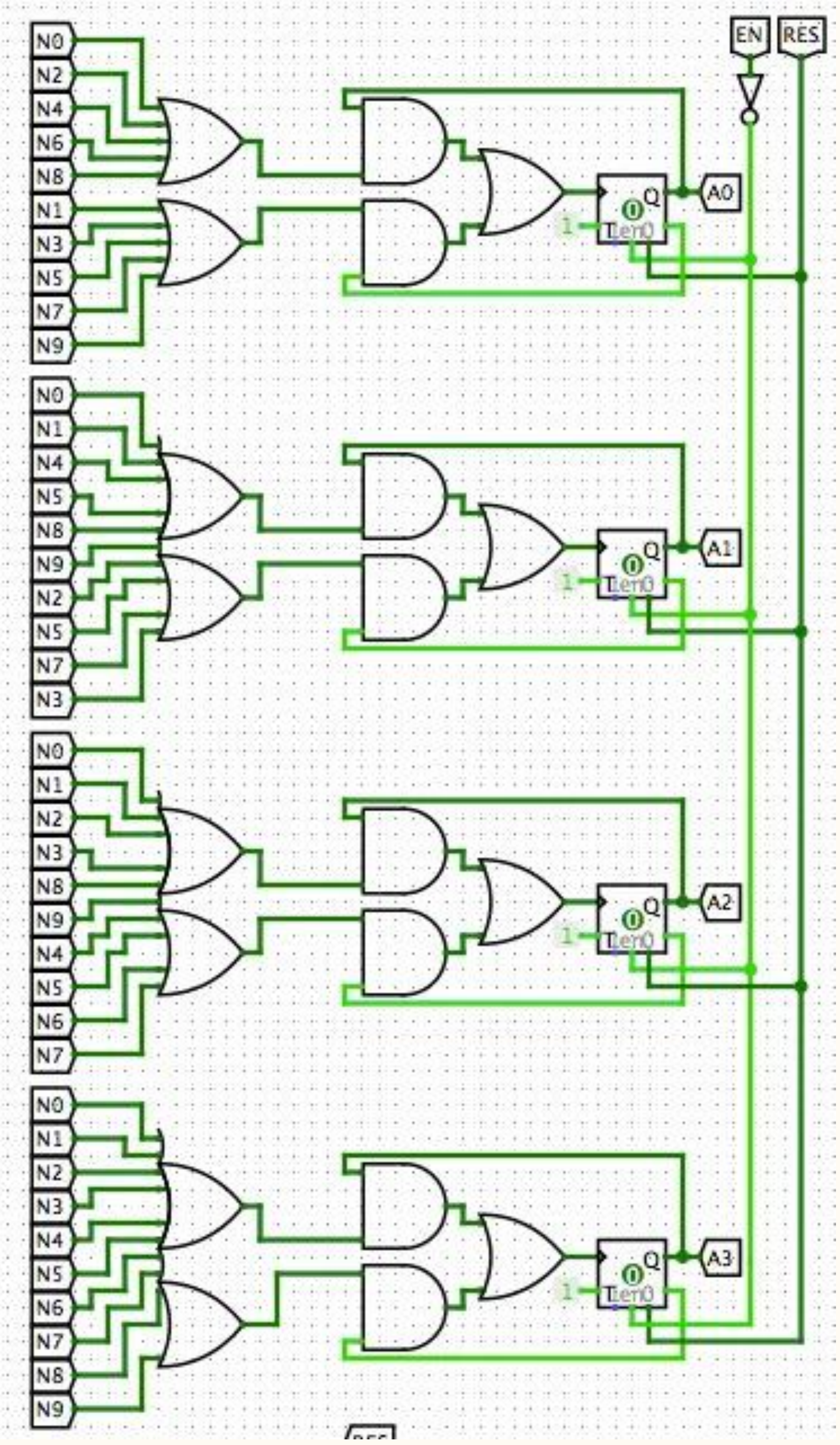
Assembly language: Where you write 10 lines of code to do what one line of Python can… but hey, at least you're *closer to the machine*! 🎉

