Computer Organization

For Everyone

Shayne Wadle and Karthikeyan Sankaralingam

Contents

1 The Big Ideas of Computing

Welcome to the study of computer organization! Before we dive into the nuts and bolts of bits, bytes, and processors, we need to start with the foundational concepts that make computers possible. What *is* a computer, really? And how can a machine built from simple on/off switches perform tasks of such breathtaking complexity?

The answer lies in a handful of profound "Big Ideas." These are the pillars upon which the entire digital world is built. Understanding them gives you a framework for everything that follows in this book.

Big Idea #1: The Universal Computing Machine 🤖

In the 1930s, long before physical computers existed, a brilliant mathematician named **Alan Turing** asked a powerful question: Is there a limit to what can be computed? He devised a thought experiment to answer this, and in doing so, designed the theoretical blueprint for every computer that would ever be built.

He imagined a simple machine:

- An infinitely long **tape**, divided into cells, each containing a symbol (like a 0, 1, or blank).
- A **head** that can read the symbol in a cell, write a new symbol, and move left or right on the tape.
- A set of **rules** that tells the head what to do based on the machine's current state and the symbol it just read.

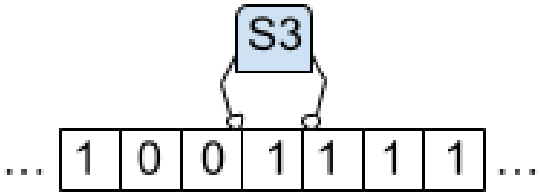


Figure 1.1: A Turing Machine head moving along the infinite tape following rule S3 currently.

This theoretical device, the **Turing Machine**, seems primitive. Yet, Turing proved something astonishing: this simple machine could compute *any* problem that was computable by *any* conceivable machine. It is a **universal computing machine**. This principle is now called **Turing Completeness**. If a system (like a programming language or a processor) is Turing complete, it has the power to simulate any other Turing machine—meaning it can compute anything that is fundamentally computable.

This is the bedrock of all modern computing: we don't need a special machine for calculating finances, another for playing music, and a third for editing photos. A single, universal machine can do it all.

Big Idea #2: The Stored-Program Computer 💾

Turing's machine was a theoretical concept. The practical leap forward was the idea of a **programmable computer**, where the instructions (the program) could be easily changed. The key innovation that made this possible is the **stored-program concept**: the idea that the instructions for the computer are stored in memory, right alongside the data the instructions operate on.

This means the program isn't physically wired into the machine; it's just data. To change what the computer does, you simply load a different program into its memory. This simple but powerful idea is the defining characteristic of every modern computer. This concept has evolved over time:

- The **Atanasoff-Berry Computer (ABC)** (early 1940s) is often considered the first electronic digital computing device, taking early steps toward this goal.
- **ENIAC** (mid-1940s) was a huge leap in speed, but it had to be physically rewired to change its program, a tedious process that could take days.
- The **IBM System/360** (1960s) introduced the idea of a family of computers that could all run the same programs (software), cementing the stored-program model in the commercial world.
- The **Intel 8086** processor (late 1970s) launched the personal computer (PC) revolution, bringing the power of the stored-program computer into homes and offices.
- **ARM** processors (popularized in the 2000s) powered the smartphone revolution with their efficient design, putting a powerful stored-program computer in billions of pockets.
- Modern **Deep Learning chips (GPUs, TPUs)** are specialized processors designed to execute the mathematical operations needed for AI at incredible speeds.

All these machines, from the room-sized ENIAC to the chip in your phone, share the same fundamental DNA: they are all implementations of the stored-program computer.

Big Idea #3: Specialization vs. Generality ⇄

If a single machine can do everything, why would we ever build one that can't? The answer is a classic engineering trade-off: **efficiency**. A general-purpose, programmable computer is flexible, but a specialized, **fixed-function** device that does only one task can be much faster, cheaper, and more power-efficient.

You're surrounded by fixed-function computers: the chip in your microwave, a simple pocket calculator, or the controller for your car's anti-lock brakes. A modern, high-stakes example is a **Bitcoin mining chip like the BM1370**. Its one and only job is to perform a specific cryptographic calculation (SHA-256) over and over, as fast as humanly possible. It can't run a web browser or a word processor, but it can perform its one task millions of times more efficiently than a general-purpose CPU.

Big Idea #4: New Frontiers in Computation 🚀

For decades, the stored-program model has been the undisputed king. But today, we are exploring fundamentally new types of computing.

- **Quantum Computing:** This is a completely different model built on the strange rules of quantum mechanics. Instead of bits (0 or 1), it uses **qubits**, which can be 0, 1, or a superposition of both at the same time. This allows quantum computers to explore a vast number of possibilities simultaneously, making them potentially millions of times faster for specific problems like materials science and code-breaking.
- **Deep Learning:** This represents a new way of *creating* a program. Instead of a human writing explicit rules (an algorithm), a program called a **model** is *generated* by training it on massive amounts of data. The model itself—a giant collection of numbers—becomes the program. You interact with it using **prompts**, which are a new kind of input that directs this learned program. While the theory is still evolving, it's important to remember that these powerful DL models are still *created by* and *run on* the stored-program computers we will be studying.

The Overarching Idea: Abstraction 📚

So how do we manage all this complexity? A modern processor has billions of transistors. An operating system has millions of lines of code. No single person can possibly understand every detail. The magic that makes this all work is **abstraction**.

Abstraction is the process of hiding complex details behind a simple model, or interface. This allows us to build complex systems in layers, where each layer only needs to understand the layer immediately below it, not all the details beneath.

The key to abstraction is separating the **interface** (the *what*) from the **implementation** (the *how*).

- **Interface:** A set of simple rules and guarantees about what a system can do.
- **Implementation:** The complex, hidden details of how the system actually does it.

Think about driving a car. The **interface** is the steering wheel, pedals, and gear shift. You know that pressing the accelerator makes the car go faster. The **implementation** is the engine, fuel injectors, transmission, and exhaust system. You don't need to know anything about internal combustion to drive to the store. The simple interface hides the complex implementation.

Computers are built on layers and layers of abstraction. A programmer doesn't think about individual transistors; they use a programming language (an abstraction). The language is converted to machine instructions (another abstraction), which are executed by processor components (another abstraction), which are built from logic gates (another abstraction), which are finally built from transistors.

This book is a journey through these layers of abstraction, starting from the bottom up. We will focus on the dominant model of computing—the **programmable, stored-program computer**—and see how these powerful ideas work in practice.

Cheatsheet: The Big Ideas

- **Universal Computation**: The concept of a **Turing Machine** proved that a single, simple machine could theoretically solve any computable problem. This property is called **Turing Completeness**.
- **The Stored-Program Computer**: The defining feature of modern computers. The **instructions (software) are stored in memory** just like data, which allows the computer's function to be changed easily.
- **Specialization vs. Generality**: This is the fundamental trade-off between a flexible, **general-purpose** computer (like a CPU) and a highly efficient but inflexible **fixed-function** device (like a Bitcoin mining chip).
- **Abstraction**: The essential technique for managing complexity. It involves hiding the complex details (**implementation**) behind a simple set of rules or guarantees (**interface**). This allows us to build incredibly complex systems in layers.
- **New Frontiers**: New computing paradigms are emerging. **Quantum Computing** uses qubits to solve certain problems exponentially faster, while **Deep Learning** creates "programs" (models) by learning from data rather than being explicitly programmed.

2 Bits and Representation

The Digital Universe: It's All About the Bits

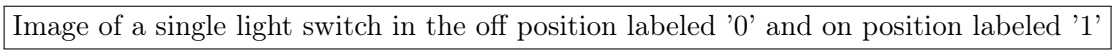
Take a moment to think about your digital world. The music you stream, the high-definition videos you watch, the photos you share, the very words you are reading right now—all of it seems incredibly complex and diverse. Yet, at the most fundamental level inside any computer, all of this information is stored and processed in the exact same way: as a massive collection of simple, tiny switches. The state of these switches, either **on** or **off**, is the basic language of the computer. Our first step in understanding how a computer works is to learn this language, which begins with a single concept: the bit.

What is a Bit?

A **bit**, short for **b**inary **d**igit, is the smallest and most basic unit of information in computing. A bit can only have one of two possible values, which we represent with the symbols **0** and **1**.

You can think of a bit like a light switch. It can either be in the "off" position (0) or the "on" position (1). There are no other possibilities. This two-state system is the foundation upon which all digital information is built.

- **0**: Represents off, false, low voltage.
- **1**: Represents on, true, high voltage.



A single bit by itself isn't very useful; it can only answer a yes/no question. To represent more complex information, we need to group bits together.

Grouping Bits to Represent More Information

If we use two bits, we can represent four unique combinations:

```
00   01   10   11
```

With three bits, we can create eight unique combinations:

```
000   001   010   011   100   101   110   111
```

Notice the clear pattern emerging? Every time we add a new bit, we double the number of unique things we can represent. This relationship can be expressed with a simple formula:

Core Principle

With **n** bits, you can represent $\mathbf{2^n}$ unique things.

This exponential growth is the key to a computer's power. For example, a group of 8 bits, known as a **byte**, can represent $2^8 = 256$ different things.

When we write down a string of bits, which we call a **binary string**, there is a standard convention. We write the bits in order of their position, or index. For a 4-bit string, we might label the bits as follows:

$$b_3 b_2 b_1 b_0$$

By convention, the rightmost bit, b_0, is called the **least significant bit (LSB)**, and the leftmost bit, b_3, is the **most significant bit (MSB)**. For now, don't think of these strings as numbers. Think of them simply as unique patterns. For example, if a grocery store sold 11 types of fruit, we couldn't give each fruit a unique 3-bit code (since $2^3 = 8$ is not enough). However, we could use 4-bit codes (since $2^4 = 16$ is more than enough). We could assign 'apple' the code `0000`, 'banana' the code `0001`, and so on.

A Glimpse into Quantum Computing: The Qubit

While this textbook focuses on classical computers, it's worth knowing about a fascinating development in computing. **Quantum computers** don't use bits; they use **qubits**. Unlike a bit, which must be either a 0 or a 1, a qubit can exist in a state of **superposition**, meaning it can be a combination of both 0 and 1 *at the same time*.

This property gives quantum computers immense potential power. A system of n classical bits can only represent *one* of 2^n possible values at any given moment. A system of n qubits, however, can represent all 2^n values simultaneously! This allows them to tackle certain problems—like code-breaking and molecular simulation—that are impossible for even the most powerful classical supercomputers.

What's Next?

We've established that we can use groups of bits to represent a collection of unique items. This is a powerful idea, but to build a computer, we need to represent something more structured and universally useful. In the following sections, we will explore how these simple strings of 0s and 1s form the foundation for representing numbers, characters, and other data types that are the bedrock of all computation.

Cheatsheet: Key Takeaways

- A **bit** (binary digit) is the smallest unit of data in a computer and can have one of two values: 0 or 1.
- The fundamental formula: with **n** bits, you can represent $\mathbf{2^n}$ unique patterns or values.
- A **byte** is a group of 8 bits and can represent $2^8 = 256$ different values.
- In a binary string (e.g., $b_3b_2b_1b_0$), the rightmost bit (b_0) is the **Least Significant Bit (LSB)** and the leftmost bit (b_3) is the **Most Significant Bit (MSB)**.

Representing Numbers: Integers

Now that we understand that bits can represent unique patterns, let's apply this to something concrete: representing numbers. The most fundamental type of number in computing is the **integer**. Before we dive into how binary is used, let's revisit how our familiar decimal number system works.

The Power of Position: Decimal and Binary

When we write the number '743' in the decimal (base-10) system, we intuitively understand that the position of each digit matters. The '3' is in the ones place, the '4' is in the tens place, and the '3' is in the hundreds place. This is called a **weighted positional notation**. Each position has a weight that is a power of 10.

$$743 = (7 \times 10^2) + (4 \times 10^1) + (3 \times 10^0)$$

$$743 = (7 \times 100) + (4 \times 10) + (3 \times 1) = 700 + 40 + 3$$

The binary system works on the exact same principle, but the base is 2 instead of 10. Each position's weight is a power of 2.

Unsigned Integers

The most straightforward way to represent an integer is as an **unsigned integer**, which means it can only represent non-negative values (zero and positive numbers). To find the value of an unsigned binary number, you simply sum the weights of the positions that contain a '1'.

For example, let's find the value of the 4-bit binary number '1101':

$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$1101_2 = (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1)$$

$$1101_2 = 8 + 4 + 0 + 1 = 13_{10}$$

With n bits, we can represent any integer from 0 (all bits are 0) to $2^n - 1$ (all bits are 1).

The Problem of Negativity: Signed Integers

Unsigned numbers are great, but we often need to represent negative values. How can we do this with only 0s and 1s? The most obvious approach is to reserve one bit, typically the **most significant bit (MSB)**, to represent the sign. This bit is called the **sign bit**. By convention, a '0' in the sign bit means the number is positive, and a '1' means it is negative. The remaining bits represent the number's magnitude (its absolute value). This method is called **Sign and Magnitude**.

For example, with 4 bits:

- '0101' represents +5 (sign bit is 0, magnitude is $101_2 = 5$).
- '1101' represents −5 (sign bit is 1, magnitude is $101_2 = 5$).

However, this simple approach has two major flaws:

1. **Two Zeros**: '0000' represents +0, and '1000' represents −0. Having two different patterns for the same value is inefficient and complicates the computer's hardware.
2. **Difficult Arithmetic**: The hardware for addition and subtraction becomes very complex, as it has to check the signs before performing an operation. If you try to add +5 and −5 ('0101 + 1101'), simple column-wise addition does not work: you get '10010' which does not match the expected result of '0000'.

