Week 6: Arithmetic and Floating Point

References: Patterson & Hennessy, ARM Ed., Chapter 3 and Appendix B

Review

Last week, we focused on **data representation for integers and characters**. We saw how unsigned binary counts naturally from 0 upward, how two's complement extends this system to include negative numbers, and how characters are encoded with ASCII and Unicode.

This week, we build on those ideas.
Real-world computing requires handling **fractions, extremely large values, and very small values**.
Scientific and engineering programs cannot survive on integers alone — they rely on floating-point arithmetic.
In addition, we study **bitwise operations**, which treat numbers as raw bit patterns rather than as quantities.
These are essential for low-level programming, systems design, and hardware control.

---

1. Bitwise Operations (§3.6)

Arithmetic interprets a binary pattern as a number.
Bitwise operations ignore numeric meaning and instead work directly on the individual bits of the pattern.
Each bit of the output depends only on the corresponding input bits.

- **AND (`&`)**: Produces `1` only if both inputs are `1`.
  - Example: 1101_2 & $1011_2 = 1001_2$.
- **OR (`|`)**: Produces `1` if either input is `1`.
  - Example: 1101_2 | $1011_2 = 1111_2$.
- **XOR** ($\oplus$): Produces `1` if the inputs differ.
  - Example: $1101_2 \oplus 1011_2 = 0110_2$.
- **NOT (`~`)**: Inverts all bits.
  - Example: '~$1101_2 = 0010_2$ (in 4-bit representation).

Shift operations

- **Logical left shift:** Shifts bits left, filling with zeros on the right. Equivalent to multiplying by powers of 2.

  - Example: $00010101_2 \ll 2 = 01010100_2$ (21 shifted left 2 = 84).
- **Logical right shift:** Shifts bits right, filling with zeros on the left. Equivalent to dividing by powers of 2.
  - Example: $10010000_2 \gg 3 = 00010010_2$ (144 shifted right 3 = 18).
- **Arithmetic right shift:** Preserves the sign bit (the leftmost bit in two's complement) while shifting right.
  - Example: 11101000_2 (–24) arithmetic right shift by $2 = 11111010_2$ (–6).

Applications of bitwise operations

- **Masking bits:** Using AND with a mask isolates certain bits.
  - Example: $10111101_2 \ \& \ 00001111_2 = 00001101_2$ extracts the lower 4 bits.
- **Clearing bits:** Mask with AND and zeros.
  - Example: $10111101_2 \ \& \ 11110000_2 = 10110000_2$ clears the lower 4 bits.
- **Setting bits:** Mask with OR and ones.
  - Example: $10010000_2 \mid 00001111_2 = 10011111_2$.
- **Toggling bits:** XOR with ones flips certain bits.

Bitwise operations are essential for device control, networking protocols, cryptography, and performance-critical software.
They are also the basis of instruction decoding and field extraction in CPU design.

Code Snippet

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

// Create a mask for bits [hi:lo], inclusive, assuming 0 <= lo <= hi < 32.
static inline uint32_t mask_hi_lo(unsigned hi, unsigned lo){
    unsigned width = hi - lo + 1;
    return (width == 32) ? 0xFFFFFFFFu : ((1u << width) - 1u) << lo;
}

static inline uint32_t get_field(uint32_t x, unsigned hi, unsigned lo){
    return (x >> lo) & ((1u << (hi - lo + 1)) - 1u);
}

static inline uint32_t set_field(uint32_t x, unsigned hi, unsigned lo, uint32_t v){
    uint32_t m = mask_hi_lo(hi, lo);
    return (x & ~m) | ((v << lo) & m);
}
```

```c
int main(void){
    uint32_t word = 0xABCD1234u;        // example packed word
    uint32_t rd   = get_field(word, 11, 8);   // extract [11:8]
    uint32_t rs   = get_field(word, 7, 4);    // extract [7:4]
    uint32_t nw   = set_field(word, 11, 8, 0xF); // set [11:8] to 0xF
    printf("rd=%u rs=%u nw=0x%08X\n", rd, rs, nw);
    return 0;
}
```

This snippet defines generic helper functions for creating masks, extracting fields, and inserting values into fields of a 32-bit word.

The `mask_hi_lo` function builds a mask covering bit positions `[hi:lo]`.
It uses bit shifting to create the correct width and then shifts left by `lo` to align the mask.
The corner case `width == 32` is special because shifting by 32 bits is undefined in C.

The `get_field` function right-shifts the word down to align the target field with the least-significant bit, then ANDs with a width mask to isolate it.

The `set_field` function clears the existing field using `~m` and then ORs in the new value after shifting it into place.

This demonstrates how CPUs decode instructions.
Instruction fields (opcode, register indices, immediate values) are simply slices of a binary word.
Masks and shifts let software extract and modify these slices without ambiguity.

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

int main(void){
    uint32_t u = 0xF0000000u;            // logical right shift: fills with 0
    int32_t  s = (int32_t)0xF0000000u;   // arithmetic right shift: sign-propagation on many

    printf("u >> 4 = 0x%08X (logical)\n", (u >> 4));
    printf("s >> 4 = 0x%08X (impl-defined for negatives)\n", (uint32_t)(s >> 4));
    return 0;
}
```

