Computer Organization and Architecture

Binary numbers

Dr Russ Ross

Utah Tech University—Department of Computing

Spring 2026

Why binary?

Everything in a computer is represented as bits.

- Programs
- Data
- Instructions
- Memory addresses
- Flags and status information

Why bits? Why not use decimal directly?

- Two stable states (0 and 1)
- Easy to distinguish electrically
- Simple logic operations
- Reliable storage and transmission

Fixed-width representation

CPU registers have fixed sizes:

- A 32-bit register holds exactly 32 bits
- A 64-bit register holds exactly 64 bits
- An 8-bit value in an 8-bit register uses all bits

What happens when a value is too big to fit?

- The CPU doesn't make it bigger
- It wraps around
- This is the **wraparound behavior** of fixed-width arithmetic

How many combinations?

With n bits, how many different values can we represent?

- 1 bit: 2 values
- 2 bits: 4 values
- 3 bits: 8 values
- n bits: 2^n values

An 8-bit register can represent $2^8 = 256$ different values.

A 32-bit register can represent 2^{32} different values.

A 64-bit register can represent 2^{64} different values.

Addition (8-bit unsigned)

Let's add 50 + 25 in 8-bit unsigned:

```
  00110010     (50 in decimal)
+ 00011001     (25 in decimal)
----------
  01001011     (75 in decimal)
```

No problem—the result fits in 8 bits.

What about 200 + 100?

```
   11001000     (200 in decimal)
 + 01100100     (100 in decimal)
-----------
 1 00101100
```

The result is 300, but it doesn't fit in 8 bits. What happens?

Overflow in unsigned arithmetic

When the result doesn't fit in the register, we get **overflow**:

```
     11001000     (200 in decimal)
   + 01100100     (100 in decimal)
  -----------
  [1]00101100     (The 1 is discarded, leaving 00101100)
```

The 9th bit overflows and is lost. We're left with `00101100` = 44.

The CPU has a **carry flag** that records whether an overflow occurred. The CPU doesn't automatically know you overflowed—you have to check the flag.

Subtraction (8-bit unsigned)

Let's subtract 75 - 50:

```
   01001011      (75 in decimal)
 - 00110010      (50 in decimal)
 ----------
   00011001      (25 in decimal)
```

What about subtracting a larger value from a smaller one, like 50 - 75?

In unsigned arithmetic, the result doesn't go negative—it wraps around. In this case, $50 - 75 = -25$ mathematically, but the hardware underflows and wraps to give us 231.

Another way to think about it: the circle model

Instead of thinking of integers as an infinite number line, think of fixed-width integers as arranged in a circle:

- For 8-bit: values 0 through 255 arranged in a circle with 0 at the top
- Adding moves clockwise around the circle
- Subtracting moves counter-clockwise
- If you "fall off the edge," you wrap to the other side seamlessly
- The CPU sets a carry flag when you cross the boundary

This matches how the hardware actually works: wraparound is automatic and expected.

Evolution of processor register widths

The “bitwidth” of a CPU determines how much data fits in a register.

Architecture	Year	Bits
MOS 6502	1975	8
Motorola 68000	1979	16
Intel 8086	1978	16
Intel 80386	1985	32
Intel Pentium Pro	1995	32
AMD64 / Intel x86-64	2003-2004	64
ARM 32-bit	1985 (ARMv1)	32
ARM 64-bit (ARMv8)	2011	64
RISC-V 32-bit (RV32)	2014	32
RISC-V 64-bit (RV64)	2014	64

In this course we’ll focus on **32-bit RISC-V**, but the concepts apply to all architectures.

AND: Turning bits off

The **AND** operation works bit-by-bit:

```
  10110101
& 10101010      (the "mask")
----------
  10100000
```

Each bit of the result is 1 only if both input bits are 1. Otherwise it's 0.

