CS161: Design and Architecture of Computer Systems
Expanded Notes – Execution Example and Binary Arithmetic

University of California, Riverside

Overview

This lecture introduces how a computer actually executes programs, what is inside the machine, and how the processor coordinates computation, memory, and control. It also begins a review of binary arithmetic: how numbers are represented in base 2, how addition works, and how signed integers are encoded using two's complement.

The material is intended as an overview. Later lectures will dive deeper into datapath design, instruction sets, and handling of corner cases.

What is Inside a Computer?

A computer is not just a processor. It is a system composed of multiple interacting parts:

- **Processors** (CPU, and sometimes GPU or accelerators).
- **Memory (RAM)** for temporary data.
- **Input/Output** devices like keyboard, mouse, monitor, network interfaces.
- **Storage** such as flash drives or hard disks for permanent data.

Modern systems are evolving. For instance, flash storage is now so fast that it connects directly over PCIe rather than through slower disk interfaces.

Inside a Processor

Within the processor, we can identify four major roles:

- **Compute:** Units that perform arithmetic or logic (add, subtract, multiply, comparisons).
- **Data Registers:** Small but very fast storage close to the compute units.
- **Memory:** Much larger storage, but relatively slow.
- **Control:** The logic that decides what happens next (if-statements, loops, jumps).

For example, consider the simple C program:

```
while (i != 2) {
    i = i + 1;
}
```

To implement this, the processor must: load `i`, compare it to 2, increment it if not equal, and loop back. This requires coordination between compute and control.

Walking Through Execution

The processor executes instructions in a cycle:

1. Load the instruction from memory.
2. Decode the operation.
3. Identify the input data (from registers or memory).
4. Perform the computation.
5. Determine the next instruction (sequential or jump).

For the loop example:

1. Load `i` into a register.
2. Subtract 2 and check if the result is zero.
3. If zero, jump to exit; otherwise increment `i`.
4. Repeat until the condition is satisfied.

This illustrates how high-level code becomes a sequence of simple operations on registers, controlled by conditional jumps.

Binary Arithmetic Basics

All of these operations ultimately rely on circuits that only handle two states: on (1) and off (0). Thus, computers represent numbers in binary (base 2).

Examples:

$$0_2 = 0_{10}, \quad 1_2 = 1_{10}, \quad 10_2 = 2_{10}, \quad 11_2 = 3_{10}, \quad 100_2 = 4_{10}.$$

A binary number is evaluated by summing powers of two:

$$dcba_2 = d \cdot 2^3 + c \cdot 2^2 + b \cdot 2^1 + a \cdot 2^0.$$

Binary Addition

Addition works like in decimal: if a sum exceeds the base, a carry is generated.

$$01_2 + 01_2 = 10_2 \quad \text{(which equals } 1 + 1 = 2_{10}\text{)}.$$

With larger numbers:

$$0011_2 + 0110_2 = 1001_2 \quad (3{+}6{=}9).$$

Each step uses three inputs: the two bits to add and a carry-in. The result is a sum bit and a carry-out. The carry-out of one column becomes the carry-in of the next.

Overflow

With n bits, only numbers from 0 to $2^n - 1$ can be represented. Adding beyond this range produces overflow. For example, with 4 bits:

$$1011_2 + 0110_2 = 0001_2 \quad \text{(11+6=17, but 17 cannot be represented in 4 bits)}.$$

Signed Numbers: Two's Complement

Unsigned numbers are straightforward, but signed numbers require a convention. The standard used in computers is **two's complement**.

In two's complement:

- The most significant bit (MSB) has a negative weight.
- For n bits, the range is from -2^{n-1} to $2^{n-1} - 1$.
- There is only one representation of zero ($0000\ldots_2$).
- Negation is easy: invert all bits and add 1.

For example, with 4 bits:

$$1000_2 = -8_{10}, \quad 1111_2 = -1_{10}, \quad 0100_2 = 4_{10}.$$

Example addition:

$$(-5) + 4 = 1011_2 + 0100_2 = 1111_2 = -1.$$

Why two's complement?

- Addition and subtraction work seamlessly.
- Only one zero exists.
- Easy to detect negativity (check MSB).
- Subtraction reduces to addition with the negated value.

Why This Matters

Understanding binary arithmetic and execution steps is fundamental for grasping what assembly instructions actually mean. When a processor executes "subtract 2," "add 1," or "is zero?", it is really just manipulating binary numbers encoded in two's complement.

This lecture sets the stage: later we will explore in detail how MIPS processors implement these operations with datapaths, control logic, and pipelines.