This program compares right shifting of an unsigned integer versus a signed integer.

For the unsigned value `0xF0000000`, shifting right fills in with `0` bits on the left. This is defined as a **logical right shift**, which is well-specified for unsigned types.

For the signed value `0xF0000000` reinterpreted as `int32_t`, the shift operation

depends on the implementation.
Many compilers propagate the sign bit (filling with ones for negative numbers), which is called an **arithmetic right shift**.
However, the C standard does not guarantee this for negative values, so the result is technically implementation-defined.

Always use **unsigned types** for portable bit-level manipulations.
Signed right shift may work as arithmetic shift on most compilers, but you cannot rely on it in cross-platform code.

---

2. Fixed-Point Representation (§3.5 and Appendix B.6)

Integers cannot express fractions.
The simplest way to store fractional values is **fixed-point representation**, where the binary point (like a decimal point in base 10) is placed at a predetermined position.

Example:
$101.101_2 = 4 + 0 + 1 + 0.5 + 0 + 0.125 = 5.625_{10}$

In this case:
- Bits to the left of the binary point represent powers of 2 ($2^2 = 4$, $2^1 = 2$, $2^0 = 1$).
- Bits to the right represent fractions ($2^{-1} = 0.5$, $2^{-2} = 0.25$, $2^{-3} = 0.125$).

Advantages of fixed-point

- Simple to implement in hardware.
- Useful in embedded systems or digital signal processing (DSP).

Limitations of fixed-point

- Precision and range are both fixed.
- Example: With 8 bits and 4 fractional bits:
  - Largest representable value = $1111.1111_2 = 15.9375$
  - Smallest increment = $0.0001_2 = \frac{1}{16} = 0.0625$
- Cannot dynamically adjust to represent both very large and very small numbers.

Fixed-point works when the range of expected values is known in advance, but it is not general-purpose enough for modern computing.
This limitation is why floating-point representation became the standard.

Code Snippet

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

typedef int32_t q16_16;

#define Q_ONE   ((q16_16)0x00010000)        // 1.0 in Q16.16
#define TO_Q(x) ((q16_16)((x) * 65536.0))   // double -> Q16.16
#define TO_D(x) ((double)(x) / 65536.0)     // Q16.16 -> double

static inline q16_16 q_add(q16_16 a, q16_16 b){ return a + b; }

static inline q16_16 q_mul(q16_16 a, q16_16 b){
    // widen to 64-bit to avoid overflow, then downshift by 16 fractional bits
    int64_t prod = (int64_t)a * (int64_t)b;
    return (q16_16)(prod >> 16);
}