Assembly



```
00000000: 01001000 11000111 11000111  H..
00000003: 00110111 00010011 00000000  7..
00000006: 00000000 01001000 11000111  .H.
00000009: 11000000 00111100 00000000  .<.
0000000c: 00000000 00000000 00001111  ...
0000000f: 00000101                    .
```



EN
RES
N0
N2
N4
N6
N8
N1
N3
N5
N7
N9
A0
N0
N1
N4
N5
N8
N9
N2
N5
N7
N3
A1
N0
N1
N2
N3
N8
N9
N4
N5
N6
N7
A2
N0
N1
N2
N3
N4
N5
N6
N7
N8
N9
A3

```
00000000: 48 c7 c7  H..
00000003: 37 13 00  7..
00000006: 00 48 c7  .H.
00000009: c0 3c 00  .<.
0000000c: 00 00 0f  ...
0000000f: 05        .
```

```
00000000  48C7C737130000    mov rdi,0x1337
00000007  48C7C03C000000    mov rax,0x3c
0000000E  0F05              syscall
```

ARM AArch64

```
mov x8, 93
mov x0, 0x1337
svc #0
```

MIPS

```
li $v0, 4001
li $a0, 0x1337
syscall
```

x86

```
mov eax, 1
mov ebx, 0x1337
int 0x80
```

z80

```
ld a, 0x37
ld b, 0x13
call 0x05
```

RISC-V

```
li a7, 93
li a0, 0x1337
ecall
```

PPC

```
li r0, 1
li r3, 0x1337
sc
```

Registers

1. General-Purpose Registers (16 registers)

- **64-bit Registers**:
  - `RAX`, `RBX`, `RCX`, `RDX` (traditional general-purpose registers)
  - `RSI`, `RDI` (used for string operations or function arguments)
  - `RBP`, `RSP` (base and stack pointers)
  - `R8` to `R15` (additional general-purpose registers in x86_64)
- These registers can also be accessed in smaller chunks:
  - **32-bit**: `EAX`, `EBX`, `ECX`, `EDX`, etc.
  - **16-bit**: `AX`, `BX`, `CX`, `DX`, etc.
  - **8-bit**: `AL`, `BL`, `CL`, `DL`, etc.

2. Special-Purpose Registers (6 primary ones)

- **Instruction Pointer**:
  - `RIP`: Holds the address of the next instruction to execute.
- **Flags Register**:
  - `RFLAGS`: Stores flags for arithmetic operations, control flow, etc.
- **Segment Registers**:
  - `CS`, `DS`, `SS`, `ES`, `FS`, `GS`: Mostly legacy, but `FS` and `GS` are still used in modern x86_64 for things like thread-local storage.