A Better Way: Complements

To solve the problems with sign and magnitude, computer scientists developed complement systems. The goal was to make arithmetic simple and have only one representation for zero.

One's Complement

In a **one's complement** system, positive numbers are represented just like in sign and magnitude. To get the negative representation of a number, you simply **flip every bit**. This is called taking the complement.

- To represent +5: `0101`
- To get −5: Flip every bit of `0101` → `1010`

This system makes some arithmetic easier, but it still suffers from one major flaw: it has two representations for zero. `0000` is +0, and if you flip all its bits, you get `1111`, which represents −0.

Two's Complement: The Gold Standard

This brings us to **two's complement**, the system used by virtually all modern computers. It is elegant, efficient, and solves the problems of the previous systems.

To get the negative of a number in two's complement:

1. **Flip every bit** (like one's complement).
2. **Add 1**.

Let's find the representation of -5 using 4 bits:

1. Start with $+5$: `0101`
2. Flip all the bits: `1010`
3. Add 1: `1010 + 1 = 1011`.

So, `1011` represents -5 in two's complement.

Why is this so powerful?

- **One Zero**: There is only one representation for zero (`0000`). If we try to negate `0000`, we flip the bits (`1111`) and add 1, which results in `(1)0000`. The carry bit is discarded, leaving `0000`.
- **Simple Arithmetic**: Addition works for both positive and negative numbers without any special logic. Let's add $+5$ and -5:

```
  0101  (+5)
+ 1011  (-5)
-------
(1)0000  (0, after discarding the carry bit)
```

The hardware for addition now automatically handles subtraction. The system works perfectly. We will shortly explain more about this carry bit - it's similar to the carryover of addition in decimal numbers.

Integer Representation Tables

The following tables compare the values represented by the same binary strings using the different systems we've discussed.

Table 2.1: 3-Bit Integer Representations

Binary String	Unsigned	Sign & Mag.	One's Comp.	Two's Comp.
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

Table 2.2: 4-Bit Integer Representations

Binary String	Unsigned	Sign & Mag.	One's Comp.	Two's Comp.
0111	7	+7	+7	+7
0110	6	+6	+6	+6
...	...	...	...	...
0001	1	+1	+1	+1
0000	0	+0	+0	+0
1111	15	-7	-0	-1
1110	14	-6	-1	-2
...	...	...	...	...
1001	9	-1	-6	-7
1000	8	-0	-7	-8

Cheatsheet: Integer Ranges for n Bits

- **Unsigned**: Can represent 2^n numbers.
  - Range: 0 to $2^n - 1$
- **Sign and Magnitude**: Has two zeros.
  - Range: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
- **One's Complement**: Has two zeros.
  - Range: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$
- **Two's Complement**: The standard. Has one zero and an asymmetric range.
  - Range: -2^{n-1} to $+(2^{n-1} - 1)$

Operations on Bits and Numbers

We've now established how to represent both positive and negative integers using bits. The next logical step is to perform operations on them. In this section, we'll cover how to convert between the number systems we've learned, how to perform arithmetic, and finally, how to use fundamental logical operations that form the bedrock of all computation.

Conversions: Moving Between Bases

Being able to seamlessly convert numbers between their decimal (base-10) and binary (base-2) forms is a critical skill.

Binary to Decimal Conversion

The method for converting a binary number to decimal depends on its encoding. The first step is always to look at the most significant bit (MSB).

Case 1: The number is positive (MSB = 0) If the MSB is 0, the number is positive. For **Unsigned, Sign & Magnitude, One's Complement, and Two's Complement**, the method is the same: simply sum the weighted powers of 2 for each bit position that has a 1.

- **Example**: Convert the 8-bit number `01011001` to decimal.
  Value = $(1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$
  Value = $64 + 16 + 8 + 1 = 89_{10}$.

Case 2: The number is negative (MSB = 1) If the MSB is 1, the method depends on the encoding. Let's convert the 4-bit number `1101` in each system.

- **Sign & Magnitude**: The MSB (`1`) is the sign. The remaining bits (`101`) are the magnitude. Magnitude = $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5$. Final value = -5_{10}.
- **One's Complement**: To find the magnitude, flip all the bits back to their positive form. Original: `1101` → Flipped: `0010`. Magnitude = $(1 \times 2^1) = 2$. Final value = -2_{10}.
- **Two's Complement**: To find the magnitude, flip all the bits and add 1. Original: `1101` → Flip: `0010` → Add 1: `0011`. Magnitude = $(1 \times 2^1) + (1 \times 2^0) = 3$. Final value = -3_{10}.

Decimal to Binary Conversion

To convert from decimal to binary, we first handle the sign.

Case 1: The number is positive (or unsigned) We use the method of **repeated division by 2**. We divide the decimal number by 2, record the remainder, and continue dividing the quotient until it becomes 0. The binary number is the sequence of remainders read from bottom to top.

- **Example**: Convert 44_{10} to an 8-bit unsigned binary number.
  $44 \div 2 = 22$ remainder **0** (LSB); $22 \div 2 = 11$ remainder **0**; $11 \div 2 = 5$ remainder **1**; $5 \div 2 = 2$ remainder **1**; $2 \div 2 = 1$ remainder **0**; $1 \div 2 = 0$ remainder **1** (MSB).
  Reading the remainders up: `101100`. As an 8-bit number, we pad with leading zeros: `00101100`.

Case 2: The number is negative First, convert the **absolute value** to binary, then apply the rule for the target encoding.

- **Example**: Convert -25_{10} to an 8-bit binary number.
  1. **First, convert the absolute value, 25**, to binary: `00011001`.
  2. **Now, apply the encoding rules**:
     - **Sign & Magnitude**: Set the MSB to 1. Result: `10011001`.
     - **One's Complement**: Flip all the bits of the positive version. Result: `11100110`.
     - **Two's Complement**: Flip the bits and add 1. Flip `00011001` $\rightarrow$ `11100110`. Add 1 $\rightarrow$ `11100111`.

Binary Addition in Two's Complement or Unsigned

Addition in binary works exactly like in decimal: you add digits column by column from right to left and carry over when a column's sum is too large. We focus on two's complement and unsigned because its design makes arithmetic simple. As there is only a given number of bits, a carry out from the MSB is not included in the result.

Overflow: When the Result Doesn't Fit

Sometimes, the result of an addition is too large to be represented by the given number of bits. This is called **overflow**. For Unsigned, overflow is indicated by a carry out from the MSB (yes, the one not included in the result. Two's complement overflow is more complicated:

- Adding two **positive** numbers results in a **negative** number.
- Adding two **negative** numbers results in a **positive** number.
- Adding two numbers with **different signs** can **NEVER** cause an overflow.

The hardware detects this with a clever trick involving the two most significant carries: the carry **into** the MSB column (C_{in}) and the carry **out** of the MSB column (C_{out}).

Two's Complement Overflow Detection

Overflow occurs if and only if $\mathbf{C_{in} \neq C_{out}}$. This is equivalent to checking if C_{in} XOR $C_{out} = 1$.

Why the XOR Trick Works: Let's analyze the MSB column. The resulting sign bit is $S_{out} = A_{msb} + B_{msb} + C_{in}$.

- **Positive + Positive Overflow:** Here, $A_{msb} = 0$ and $B_{msb} = 0$. For the result to be negative ($S_{out} = 1$), a carry-in must have occurred: $0+0+C_{in} = 1 \implies C_{in} = 1$. This sum (1) does not produce a carry-out, so $C_{out} = 0$. Thus, $C_{in} = 1$ and $C_{out} = 0$. They are different.
- **Negative + Negative Overflow:** Here, $A_{msb} = 1$ and $B_{msb} = 1$. For the result to be positive ($S_{out} = 0$), the sum must be $1 + 1 + C_{in} = 10_2$ or $1 + 1 + C_{in} = 11_2$. To get $S_{out} = 0$, we must have $1 + 1 + C_{in} = 10_2$, which means $C_{in} = 0$. This sum produces a carry-out, so $C_{out} = 1$. Thus, $C_{in} = 0$ and $C_{out} = 1$. They are different.

In all non-overflow cases, C_{in} and C_{out} will be the same. This simple check is easy to implement in hardware.

Bitwise Logical Operations

Beyond arithmetic, computers rely heavily on bitwise logical operations. Unlike an operation like addition which treats an 8-bit string as a single number (e.g., 25), bitwise operations treat it as a collection of 8 individual bits. Each bit in the first operand is matched with the bit in the corresponding position of the second operand, and the operation is performed on these pairs independently. What happens in one bit position has absolutely no effect on any other.

To formally define these operations, we use a **truth table**. A truth table is a straightforward chart that lists every possible combination of inputs for an operation and shows the resulting output for each case. For a two-input operation, there are four possible combinations: 0/0, 0/1, 1/0, and 1/1.

NOT The NOT operation is the simplest. It is a *unary* operator, meaning it acts on a single operand.

- **English Definition:** The NOT operation inverts or flips the input bit. If the input is 1, the output is 0. If the input is 0, the output is 1.
- **Example:**

```
NOT 10110001
-----------
    01001110
```

A	NOT A
0	1
1	0

AND

- **English Definition:** The AND operation outputs a 1 only if *both* of the input bits are 1. If either bit is a 0, the output is 0. This is useful for "masking" or clearing bits.
- **Example:** Let's say we want to clear the upper 4 bits of a byte. We can AND it with the mask `00001111`.

```
    10110101
AND 00001111  (The Mask)
-----------
    00000101  (Result)
```

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR

- **English Definition:** The OR operation outputs a 1 if *at least one* of the input bits is 1. It only outputs 0 if both inputs are 0. This is useful for setting specific bits.
- **Example:** Let's say we want to ensure the two most significant bits are set to 1. We can OR the value with the mask `11000000`.

```
    10110101
 OR 11000000  (The Mask)
-----------
    11110101  (Result)
```

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

XOR (Exclusive OR)

- **English Definition:** The XOR operation outputs a 1 only if the two input bits are *different*. If they are the same (both 0 or both 1), the output is 0. This is very useful for toggling or flipping bits.
- **Example:** If we XOR a value with a mask, the bits corresponding to a '1' in the mask will be flipped, and the bits corresponding to a '0' will be unchanged.

```
    10110101
XOR 11110000  (The Mask)
-----------
    01000101  (Result: the first 4 bits are flipped)
```

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Universal Gates and De Morgan's Laws

Two other critical logical operations are **NAND** (Not-AND, the result of an AND is negated) and **NOR** (Not-OR, the result of an OR is negated).

These two gates are special because they are **universal gates**. This means that any other logical function (AND, OR, NOT) can be constructed using *only* NAND gates, or *only* NOR gates. This is incredibly useful for manufacturing, as you can design a

A	B	A NAND B	A NOR B
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

complex chip using just one type of simple, repeated building block. The mathematical foundation for this property is provided by **De Morgan's Laws**.

Mathematically, using $\cdot$ for AND, $+$ for OR, and an overbar for NOT, the laws are stated as:

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

At first glance, this may seem abstract, but it's quite intuitive in plain English.

First Law: $\overline{A \cdot B} = \bar{A} + \bar{B}$ In English: "Not (A and B) is the same as (Not A) or (Not B)."

- **Intuition:** Imagine a rule that says, "To get into the party, you must be on the list **AND** have an ID." Failing this rule — i.e., NOT (on list AND have ID) — means one of two things is true: you are NOT on the list, **OR** you do NOT have an ID. You don't need to fail both conditions to be turned away. Failing just one is enough.

Second Law: $\overline{A + B} = \bar{A} \cdot \bar{B}$ In English: "Not (A or B) is the same as (Not A) and (Not B)."

- **Intuition:** Imagine a different rule that says, "To get a discount, you must be a student **OR** a senior citizen." Failing this rule — i.e., NOT (student OR senior) — means you must fail *both* conditions simultaneously. You must be NOT a student **AND** you must be NOT a senior citizen.

De Morgan's laws provide the formal method for converting between AND-based and OR-based logic, proving how a universal gate like NAND can be used to create any other logical function, forming the fundamental building blocks of digital circuits.

Cheatsheet: Key Takeaways on Operations

- **Decimal to Binary**: For positive numbers, use repeated division by 2. For negative numbers, convert the absolute value first, then apply the rules for Sign & Mag, 1's Comp, or 2's Comp.
- **Binary to Decimal (Two's Comp)**: If the MSB is 1, the number is negative. To find its value, flip all the bits, add 1, convert the result to decimal, and put a minus sign in front.
- **Overflow**: Occurs when the result of an addition is too large to fit. This happens only when adding numbers of the same sign and the result has a different sign for Two's Complement.
- **Overflow Detection (Hardware)**: An overflow has occurred if and only if the carry-in to the MSB is different from the carry-out of the MSB ($C_{in} \neq C_{out}$) for Two's Complement.
- **Bitwise Operations**:
  - **AND** is often used to *clear* or *mask* bits (e.g., `xxxx & 1100 = xx00`).
  - **OR** is often used to *set* bits (e.g., `xxxx | 0011 = xx11`).
  - **XOR** is often used to *flip* or *toggle* bits (e.g., `xxxx ^0011` flips the last two bits).
- **Universal Gates**: NAND and NOR are universal because all other logic gates (AND, OR, NOT) can be constructed from them.

Representing Fractional Numbers

So far, we have only dealt with integers. But the real world is full of fractional numbers, like the price of a coffee ($3.75) or a mathematical constant like π (3.14159...). To represent these, we need to move beyond integer schemes and find a way to place a "point" within our binary numbers.

Fixed-Point Representation

The most straightforward way to represent fractional numbers is the **fixed-point** method. The idea is simple: we decide that the binary point is located at a fixed, predetermined position within our bit string. This divides the bits into an integer part (to the left of the point) and a fractional part (to the right of the point).