Common pattern: Use AND with a **mask** to turn specific bits off:

```
value & 0xF0    // Keep high nibble, turn low nibble to 0
value & 0x01    // Isolate the least-significant bit (odd/even test)
```

Practical example: Test if a number is odd or even:

- 5 `&` 1 $\rightarrow$ 1 (odd)
- 14 `&` 1 $\rightarrow$ 0 (even)

OR: Turning bits on

The **OR** operation works bit-by-bit:

```
  10110101
| 10101010      (the "mask")
----------
  10111111
```

Each bit of the result is 1 if either input bit is 1. Otherwise it's 0.

Common pattern: Use OR with a **mask** to turn specific bits on:

```
value | 0x80    // Turn on the high bit
value | 0x01    // Turn on the least-significant bit
```

XOR: Flipping bits

The **XOR** (exclusive OR) operation works bit-by-bit:

```
  10110101
^ 10101010      (the "mask")
----------
  00011111
```

Each bit of the result is 1 if the input bits are different, 0 if they're the same.

Common pattern: Use XOR with a **mask** to toggle (flip) specific bits:

```
value ^ 0xFF    // Flip all bits (same as bitwise NOT for 8 bits)
value ^ 0x01    // Toggle the least-significant bit
```

Key property: Applying XOR twice with the same mask returns to the original value. This makes XOR useful for encryption: XOR plaintext with a keystream to encrypt, then XOR the ciphertext with the same keystream to decrypt.

NOT: Bitwise complement

The **NOT** operation flips every bit:

```
~ 10110101
----------
  01001010
```

This is different from logical NOT (`!`). Bitwise NOT works on every bit individually.

Note: `~1` is not 0, it's `0xFFFFFFFE` on a 32-bit system (31 ones and one zero).

Where do we use these?

These operations seem strange at first, but they show up everywhere in computer science:

Permissions and flags:

- Unix file permissions: read (4), write (2), execute (1)
- Use AND to check if a bit is set
- Use OR to add permissions
- Use XOR to toggle permissions

Hardware registers:

- Each bit of a register controls a feature
- Use AND to read a specific bit
- Use OR to enable a feature
- Use AND with NOT to disable a feature

Where do we use these?

Networking:

- IP address masks (e.g., `192.168.1.0 & 255.255.255.0`)
- Protocol headers with packed fields

Graphics:

- RGB colors: Pack three 8-bit channels into one 32-bit value
  - Red shifted left 16 bits: `R << 16`
  - Green shifted left 8 bits: `G << 8`
  - Blue in low bits: `B`
  - Combined: `(R << 16) | (G << 8) | B`
- To extract a channel, use masking and shifting (e.g., extract blue with `color & 0xFF`)
- Transparency and alpha blending
- Pixel manipulation

Where do we use these?

These are standard in high-level languages:

These aren't assembly-only tricks. C, Java, Python, etc. all support bitwise operations:

C:

```
int flags = 0;
flags |= (1 << 2);          // Set bit 2
if (flags & (1 << 2)) { }   // Check if bit 2 is set
flags &= ~(1 << 2);         // Clear bit 2
```

Python:

```
flags = 0
flags |= (1 << 2)
if flags & (1 << 2):
    pass
```

The problem with unsigned

Unsigned integers can only represent non-negative values (0 and up).

What if we want to represent negative numbers in fixed-width bits?

- We can't add a minus sign
- We need a scheme that works with the same hardware
- And addition/subtraction must still work correctly

How would you encode -1 in 8 bits if positive numbers use 0-127?

How would you encode -128?

Two's complement representation

Two's complement is a scheme where:

- Half the values are non-negative (0 and up)
- Half the values are negative
- The representation wraps around in a circle
- Addition and subtraction work identically whether the numbers are signed or unsigned

For 8-bit two's complement:

- Range: -128 to +127 (not 0 to 255)
- Bit 7 (the high bit) is the **sign bit**: 1 = negative, 0 = non-negative
- The rest of the bits encode the magnitude

Which representation is which? Draw it as a circle with zero at the top.