3. Floating-Point and Vector Registers (32 registers)

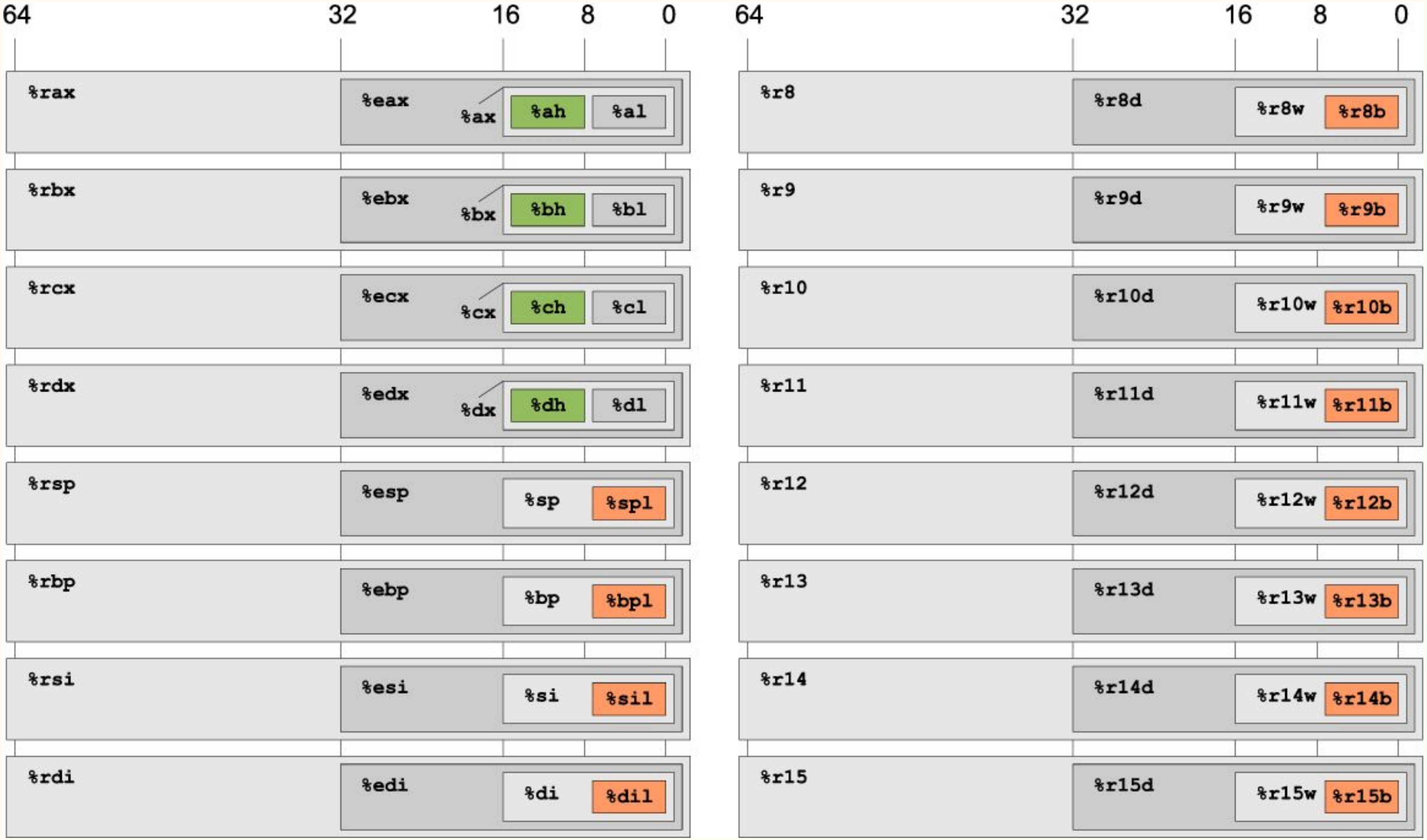
- **XMM Registers (128-bit)**:
  - `XMM0` to `XMM15`: Used for SIMD (Single Instruction, Multiple Data) operations.
- **YMM Registers (256-bit)**:
  - `YMM0` to `YMM15`: Used with AVX (Advanced Vector Extensions).
- **ZMM Registers (512-bit)**:
  - `ZMM0` to `ZMM31`: Available on processors with AVX-512 support.
- **FPU Registers**:
  - `ST0` to `ST7`: Legacy floating-point registers from the x87 FPU stack.

4. Control and Debug Registers

- **Control Registers (4 primary ones)**:
  - `CR0`, `CR2`, `CR3`, `CR4`: Used for system-level settings like memory management.
- **Debug Registers (8 registers)**:
  - `DR0` to `DR7`: Used for setting hardware breakpoints and debugging.

5. Other Specialized Registers

- **Model-Specific Registers (MSRs)**: Configuration and performance monitoring.
- **Test Registers** (legacy): Rarely used today.
- **Performance Counters**: Used for profiling and optimization.