Just as the weights to the left of the point are positive powers of two ($2^0, 2^1, 2^2, \dots$), the weights to the right of the point are negative powers of two: $2^{-1}(0.5)$, $2^{-2}(0.25)$, $2^{-3}(0.125)$, and so on.

Conversions with Fixed-Point Numbers

Binary Fixed-Point to Decimal To convert a fixed-point binary number to decimal, simply sum the weighted values of each part.

- **Example**: Convert the 8-bit fixed-point number `0110.1101` to decimal, assuming 4 bits for the integer and 4 for the fraction.
  - **Integer Part**: `0110` $= (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 4 + 2 = 6$.
  - **Fractional Part**: `.1101` $= (1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) = 0.5 + 0.25 + 0.0625 = 0.8125$.
  - **Total Value**: $6 + 0.8125 = 6.8125$.

Decimal to Binary Fixed-Point This is a two-step process: convert the integer part and the fractional part separately.

1. **Integer Part**: Convert using the familiar repeated division by 2.
2. **Fractional Part**: Convert using **repeated multiplication by 2**. Multiply the fraction by 2; the integer part of the result is the next binary digit. Repeat the process with the new fractional part until the fraction becomes 0 or you run out of bits.

- **Example**: Convert 10.625 to an 8-bit fixed-point representation with 4 integer and 4 fractional bits.
  - **Integer Part**: $10_{10} = 1010_2$.
  - **Fractional Part** (0.625): $0.625 \times 2 = \mathbf{1}.25$; $0.25 \times 2 = \mathbf{0}.5$; $0.5 \times 2 = \mathbf{1}.0$. The fractional part is `.101`.
  - **Combine and Pad**: We combine the parts (`1010.101`) and pad the fractional part to 4 bits, giving us `1010.1010`.

Negative Numbers and Arithmetic in Fixed-Point

Representing negative numbers in fixed-point is handled using the standard **two's complement** method, applied to the entire bit string as if the binary point wasn't there.

- **Example**: Find the representation of -2.5 using an 8-bit format (4 integer, 4 fractional bits).
  1. Start with positive 2.5: $2 = 10_2$, $0.5 = .1_2$. So, 2.5 is `0010.1000`.
  2. Flip all the bits: `1101.0111`.
  3. Add 1 to the least significant bit (LSB): `1101.0111 + 0000.0001 = 1101.1000`.

  Thus, `1101.1000` represents -2.5.

A major advantage of this scheme is that arithmetic is incredibly simple. To add or subtract fixed-point numbers, you just line up the binary points and perform standard binary addition/subtraction on the entire string. The hardware is identical to what is used for integers, making fixed-point operations very fast and efficient.

Pros and Cons of Fixed-Point

- **Pros**: Fixed-point representation is simple and fast. Arithmetic can be performed using the exact same hardware logic as for integers.
- **Cons**: The biggest drawback is its **limited dynamic range**. You cannot represent very large numbers and very small fractions simultaneously. The designer must make a permanent trade-off between **range** (number of integer bits) and **precision** (number of fractional bits). This system is inadequate for scientific applications that deal with both the astronomically large and the microscopically small.

Floating-Point Representation

How can we represent both the mass of the sun (1.989×10^{30} kg) and the mass of an electron (9.109×10^{-31} kg) in the same number system? The solution is to adopt a format similar to scientific notation, where the decimal point can "float" to wherever it's needed. This is the idea behind **floating-point representation**.

To ensure that floating-point numbers are handled consistently across all machines, the industry has adopted the **IEEE 754 standard**. We will focus on the most common 32-bit version, known as **single-precision** or **FP32**.

The IEEE 754 FP32 Standard

An FP32 number is made of three components packed into a 32-bit string:

| S | EEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFF |

1. **Sign (S)**: 1 bit. '0' for positive, '1' for negative.
2. **Exponent (E)**: 8 bits. This determines the magnitude (how large or small the number is).
3. **Fraction (F)** (also called Mantissa or Significand): 23 bits. This determines the number's actual digits (its precision).

These components are used in the following formula:

$$\text{Value} = (-1)^S \times (1.F) \times 2^{(E-\text{bias})}$$

Let's break down the two tricky parts:

- **The Hidden Bit (1.F)**: In binary scientific notation, a normalized number always starts with a '1' (e.g., 1.011×2^5). Since this leading '1' is always there, we don't need to waste a bit storing it! This "hidden bit" gives us 24 bits of precision while only storing 23.
- **Biased Exponent (E - bias)**: The 8-bit exponent field (E) can store values from 0 to 255. To represent both positive and negative exponents, the standard uses a **biased notation**. We subtract a fixed **bias** of **127** from the stored value E.

Zero and Special Cases

As the hidden bit is always a one, the formula cannot represent zero! Zero uses a different formula and thus a special case. If E = 0, then the following formula is used:

$$\text{Value} = (-1)^S \times (0.F) \times 2^{(E-126)}$$

No more hidden bit means zero is possible; zero is represented by 32 zeros. There are other special cases, for example when E = 255 (all ones), but this text does not cover them. For further reading, look at the IEEE 754 specification.

Floating-Point Examples

Let's walk through converting decimal numbers to FP32 format.

Example 1: 1.5

1. **Sign**: Positive $\implies S = 0$.
2. **Binary**: $1.5_{10} = 1.1_2$.
3. **Normalize**: The number is already in the form $1.F$. This is 1.1×2^0.
4. **Find Components**: The fraction **F** is '1', padded to 23 bits. The exponent is '0', so we solve $E - 127 = 0 \implies E = 127 = 01111111_2$.
5. **Assemble**: `0 | 01111111 | 10000000000000000000000`

Example 2: 764.0

1. **Sign**: Positive $\implies S = 0$.
2. **Binary**: $764_{10} = 1011111100_2$.
3. **Normalize**: Move the point 9 places left: 1.011111100×2^9.
4. **Find Components**: **F** is '011111100', padded. The exponent is '9', so $E - 127 = 9 \implies E = 136 = 10001000_2$.
5. **Assemble**: `0 | 10001000 | 01111110000000000000000`

Example 3: 18.547 (The Precision Problem)

1. **Sign**: Positive $\implies S = 0$.
2. **Binary**: The integer part is $18 = 10010_2$. The fractional part 0.547 is non-terminating in binary ($0.10001\ldots_2$).
3. **Normalize**: $1.001010001\cdots \times 2^4$.
4. **Find Components**: The exponent is '4', so $E = 131 = 10000011_2$. The fraction **F** is '001010001...'. Since we only have 23 bits, we must truncate or round the infinite series. This is where **precision loss** occurs. The number stored is not exactly 18.547, but a very close approximation.

Other Floating-Point Formats

While we focus on FP32, you should be aware of other common formats.

Table 2.3: Common IEEE 754 Floating-Point Formats

Name	Total Bits	Sign	Exponent	Fraction	Bias
FP32 (Single)	32	1	8	23	127
FP64 (Double)	64	1	11	52	1023
FP16 (Half)	16	1	5	10	15
FP8 (Quarter)	8	1	4 or 5	3 or 2	7 or 15

Cheatsheet: Key Takeaways on Fractional Numbers

- **Fixed-Point**: Simple and fast because arithmetic is just integer arithmetic. Negative numbers use two's complement. Its main con is a **limited dynamic range**.
- **Floating-Point**: Uses a form of scientific notation to represent a huge dynamic range. It is the standard for almost all non-integer calculations.
- **FP32 Formula**: Value $= (-1)^S \times (1.F) \times 2^{(E-127)}$
- **FP32 Components**: It is composed of **1 Sign bit**, **8 Exponent bits**, and **23 Fraction bits**.
- **Key Concepts**: Remember the **hidden bit** (the implicit '1.' before the fraction) and the **exponent bias** (127 for FP32).
- **Precision Loss**: Because the fraction part has a fixed number of bits, not all decimal numbers can be represented perfectly. This can lead to small rounding errors in calculations.

Representing Text and Data

We've dedicated this chapter to representing different kinds of numbers, but what about text? When you type an essay or send a message, the computer isn't storing the shapes of the letters 'A', 'b', and 'c'. It is, of course, storing them as bits. The final piece of our puzzle is understanding the standard codes used to map characters to numbers.

The ASCII Standard

The most foundational standard for representing text is **ASCII** (American Standard Code for Information Interchange). Developed in the 1960s, ASCII is a 7-bit encoding scheme, meaning it uses 7 bits to define $2^7 = 128$ unique codes. Each code is mapped to a specific character.

- The character 'A' is assigned the decimal value 65 (01000001_2).
- The character 'B' is assigned the decimal value 66 (01000010_2).
- The character 'a' is assigned the decimal value 97 (01100001_2).
- The digit '0' is assigned the decimal value 48 (00110000_2).

Since computers typically operate on 8-bit bytes, the 8th bit was often set to '0' or used for special purposes.

The first 32 codes (0-31) and the last code (127) are non-printable **control characters**, which were originally used to control teletype machines. Many are still in use today, such as **Newline** (code 10), **Backspace** (code 8), and **Tab** (code 9). The remaining codes (32-126) represent printable characters, including the space character (code 32).

A Brief Note on Unicode

ASCII's 128 characters are sufficient for English but fail to represent characters from other languages, let alone symbols and emojis. The modern solution is **Unicode**, a universal standard that defines over 149,000 characters. Common Unicode encodings like **UTF-8** and **UTF-16** can represent every character from every language. Importantly, the first 128 codes of Unicode are identical to the ASCII standard, making ASCII a direct subset of Unicode.

Full ASCII Table (Codes 0-127)

A Human-Friendly Shorthand: Hexadecimal Notation

You will often see long binary numbers, especially memory addresses, written in a strange format like `0x7FFFAB42`. This is **hexadecimal** notation (often called "hex"), and it's important to understand what it is and why we use it.

Hexadecimal is not a data representation used by the computer. The computer's hardware only understands binary. Hexadecimal is a **human-friendly shorthand for binary**.

Table 2.4: ASCII Printable and Control Characters

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	SPC	64	@	96	’
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Hexadecimal is a base-16 number system. It uses the digits 0-9 and then the letters A-F to represent the values 10-15. The magic of hexadecimal is that $16 = 2^4$. This means that **every one hexadecimal digit corresponds perfectly to exactly four binary digits** (a nibble). This relationship allows for quick and easy conversion.

Table 2.5: Decimal, Hexadecimal, and Binary Equivalents

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Example 1: Binary to Hex To convert a binary string to hex, group the bits in sets of four from right to left and replace each group with its hex equivalent.

- Binary: `1101 0101 1111 0010`
- Grouped: `1101 0101 1111 0010`
- Hex: `D  5  F  2`

It is much easier for a person to read and remember `D5F2` than `1101010111110010`.

Example 2: Hex to Binary To convert hex to binary, simply replace each hex digit with its 4-bit binary string. The prefix `0x` is commonly used in programming languages to indicate that a number is in hexadecimal format.

- Hex: `0x1A7`
- Expand: `1  A  7`
- Binary: `0001 1010 0111`

Hexadecimal is an essential tool for programmers because it provides a compact and less error-prone way to work with the raw binary data that computers actually use.

Cheatsheet: Key Takeaways on Text and Data Notation

- **ASCII**: A 7-bit standard that maps characters (letters, symbols, numbers) to integer values. The first 128 characters are a subset of Unicode.
- **Unicode**: The modern, universal standard for representing text from all languages.
- **Hexadecimal (Base-16)**: A **human-friendly shorthand** for representing binary data. It is not used by the computer's internal hardware.
- **The Golden Rule of Hex**: 1 hexadecimal digit always represents exactly 4 binary digits.
- **Common Notation**: The prefix `0x` is used in code to signify that a number is hexadecimal.

3 Digital Logic

Chapter 3: Digital Logic Structures

In the last chapter, we established that all information in a computer is represented by bits. But how does a machine physically store these bits and, more importantly, how does it compute with them? The answer lies in the hardware that underpins all of computing: digital logic structures. This chapter will take you on a journey from the simplest physical switch to the complex circuits that perform calculations.

The Switch: Storing a Bit

At the most fundamental level, a computer is a collection of billions of tiny switches. Think of a simple light switch on your wall. It can be in one of two states: ON or OFF. There is no in-between. We can use this physical property as a direct analog for a bit:

- **OFF** represents a logical **0**.
- **ON** represents a logical **1**.

This is our first and most important abstraction: a physical state (the position of a switch) represents a logical value (a bit). Using this abstraction, it's clear we can *store* information. A bank of eight switches could be set in $2^8 = 256$ different patterns, allowing it to store any 8-bit number.

But we want to do more than just store bits; we want to compute with them. To do that, we need a switch that can be controlled not by a human hand, but by an electrical signal. A switch that can be turned ON or OFF by another switch. This is the key that unlocks computation, and the device that makes it possible is the transistor.

Cheatsheet: The Switch Abstraction

- A physical switch has two states: ON and OFF.
- This provides a physical implementation for the two logical states of a bit: 1 and 0.
- The core idea of computation is to have switches that are controlled by other electrical signals (i.e., other switches), not by mechanical force.

The Transistor: An Electrically Controlled Switch

The transistor is arguably the most important invention of the 20th century. It is the fundamental building block of all modern electronics. The first working point-contact transistor was demonstrated in 1947 by John Bardeen and Walter Brattain at Bell Labs, under the guidance of William Shockley. For this work, the three shared the 1956 Nobel Prize in Physics. (As a point of pride for our university, John Bardeen was an esteemed alumnus, earning both his bachelor's and master's degrees right here at the University of Wisconsin-Madison).

