Week 5: Data Representation

References: Patterson & Hennessy, ARM Ed., Chapter 2

Review

Last week, we studied the **building blocks of digital logic**: gates, combinational circuits such as multiplexers, decoders, and adders, and sequential circuits such as latches, flip-flops, and registers.
These components provided the hardware foundation for how a CPU performs arithmetic and controls the flow of data.

This week, we turn from **how** the circuits operate to **what** they operate on: data itself.
The 0s and 1s flowing through logic gates are not arbitrary.
They represent integers, signed and unsigned values, characters, and eventually floating-point numbers.
Understanding how data is encoded in binary is essential for bridging the gap between high-level programming concepts and the physical machine.

---

1. Bits, Bytes, and Words (§2.1 – §2.3)

The most fundamental unit of information in a computer is the **bit**, which can hold a value of `0` or `1`.
Physically, this is implemented as a low or high voltage.

- A **byte** is a collection of 8 bits.
  The byte is the smallest *addressable unit* of memory in most architectures.
  Even if only a single bit is required, the memory system allocates at least one byte.
- A **word** is the "natural" data size for a processor.
  A word matches the width of the CPU registers and datapath.
  - In a 32-bit ARMv7 processor, a word is 32 bits (4 bytes).
  - In a 64-bit ARMv8 processor, a word is 64 bits (8 bytes).

For example, the decimal value 300_{10} is equal to 100101100_2.
This requires 9 bits to store.
Since memory can only address whole bytes, at least 2 bytes are required.

Memory alignment

Processors often require that words be stored at memory addresses that are multiples of the word size.

For example, a 4-byte word should be placed at an address divisible by 4.
Misaligned accesses may be slower or may even cause exceptions on some systems.
Alignment ensures efficient memory access.

---

2. Unsigned Binary Numbers (§2.4)

Binary numbers are represented using **positional notation**, similar to decimal numbers but with base 2.
Each position corresponds to a power of 2.

For an n-bit binary number:

$$\text{Value} = (b_{n-1} \times 2^{n-1}) + (b_{n-2} \times 2^{n-2}) + \cdots + (b_1 \times 2^1) + (b_0 \times 2^0)$$

Example:

$$1101_2 = (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13_{10}$$

Range of unsigned values

- An n-bit unsigned integer can represent values from 0 to $2^n - 1$.
- Examples:
  - 8-bit unsigned: $0 \to 255$
  - 16-bit unsigned: $0 \to 65{,}535$
  - 32-bit unsigned: $0 \to 4{,}294{,}967{,}295$

Overflow in unsigned arithmetic

Since registers have fixed width, results that exceed the maximum representable value **wrap around**.

Example (8-bit):

$$11111111_2\ (255) + 1 = 00000000_2\ (0)$$

The 9th bit is discarded, producing wraparound.

In C and most modern languages, unsigned arithmetic is explicitly defined as modulo 2^n.

---

3. Signed Numbers (§2.4)

Unsigned binary cannot represent negative values.
Several historical methods were developed to represent signed integers, but modern systems universally use **two's complement**.

3.1 Sign-and-Magnitude

The most intuitive scheme uses the leftmost bit to indicate sign ($0 =$ positive, $1 =$ negative), with the remaining bits giving the magnitude.

- Example (4-bit): $1001_2 = -9$.
- Problem: Two encodings for zero ($0000 = +0$, $1000 = -0$).

3.2 One's Complement

Negative numbers are obtained by flipping all the bits of the positive representation.

- Example (4-bit): $+5 = 0101$, $-5 = 1010$.
- Problem: Still has two encodings for zero.

3.3 Two's Complement (standard today)

Negative numbers are represented by inverting all bits of the positive number and then adding 1.

- Example (4-bit):
  - $+6 = 0110$
  - $-6 = \text{invert}(0110) = 1001 + 1 = 1010$

Range of two's complement

An n-bit two's complement number represents values from -2^{n-1} to $2^{n-1} - 1$.

- 4-bit: $-8 \ \rightarrow\ +7$
- 32-bit: $-2{,}147{,}483{,}648 \ \rightarrow\ +2{,}147{,}483{,}647$

Why two's complement?

