Hardware, part 2

Microprocessors

- Logic gate circuits allow us to compute Boolean functions **very** fast

  limited by propagation delay in copper (nanoseconds per meter) and in transistors (picoseconds)

- Boolean functions can model essentially anything we can compute today.

But

- we cannot design and manufacture a new IC for each algorithm or computing task
- we need **many** logic gates, even for simple things

  ~100k transistors for a 64-bit integer division

  for context, modern microprocessors have 1-200 billion transistors

$\longrightarrow$ We break down complex algorithms into simple steps.

Components in a microprocessor

- Logic gates
- A **clock**
- **Memory**
- Input and output devices

A simple model

- Memory is N bits $x \in \{0,1\}^N$ (e.g. for 16 GB, $N \simeq 128 \times 10^9$)
- At every **clock cycle** (e.g. 1.2 GHz), we update the memory:

$$x'_i \leftarrow f_i(x) \qquad \forall i = 0, \ldots, N$$

- To simplify the model
  - Some of the memory comes from input devices
  - Some of the memory is sent to output devices

Issue with the simple model

In this model, we update the whole memory at every clock cycle:

- That would be $128 \times 10^9 \times 1.2 \times 10^9 = 153.6 \times 10^{18}$ b/s

$$\simeq 19,200,000,000 \text{ GB/s}$$

- As of 2025, memory maxes out at ~ 800 GB/s

Therefore, we cannot have so many different Boolean functions f_i

A more realistic model

Instead, at each cycle, the computer executes one of a limited set of **instructions** in a **microprocessor**. Ex.: “Central Processing Unit” (CPU), “Graphics Processing Unit” (GPU).

Instructions are read sequentially from memory and they can be:

- a memory read / write (a tiny amount, like 512 bits)
- 64-bit arithmetic (`+`, `-`, `×`, `/`, …)
- a comparison (`<`, `>`, `=`, …)
- a branch (`if`, `while`, …) which alters the control flow of instructions

Instruction Set Architectures (ISA)

An **ISA** specifies:

- How the machine is organized (memory, etc.)
- What instructions are available
- How instructions are encoded into bits

Two major ISAs in practice:

- `x86_64` (aka. x64, x86_64, AMD64): Intel® and AMD® 64-bit CPUs
- `AArch64` (aka. ARM64): ARM®-based 64-bits CPUs (most phones, Apple M1 – M4)

Many older or less-prominent ISAs:

x86, Itanium, ARMv7, RISC-V, PowerPC, …

```c
int f(int a, int b, int c)
{
    return (a * b) / c;
}
```

x86_64:

89 f8 89 d1 0f af c6 99 f7 f9 c3

```
f:
  mov eax, edi    # 89 f8
  mov ecx, edx    # 89 d1
  imul eax, esi   # 0f af c6
  cdq             # 99
  idiv ecx        # f7 f9
  ret             # c3
```

↑ assembly ↑

AArch64:

1b 01 7c 00 1a c2 0c 00 d6 5f 03 c0

```
f:
  mul w0, w0, w1    # 1b 01 7c 00
  sdiv w0, w0, w2   # 1a c2 0c 00
  ret               # d6 5f 03 c0
```

Assembly

- Assembly is the lowest-level programming language
- Usually in 1:1 correspondence with binary encoding of instructions
- Typically, one line per instruction

Instructions (x86_64)

```
f:
  mov eax, edi    # 89 f8
  mov ecx, edx    # 89 d1
  imul eax, esi   # 0f af c6
  cdq             # 99
  idiv ecx        # f7 f9
  ret             # c3
```

`mov` a, b	move	$a \leftarrow b$
`imul` a, b	signed integer multiply	$a \leftarrow a \times b$
`idiv` a	signed integer divide	eax $\leftarrow$ eax $/b$
`cdq`	convert double-word (32 bits) to quad-word (64 bits)	sign-extend eax into edx:eax
`ret`	return	return to calling function

Instructions (AArch64)

```
f:
  mul w0, w0, w1    # 1b 01 7c 00
  sdiv w0, w0, w2   # 1a c2 0c 00
  ret               # d6 5f 03 c0
```

`mul` a, b, c	multiply	$a \leftarrow b \times c$
`sdiv` a, b, c	signed integer divide	$a \leftarrow b / c$
`ret`	return	return to calling function

Registers

x86_64:

```
f:
  mov eax, edi     # 89 f8
  mov ecx, edx     # 89 d1
  imul eax, esi    # 0f af c6
  cdq              # 99
  idiv ecx         # f7 f9
  ret              # c3
```

AArch64:

```
f:
  mul w0, w0, w1    # 1b 01 7c 00
  sdiv w0, w0, w2   # 1a c2 0c 00
  ret               # d6 5f 03 c0
```

- small, fixed set of variables that can be accessed instantly
- 16 (x86_64) or 31 (AArch64) general-purpose 64-bit registers
- plus special registers and flags (not accessible directly)
- plus larger registers for extended operations (e.g. non-integer numbers)

General-purpose registers (x86_64)

- sixteen 64-bit registers:

  `rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15`

- we can access the lower 32 bits separately:

  `eax, ebx, ecx, edx, ebp, esp, esi, edi, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d`

- we can access the lower 16 bits separately:

  `ax, bx, cx, dx, bp, sp, si, di, r8w, r9w, r10w, r11w, r12w, r13w, r14w, r15w`

- we can access the lower 8 bits separately:

  `al, bl, cl, dl, bpl, spl, sil, dil, r8b, r9b, r10b, r11b, r12b, r13b, r14b, r15b`

- we can access bits 8-15 separately for some registers:

  `ah, bh, ch, dh`

Example:

<table>
<tr><th>bits</th><th>63…56</th><th>55…48</th><th>47…40</th><th>39…32</th><th>31…24</th><th>23…16</th><th>15…8</th><th>7…0</th></tr>
<tr><td>64</td><td colspan="8">rax</td></tr>
<tr><td>32</td><td colspan="4"></td><td colspan="4">eax</td></tr>
<tr><td>16</td><td colspan="6"></td><td colspan="2">ax</td></tr>
<tr><td>8</td><td colspan="6"></td><td>ah</td><td>al</td></tr>
</table>

General-purpose registers (AArch64)

- thirty-one 64-bit registers:

  `x0`, …, `x30`

- we can access the lower 32 bits separately:

  `w0`, …, `w30`

- register 31 (`x31`, `w31`) is read-only (zero in most cases)

Example:

bits	63…56	55…48	47…40	39…32	31…24	23…16	15…8	7…0
64	x0							
32					w0			

Note:

- In both cases, registers are treated as integer numbers
- We cannot (directly) address individual bits
- When it matters, the instruction specifies whether the register is signed or not:

x86_64:

```
 idiv ecx        # f7 f9    (signed)
div ecx         # f7 f1    (unsigned)
```

AArch64:

```
 sdiv w0, w0, w2  # 1a c2 0c 00   (signed)
udiv w0, w0, w2  # 1a c2 08 00   (unsigned)
```

Memory

```c
int g(int *a, int *b)
{
    return *a + *b;
}
```

x86_64:

```
g:
  mov eax, DWORD PTR [rsi]
  add eax, DWORD PTR [rdi]
  ret
```

AArch64:

```
g:
  ldr w2, [x0]
  ldr w0, [x1]
  add w0, w2, w0
  ret
```

Memory

- From a process' perspective, memory is seen as a single long array of bytes

  (8 bits, treated as a single signed or unsigned integer)

- Like registers, memory can be accessed in larger chunks

  (e.g. 16, 32 or 64 bits integer)

- But the smallest addressable unit is the byte

Byte ordering

address	0	1	2	3	...	239	240	241	242	243	244	...
value (hex)	ef	cd	ab	89	...	ff	a0	a1	a2	a3	42	...

- the byte at address 240 is (hex) `a0` = (decimal) 160
- the byte at address 241 is (hex) `a1` = (decimal) 161
- the byte at address 242 is (hex) `a2` = (decimal) 162
- the byte at address 243 is (hex) `a3` = (decimal) 163

Q: What is the value of the 32-bit integer at address 240?

A: It depends!

Byte ordering / “Endianess”

address	0	1	2	3	…	239	240	241	242	243	244	…
value (hex)	ef	cd	ab	89	…	ff	a0	a1	a2	a3	42	…

- “big-endian” (BE): 32-bit int at 240 is (hex) `a0 a1 a2 a3`
  = (decimal) $160 \times 2^{24} + 161 \times 2^{16} + 162 \times 2^{8} + 163$
  = (decimal) 2,694,947,491
- “little-endian” (LE): 32-bit int at 240 is (hex) `a3 a2 a1 a0`
  = (decimal) $163 \times 2^{24} + 162 \times 2^{16} + 161 \times 2^{8} + 160$
  = (decimal) 2,745,344,416

- `x86_64` is LE
- `AArch64` is LE by default (LE-only on Windows, MacOS, Linux)

Visualizing little-endian”

address	…	244	243	242	241	240	239	…	3	2	1	0
value (hex)	…	42	a3	a2	a1	a0	ff	…	89	ab	cd	ef

Bit ordering

Because we cannot address individual bits on a CPU (smallest chunk is a byte), bit ordering does not matter here.

However the same problem crops up in other contexts (USB, Ethernet, Wifi, ...)

Memory access notation

- In assembly, accessing memory is denoted using “[” and “]”
  - Moving the value 240 into a register:

```
 mov eax, 240 # eax = 240
ldr w0, 240  # w0  = 240
```

  - Moving the 4 bytes of memory at address 240 into a register:

```
 mov eax, DWORD PTR [240] # eax = (hex) a3a2a1a0
ldr w0, [240]             # w0  = (hex) a3a2a1a0
```

```c
int g(int *a, int *b)
{
    return *a + *b;
}
```

x86_64:

```
g:
  mov eax, DWORD PTR [rsi]
  add eax, DWORD PTR [rdi]
  ret
```

AArch64:

```
g:
  ldr w2, [x0]
  ldr w0, [x1]
  add w0, w2, w0
  ret
```