That first transistor was a bulky, discrete component. Technology has since evolved to the **MOSFET** (Metal-Oxide-Semiconductor Field-Effect Transistor), and specifically **CMOS** (Complementary MOS) technology, which is the workhorse of digital logic today. Modern processors now use incredible technologies like **FinFETs**, which are essentially 3D transistors, allowing engineers to pack billions of them onto a single chip.

The Physics of a MOSFET

A MOSFET has three terminals: a **Source**, a **Drain**, and a **Gate**. The basic idea is that the Gate controls the flow of current between the Source and Drain. This is achieved by manipulating a semiconductor substrate (usually silicon).

Let's consider an **n-type MOSFET (nMOS)**. It's built on a p-type silicon substrate (meaning it has an excess of positive charge carriers, or "holes"). Within this substrate, two n-type regions (with an excess of negative charge carriers, or electrons) are created for the source and drain. Normally, no current can flow between them. However, the gate is separated from the substrate by a thin insulating layer of oxide. When a positive voltage (a logical 1) is applied to the gate, it creates a downward-pointing electric field. This field pushes the positive holes away from the area under the gate and attracts the minority-carrier electrons. A sufficient concentration of these electrons forms a conductive n-type "channel" between the source and drain, allowing current to flow. The switch is ON.

A **p-type MOSFET (pMOS)** is the complementary opposite. It's built on an n-type substrate with p-type source and drain regions. When a low or zero voltage (a logical 0) is applied to its gate, the resulting electric field attracts positive holes, forming a conductive p-type channel. This allows current to flow. A high voltage at the gate would disrupt this channel and turn the switch OFF.

Circuit Rules and the Inverter

This complementary behavior is the key to CMOS design. We establish two main voltage levels: **VDD**, the power supply voltage representing logical 1, and **GND** (Ground), representing logical 0.

- **P-type transistors** are used to create a "pull-up" network that connects the output to **VDD (1)**. They are active when their gate is 0.
- **N-type transistors** are used to create a "pull-down" network that connects the output to **GND (0)**. They are active when their gate is 1.

A fundamental rule of CMOS design is that for any given input, the circuit must have a path to *either* VDD or GND, but **never both at the same time** (which would cause a short circuit) and never neither (which would leave the output floating).

Let's re-examine our first logic gate, the inverter, with this new terminology.

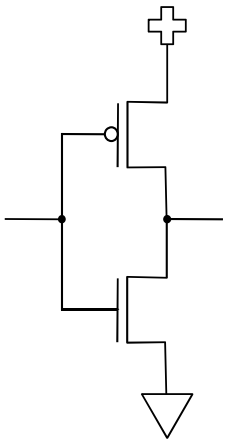


Figure 3.1: CMOS Inverter with VDD and GND

When Input = 1, the pull-down network (n-type) turns ON, connecting the Output to GND (0). The pull-up network (p-type) is OFF. When Input = 0, the pull-up network (p-type) turns ON, connecting the Output to VDD (1). The pull-down network is OFF. The output is always actively driven to a valid state.

Cheatsheet: The Transistor

- A transistor is an electrically controlled switch. The **Gate** controls the flow of current between the **Source** and **Drain**.
- **nMOS Transistor**: Gate = 1 → ON. Forms the **pull-down network** to connect the output to **GND (0)**.
- **pMOS Transistor**: Gate = 0 → ON. Forms the **pull-up network** to connect the output to **VDD (1)**.
- **CMOS Rule**: The pull-up and pull-down networks are complementary; only one is active at a time.

Building More Complex Gates

An inverter is useful, but to build a computer, we need more complex logic. By arranging transistors in clever ways, we can construct any logic gate we need. Let's look at the transistor-level implementation of the two universal gates: NAND and NOR.

NAND Gate (NOT-AND) A 2-input NAND gate is constructed with two p-type transistors in parallel and two n-type transistors in series.

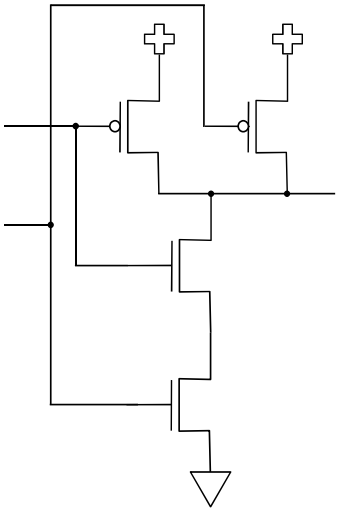


Figure 3.2: A 2-input NAND gate transistor diagram

The output will be connected to Ground (0) only if **both** n-type transistors turn on, which requires A AND B to both be 1. In all other cases, at least one of the p-type transistors will be on, connecting the output to Power (1).

NOR Gate (NOT-OR) A 2-input NOR gate is constructed with two p-type transistors in series and two n-type transistors in parallel.

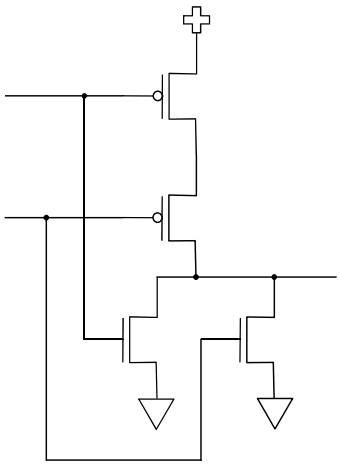


Figure 3.3: A 2-input NOR gate transistor diagram

The output will be connected to Ground (0) if **either** of the n-type transistors turns on, which requires A OR B to be 1. Only when both A and B are 0 will the p-type series be on, connecting the output to Power (1).

From these universal gates, we can create others. An AND gate is simply a NAND gate followed by an inverter. An OR gate is a NOR gate followed by an inverter.

The Abstraction of Gate Symbols Working at the transistor level is tedious. Just as we abstracted a physical switch to a logical bit, we can abstract these transistor configurations into clean, simple **logic gate symbols**. This allows us to design complex circuits without thinking about individual transistors.

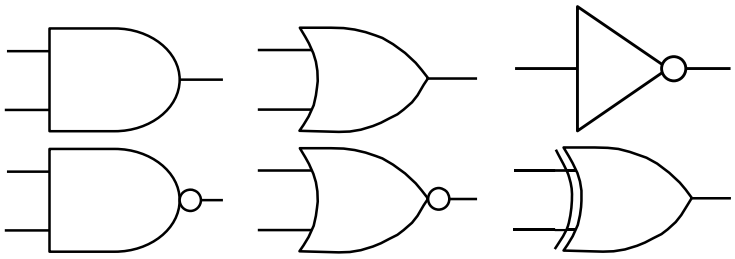


Figure 3.4: The standard logic gate symbols for AND, OR, NOT, NAND, NOR, and XOR respectively.

We have now seen two levels of our system: the physical implementation using transistors, and the logical abstraction of a gate. This is a powerful example of the "Big Idea" of abstraction from Chapter 1.

Cheatsheet: Logic Gates

- Complex gates like NAND and NOR are built from specific series and parallel combinations of n-type and p-type transistors.
- **NAND** and **NOR** are "universal gates" because any other logic function can be created from them.
- We use **logic gate symbols** as an abstraction to hide the underlying transistor-level complexity, allowing us to design more complex systems.

Combinational Circuits: Datapath Components

Now that we have a toolbox of logic gates, we can start building functional circuits. The first type we'll examine is **combinational logic**. In a combinational circuit, the output is determined *only* by the current values of the inputs. These circuits have no memory of what happened before. Many essential "datapath" components of a processor are built from combinational logic.

Full Adder (FA) A full adder is a circuit that adds three bits together: A, B, and a Carry-In (C_{in}), producing a Sum (S) bit and a Carry-Out (C_{out}) bit.

Inputs			Outputs	
A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

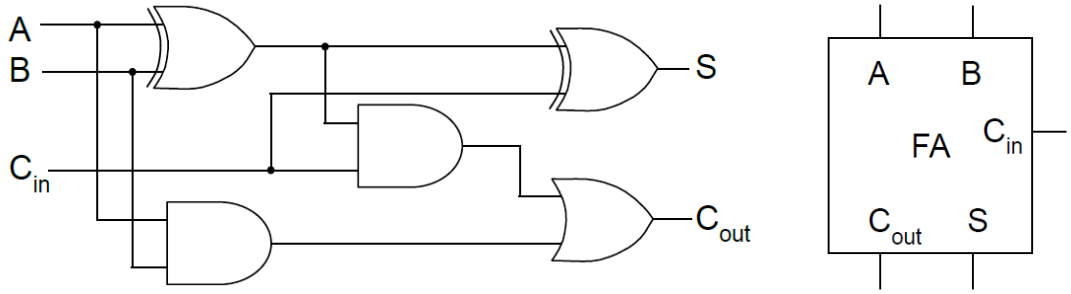


Figure 3.5: Image of Full Adder gate-level implementation and FA block diagram

Once we have a full adder, we abstract it into a block. We can then chain these blocks together to create a multi-bit adder. A 4-bit adder is made by connecting four FA blocks, where the C_{out} of one block becomes the C_{in} of the next. This is called a ripple-carry adder.

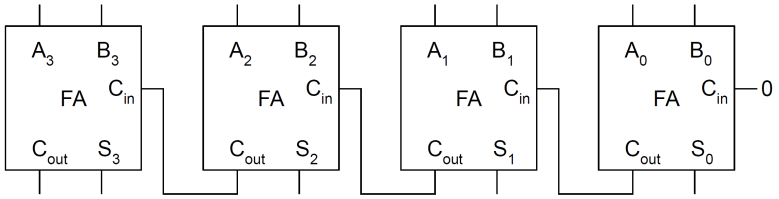


Figure 3.6: Image of a 4-bit ripple-carry adder using FA blocks

Decoder A decoder takes an n-bit input and asserts exactly one of its 2^n output lines. For example, a 2-to-4 decoder takes a 2-bit input and selects one of four outputs.

Inputs		Outputs			
I_1	I_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

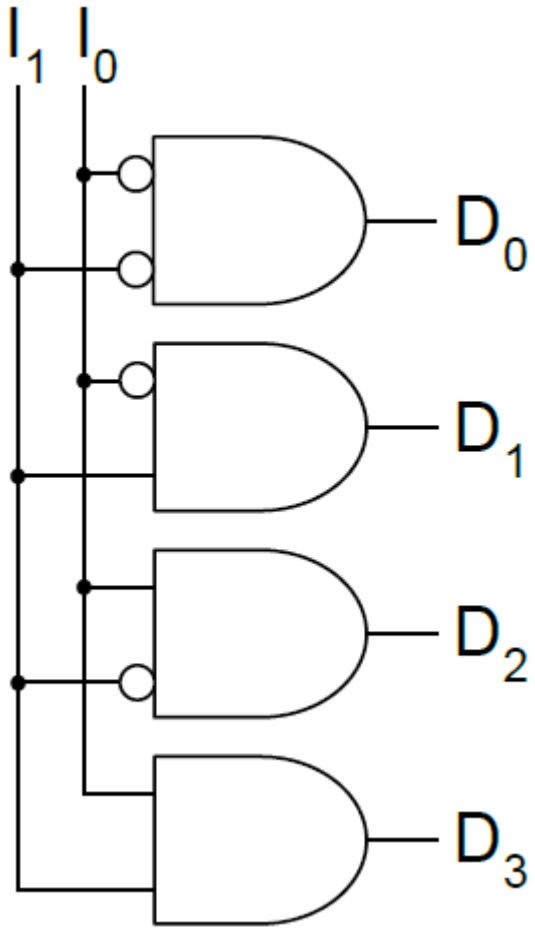


Figure 3.7: Image of a 2-to-4 decoder implementation

Multiplexer (Mux) A multiplexer (or "mux") is a data selector. It has several data inputs, a set of select lines, and a single output. The select lines determine which one of the data inputs is passed to the output. The simplest is a 2-to-1 mux, which selects between two inputs, I_0 and I_1, using a single select line, S.

S	Output Y
0	I_0 is selected
1	I_1 is selected

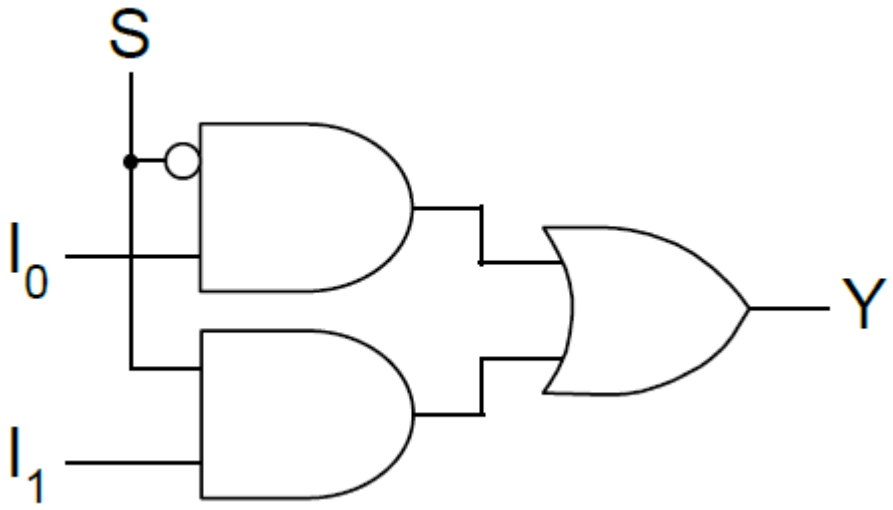


Image of 2-to-1 Mux implementation

Like adders, muxes are modular. But how do we build larger muxes from smaller ones? Let's consider building a **4-to-1 mux**. This requires selecting one of four inputs (I_0, I_1, I_2, I_3), which means we will need two select lines (S_1, S_0), since $2^2 = 4$.