int main(void){
    q16_16 a = TO_Q(1.25);   // 1.25
    q16_16 b = TO_Q(2.50);   // 2.50
    q16_16 s = q_add(a, b);
    q16_16 m = q_mul(a, b);
    printf("sum=%f mul=%f\n", TO_D(s), TO_D(m));
    return 0;
}
```

This code implements a Q16.16 fixed-point format using a signed 32-bit integer.

- The upper 16 bits represent the integer part.
- The lower 16 bits represent the fractional part.

`TO_Q` multiplies a floating-point number by 2^16 and stores it in the fixed-point type.
`TO_D` divides by 2^16 to recover the floating-point equivalent.

Addition is straightforward because both numbers share the same scale.
Multiplication is more subtle: multiplying two Q16.16 values produces a Q32.32 intermediate result.
This is stored in a 64-bit integer, then shifted right by 16 to scale back down to Q16.16.

Fixed-point arithmetic trades dynamic range for simplicity and determinism.
It is especially useful in embedded systems where floating-point hardware is unavailable or too costly.

However, it requires careful scaling and awareness of overflow.

---

3. IEEE 754 Floating-Point Standard (§3.5)

To handle a vast range of values with limited bits, computers use **floating-point numbers**, modeled after scientific notation.

General form:
$(-1)^{\text{sign}} \times 1.\text{fraction} \times 2^{(\text{exponent}-\text{bias})}$

32-bit single precision

- 1 sign bit.
- 8 exponent bits (bias = 127).
- 23 fraction bits (also called mantissa).

64-bit double precision

- 1 sign bit.
- 11 exponent bits (bias = 1023).
- 52 fraction bits.

Example: Represent 6.5 in single precision

- Step 1: Convert to binary $\rightarrow 110.1_2$.
- Step 2: Normalize $\rightarrow 1.101_2 \times 2^2$.
- Step 3: Sign = 0 (positive).
- Step 4: Exponent $= 2 + 127 = 129 = 10000001_2$.
- Step 5: Fraction = .101 $\rightarrow$ 101000 ... (padded to 23 bits).

Final representation:
0 10000001 10100000000000000000000

Floating-point allows the same 32-bit pattern to represent values as small as approximately 10^{-38} and as large as approximately 10^{38}.
This enormous dynamic range is what makes floating-point indispensable in scientific and engineering computation.

Code Snippet

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

static uint32_t float_to_u32(float f){
    uint32_t u;
    memcpy(&u, &f, sizeof u);
    return u;
}

static float u32_to_float(uint32_t u){
    float f;
    memcpy(&f, &u, sizeof f);
    return f;
}

int main(void){
    float f = 6.5f; // example from notes
    uint32_t u = float_to_u32(f);
    uint32_t sign = (u >> 31) & 0x1u;
    uint32_t exp  = (u >> 23) & 0xFFu;
    uint32_t frac =  u        & 0x7FFFFFu;

    printf("f=%g raw=0x%08X sign=%u exp=0x%02X frac=0x%06X\n", f, u, sign, exp, frac);

    // Rebuild the same float from the bits
    float g = u32_to_float(u);
    printf("rebuilt=%g\n", g);
    return 0;
}
```

This snippet reinterprets a `float` as a 32-bit unsigned integer without breaking C's strict aliasing rules.
The conversion is done safely by copying the bytes with `memcpy`.

Once in integer form, the code isolates the sign, exponent, and fraction fields by shifting and masking.
These fields correspond exactly to the IEEE 754 single-precision layout:

- Sign: 1 bit at position 31.
- Exponent: 8 bits at positions [30:23].
- Fraction: 23 bits at positions [22:0].

Finally, the program reconstructs the original float by reversing the process.

This shows that a floating-point number is not "magical."
It is just a 32-bit binary word with fields that can be dissected and manipulated.
The representation directly matches the IEEE 754 standard.

---

4. Floating-Point Arithmetic (§3.5)

Floating-point arithmetic is more complex than integer arithmetic because of normalization and rounding.

Addition and subtraction

1. Align exponents (shift the smaller number).
2. Add or subtract the fractions.
3. Normalize the result (shift until it is in `1.x` form).
4. Round if necessary.

Example: 1.5 + 2.25

- $1.5 = 1.1_2 \times 2^0$
- $2.25 = 1.001_2 \times 2^1$
- Align $\rightarrow$ $1.5 = 0.11_2 \times 2^1$
- Add $\rightarrow$ $0.11_2 + 1.001_2 = 1.111_2$
- Normalize $\rightarrow$ $1.111_2 \times 2^1 = 3.75_{10}$

Multiplication and division

- Multiply or divide fractions.
- Add or subtract exponents.
- Normalize result.

Rounding modes

Since fraction bits are limited, results must be rounded.
IEEE 754 defines four modes:

- Round to nearest, ties to even (default).
- Round toward 0.
- Round toward $+\infty$.
- Round toward $-\infty$.

Rounding ensures predictable and consistent results across platforms.
It also means floating-point arithmetic is not exact — tiny errors can accumulate in long computations.

Code Snippet

```c
#include <stdio.h>
#include <float.h>
#include <math.h>

int main(void){
    float a = 0.1f, b = 0.2f, c = 0.3f;
    float s = a + b;

    printf("0.1f + 0.2f = %.9f\n", s);
    printf("0.3f         = %.9f\n", c);
    printf("equal? %s\n", (s == c) ? "yes" : "no");

    printf("FLT_EPSILON = %.9g\n", FLT_EPSILON);
    printf("|s - 0.3|   = %.9g\n", fabsf(s - c));
    return 0;
}
```

This snippet adds `0.1f + 0.2f` and compares the result to `0.3f`.