64
32
16
8
0
%rax
%eax
%ax
%ah
%al
%rbx
%ebx
%bx
%bh
%bl
%rcx
%ecx
%cx
%ch
%cl
%rdx
%edx
%dx
%dh
%dl
%rsp
%esp
%sp
%spl
%rbp
%ebp
%bp
%bpl
%rsi
%esi
%si
%sil
%rdi
%edi
%di
%dil
%r8
%r8d
%r8w
%r8b
%r9
%r9d
%r9w
%r9b
%r10
%r10d
%r10w
%r10b
%r11
%r11d
%r11w
%r11b
%r12
%r12d
%r12w
%r12b
%r13
%r13d
%r13w
%r13b
%r14
%r14d
%r14w
%r14b
%r15
%r15d
%r15w
%r15b

Operations

1. Arithmetic Operations

- **Description**: Perform basic mathematical computations.
- **Examples**:
  - `ADD`, `SUB` – Addition and subtraction.
  - `MUL`, `IMUL` – Unsigned and signed multiplication.
  - `DIV`, `IDIV` – Unsigned and signed division.
  - `INC`, `DEC` – Increment and decrement.
  - `ADC`, `SBB` – Add and subtract with carry/borrow.

2. Logical Operations

- **Description**: Perform bitwise and logical computations.
- **Examples**:
  - `AND`, `OR`, `XOR` – Bitwise AND, OR, and XOR.
  - `NOT` – Bitwise negation.
  - `TEST` – Perform a bitwise AND and set flags without storing the result.
  - `CMP` – Compare two values by subtracting and setting flags.

3. Data Movement Instructions

- **Description**: Transfer data between registers, memory, and I/O.
- **Examples**:
  - `MOV` – Move data between registers and memory.
  - `PUSH`, `POP` – Push and pop values onto/from the stack.
  - `LEA` – Load the effective address of a memory operand.
  - `XCHG` – Exchange the contents of two locations.
  - `CMOVcc` – Conditional move based on flags (e.g., `CMOVE`, `CMOVNE`).

4. Control Flow Instructions

- **Description**: Alter the flow of execution.
- **Examples**:
  - `JMP` – Unconditional jump.
  - `JE`, `JNE`, `JG`, `JL`, etc. – Conditional jumps based on flags.
  - `CALL`, `RET` – Call a procedure and return from it.
  - `LOOP` – Loop with a counter.

5. String and Memory Operations

- **Description**: Operate on strings and memory blocks efficiently.
- **Examples**:
  - `MOVSB`, `MOVSW`, `MOVSD` – Move string data.
  - `STOSB`, `STOSW`, `STOSD` – Store string data.
  - `LODSB`, `LODSW`, `LODSD` – Load string data.
  - `CMPSB`, `CMPSW`, `CMPSD` – Compare string data.
  - `SCASB`, `SCASW`, `SCASD` – Scan string data.

6. Shift and Rotate Instructions

- **Description**: Shift and rotate bits in registers or memory.
- **Examples**:
  - `SHL`, `SHR` – Shift left and right logically.
  - `SAR` – Shift right arithmetically.
  - `ROL`, `ROR` – Rotate bits left and right.
  - `RCL`, `RCR` – Rotate bits through the carry flag.

7. Input/Output Instructions

- **Description**: Read from or write to I/O ports.
- **Examples**:
  - `IN`, `OUT` – Read from and write to an I/O port.
  - `INSB`, `INSW`, `INSD` – Input from port to string.
  - `OUTSB`, `OUTSW`, `OUTSD` – Output string to port.

8. Floating-Point and SIMD Instructions

- **Description**: Perform floating-point arithmetic and vectorized operations.
- **Examples**:
  - `FADD`, `FSUB`, `FMUL`, `FDIV` – Floating-point arithmetic.
  - `MOVAPS`, `ADDPS`, `MULPS` – SIMD operations with packed single-precision floats.
  - `PADDQ`, `PSLLD` – Integer SIMD operations.
  - `SQRTPS`, `MINPS` – Specialized SIMD instructions.

9. System-Level Instructions

- **Description**: Manage processor state, system calls, and privileged operations.
- **Examples**:
  - `SYSCALL`, `SYSRET` – System call and return (Linux and Windows).
  - `CPUID` – Get processor information.
  - `HLT` – Halt the processor.
  - `INT n` – Trigger a software interrupt.
  - `IRET` – Return from an interrupt handler.