- Only one representation for zero.
- Addition and subtraction use the **same adder circuits** as unsigned numbers.
- The carry out from the most significant bit can be ignored.

Code Snippet

```
#include <stdint.h>
#include <stdio.h>

int8_t negate8(int8_t x){
    return (int8_t)((~x) + 1);    // two's complement negation in 8 bits
}

int main(void){
    int8_t a = 53;                // 0b0011_0101
    int8_t na = negate8(a);       // should be -53
    printf("a=%d neg(a)=%d\n", a, na);
    return 0;
}
```

---

4. Arithmetic in Two's Complement (§2.4)

Two's complement arithmetic follows the same binary addition rules, with results interpreted as signed or unsigned depending on context.

Example of overflow

$0111_2\ (7) + 0001_2\ (1) = 1000_2\ (-8)$

The 4-bit signed range is $-8 \rightarrow +7$, so the result has overflowed.

Detecting overflow

- If two positive numbers yield a negative result, overflow has occurred.
- If two negative numbers yield a positive result, overflow has occurred.
- If the operands have different signs, overflow cannot occur.

Code Snippet

```
#include <stdint.h>
#include <stdio.h>

int add_overflows(int32_t a, int32_t b){
    int32_t s = a + b;
    return ((a ^ s) & (b ^ s)) < 0;  // same signs in, different sign out
}
```

```c
int main(void){
    int32_t x = 1<<30;   // large positive
    int32_t y = 1<<30;   // large positive
    int32_t s = x + y;
    printf("s=%d overflow=%d\n", s, add_overflows(x, y)); // expect overflow=1
    return 0;
}
```

Subtraction

Subtraction is performed by adding the two's complement of the subtrahend.

Example: $5 - 3$ is equivalent to $5 + (-3)$.

Code Snippet

```c
#include <stdint.h>
#include <stdio.h>

int32_t sub_via_add(int32_t a, int32_t b){
    return a + (~b + 1);  // two's complement subtraction
}

int main(void){
    int32_t a = 5, b = 3;
    printf("a-b=%d sub_via_add=%d\n", a-b, sub_via_add(a,b));
    return 0;
}
```

---

5. Endianness (§2.6, §2.8 supplement)

Endianness refers to the ordering of bytes when multi-byte data is stored in memory.

- **Little-endian:** least significant byte stored at the lowest memory address.
  This is the default on ARM and x86.

- **Big-endian:** most significant byte stored at the lowest address.

Example: Storing the 32-bit word `0x12345678` at address `0x1000`.

- Little-endian:
  - `0x1000: 78`

  - `0x1001: 56`
  - `0x1002: 34`
  - `0x1003: 12`
- Big-endian:
  - `0x1000: 12`
  - `0x1001: 34`
  - `0x1002: 56`
  - `0x1003: 78`

Endianness is important for interpreting memory dumps, file formats, and network protocols.

---

6. Characters and ASCII (§2.9)

Computers must also represent non-numeric data such as text.
This is achieved by mapping characters to integers.

- **ASCII (American Standard Code for Information Interchange):** A 7-bit encoding for 128 characters, including letters, digits, punctuation, and control codes.
  - Examples: `'A' = 65`, `'a' = 97`, newline = 10.
- **Extended ASCII:** Uses 8 bits to allow 256 characters.
  Different systems defined different extended sets.
- **Unicode:** Defines a much larger set of characters for all writing systems.
- **UTF-8:** A variable-length encoding for Unicode.
  Backward compatible with ASCII: the first 128 characters have identical byte values.

Transition to Week 6

This week we established how **integers and characters** are represented inside a computer.
We examined unsigned binary numbers, extended the system to negatives with two's complement, and explored how overflow arises in fixed-width arithmetic.
We also saw how characters are encoded in ASCII and Unicode, connecting binary storage to human-readable text.

Next week, we expand this foundation in two directions.
First, we study **bitwise operations**, which treat numbers as raw bit patterns and allow direct manipulation of individual bits — a key tool for systems programming and hardware design.
Second, we turn to **floating-point representation**, which extends binary arithmetic to fractions and real numbers using the IEEE 754 standard.
Together, these topics prepare us to handle both **low-level bit manipulation** and the **wide range of values** required in real-world scientific and engineering applications.