In decimal, we expect equality.
In binary floating-point, neither 0.1 nor 0.2 can be represented exactly.
Their approximations add to a value slightly greater than 0.3.

The program prints both results to nine decimal places, showing the discrepancy.
It then compares them for equality and finds that the result is not equal.

Finally, it prints `FLT_EPSILON`, the smallest value such that `1.0f + FLT_EPSILON > 1.0f`.
It also shows the absolute error between the sum and 0.3, which is on the order of `FLT_EPSILON`.

Floating-point arithmetic is approximate.
Errors accumulate not because of bugs but because of fundamental limits of

binary representation.
Machine epsilon quantifies the precision limits for a given floating-point format.

---

5. Special Values in IEEE 754 (§3.5)

Floating-point has special bit patterns to handle cases outside normal arithmetic.

- **Zero:** Exponent = 0, fraction = 0.
  There are +0 and –0, but they behave the same in most operations.
- **Denormalized numbers:** Exponent = 0, fraction $\neq$ 0.
  These represent values very close to zero, filling the gap between zero and the smallest normalized number.
- **Infinity:** Exponent = all 1s, fraction = 0.
  Results from overflow or division by zero.
  Can be $+\infty$ or $-\infty$.
- **NaN (Not a Number):** Exponent = all 1s, fraction $\neq$ 0.
  Used for invalid operations such as `0/0`, ∞ - ∞, or the square root of a negative number.

These rules make floating-point arithmetic robust and allow programs to continue executing even when encountering unusual values.
Instead of crashing, computations produce Infinity or NaN, which can then be tested and handled in software.

Code Snippet

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

int main(void){
    double z = 0.0;
    double inf = 1.0 / z;          // +Infinity
    double nan = 0.0 / 0.0;        // NaN

    printf("inf: isinf=%d isnan=%d isfinite=%d sign=%d\n",
           isinf(inf), isnan(inf), isfinite(inf), signbit(inf));

    printf("nan: isinf=%d isnan=%d isfinite=%d\n",
           isinf(nan), isnan(nan), isfinite(nan));

    int cls = fpclassify(nan);
    printf("fpclassify(nan)=%d (FP_NAN=%d)\n", cls, FP_NAN);
```

```
    return 0;
}
```

This program generates special floating-point values by dividing by zero and zero over zero.

- `1.0 / 0.0` produces positive infinity.
- `0.0 / 0.0` produces NaN.

The functions `isinf`, `isnan`, and `isfinite` check for these conditions.
`signbit` detects whether a value is negative at the bit level, even for zero.
`fpclassify` returns a category such as `FP_NAN`, `FP_INFINITE`, `FP_ZERO`, `FP_SUBNORMAL`, or `FP_NORMAL`.

IEEE 754 defines explicit encodings for Infinity and NaN so that computations can continue without crashing.
Software can detect these values and handle them explicitly, which is vital in scientific computing.

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

int main(void){
    printf("FLT_MIN      = %.10e (smallest normalized)\n", FLT_MIN);
    printf("FLT_TRUE_MIN = %.10e (smallest subnormal)\n", FLT_TRUE_MIN);

    float x = FLT_MIN;
    for(int i=0;i<5;i++){
        x *= 0.25f; // progressively underflow
        printf("x = %.10e\n", x);
    }
    return 0;
}
```

This program prints the smallest normalized float (`FLT_MIN`) and the smallest positive subnormal (`FLT_TRUE_MIN`).

It then multiplies `FLT_MIN` repeatedly by 0.25.
As the result decreases, it falls below the normalized range but remains representable as a subnormal.
Eventually, continued scaling underflows all the way to zero.

This demonstrates **gradual underflow**, where precision is lost gradually rather than suddenly at the edge of representable numbers.

Subnormals ensure that the floating-point number line is continuous.
Without them, the gap between zero and the smallest normalized number would be enormous, causing catastrophic loss of precision in some calculations.

6. Transition to Next Week

This week, we expanded our understanding of binary arithmetic to include **bitwise manipulation, fractions, and floating-point numbers**.
We examined how the IEEE 754 standard represents real numbers, how arithmetic is performed, and how special cases like infinity and NaN are handled.

Next week, we move back into **CPU architecture**, specifically the **datapath**.
We will connect the circuits we studied in Weeks 3–4 with the arithmetic operations we studied in Weeks 5–6.
This is where hardware and data representation come together to execute programs.