10. Miscellaneous Instructions

- **Description**: Instructions that don't fit cleanly into other categories.
- **Examples**:
  - `NOP` – No operation.
  - `PAUSE` – Hint to the CPU to reduce power or delay.
  - `XLAT` – Translate a byte using a lookup table.
  - `UD2` – Undefined instruction (for debugging purposes).

syscall

What is `syscall`?

- The `syscall` instruction is used to make **system calls** in x86_64 architecture.
- It transitions control from user mode to kernel mode, allowing programs to request services from the operating system (e.g., file I/O, process management).

How `syscall` Works

1. **Registers Used for Arguments**:
   - System call number: `RAX`
   - Arguments:
     - `RDI`: First argument
     - `RSI`: Second argument
     - `RDX`: Third argument
     - `R10`: Fourth argument
     - `R8`: Fifth argument
     - `R9`: Sixth argument

How `syscall` Works

2. **Registers Affected**:
    - Return value: Stored in `RAX` after the syscall.
    - Flags: `RFLAGS` may change based on syscall results.

How `syscall` Works

3. **Instruction Flow**:

    - Load the syscall number into `RAX`.
    - Load any required arguments into the appropriate registers.
    - Execute `syscall`.
    - Check the return value in `RAX`.

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	fconst char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot			

What is the `call` Instruction?

- The `call` instruction is used to invoke a **subroutine** (function).
- It performs two key tasks:
  1. **Pushes the return address** (the address of the next instruction after `call`) onto the stack.
  2. **Transfers control** to the subroutine by jumping to the specified address.

How `call` Works

1. **Push Return Address**:
   - The address of the instruction immediately after the `call` is pushed onto the stack.
   - This ensures the program knows where to return after the subroutine finishes.
2. **Jump to Target Address**:
   - Control is transferred to the subroutine by jumping to the target address or label.

Key Points:

1. **Pairs with `ret`**:
   - The subroutine uses `ret` to pop the return address from the stack and jump back.
2. **Indirect Calls**:
   - You can use `call` with a register or memory address for indirect subroutine calls:
   -
```
mov rax, my_function
call rax
```

Writing, Compiling

```
> vi input.s

  .intel_syntax noprefix
  .globl _start

 _start:
     mov rax, 60
     mov rdi, 1337
     syscall
```

1a. assemble

- as -o output.o input.s

What is an Object File?

- An intermediate binary file produced by the assembler (e.g., `as` in GNU toolchain).
- Contains **machine code** and metadata required for linking and creating an executable.

Key Sections in an Object File

Section	Purpose
.text	Contains the **machine code** (instructions) generated from the assembly source.
.data	Stores initialized **global and static variables** (e.g., `int x = 42;`).
.bss	Holds uninitialized variables, which are zero-initialized at runtime (e.g., `int y;`).
.rodata	Contains **read-only data** such as string literals (e.g., `"Hello, World!"`).
.symtab	Symbol table, listing functions and global variables with associated metadata (names, addresses, etc.).
.rel.text	Relocation table for `.text`, indicating places where addresses need adjustment during linking.
.debug	Debugging information (optional, if compiled with debug flags).

Common Tools to Inspect Object Files

1. **`objdump`**: Disassembles and analyzes the object file.
   - Example: `objdump -d input.o` (disassembles the `.text` section).
2. **`readelf`**: Displays the object file structure.
   - Example: `readelf -a input.o` (shows all sections, symbols, and relocation info).

Takeaways

- **Object files are not executables** but are crucial for the linking stage.
- They combine **code, data, and metadata** to facilitate building the final binary.

1b. Manual linking

- ld -o my-elf output.o

Definition:

- Linking is the process of combining **object files** and **libraries** into a single executable binary.
- It resolves references between symbols (e.g., functions, variables) defined in different files.

Two Types of Linking:

1. **Static Linking**:
   - Libraries are directly embedded into the executable.
   - Produces a standalone binary but increases size.
2. **Dynamic Linking**:
   - External libraries are loaded at runtime.
   - Reduces binary size but depends on system-installed libraries.