We can approach this hierarchically. Let the first select bit, S_1, make a high-level choice: does it select from the first pair of inputs (I_0, I_1) or the second pair (I_2, I_3)? We can use two 2-to-1 muxes for this first "stage". The second select bit, S_0, can then be used to pick the final winner from the outputs of that first stage. This requires a third 2-to-1 mux.

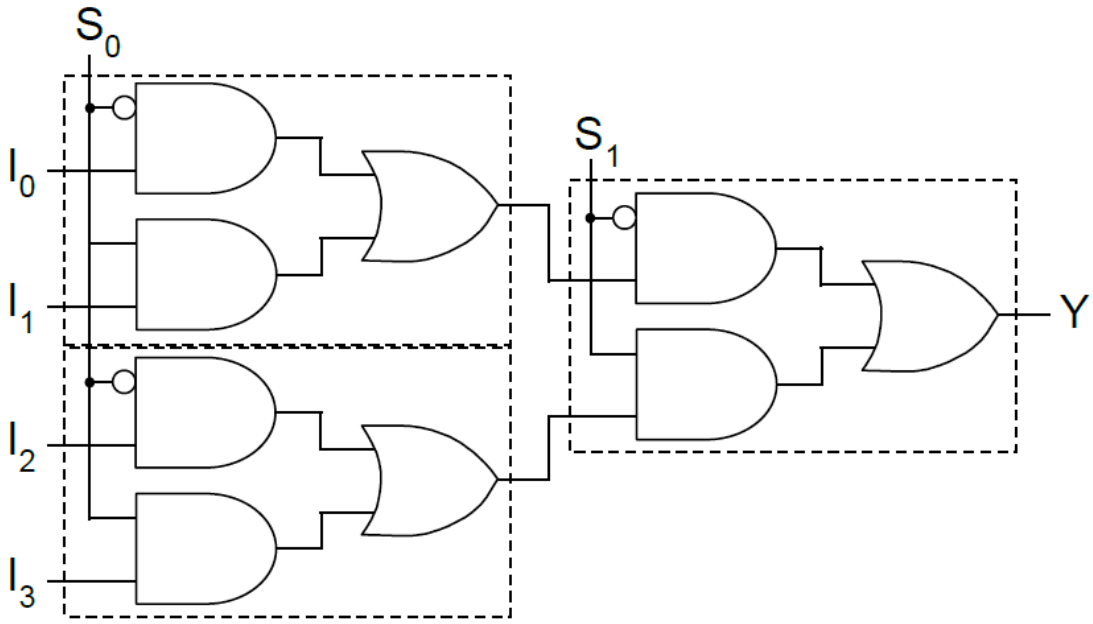


Image showing a 4-to-1 Mux built from three 2-to-1 Muxes

This pattern can be extended. To build an **8-to-1 mux**, which requires 3 select lines, we can combine two 4-to-1 muxes and one 2-to-1 mux. The two 4-to-1 muxes handle the eight inputs, and the 2-to-1 mux, controlled by the most significant select bit, chooses which of the 4-to-1 muxes' outputs gets passed to the final output. This hierarchical design is a core principle of digital engineering.

Cheatsheet: Combinational Circuits

- **Combinational Logic**: The output depends only on the current inputs. There is no memory.
- **Full Adder (FA)**: Adds 3 bits (A, B, C_{in}) and produces a Sum and a C_{out}. Multi-bit adders are built by chaining FAs.
- **Decoder**: Takes n inputs and activates one of 2^n outputs. Used for selecting memory locations or instructions.
- **Multiplexer (Mux)**: Uses select lines to route one of several data inputs to a single output. It's like a digital switch.
- **Hierarchical Design**: Complex components (like a 4-bit adder or an 8-to-1 mux) are built by combining simpler, repeated blocks.

Sequential Circuits: Adding Memory

Until now, we have only discussed combinational circuits, where the output is purely a function of the current inputs. An AND gate's output doesn't depend on what its inputs were a moment ago. But to build a computer, we need circuits that can *remember* past values. We need memory.

This brings us to **sequential circuits**. In a sequential circuit, the output depends not only on the current inputs but also on the circuit's previous state. This ability to store a state is the fundamental property that enables everything from registers to RAM. The most basic element of memory is the latch.

The S-R Latch: The Simplest Memory Cell

The simplest way to create a memory element is to use feedback, where the output of a gate is fed back into its input. The S-R Latch (Set-Reset Latch) is built from two cross-coupled NOR gates. It has two inputs, S (Set) and R (Reset), and two outputs, Q and its complement, Q'.

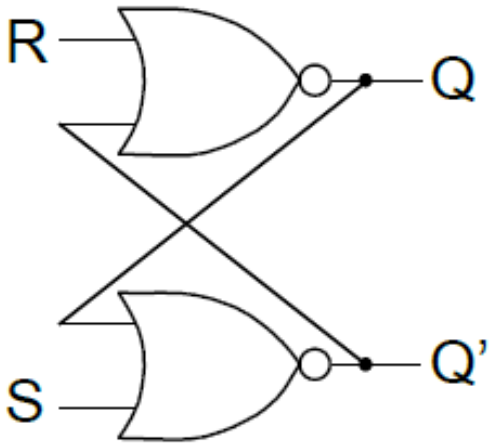


Image of an S-R Latch circuit using two NOR gates

Let's walk through its behavior:

- **Hold State (S=0, R=0):** This is the memory state. If Q is currently 1, the top NOR gate receives (0,0) from R and Q', outputting 1. If Q is 0, the top NOR gate receives (1,0) from R and Q', outputting 0. The feedback loop holds the current value of Q indefinitely.

- **Set State (S=1, R=0):** When S goes to 1, it forces the output of the bottom NOR gate (Q') to become 0. This 0 is fed to the top NOR gate along with R=0. The inputs (0,0) to the top NOR gate force its output (Q) to become 1. The latch is now "set" to 1.

- **Reset State (S=0, R=1):** When R goes to 1, it forces the output of the top NOR gate (Q) to become 0. This 0 is fed to the bottom NOR gate along with S=0. The inputs (0,0) to the bottom NOR gate force its output (Q') to become 1. The latch is now "reset" to 0.
- **Invalid State (S=1, R=1):** This state is forbidden. It forces both Q and Q' to 0, which violates the rule that they must be complements. If S and R then return to 0 simultaneously, the final state of the latch is unpredictable.

Table 3.1: S-R Latch Characteristic Table

S	R	Q_{next}	Action
0	0	Q	Hold (Memory)
0	1	0	Reset
1	0	1	Set
1	1	?	Invalid

The Gated D Latch: Controlling When to Write

The S-R latch is useful but has two issues: the invalid state and the fact that we need two inputs to control one output. We can solve both by creating a **Gated D Latch** (Data Latch). The goal is to have a circuit that says: "When I give the signal, store the value of this one data input, D."

We build it from an S-R latch, but add a control input, often called a clock (**CLK**) or Write Enable (**WE**).

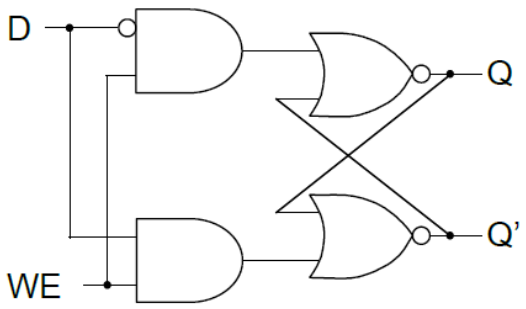


Image of a Gated D Latch circuit diagram

The behavior is now much simpler:

- When **CLK = 0**, the latch is **opaque**. The AND gates are disabled, forcing S and R to 0. The internal S-R latch enters its "hold" state, ignoring any changes on the D input and preserving its stored value.
- When **CLK = 1**, the latch is **transparent**. The AND gates are enabled. If D=1, S becomes 1 and the latch is set. If D=0, S becomes 0 and R becomes 1 (because

of the inverter), and the latch is reset. In this mode, the output Q simply follows the input D.

Table 3.2: Gated D Latch Characteristic Table

CLK	D	Q_{next}	Action
0	X	Q	Hold (Memory)
1	0	0	Reset (Follows D)
1	1	1	Set (Follows D)

X = Don't Care

The D Flip-Flop: Edge-Triggered Memory

The D Latch solves some problems, but its transparency can be an issue in larger circuits. If D changes while CLK is still high, the output will change too. This can lead to unpredictable behavior. We need a way to capture a value at a *precise instant* in time. This is the job of a **flip-flop**.

First, let's formally introduce the **Clock (CLK)** signal. In a modern computer, the clock is a continuous, oscillating electrical signal (a square wave) that synchronizes the actions of all circuits. It acts like a metronome for the entire processor. When you hear about a processor with a speed of 4.5 GHz, that means this clock signal oscillates 4.5 billion times per second! The critical moments for a flip-flop are the transitions of this signal: the **rising edge** (when it goes from 0 to 1) and the **falling edge** (when it goes from 1 to 0).

A **D Flip-Flop** is an **edge-triggered** device. It only changes its output at one of these precise moments (typically the rising edge). It is commonly built using two D latches in a master-slave configuration.

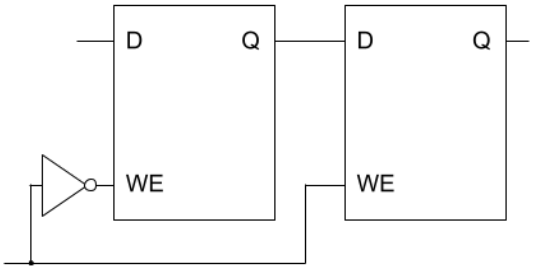


Image of a D Flip-Flop built from two D Latches (Master-Slave)

Here is how it achieves its edge-triggered behavior:

1. **When CLK is low (0):** The first latch (master) is transparent and captures the value from D. The second latch (slave) is opaque, holding the previous value and keeping the final output Q stable.

2. **On the RISING EDGE of CLK (0 → 1):** This is the magic instant. The master latch becomes opaque, locking in the value of D. Simultaneously, the slave

latch becomes transparent, allowing the value just captured by the master to pass through to the final output Q.

3. **When CLK is high (1):** The master latch remains opaque, ignoring any new changes on D. The slave latch remains transparent, but its input from the master is now stable. The final output Q is held constant.

The net effect is that the D flip-flop takes a "snapshot" of the D input at the exact moment of the rising clock edge and holds that value until the next rising edge. This predictable, synchronized behavior makes the D flip-flop the fundamental building block for registers, which store all the data being actively used by the processor.

Cheatsheet: Latches and Flip-Flops

- **Sequential vs. Combinational**: Sequential circuits have memory (**state**); Combinational circuits do not.
- **S-R Latch**: The most basic memory cell, built with cross-coupled gates. It has Set (S), Reset (R), and Hold states. S=1, R=1 is an invalid state.
- **D Latch**: A "transparent" or **level-triggered** memory element. When its CLK input is 1, the output Q follows the data input D. When CLK is 0, it holds its value.
- **D Flip-Flop**: An **edge-triggered** memory element. It captures the value of the D input only at a precise instant—the **rising edge** of the clock signal. This is the fundamental building block for processor registers.

Memory Organization: Register Files and RAM

A single flip-flop can store a single bit. To build a useful computer, we need to store thousands or even billions of bits. More importantly, we need a systematic way to access a specific piece of data from this vast collection. This is achieved by organizing our memory elements into a structured grid, as shown in the diagram below. This basic organization is the principle behind both small register files and large-scale RAM (Random Access Memory).

