

### Instruction Set Architecture (ISA)

Every CPU has a set of commands it understands known as its Instruction Set Architecture or ISA. Two ISAs are **very** common:

- 1.
- 2.

An ISA defines the function of the hardware in the CPU.

### CPU Registers

Each CPU core has an extremely limited number of \_\_\_\_\_ that are used for general purpose CPU operations:

- x64: 16 registers of 64 bits
- ARMv8: 31 registers of 64 bits

With very few exceptions \_\_\_\_\_.

### Instruction Sets

Every ISA defines a set of instructions that a CPU can execute:

<b>Move:</b>	MOV, XCHG, PUSH, POP, ...
<b>Arithmetic (int):</b>	ADD, SUB, MUL, DIV, NEG, CMP, ...
<b>Logic:</b>	AND, OR, XOR, SHR, SHL, ...
<b>Control Flow:</b>	JMP, LOOP, CALL, RET, ...
<b>Synchronization:</b>	LOCK, ...
<b>Floating Point:</b>	FADD, FSUB, FMUL, FDIV, FABS, ...

ARM processors have significantly fewer instructions and are known as \_\_\_\_\_ while x64 processors have a greater set of instructions and known as \_\_\_\_\_.

**Q:** Advantages of RISC / CISC?

### CPU Instruction in a Real Program

	04.c	gcc 04.c objdump -d ./a.out
3	int main() {	f3 0f 1e fa            endbr64 55                    push   %rbp 48 89 e5             mov    %rsp,%rbp 48 83 ec 10          sub    \$0x10,%rsp
4	int a = 0;	c7 45 fc 00 00 00 00   movl   \$0x0, -0x4(%rbp)
5	a = a + 3;	83 45 fc 03            addl   \$0x3, -0x4(%rbp)
6	a = a - 2;	83 6d fc 02            subl   \$0x2, -0x4(%rbp)
7	a = a * 4;	c1 65 fc 02            shll   \$0x2, -0x4(%rbp)
8	a = a / 2;	8b 45 fc             mov    -0x4(%rbp), %eax 89 c2                mov    %eax, %edx c1 ea 1f             shr    \$0x1f, %edx 01 d0                add    %edx, %eax d1 f8                sar    %eax 89 45 fc             mov    %eax, -0x4(%rbp)
9	a = a * 5;	8b 55 fc             mov    -0x4(%rbp), %edx 89 d0                mov    %edx, %eax c1 e0 02             shl    \$0x2, %eax 01 d0                add    %edx, %eax 89 45 fc             mov    %eax, -0x4(%rbp)
10	printf("Hi");	48 8d 3d f0 0d 00 00   lea    0xdf0(%rip), %rdi # 2004 <_IO_stdin_used+0x4> b8 00 00 00 00        mov    \$0x0, %eax e8 42 fe ff ff        callq  1060<printf@plt>
11	a = a * 479;	8b 45 fc             mov    -0x4(%rbp), %eax 69 c0 df 01 00 00    imul  \$0x1df, %eax, %eax 89 45 fc             mov    %eax, -0x4(%rbp)

**Program Counter (PC):**

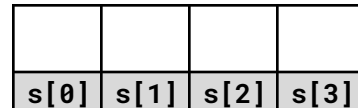
**Operation Timings:**

## Endianness:

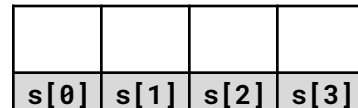
One major difference between ISAs is how multi-byte characters are stored. Knowing that `sizeof(int) == 4`, what do we expect from the following program?

```
04b.c
4 int i = 3 + (2 << 8) + (1 << 16); // 66051
5 char *s = (char *)&i;
6 printf("%02x %02x %02x %02x\n", s[0], s[1], s[2], s[3]);
```

x86/x64 - Big Endian:



ARM/A64 - Little Endian:



What is "Host Byte Order"? What is "Network Byte Order"?

## Beyond Characters: Files and File Types

Using binary digits, often represented as characters using an encoding like UTF-8, we can build more complex file types.

### File Extensions: An Easy Identifier

The most common way to identify the contents of a file is by the **file extension**. The file extension is defined as:

Examples:

cs240.png	mp1.c	mp1.h	taylor.swift.mp4
-----------	-------	-------	------------------

Which files are "plain text files"?

## Memory Hierarchy:

The third foundation of a computer system is the memory -- the storage of data to be processed by our CPU. There are many different types of common memory in a system:

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.

## Sample Programs:

```
04cr.c
16 for (unsigned int c = 0; c < SIZE; c++) {
17     for (unsigned int r = 0; r < SIZE; r++) {
18         array[(r * SIZE) + c] = (r * SIZE) + c;
19     }
20 }
```

## Sample Program #2:

```
04rc.c
16 for (unsigned int r = 0; r < SIZE; r++) {
17     for (unsigned int c = 0; c < SIZE; c++) {
18         array[(r * SIZE) + c] = (r * SIZE) + c;
19     }
20 }
```

What is different about `04cr.c` and `04rc.c`?

Running Times: `04cr.c` (Program #1):

`04rc.c` (Program #2):