Why Linking Matters?

- Combines code from multiple sources.
- Resolves function and variable dependencies.
- Optimizes and prepares a binary for execution.

Key Steps Performed by `ld`:

1. **Symbol Resolution**:
   - Matches undefined symbols (e.g., `printf`) to their definitions in libraries or other object files.
2. **Relocation**:
   - Adjusts memory addresses for symbols and code to match the final binary layout.
3. **Section Merging**:
   - Combines similar sections (e.g., `.text`, `.data`) from different object files.
4. **Library Linking**:
   - Includes required library functions based on symbol usage.

1c. Pull out code

- objcopy --dump-section .text=code my-elf

2. Assemble and Link one step

- gcc -nostdlib -static -o my-elf input.s

3. Compile straight to bytes

```
> vi input.s

  BITS 64

  start:
     mov rdi, 1337
     mov rax, 60
     syscall
```

- nasm -f bin input.s

4. pwntools

```python
from pwn import asm, context

# Set architecture (x86_64 for 64-bit systems)
context.arch = 'amd64'

# Assembly instructions
assembly_code = """
mov rax, 60      ; syscall: exit
mov rdi, 0       ; exit code: 0
syscall          ; make the syscall
"""

# Assemble the code
machine_code = asm(assembly_code)

# Print the resulting machine code in hex format
print("Assembly Code:")
print(assembly_code)

print("\nMachine Code:")
print(machine_code.hex())
```

Running and Debugging

- **gdb**: General-purpose debugger for assembly and other languages.
- **pwntools**: Python library with built-in debugging utilities.
- **strace**: Traces system calls for insight into program behavior.

What is gdb?

- The GNU Debugger (gdb) allows you to:
  - Step through assembly instructions.
  - Inspect registers and memory.
  - Set breakpoints to pause execution.

Key Commands:

Command	Description
gdb <program>	Start debugging a program.
break <addr>	Set a breakpoint at an address or function.
run	Run the program until a breakpoint.
stepi or si	Execute the next assembly instruction.
info registers	Display all register values.
x/<n> <addr>	Examine memory at an address (<n> bytes).

```
gdb program
> break *0x401000          # Set a breakpoint at an address
> run                      # Run the program
> stepi                    # Step through instructions
> info registers           # Check register state
> x/10xw $rsp              # Examine 10 words at RSP
```

What is `pwntools`?

- Python library for binary exploitation and debugging.
- Provides tools for **dynamic debugging** using scripts.

What is `strace`?

- A tool to trace **system calls** made by a program.
- Helps debug issues related to:
  - File I/O.
  - Memory allocation.
  - Permissions or resource errors.

The Stack

Definition:

- The **stack** is a region of memory used for temporary storage in programs.
- It operates in a **Last In, First Out (LIFO)** manner.

Key Characteristics:

1. **Dynamic Allocation**:
    - Automatically allocates and deallocates memory during function calls.
2. **Directional Growth**:
    - On x86_64, the stack grows **downward** (toward lower memory addresses).
3. **Managed by Registers**:
    - **RSP**: Stack Pointer (points to the top of the stack).
    - **RBP**: Base Pointer (used for referencing local variables).

Why Use the Stack?

- **Function Calls**: Store return addresses, arguments, and local variables.
- **Temporary Storage**: Efficient for short-lived data.
- **Control Flow**: Helps manage recursive and nested functions.

Basic Operations:

1. **Push**: Adds data to the top of the stack.
   - Decreases RSP.

```
push rax ; Store RAX on the stack
```

2. **Pop**: Removes data from the top of the stack.
   - Increases RSP.

```
pop rax ; Restore the top value into RAX
```

Function Call Example:

1. Caller pushes arguments onto the stack.
2. The return address is pushed automatically during `call`.
3. The callee allocates space for local variables.

```
High Memory Addresses
|   Arguments       |  <- Caller pushes arguments
|   Return Address  |  <- CALL instruction pushes return address
|   Local Variables|
|-----------------/  <- RSP (Stack Pointer)
Low Memory Addresses
```

Calling Conventions:

- Defines how arguments, return values, and stack management are handled.
- **x86_64 Linux (System V ABI)**:
  - **Registers**: First 6 arguments in `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.
  - **Stack**: Additional arguments and return address.

