# LECTURE 2

# INTEGERS (CONTINUED)

## Two's complement:

- Given a single $n$-bit pattern,
  - let $u$ be its unsigned value
  - let $s$ be its signed value,
- If bit $(n-1) = 0$, then:
  - $s := u$
- If bit $(n-1) = 1$, then:
  - $s := u - 2^n$

4-bit example:

| bit pattern | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| unsigned $u$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| signed $s$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

- bit $(n-1) = 0 \quad \Rightarrow \quad s = u$
- bit $(n-1) = 1 \quad \Rightarrow \quad s = u - 2^n$

In general:

| bit pattern | 00 … 0 | | 01 … 1 | 10 … 0 | | 11 … 1 |
|---|---|---|---|---|---|---|
| unsigned $u$ | 0 | ... | $(2^{n-1}) - 1$ | $(2^{n-1})$ | ... | $(2^n) - 1$ |
| signed $s$ | 0 | | $(2^{n-1}) - 1$ | $-(2^{n-1})$ | | $-1$ |

- Unsigned: $u \in \{0, \ldots, (2^n) - 1\}$
- Signed: $s \in \{-(2^{n-1}), \ldots, -1, 0, \ldots (2^{n-1}) - 1\}$

| $n$ bits | $-2^{\text{bits}-1}$ (min) | $2^{\text{bits}-1}-1$ (max) |
|---:|---:|---:|
| 8 | -128 | 127 |
| 16 | -32768 | 32767 |
| 32 | -2,147,483,648 | 2,147,483,647 |
| 64 | $\simeq -9.10^{18}$ | $\simeq 9.10^{18}$ |
| 128 | $\simeq -2.10^{38}$ | $\simeq 2.10^{38}$ |

**Conversely:**

- if $s \geq 0$ $\quad s \in \{0, \ldots, (2^n) - 1\}$
  - represent with bit pattern of $u = s$.
- if $s < 0$ $\quad s \in \{-(2^{n-1}), \ldots, (2^{n-1}) - 1\}$
  - represent with bit pattern of $u = 2^n - |s|$.
- if $s \notin \{-(2^{n-1}), \ldots, (2^{n-1}) - 1\}$
  - cannot represent, need larger $n$

# Sign extension

Let us represent $s = -5$ in $n$-bit signed binary (two's complement):

$$u = 2^n - |s| = 2^n - 5$$

| $n$ | $s$ | $u$ | bit pattern |
|---|---|---|---|
| 4 | -5 | 11 | 1011 |
| 5 | -5 | 27 | 11011 |
| 6 | -5 | 59 | 111011 |
| 7 | -5 | 123 | 1111011 |
| 8 | -5 | 251 | 11111011 |
| 9 | -5 | 507 | 111111011 |
| 10 | -5 | 1019 | 1111111011 |
| 11 | -5 | 2043 | 11111111011 |
| 12 | -5 | 4091 | 111111111011 |

# Increasing the number of bits

**To convert an $n$-bit number to an $(n + k)$-bit number** $(k \geq 0)$:

- Unsigned:
  - Additional high-order (leftmost) bits are set to zero
- Signed ("sign extension"):
  - Additional high-order (leftmost) bits are set to the value of bit $(n - 1)$

# Base 16

Hexadecimal digits: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`

| binary | hexadecimal | decimal |
|---|---|---|
| 0000 0000 | 00 | 0 |
| 0000 0001 | 01 | 1 |
| 0000 0010 | 02 | 2 |
| 0000 0011 | 03 | 3 |
| 0000 0100 | 04 | 4 |
| 0000 0101 | 05 | 5 |
| 0000 0110 | 06 | 6 |
| 0000 0111 | 07 | 7 |
| 0000 0000 | 08 | 8 |
| 0000 1001 | 09 | 9 |
| 0000 1010 | 0a | 10 |
| 0000 1011 | 0b | 11 |
| 0000 1100 | 0c | 12 |
| 0000 1101 | 0d | 13 |
| 0000 1110 | 0e | 14 |
| 0000 1111 | 0f | 15 |

| binary | hexadecimal | decimal |
|---|---|---|
| 0001 0000 | 10 | 16 |
| 0001 0001 | 11 | 17 |
| 0001 0010 | 12 | 18 |
| 0001 0011 | 13 | 19 |
| 0001 0100 | 14 | 20 |
| 0001 0101 | 15 | 21 |
| 0001 0110 | 16 | 22 |
| 0001 0111 | 17 | 23 |
| 0001 0000 | 18 | 24 |
| 0001 1001 | 19 | 25 |
| 0001 1010 | 1a | 26 |
| 0001 1011 | 1b | 27 |
| 0001 1100 | 1c | 28 |
| 0001 1101 | 1d | 29 |
| 0001 1110 | 1e | 30 |
| 0001 1111 | 1f | 31 |