- Outside the circle: unsigned interpretation (0-255)
- Inside the circle: signed interpretation (-128 to +127)

Two's complement arithmetic

The brilliant part: **addition and subtraction work the same** regardless of whether the numbers are signed or unsigned.

```
   01010011     (83 in decimal)
 + 11111101     (-3 in two's complement)
-----------
[1]01010000     (80 in decimal, overflow bit discarded)
```

The CPU computes: $83 + (-3) = 80$. It doesn't need to know the numbers are signed—the math just works.

Overflow in signed arithmetic

With unsigned numbers, overflow sets a carry flag.

With signed numbers, we get **signed overflow** when the result is too large or too small to fit:

```
  01111111      (127, the largest positive 8-bit number)
+ 00000001      (1)
----------
  10000000      (-128 in signed, but we wanted 128)
```

The result wrapped around to a negative number! The CPU has a **signed overflow flag** (V flag) to detect this.

Different flags for different contexts: always check the one you need.

Computing negatives: Flip and add one

To compute $-x$ from x, use this procedure:

1. Flip all the bits (bitwise NOT)
2. Add 1

Example: negating 3 in 4 bits

```
   0011        (3)
   1100        (flip all bits)
+  0001        (add 1)
--------
   1101        (-3 in two's complement)
```

Special cases:

- Negating 0 yields 0 (the extra carry bit is discarded)
- Negating the most negative value (e.g., -8 in 4 bits) wraps to itself

Column interpretation

The key insight: the most significant bit column is **negative** instead of positive.

For 4-bit signed numbers, the columns are: -8, 4, 2, 1

Example: What is `1101` in signed two's complement?

- $1 \times (-8) + 1 \times 4 + 0 \times 2 + 1 \times 1 = -8 + 4 + 1 = -3$

This is why the range for 4-bit signed is -8 to +7 (the most negative value has no positive counterpart).

Sign extension

When expanding a signed value to a larger width, replicate the sign bit:

- Positive numbers (sign bit $= 0$): add leading zeros
- Negative numbers (sign bit $= 1$): add leading ones

Example: Extend `1101` (-3 in 4 bits) to 8 bits:

```
1101            (4-bit: -3)
11111101        (8-bit: still -3)
```

Why this works: when you add the new leading 1, it contributes $-(2^7) = -128$ to the value. But the old sign bit, now in the 2^6 position, contributes $+64$. The net change is $-128 + 64 = -64$. Meanwhile, we've shifted the old value left by one place, contributing another -64 offset. These exactly cancel out, preserving the value.

Shifts and multiplication by powers of 2

Left shifting a value (signed or unsigned) effectively multiplies it by 2:

```
    0101    (5 in decimal)
    1010    (5 << 1 = 10 in decimal)
```

This works identically for negative two's complement values:

```
    1011    (-5 in 4-bit signed)
    0110    (-5 << 1 = -10 in 4-bit signed, wrapping)
```

Right shifting a signed value arithmetically (copying the sign bit) divides by 2, rounding toward negative infinity.

Subtraction in hardware

The CPU can implement subtraction using the adder and minimal extra logic:

To compute $A - B$:

1. Flip all bits of B (bitwise NOT)
2. Add 1 (negate B)
3. Add the result to A using the normal adder: $A + (-B)$

The "clever trick": use the carry-in of the adder to add the 1, avoiding a separate increment step.

Why this matters: the same hardware adder works for both addition and subtraction.

Carry vs. overflow flags

The CPU sets flags to help detect errors:

Carry flag (unsigned context): * Set when addition produces a carry out, or subtraction requires a borrow * Signals wraparound in unsigned arithmetic

Overflow flag (signed context): * Set when the result crosses the signed boundary (e.g., positive $+$ positive $\rightarrow$ negative) * Signals when a signed result is out of range

These are separate flags for different use cases. Always check the appropriate one for your context.