Prologue (Callee Setup):

Save the previous base pointer:

```
push rbp
mov rbp, rsp
```

Allocate space for local variables:

```
sub rsp, <size>
```

Epilogue (Callee Cleanup):

Deallocate local variables:

```
add rsp, <size>
```

Restore the base pointer and return:

```
pop rbp
ret
```

```
_start:
    ; Push values onto the stack
    mov rax, 42
    push rax          ; Store 42 on the stack
    mov rax, 100
    push rax          ; Store 100 on the stack

    ; Pop values off the stack
    pop rbx           ; RAX = 100
    pop rcx           ; RCX = 42

    ; Exit syscall
    mov rax, 60
    xor rdi, rdi
    syscall
```

> demos

\> challenges

1. Hello, World! (Data Movement + String Operations)

- **Goal**: Print "Hello, World!" to the screen using a syscall.
- **Instructions**: `mov`, `syscall`.
- **Hints**:
  - Use the `write` syscall (`rax = 1`) with the string in memory.
  - Pass the file descriptor (`stdout` = 1), string pointer, and length.

2. Add Two Numbers (Arithmetic Operations)

- **Goal**: Prompt the user to input two numbers, add them, and print the result.
- **Instructions**: `add`, `mov`, `syscall`.
- **Hints**:
  - Use the `read` syscall to get input.
  - Convert ASCII input to integers and use `add`.

3. Compare Two Numbers (Control Flow)

- **Goal**: Compare two user-provided numbers and print which one is larger.
- **Instructions**: `cmp`, `jne`, `jl`, `jmp`.
- **Hints**:
  - Use `cmp` to compare values and conditional jumps (`jl`, `jg`) to handle output.

4. Implement a Simple Loop (Control Flow + Arithmetic)

- **Goal**: Print numbers 1 through 10 in a loop.
- **Instructions**: `mov`, `add`, `cmp`, `jmp`.
- **Hints**:
  - Use a counter in a register.
  - Use `cmp` and `jmp` to create a loop.

5. Bitwise Manipulation (Logical Operations)

- **Goal**: Toggle the case of a string (convert uppercase to lowercase and vice versa).
- **Instructions**: `xor`, `and`, `or`.
- **Hints**:
  - Use bitwise `xor` with `0x20` to toggle case.
  - Loop through each character in the string.

6. Shift and Rotate (Bitwise Operations)

- **Goal**: Multiply a number by 16 using a left shift.
- **Instructions**: `shl`, `sar`, `mov`.
- **Hints**:
  - Use `shl` to shift bits to the left.
  - Print the result using the `write` syscall.

7. Basic String Reverse (String and Memory Operations)

- **Goal**: Reverse a user-provided string.
- **Instructions**: `movsb`, `rep`, `jmp`.
- **Hints**:
  - Use pointers to swap characters in memory.
  - Iterate until the midpoint of the string.

8. Implement an XOR Cipher (Logical Operations)

- **Goal**: Encrypt a string using an XOR cipher with a fixed key.
- **Instructions**: `xor`, `mov`, `loop`.
- **Hints**:
  - XOR each character with a key (e.g., `0xAA`).
  - Print the encrypted result.

9. Smallest Number in an Array (Arithmetic + Loops)

- **Goal**: Find the smallest number in an array of integers.
- **Instructions**: `cmp`, `mov`, `jmp`.
- **Hints**:
  - Use a register to store the smallest number.
  - Iterate through the array with a loop, updating the register when a smaller number is found.

10. Fibonacci Sequence (Advanced Control Flow + Arithmetic)

- **Goal**: Compute and print the first 10 numbers in the Fibonacci sequence.
- **Instructions**: `mov`, `add`, `push`, `pop`, `jmp`.
- **Hints**:
  - Use two registers to store the last two Fibonacci numbers.
  - Loop to calculate and print each new number.