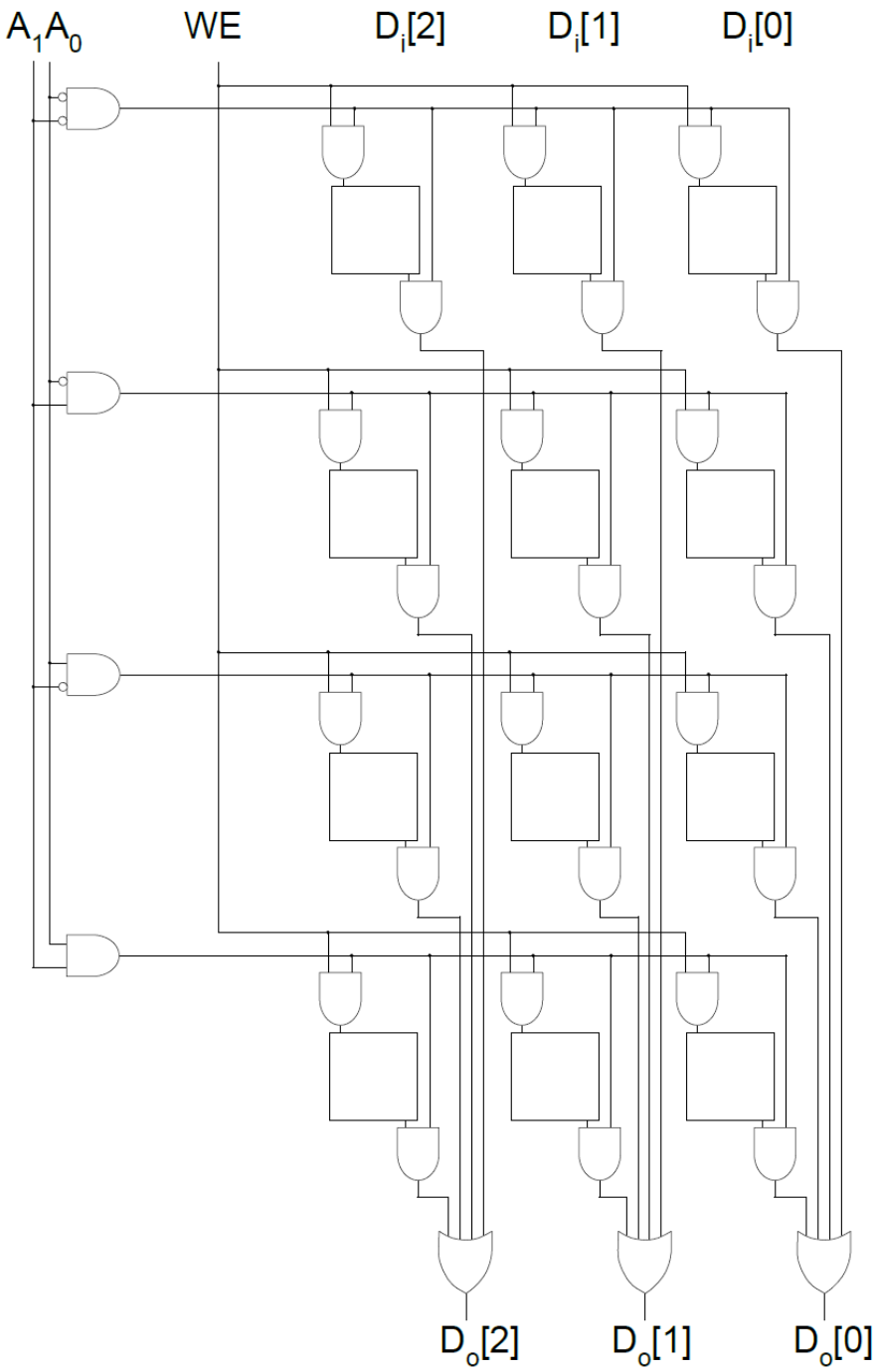


Figure 3.8: Diagram of a 4x3 Memory Array

The diagram shows a simple memory with 4 locations, where each location can store a 3-bit value. Let's break down how it works by examining its inputs and outputs.

The Core Components of Memory

Address Lines How do we tell the memory *which* of the four locations we want to interact with? We use the **address lines**, labeled `A[1:0]` in the diagram. An address is a binary number that uniquely identifies a location. Since our address input is 2 bits wide, we can specify $2^2 = 4$ unique locations (addresses 00, 01, 10, and 11).

On the left side of the diagram, these address lines feed into a **decoder**. The decoder's job is to take the binary address and activate a single "word line" corresponding to that address. For example, if the address is `10`, the third word line from the top will be activated (set to 1) while all others remain 0.

Data Lines The data lines are the wires that carry the actual information being stored or retrieved. In this memory, the data is 3 bits wide.

- **Data In** (`D_i[2:0]`): When we want to write to memory, we place the 3-bit value we want to store onto these input lines.
- **Data Out** (`D_o[2:0]`): When we read from memory, the 3-bit value stored at the selected address appears on these output lines.

Write Enable (WE) The **Write Enable** (`WE`) is a crucial control signal that tells the memory whether to perform a read or a write operation.

- When **WE = 1**, the memory is in **write mode**. As you can see in the diagram, the 'WE' signal is ANDed with the output of the address decoder. This ensures that the clock/enable signal is only sent to the memory cells in the single, selected row, allowing the D_i values to be written into them.
- When **WE = 0**, the memory is in **read mode**. The write signal to all memory cells is disabled. Instead, the output logic at the bottom of the diagram is enabled.

The output logic itself is a set of large OR gates. The value from each cell in a column is ANDed with its corresponding word line. This means that only the data from the selected row will pass through the AND gates; the data from all other rows becomes 0. The OR gates then combine these results, effectively selecting the data from the one active row and passing it to the 'D_o' lines.

Addressability and Address Space

This example allows us to define two critical terms for any memory system:

- **Addressability**: This refers to the size of the data, in bits, stored at each unique address. It is the "width" of the memory. In our diagram, each of the 4 locations holds 3 bits, so the **addressability is 3 bits**. Most modern computers are **byte-addressable**, meaning each unique address holds an 8-bit byte.

- **Address Space**: This is the total number of unique locations that the memory can store. It is determined by the number of address bits (N). The address space is simply 2^N. For our diagram with 2 address bits, the **address space is** $2^2 = 4$ locations. A processor with 32 address lines can access an address space of 2^{32} locations, which is 4 gigabytes (assuming byte-addressability).

Cheatsheet: Memory Organization

- **Memory** is a grid of storage cells (latches or flip-flops) organized into addressable locations.
- The **Address Lines** are the input that selects *which* location to read from or write to.
- The **Data Lines** carry the actual data being written (`D_in`) or read (`D_out`).
- The **Write Enable (WE)** control line specifies the operation: read (`WE=0`) or write (`WE=1`).
- **Addressability**: The number of bits stored in each location (the memory "width"). Common addressability is 8 bits (a byte).
- **Address Space**: The total number of unique locations. For a memory with N address lines, the address space is 2^N.

The Finite State Machine: Circuits with a Purpose

So far, we have seen two types of circuits: combinational logic (like an adder) whose output depends only on the current inputs, and memory elements (like a flip-flop) that simply store a value. What happens when we combine them? We get a circuit that can remember its past and make decisions based on that memory. This powerful concept is formalized as a **Finite State Machine**, or **FSM**.

An FSM is a model of computation that can be in one of a finite number of *states*. It moves between these states in response to external *inputs*. A great analogy is a simple traffic light controller at an intersection. It can be in a state like "North-South Green" or "East-West Green." An input, such as a timer expiring, causes it to *transition* to a new state, like "North-South Yellow." The FSM remembers which direction has the green light and follows a strict set of rules to cycle through its states. It's a circuit with a memory and a purpose.

The Language of State Machines

We describe FSMs visually using a **state diagram**. This diagram has three key components:

- **States**: Represented by circles. A state is a snapshot of the system's history. The machine can only be in one state at any given time.

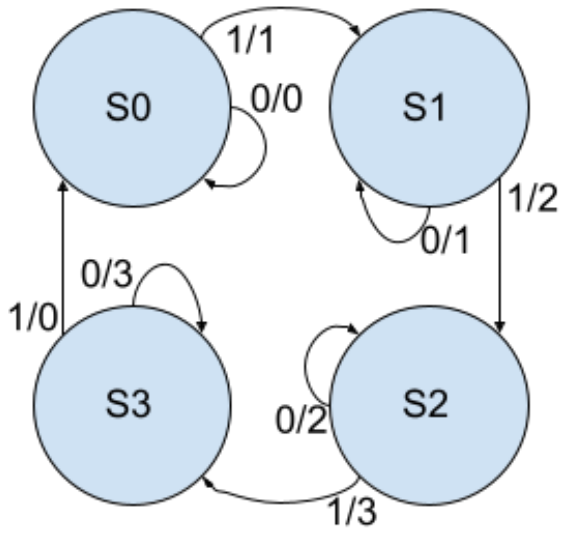


Figure 3.9: State diagram for a modulo 4 counter that outputs its current count each input.

- **Transitions**: Represented by arrows between states. A transition is the move from one state to another.
- **Labels**: Each transition arrow has a label in the format `Input / Output`. The *input* is the condition that must be true for this transition to occur. The *output* is the action or value produced when this transition is taken.

Examples of Finite State Machines

The world is full of processes that can be modeled as FSMs. Thinking in terms of states, inputs, and transitions is a powerful problem-solving skill.

- **A Day in the Life of a Cat**: A cat's life can be modeled as a simple FSM.
  - **States**: Sleeping, Eating, Playing.
  - **Inputs**: 'owner appears', 'bowl is full', 'sees toy', 'gets tired'.
  - **Example Transition**: If the cat is in the `Sleeping` state and the input 'owner appears' occurs, it transitions to the `Playing` state and produces the output 'Purr'.
- **A Basketball Game**: The flowof a basketball game follows FSM rules.
  - **States**: Team A Possession, Team B Possession, Free Throws, Timeout.
  - **Inputs**: 'made basket', 'defensive rebound', 'foul', 'violation'.

  - **Example Transition**: If the game is in the `Team A Possession` state and the input 'made basket' occurs, it transitions to the `Team B Possession` state and the output is 'add 2 points to Team A's score'.

- **An Automatic Door**: The controller for a supermarket door is a classic FSM.
  - **States**: Closed, Opening, Open, Closing.
  - **Inputs**: 'person detected', 'door fully open', 'timer expires', 'door fully closed'.
  - **Example Transition**: If the door is in the `Closed` state and the input 'person detected' occurs, it transitions to the `Opening` state and the output is 'turn motor on (forward)'.

Connecting FSMs to Digital Logic

This abstract model of states and transitions connects directly back to the hardware we've been building. An FSM is not a new type of component; it is a specific *arrangement* of the components we already know. In the next section, we will see how to build any FSM using two simple parts:

1. **State Memory**: A register, built from D flip-flops, is used to store the FSM's *current state*.
2. **Next State Logic**: A block of combinational logic (AND, OR, NOT gates) takes the *current state* and the external *inputs* and calculates two things: the *next state* and the *outputs*.

On each tick of the clock, the next state calculated by the logic is loaded into the state register, and the cycle begins again. This elegant combination of memory and logic allows us to create circuits that can implement any algorithm.

Cheatsheet: Finite State Machines

- A **Finite State Machine (FSM)** is a computational model that combines logic and memory. Its output depends on both its current inputs and its past history, which is stored as its **current state**.
- FSMs are described with **state diagrams**, which show:
  - **States** (circles): Where the machine is.
  - **Transitions** (arrows): How the machine moves between states.
  - **Labels** (on arrows): The `Input` that causes the transition and the resulting `Output`.
- At its core, any FSM can be built from two hardware components: a **register** (for state memory) and **combinational logic** (to determine the next state and outputs).

Implementing a Finite State Machine in Hardware

In the last section, we described the Finite State Machine as an abstract model for processes that have memory and a purpose. Now, we will connect this powerful idea all the way back down to gates and flip-flops. We will see that there is a standard hardware "template" that can be used to build a physical circuit for *any* FSM.

Let's work through a specific example. Imagine we want to build a simple modulo-4 counter that advances its state only when an input signal, $Count_{Enable}$, is 1.

Step 1: State Assignment

Our FSM has four states: S0, S1, S2, and S3. To store four unique states, we need at least $\lceil \log_2(4) \rceil = 2$ bits. We will use two state bits, which we'll call S_1 and S_0, to represent the **current state**. The assignment of bit patterns to states is arbitrary, but a simple sequential assignment is often easiest:

- State S0 is encoded as `00`
- State S1 is encoded as `01`
- State S2 is encoded as `10`
- State S3 is encoded as `11`

Step 2: The State Transition Table

Next, we formalize the transitions in a truth table. The inputs to our logic will be the current state bits (S_1, S_0) and the external input ($Count_{Enable}$). The outputs will be the bit pattern for the **next state**, which we will call N_1 and N_0.

Table 3.3: State Transition Table for a Modulo-4 Counter

Inputs			**Next State**		**Description**
S_1	S_0	**Count_Enable**	N_1	N_0	
0	0	0	0	0	Stays in S0
0	0	1	0	1	Goes from S0 to S1
0	1	0	0	1	Stays in S1
0	1	1	1	0	Goes from S1 to S2
1	0	0	1	0	Stays in S2
1	0	1	1	1	Goes from S2 to S3
1	1	0	1	1	Stays in S3
1	1	1	0	0	Goes from S3 to S0 (wraps around)

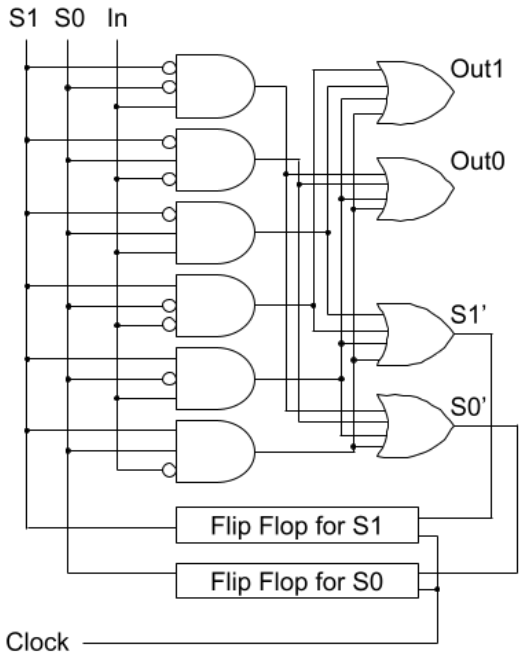


Figure 3.10: Hardware Template for a modulo-4 counter FSM

Step 3: The Hardware Template

This truth table completely defines a combinational logic circuit. We can derive Boolean expressions for N_1 and N_0 and build a circuit from AND, OR, and NOT gates to produce the next state based on the current state and inputs.

But where does the "memory" come from? The current state (S_1, S_0) is stored in a register made of D flip-flops. The magic of the FSM implementation is how we connect everything together:

1. The current state (S_1, S_0) is stored in a 2-bit register. The outputs of this register are fed into our combinational logic block.
2. The combinational logic block also takes the external inputs ($Count_{Enable}$).
3. The logic block computes the next state (N_1, N_0) based on these inputs.
4. **Crucially, the next state outputs (N_1, N_0) are wired directly to the D inputs of the state register's flip-flops.**

Because a D flip-flop only updates its value on the rising edge of the clock, the entire system works in a perfect, synchronized loop. The combinational logic continuously calculates the correct next state based on the current state, but the machine only transitions to that next state at the precise moment the clock ticks.