Comparisons via subtraction

To compare two values, subtract one from the other and interpret the result:

- If $A - B = 0$, then $A = B$
- If $A - B > 0$ (result is positive), then $A > B$
- If $A - B < 0$ (result is negative), then $A < B$

The CPU uses the sign bit and overflow flag to determine whether the result is truly positive or negative (in case of signed overflow).

This is why comparisons and conditionals are cheap: they're just subtraction plus flag checking.

Approximating magnitude

Every 10 binary digits represents roughly $2^{10} = 1024 \approx 10^3$:

- 10 bits $\approx$ 1,000 (one thousand)
- 20 bits $\approx$ 1,000,000 (one million)
- 30 bits $\approx$ 1,000,000,000 (one billion)

You can use this to quickly estimate the magnitude of a number given its bit width.

For example, how many bits do you need to represent 1 million?

- 20 bits gives $\approx$ 1M, but we need a few more
- $2^{20} = 1,048,576$

16 million colors

An image pixel in RGB uses three 8-bit color channels:

- Red: 0-255 (256 values)
- Green: 0-255 (256 values)
- Blue: 0-255 (256 values)

How many distinct colors can we represent?

$256 \times 256 \times 256 = 2^{24}$ possibilities.

That's why we say "16 million colors" (approximately).

Can you write out the exact number?

4 billion IPv4 addresses

An IPv4 address is a 32-bit number.

2^{32} possible addresses $\approx$ 4.3 billion.

Example: `192.168.1.1` in binary is:

```
11000000.10101000.00000001.00000001
```

(Each octet is one byte.)

We've run out of IPv4 addresses because there are more than 4 billion devices on the internet. That's one reason we have IPv6 (128-bit addresses).

A challenge: USA population

The USA has approximately 330 million people.

How many bits would you need to assign a unique ID to every person in the USA?

- $2^{28} =$ 268 million
- $2^{29} =$ 536 million

What's the answer?

Hexadecimal: human-friendly binary

Binary is hard to read due to long strings of bits:

```
10110101010101110110100010101010
```

Hexadecimal (base 16) is the standard abbreviation:

- Each hex digit represents exactly 4 binary bits
- A byte (8 bits) is exactly 2 hex digits
- Much more compact and easier to read
- Widely used in memory dumps, debugging, and low-level inspection

Hex digits: 0–9, A–F (where A=10, B=11, …, F=15)

Hexadecimal: human-friendly binary

Example: Group binary into 4-bit chunks:

```
    1011 0101 0101 0111 0110 1000 1010 1010
      B    5    5    7    6    8    A    A
```

Result: `0xB557685A`

The transparency is key: you can see the binary pattern directly in the hex (many F's = many 1 bits, 0's = 0 bits).

Octal (base 8) exists but is less convenient with bytes (8 bits is not a multiple of 3). You'll see it in Unix file permissions (`0755`) and legacy code, but hexadecimal dominates modern usage.

Key ideas

- Everything in a computer is bits, arranged in fixed-width registers
- **Fixed-width arithmetic causes wraparound**: values wrap around at boundaries
- **Bitwise operations** (AND, OR, XOR, NOT) are fundamental and implemented at the hardware level as logic gates
- **Two's complement**: elegant scheme for signed integers where:
    - The most significant bit is negative instead of positive
    - The same adder hardware works for both signed and unsigned
    - Subtraction is implemented as negation (flip bits, add 1) followed by addition
- **Sign extension**: when expanding to a larger width, replicate the sign bit to preserve the value
- **Magnitude estimation**: every 10 bits ≈ 3 decimal orders of magnitude (since $2^{10} \approx 10^3$)
- **Comparisons and conditionals** are just subtraction followed by flag checking
- **Hexadecimal** is the human-friendly notation: each hex digit = 4 bits, a byte = 2 hex digits
- These concepts apply universally across all CPU architectures, not specific to any one ISA