| binary | hexadecimal | decimal |
|---|---|---|
| 0010 0000 | 20 | 32 |
| 0010 0001 | 21 | 33 |
| 0010 0010 | 22 | 34 |
| 0010 0011 | 23 | 35 |
| 0010 0100 | 24 | 36 |
| 0010 0101 | 25 | 37 |
| 0010 0110 | 26 | 38 |
| 0010 0111 | 27 | 39 |
| 0010 0000 | 28 | 40 |
| 0010 1001 | 29 | 41 |
| 0010 1010 | 2a | 42 |
| 0010 1011 | 2b | 43 |
| 0010 1100 | 2c | 44 |
| 0010 1101 | 2d | 45 |
| 0010 1110 | 2e | 46 |
| 0010 1111 | 2f | 47 |

- Pros:
  - Directly maps to binary numbers:

    hex `12f3` = binary `0001 0010 1111 0011`
  - More compact than binary
  - Directly maps to bytes:

    two hex digits = one byte
- Cons:
  - Not human-friendly (esp. for arithmetic)

# CHARACTERS AND TEXT

How do we map bit patterns to characters in order to form text?

- Many standards
- Some similaritites
- Some incompatibilities

# ASCII (1963-)

- American Standard Code for Information Interchange
- Each character stored stored in 1 byte (8 bits, 256 possible characters)
- 128 standardized characters
- Many derivatives specify the remaining 128

| | 2nd hex digit | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| 0 | | | | | | | | | | | \t | \n | | \r | | |
| 1 | | | | | | | | | | | | | | | | |
| 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | |

1st hex digit:

# Unicode (1988-)

- Associates “code points” (roughly, characters) to integers
- Up to 1,112,064 code points (currently 149,186 assigned)
- First 128 code points coincide with ASCII
- Multiple possible encodings into bytes (“transmission formats”):
  - UTF-8
    - First 128 code points encoded into a single byte (backward compatible with ASCII)
    - Sets most significant bit (bit 7) to 1 to signify “more bytes needed”
    - Up to 4 bytes per code point
    - Default on BSD, iOS/MacOS, Android/Linux and on most internet communications
  - UTF-16
    - Code points are encoded by either two or four bytes
    - Default on Windows, for Java code, and for SMS

# Unicode (1988-)

- Aims at encoding all languages:
  - including extinct ones
  - left-to-right, right-to-left or vertical
  - and more (emojis 🤷)
- Some “characters” require multiple code points (flag emojis, skin tone modifiers)
- What is even a “character”? (code point, glpyh, grapheme, cluster)
- Unicode is extremely complicated
- Latest version (v15.0.0, 2022) specification is 1,060 pages

# HARDWARE

Logic gates allow us to compute Boolean functions “instantly” (subject to physical limits).

But we need **many** logic gates, even for simple things (like 64-bit integer division).

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

# COMPONENTS IN A COMPUTER

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

# A first abstraction

- Memory is $N$ bits $x \in \{0,1\}^N$ (e.g. for 16 GB, $N \simeq 128.10^9$)
- At every **clock cycle** (e.g. 1.2 GHz), we update the memory:

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

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

# A more realistic model

- We cannot 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
    $$\simeq 153,600,000,000 \text{ GB/s}$$
  - As of 2023, memory maxes out at $\sim 800$ GB/s
  - Instead, at each cycle, we only read/write a tiny amount of memory
- We cannot have too many different Boolean function $f_i$
  - Instead, at each cycle, the computer executes one of a limited set of **instructions** in a "processor" (aka. "Central Processing Unit", CPU), e.g.
    - memory read / write
    - 64-bit arithmetic (+, -, ×, /, …)
    - comparison (<, >, =, …)
    - branch (`if`, `while`, …)

# 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 (phones, Apple M1 & M2)

Many older and 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
```

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

- Assembly is the lowest-level programming language
- Assembly is 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
- special registers and flags (not accessible directly)
- larger registers for extended operations (e.g. non-integer numbers)

# 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:

| bits | 63…56 | 55…48 | 47…40 | 39…32 | 31…24 | 23…16 | 15…8 | 7…0 |
|---|---|---|---|---|---|---|---|---|
| 64 | rax | | | | | | | |
| 32 | | | | | eax | | | |
| 16 | | | | | | | ax | |
| 8 | | | | | | | ah | al |

# 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 | | | |

# 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)
- Like registers, memory can be accessed in larger chunks (16, 32 or 64 bits)
- 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)

# Bit ordering

Because we cannot access 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
```