Review Document: Number Systems and Binary Arithmetic

1 Positional Notation and Radix

In a positional numeral system with base b, a number V is written as a sequence of digits:

$$d_{m-1}\, d_{m-2}\, \ldots\, d_1\, d_0.$$

Each digit d_i has an integer value symbol_value(d_i). The value of the entire number is computed by summing each digit multiplied by a power of b:

$$\text{value}\big(\text{number}_b\big) \;=\; \sum_{i=0}^{m-1} \Big(\text{symbol\_value}(d_i)\Big) \;\times\; b^i.$$

1.1 Examples of Common Bases

- **Binary (base 2)**: digits $\{0, 1\}$.
- **Octal (base 8)**: digits $\{0, 1, 2, 3, 4, 5, 6, 7\}$.
- **Decimal (base 10)**: digits $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- **Hexadecimal (base 16)**: digits $\{0, 1, \ldots, 9, A, B, C, D, E, F\}$.

For instance, 13_{10} in decimal is D_{16} in hexadecimal (since D corresponds to 13).

2 Converting Numbers Between Bases

2.1 Decimal to Another Base (Repeated Division)

To convert a **decimal** integer to base b:

1. **Divide** the number by b.
2. **Record** the remainder (this is the least significant digit).
3. **Update** the number to be the integer quotient of that division.
4. **Repeat** until the quotient is zero.
5. The base-b digits appear in **reverse order** of remainders.

Example: Converting 26_{10} to Hex (base 16)

$$26 \div 16 = 1 \text{ remainder } 10 \quad (\text{which is } A_{16})$$

$$1 \div 16 = 0 \text{ remainder } 1$$

Reading remainders **backwards**, we get $1A_{16}$. Hence, $26_{10} = 1A_{16}$.

Example: Converting 11_{10} to Binary (base 2)

$$11 \div 2 = 5 \text{ remainder } 1$$

$$5 \div 2 = 2 \text{ remainder } 1$$

$$2 \div 2 = 1 \text{ remainder } 0$$

$$1 \div 2 = 0 \text{ remainder } 1$$

Reading remainders in **reverse** order: 1011_2. Indeed, $11_{10} = 1011_2$.

2.2 Base b to Decimal (Polynomial Expansion)

If you have a number in base b, say $d_{m-1}d_{m-2}\ldots d_0$, you can convert it to decimal by evaluating:

$$\sum_{i=0}^{m-1} \text{symbol\_value}(d_i) \times b^i.$$

Example: 1011_2 to Decimal

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = 11_{10}.$$

3 Binary Arithmetic: Addition and Subtraction

3.1 Unsigned Binary Addition

When adding two binary numbers, you start from the least significant bit (rightmost), move left, and **carry** a 1 if the sum of bits in a column is ≥ 2.

Example (3-bit addition): $(011)_2 + (101)_2$

Carry In	Bit A	Bit B	Sum	Carry Out
–	0	1	1	0
0	1	0	1	0
0	1	1	0	1

Result: $(011)_2 + (101)_2 = (1000)_2$. The final *carry out* is 1, indicating an overflow if you only have 3 bits to store the result.

3.2 Unsigned Binary Subtraction

For subtraction, if a bit cannot be subtracted (e.g., $0 - 1$), you **borrow** from the next more significant bit.

Example (3-bit subtraction): $(100)_2 - (001)_2$

Least significant bit: $0 - 1$ is not possible without borrow.

Borrow 1 from the next column, turning it into 2_2 in that position: effectively

$$10_2 - 1_2 = 1_2.$$

4 Signed Number Representations

4.1 One's Complement

- Negative numbers are formed by **inverting** each bit of the positive counterpart.
- Each binary digit d_i is replaced by $\bar{d}_i$.

For a 3-bit example: $+3 = 011_2$, so -3 in one's complement is 100_2.

4.2 Two's Complement

- Negative numbers are formed by **inverting** (one's complement) then **adding 1**.
- This representation is widely used because the same hardware circuits can do addition/subtraction for both positive and negative numbers.

Example (3-bit two's complement for -3):

$$+3 = 011_2$$

Invert bits: $011_2 \rightarrow 100_2$

Add $1 : 100_2 + 001_2 = 101_2$.

Hence, $-3 = 101_2$ in 3-bit two's complement.

5 Practical Highlights

1. **Overflow**:
   - In *unsigned* arithmetic, adding two values can exceed the representable range.
   - In *two's complement* arithmetic, overflow can happen when adding two numbers with the same sign yields a result that doesn't fit within the bit width.

2. **Hex $\leftrightarrow$ Binary**:
    - One hex digit maps directly to 4 binary bits.
    - For example, F_{16} corresponds to 1111_2.
    - This makes hex an efficient shorthand for binary.
3. **Fixed-Width Formats** (e.g., 8-bit, 16-bit, 32-bit):
    - Computers typically store integers in fixed bit sizes.
    - Leading zeros (for unsigned) or sign extensions (for signed) fill the extra bits.

6 Conversion Algorithms at a Glance

1. **Decimal $\rightarrow$ Binary (or base b)**:
    - Repeated **divide** by b, track remainders, read them in reverse.
2. **Binary (or base b) $\rightarrow$ Decimal**:
    - Polynomial expansion: each digit times b^i.
3. **Hex $\leftrightarrow$ Binary**:
    - Split the binary number into groups of 4 bits (from right to left).
    - Convert each group to a single hex digit.

6.1 Example in LaTeX

If you want to show your intermediate steps nicely in LaTeX, consider:

$$26_{10} \div 16 = 1 \text{ remainder } 10 \quad \Rightarrow \quad \text{digit}_0 = A$$

$$1_{10} \div 16 = 0 \text{ remainder } 1 \quad \Rightarrow \quad \text{digit}_1 = 1$$

Reading digits from last to first: $1A_{16}$.

7 Study Tips

- Practice with small examples first (3-bit or 4-bit).
- Check each step carefully for borrowing or carry bits.
- Relate your results to actual hardware or assembly instructions by seeing how the CPU flags overflow and carry in real architectures.
- Remember the difference between signed (two's complement) and unsigned operations; the same bit pattern can represent different decimal values depending on interpretation.

8 Summary

- **Positional Notation**: Every base b number is expanded via digits times powers of b.
- **Subscripts/Prefixes**: Clarify the base (e.g., 101_2, 0xA, 0b1011).
- **Arithmetic**: Binary addition/subtraction uses carry and borrow. Overflow is inevitable when results exceed the bit width.
- **Signed Representation**: Modern systems typically use two's complement because of its unified handling of negative and positive integers.
- **Conversions**: Apply repeated division or polynomial expansions to move between bases.

Use these core principles to interpret numeric values at the hardware level, debug assembly code, or reason about compiler output. Mastery of these foundational concepts will inform all your work with digital logic and computer organization going forward.