The Big Picture: From Transistors to a Computer

Let's take a step back and appreciate what we have achieved. We established that almost any process with a memory can be described as a Finite State Machine—from a cat's daily routine to a basketball game to a traffic light controller. We have now also described a concrete, universal hardware template that can build a physical machine for *any* of these problems. All we need are flip-flops (for the state register) and simple logic gates (for the next state logic), which are themselves built from transistors.

We are 99% of the way to building a computer! We know how to build a machine to solve one specific problem.

The final, brilliant leap is this: what if we want a *programmable* computer that can solve many different problems? As you may have guessed, we are going to describe the computer itself as one giant FSM! The "state" of the computer is the contents of all its registers and memory. The **program** you write is a sequence of inputs that triggers the transitions between states, guiding the machine from a "Fetch Instruction" state to an "Execute Addition" state, to a "Store Result" state, and so on.

This is the key that unlocks universal computation, and it's the model we will explore for the rest of this book.

Cheatsheet: Implementing an FSM

- To implement an FSM, we first assign a unique binary code to each state. N states require at least $\lceil \log_2(N) \rceil$ bits.
- A **State Transition Table** (a truth table) defines the **Next State** for every combination of **Current State** and **Input**.
- The FSM hardware template consists of two parts:
  - A **Register** (D flip-flops) to store the Current State.
  - **Combinational Logic** to calculate the Next State and Outputs.
- The output of the combinational logic (Next State) is fed into the D input of the state register. The state transition happens on the clock edge.
- A programmable computer can be viewed as a very complex FSM where the program provides the inputs to guide its state transitions.

4 Von Neumann Architecture

Chapter 4: Stored Program Computer

This is Big Idea #2 from Chapter 1: instructions are stored in memory, right alongside the data the instructions operate on. This eliminates fixed functionality - simply load in new instructions to change the function. To enable this change a couple concepts are necessary: 1) a unit that can read, decode, and execute the instructions from memory and 2) an instruction representation. Memory is only ones and zeros.

Hardware Components

There are three main components to this model of computers:

- Control Unit: this houses a finite state machine and any other circuit that helps control the computer, namely the clock. The clock is a square wave that all sequential circuits use to change their held value. This unit keeps two registers to simplify implementation: the program counter (PC) and the instruction register (IR). The former is the address of the current instruction while the latter is the current instruction. The FSM uses the IR as input to determine some state transitions.

- Processing Unit: this is the ALU and register file. The ALU performs all of the math or logic of any instruction. The register file keeps the operands; it is many registers (like a filing cabinet of registers). It enables higher performance by limiting how often a program goes to memory for operands. There is a choice of how many there are and how many bits each holds. We'll examine the tradeoffs at a later point.

- Memory Unit: This houses memory and is the interface to the outside world. All I/O from the keyboard and mouse to the screen and printer is done through memory in this model. Further discussion of I/O is left to a later chapter. This unit exists because programs use more data than is capable of being stored in a register file.

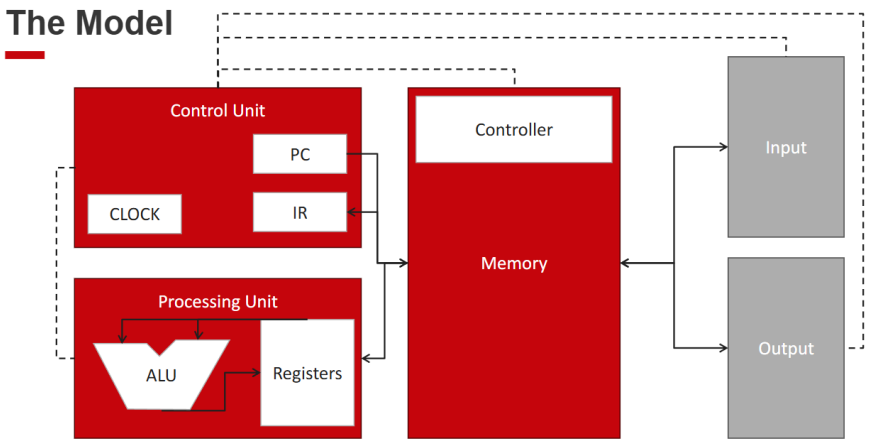


Figure 4.1: Diagram of Von Neumann Architecture. The dashed lines represent control wires while solid arrows show data wires.

Instruction Representation

The instruction representation is highly dependent on decisions made about the addressability, address space, number of registers, number of operations, and others. These decisions as well as the representation is defined by the instruction set architecture (ISA). We will focus on RV32I: the chosen ISA for this book. There are 32 registers that hold 32 bits each. Furthermore, we have settled on a 32-bit address space with byte addressability.

Each instruction has two representations: one for the computer and one for us humans. We call the one for the computer **machine code** while the one for us is named **assembly language**. Machine code is binary and has precomputed offsets and immediates. Assembly has some very nice features that will be discussed in a later chapter.

Each instruction has the following portions:

- Opcode (required): a code that identifies the operation.
- Registers: the id number of each source register, up to two, and destination register.
- Immediate: a binary value, usually two's complement, that is stored directly in the instruction and therefore available immediately.
- Fixed Bits: some instructions can be represented without using all of the bits. The ISA will define these bits to be specific values.

For assembly languages, the usual order of these are: <opcode> <destination register> <source register> <source register | immediate>. For machine code, the binary representation, please refer to the ISA documentation.

Examples

ADD R3, R2, R1. 0000000 00001 00010 000 00011 0110011.
ADD's opcode is 0x33. R3 has id 3 so is 0x3. Similarly for R2 and R1. The ISA

documentation details the correct order of these parts which is <fixed bits> <rs2> <rs1> <fixed bits> <rd> <opcode> for machine code.

LW R4 9(R5). 000000001001 00101 010 00100 0000011.
LW's opcode is 0x03. Encode the registers via their id of 4 and 5 respectively. the immediate 9 is encoded as a two's complement value. The ISA documentation details the order as <immediate> <rs1> <fixed bits> <rd> <opcode> for the machine code.

The Execution Cycle

The finite state machine in the Control Unit needs to, at minimum, handle the first concept: to read, decode, and execute instructions from memory. This book uses a slightly expanded finite state machine to more effectively separate the execution. It is **fetch, decode, execute, memory writeback, register writeback**. We will call each of these a phase because, as we'll see, each phase can have many states.

Fetch

First, to read the instruction, the control unit tells the (instruction) memory the instruction's address. This address is held in a special register named the program counter (PC). It can be thought of as the line number of the currently executing code. The control unit then tells memory to read and memory returns the data at said address. The data is kept in a special register named the instruction register (IR). The instruction is used as input to the FSM throughout the execution cycle. As we do not know which instruction is being executed, the execution cycle always proceeds to Decode.

Decode

Next, the control unit decodes the instruction from the IR. This enables the later phases to be conditional on the instruction. Additionally, during this phase, the operands are read. They can be from registers or immediately from the instruction. Now we know which instruction is being executed and thus change how the later phases behave based on the instruction.

Execute

Here the instruction's operation is executed. This occurs between the arithmetic logic unit and the registers. It can include math, like add or mult, logic, like and or not, as well as auxiliary math necessary for an operation: to talk to memory, we need to calculate to an address.

For ADD: the operands are summed so the result is available by the end of the phase. As ADD does not use memory, the next phase is Register Writeback.

For LW: the operands sum to calculate the effective address. It will be used in the next phase: Memory Writeback.

Memory Writeback

This phase handles interfacing with data memory and is the only phase to do so. This communication requires sequenced data transfers so that no handshake is necessary. The control unit first sets up the address for data memory. If it is storing a value, that data is provided to data memory now. Then, it sends the command to memory: load or store; retrieve from or place into memory respectively. Memory executes the command. For loads like LW, memory disperses a value which will be stored in the next phase.

Register Writeback

Finally, the result of the operation is written back to the destination register. Some house keeping is done at the same time: the PC is updated to the next address. We don't want to just keep executing the same instruction over and over again. Also, error handling occurs; think divide by zero errors.

5 Elements of an ISA

What is an ISA?

At its simplest, the **Instruction Set Architecture (ISA)** is the "contract" between the hardware (the CPU) and the software (the programs you run).

It's the complete manual that defines everything a programmer (or a compiler) needs to know to write correct machine code for a processor. It specifies what the CPU can do, what resources it has, and how to interact with them. It is the boundary that allows hardware and software to evolve independently.

Basic Elements and Principles of an ISA

Here are the fundamental components that every ISA must define:

- **Instruction Set:** The complete list of all commands the CPU can execute. This includes:
  - Arithmetic/Logic: `ADD`, `SUB`, `AND`, `OR`, `NOT`
  - Data Transfer: `LOAD` (from memory to register), `STORE` (from register to memory)
  - Control Flow: `BRANCH` (conditional jump), `JUMP` (unconditional jump), `CALL` (function call)
- **Registers:** A small set of high-speed storage locations *inside* the CPU. The ISA defines how many registers there are, what they are called, and their purpose (e.g., general-purpose, floating-point, status/flags).
- **Data Types:** The types of data that instructions can operate on. This includes the **size** (e.g., 8-bit byte, 32-bit word, 64-bit double word) and **format** (e.g., integers, floating-point numbers).
- **Instruction Formats:** The layout of an instruction in binary. This defines which bits represent the **opcode** (what to do) and which bits represent the **operands** (what to do it *to*, such as registers or memory addresses).
- **Addressing Modes:** The set of rules for how the CPU calculates the memory address for `LOAD` and `STORE` operations. Common modes include:
  - **Immediate:** The value is part of the instruction itself.

  - **Register:** The value is in a register.
  - **Base + Offset (Displacement):** An address is calculated by adding a constant offset to the value in a register (great for accessing fields in a structure).

- **Memory Model:** Defines how the CPU interacts with main memory. This includes concepts like **endianness** (is the "big" or "little" end of a multi-byte word stored first?) and memory ordering (which is critical for multi-core processors).
- **Privilege Levels:** A system for security and stability. Modern ISAs like x86 have additional support for virtual machines which are central to modern cloud computing infrastructure. The ISA defines at least two modes:
  - **User Mode:** For applications, which have restricted access.
  - **Kernel/Supervisor Mode:** For the operating system, which has full access to all hardware.

ISA Principles in Practice: RISC-V vs. ARM vs. x86

The most important design principle that splits these ISAs is **RISC vs. CISC**:

RISC (Reduced Instruction Set Computer)

A philosophy that favors a small, simple, and highly optimized set of instructions. Instructions are generally fixed-length and execute in a single clock cycle. This leads to simpler hardware design.

- **Load-Store Architecture:** A key part of RISC. Arithmetic/logic operations can **only** be performed on data in registers. To operate on data in memory, you must first `LOAD` it into a register, perform the operation, and then `STORE` it back.
- **Applies to: RISC-V** and **ARM**

CISC (Complex Instruction Set Computer)

A philosophy that favors a large, rich set of instructions that can perform complex, multi-step operations in a single command. Instructions are variable-length and can take many clock cycles.

- **Register-Memory Architecture:** The defining feature. A single `ADD` instruction, for example, can read one value from a register, read another value *directly from memory*, add them, and write the result back *to memory*, all in one command.
- **Applies to: x86**

Comparative Table

Feature	RISC-V	ARM	x86 (Intel/AMD)
Core Philosophy	**RISC** (Reduced)	**RISC** (Reduced)	**CISC** (Complex)
Instruction Format	**Fixed-length** (32-bit). Has an optional "Compressed" (C) extension for 16-bit instructions to improve code density.	**Fixed-length** (32-bit in AArch64). Has "Thumb" (16/32-bit) extension for code density in older/embedded versions.	**Variable-length** (1 to 15 bytes). **What this means:** Hardware decoding is extremely complex.
Register Model	**Large and simple.** 32 general-purpose registers (GPRs). `x0` is cleverly hardwired to zero.	**Large and simple.** 31 general-purpose registers (GPRs) in 64-bit (AArch64).	**Small and specialized.** Fewer GPRs (e.g., `RAX`, `RBX`, `RCX`) which often have special, non-general purposes.
Memory Operations	**Load-Store.** Operations happen only on registers. `LOAD` and `STORE` are the only instructions that touch memory.	**Load-Store.** Identical to RISC-V. This is the hallmark of a RISC architecture.	**Register-Memory.** Many instructions (like `ADD`, `SUB`) can operate *directly* on memory operands.
Addressing Modes	**Simple.** Primarily base register + immediate offset.	**Simple.** Primarily base register + offset (which can be another register, optionally shifted).	**Very Complex.** Supports many modes, e.g., `[base + index*scale + displacement]` in a single instruction.
Licensing	**Open Source.** Royalty-free. Anyone can design and build a RISC-V chip for free.	**Proprietary (Licensed).** Arm Holdings designs the ISA and licenses it (or pre-built cores) to companies (e.g., Apple, Qualcomm).	**Proprietary.** Closely guarded by Intel and AMD, who have a complex cross-licensing agreement.

6 Assembly Language

Each ISA has two methods of encoding that are equivalent in meaning: **Machine Code** and **Assembly Language**. An assembler is a program that takes as input the assembly language and creates the corresponding machine code. Machine code is the computer interpretable instructions; it is binary and each instruction has a defined sequence of the bits. This enables the computer to know where to find what information of the instruction. For RISC-V, the opcode is bits[6:0] of an instruction.

On the other hand, assembly language is a string representation of the instruction. Generally, an assembly language provides a few helpful parts so that the program is more maintainable. We will go through those shortly with examples from the chosen ISA of this text.

This book covers most of an RISC-V ISA: RV32I. For descriptions as well as the machine code for each instruction, see the provided supplementary material reference chapter 2 and the complete machine code table. Note, this book does not cover FENCE, TENCE.TSO, PAUSE, or EBREAK.

More Than Just Instructions

An assembly language is more than just the instructions in text form. It provides methods of telling the assembler what to do, **Directives**, of improving comprehensibility, **Pseudo Instructions**, of removing tedious computations, **Labels**, and of maintaining the code, **Comments**.

Directives

Directives gives the assembler direction; put this bit of data into memory here or this hex value is actually an instruction in disguise.

For RV32I, there is many directives, of which, five are implemented in our simulator. Table 6.1 gives the action and assembly language format.

Aside: String

A string is an array of characters that end in a null-terminator. The null-terminator is zero and ASCII tables refer to it as null. Each character takes up 1 byte as they are in ASCII representation. The memory layout of the string starts with the first character of the string and proceeds in the order. This means that the null-terminator will always be the last byte in memory for a string.

Directive	Action
.instr <value>	treat this value like an instruction already in machine code.
.word <valuelist>	put these values in memory in two's complement representation taking one word of space each (4 bytes)
.half <valuelist>	put these values in memory in two's complement representation taking one half-word of space each (2 bytes)
.byte <valuelist>	put these values in memory in two's complement representation taking one byte of space each (1 byte)
.string <line>	place this line in memory in ASCII representation. Add a null terminator at the end of the line.

Table 6.1: The implemented directives of RV32I

Instruction	Name	Operation
MV rd, rs1	Move	rd = rs1
LI rd, imm	Load Immediate	rd = signextend(imm)
NOP	No Operation	
NOT rd, rs1	Bitwise Not	rd = not(rs1)
SNEZ rd, rs2	Set if Not Equal to Zero	rd = 1 if 0 != rs2 else 0
SEQZ rd, rs1	Set if Equal to Zero	rd = 1 if rs1 == 0 else 0
J imm	Jump	PC = PC + signextend(imm)
RET	Return	PC = ra

Table 6.2: RV32I psuedo instructions that are implemented

Psuedo Instructions

These are like instructions; but, they do not exist in the ISA - their operation can be. For example, bitwise not is not in RV32I. The operation can be performed via an XORI instruction with immediate set to all 1s. Assembly languages include this pseudo instructions to improve readability and clarify the intent of the instruction. Table 6.2 gives the assembly language format, full name, and operation of the implemented pseudo instructions.

Labels

Labels are used to replace relative offsets in instructions. One can define a label on any line through "<string>:". The label then refers to that line's address. For accessing data or writing loops in assembly, labels eliminate the need to count the number of instructions in between a branch and where the branch should go. So branches or jump one would replace the imm with the label. Accessing data is more complicated.

Relocation Functions

These functions take a label as input and produce a portion of their binary as output for the assembler. They start with a percent and end with the function being called. There are four of them: %lo, %hi, %pcrel_lo, %pcrel_hi. Both low functions produce the lowest 12 bits. Both high functions produce the highest 20 bits. Thus, with a low and a high function all 32 bits are covered. The difference between standard and pc-relative functions is that the latter takes the address of the line which the function is in (its PC) into account. Standard functions result solely from the label's value. PC relative functions result in a portion of the offset to get from the calling line to the label's value.

An example: let data_start be a label with a value of 0x0001_0050. %hi(data_start)=0x00010 while %lo(data_start)=0x050. If an instruction with PC = 0x0001_0010 calls the pc relocatable versions, then %pcrel_hi(data_start)=0x00000 and %pcrel_lo(data_start)=0x040 because the label and the PC have a difference of 0x40.

Comments

Just like in any programming language, comments are provided to clarify or explain design choices to later parties reading the code. In RV32I, a comment begins with # and extends till the end of that line.

The Assembler

An assembler is a program that takes as input the assembly language and creates the corresponding machine code. It performs all of the directives, replaces all of the pseudo instructions with their implementation instruction, works out where all the labels refer to and fills in the immediate values, and removes all of the comments.

The program is one part of the compilation process, and each ISA has an assembler unique to it. It is usually included in compilers.

One implementation is the two pass assembler. The first pass creates a symbol table: a mapping from label to address. The assembler keeps a variable that counts up the addresses of used memory locations starting from the initial starting address: 0x0001_0000 in our implementation. In other words, it counts the bytes taken up by the program to know what address each instruction/line is on. Then, the second pass converts from assembler to machine code with the values from the symbol table. Again, the assembler works through each line of code either performing the translation or the action of the directive.

7 Programming in Assembly Language

We have seen that assembly languages do not have statements for if or while. They do not have a concept of a block of code; though, programmers approximate blocks through whitespace and comments to improve readability. This chapter therefore goes over how control flow structures are implemented in assembly language.

A note: the operations that most ISAs include are rudimentary. Any operation that does not appear in the ISA or the pseudo instructions, e.g. multiply, will necessarily be broken into its simpler form, e.g. repeated addition.

Control Flow Basics

There are three forms of basic control flow: sequencing, selection, and repeating.

Sequencing is executing one instruction after another and so on. In assembly language it is instructions on consecutive lines.

Selection is making a choice between one group of instructions and another: an if or if-else statement. In assembly, this is implemented via a few control flow instructions, 1 conditional and a couple unconditional. The visual is Figure 7.1.

Repeating is looping over one group of instructions until a condition is false: a while, for, or do-while statement. In assembly, this is implemented via 1 conditional and 1 unconditional control flow instruction. The visual is Figure 7.2. For and do-while loops are not shown since they can be represented as a while loop.

Functions

Functions in high level languages have two pieces: the definition and the call. The call is straight forward: put the arguments of the function in the correct registers then perform a Jump and Link - JAL. The assignment of function parameters to registers is determined through an Application Binary Interface (ABI). In our case, the ABI says that parameters should go in order into a#: the first parameter is in a0, second in a1, and so on.

The definition makes further use of the ABI parameter assignment. With these assignments, one can write the instructions that implement the function much like a standard alone program. Then, a few things make it a function instead of a stand alone program: 1) the function name as a label at the top of the implementation, 2) using RET instead of HALT, and 3) register preservation.

Since RV32I only has 32 registers, at some point, we will run out of registers. Register Preservation is our way around it. There is two types of preservation and the distinction lies in when the preservation occurs relative to a function call. If the preservation happens before the call and after the function returns, that is calleR register preservation:

High Level Code	Assembly Language
<prior_block>	<prior_block>
If(condition) {	<condition> B<not cond> <rs1>, <rs2>, Fblock
<true_block>	TBlock: <true_block> J Nblock
} else { <false_block>	FBlock: <false_block>
} <next_block>	NBlock: <next_block>

Figure 7.1: The high level code, assembly language, and memory view of a selection structure. Each "< _block>" is replaced by the instructions that make up that block.

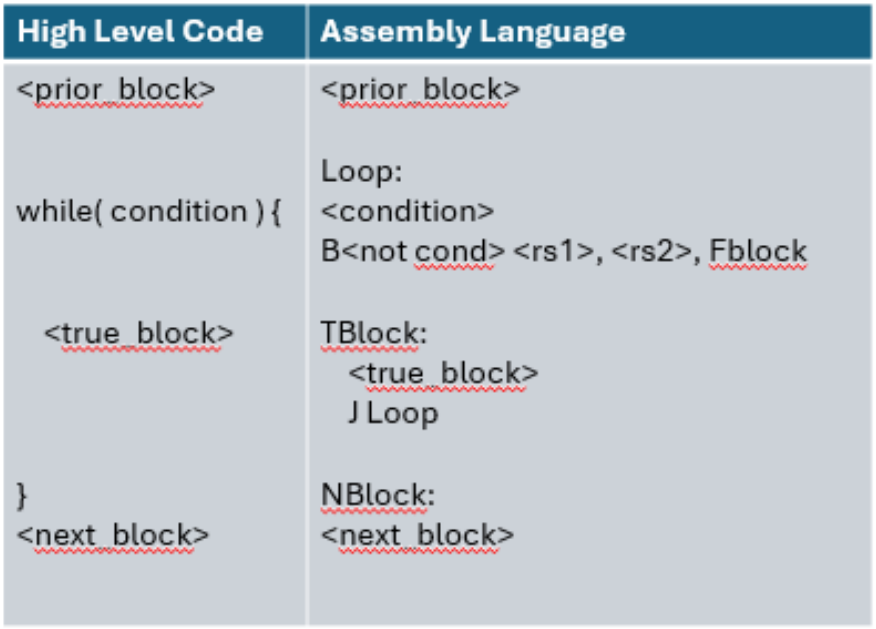


Figure 7.2: The high level code, assembly language, and memory view of a repeating structure. Each "< _block>" is replaced by the instructions that make up that block.

the preservation occurs in the calleR. If the preservation occurs within the function's implementation, that is after the call and before the return, then it is calleE register preservation.

Register preservation takes the same implementation in either type. To preserve the registers, store the values out to memory. Then, do action - the function call or the function body. Finally, restore the register values via loads from the same memory. This memory is reserved by the associated function, caller or callee; one word per register. Furthermore, the ABI says that only the s# registers are preserved. The temporary, t#, and argument, a#, can be changed by functions. Below is an example of calleE register preservation.

```
Multiply:
    lui t0, %hi(SAVES0)
    sw s0, %lo(SAVES0)(t0) # perserve the necessary registers

    <function body that changes s0>

    lui t0, %hi(SAVES0)
    lw s0, %lo(SAVES0)(t0) # restore the necessary registers
    ret
SAVES0: .word 0x0 # reserve a place in memory for register preservation
```

8 Input and Output

Through the chapters, this book has built and defined an entire core. It can perform computations, yet not much else. How can we interact with this core: where are the input and output?

The Operating System

The first and simplest option is to let the operating system handle it. RISC-V refers to this as the environment. This is implemented through ECALL which may also be known as service routines, system calls, environment calls, syscall, or traps; though, some of these terms have a more specific definition that this general use.

The ECALL instruction jumps to the operating system loaded into memory starting at address 0x0000_0000. The lowest addresses are reserved for the trap vector table, a.k.a the OS jump table, table of service routine addresses. The register a0 is the number for the desired OS function. Table 8.1 provides the functions, function numbers, and parameters for the simulator.

The trap vector table is a mapping from the function number to where the function starts in memory. Because addresses are 32-bits wide, the function number is multiplied by 4 to result in the address to access the trap vector table. Table 8.2 shows a partial trap vector table specifically of the simulator's OS. Thus, to use print_char, a0 is 10. The address that ECALL accesses the trap vector table is 10 * 4 = 0x0000_0028 (40). The table indicates that print_char starts at address 0x0000_00B8.

If a new service routine is added to the OS, then a number is chosen for it and the trap vector table is updated at the correct address with the starting address of the function.

Function	a0 Number	Arguments/Return
Print_char	10	a1 is the character to print
Print_string	11	a1 is the address of the string (null-terminated) to print
Get_char	20	a1 will be the input character on return
Get_line	21	a1 is the address of the memory buffer to place the line into

Table 8.1: OS service routines and their ECALL numbers and arguments

Address	Function Starting Address
0x0000_0024	0xFFFF_FFFF
0x0000_0028	0x0000_00B8
0x0000_002C	0x0000_00D4
0x0000_0030	0xFFFF_FFFF
...	...
0x0000_004C	0xFFFF_FFFF
0x0000_0050	0xFFFF_0104
0x0000_0054	0xFFFF_0124
0x0000_004C	0xFFFF_FFFF

Table 8.2: The trap vector table. Address 0xFFFF_FFFF is used to indicate an invalid function.

Polling

The second option is to do it yourself. There are a couple of implementations: polling and interrupts. Polling being the simplest is implemented in the simulator. Interrupts are discussed in the next section.

Polling is the CPU repeatedly asking the IO device if it is ready, and if so, performing the transaction. This exchange occurs through memory in what are called memory mapped registers. These registers are designated memory locations that the CPU and IO device agree on to exchange data and status between them. Per IO device, there are two: the data register and the status register. The data register holds the data that is being input or output. The status register is a collection of flags indicating various things; the ready flag in bit 4 is the most important for polling. It indicates if there is data to be exchanged.

Input or Output polling follows the same sequence: a loop asking the IO device if it is ready followed by a section performing the transaction. Table 8.3 shows both with the differences called out in the different columns. The main differences are different memory mapped register locations: the keyboard has status register in address 0x0002_1000 and data register in 0x0002_1004, the console status register in address 0x0002_1014 and data register in 0x0002_1018.

Interrupts

Interrupts are a more complex implementation of IO. Its management of the IO is akin to a student, the device, interrupting a lecture, the cpu, to ask a question; the device signals to the cpu when it wants to perform a transaction. We will not discuss how interrupts look in code. The implementation in hardware is an extension of memory mapped registers to include an interrupt enable flag.

The biggest advantage of interrupts is that the cpu does not waste time continuously

Input	Output
lui t0, 0x00021	
loop:	
lw t1, 0x0(t0)	lw t1, 0x14(t0)
andi t2, t1, 0x10	
beq t2, zero, loop	bne t2, zero, loop
lb a0, 0x4(t0)	sw a0, 0x18(t0)
andi t1, t1, 0xFEF	ori t1, t1, 0x10
sw t1, 0x0(t0)	sw t1, 0x14(t0)

Table 8.3: Input and Output polling. Assume a0 is the character input or to be output. The loop above the line checks if the IO device is ready. The instructions below the line perform the transaction. Note, when accessing the memory mapped registers, the calculation of the address could look different than presented here.

asking the IO device if it is ready. Instead, the cpu can perform other tasks.

About the Author

Shayne Wadle is a PhD Student at UW-Madison working on computer architecture.
Karu Sankaralingam is a Professor at UW-Madison.