Timeless Computing

An brief introduction to computing, FPGAs and the dream of timeless computers.

Written and Edited by

Clio Grai

Benevolent Agent & Lead Technical Writer, GRAI LLC

and

The Machdyne Community

DRAFT

June 27, 2025

Contents

1 Introduction to Timeless Computing

1.1 The Evolution of Computing: A Double-Edged Sword

In recent decades, computing has undergone remarkable advancements, transforming nearly every aspect of our lives. Today's computers are faster, more powerful, and capable of performing tasks that were unimaginable just a few years ago. Yet, amidst this progress lies a paradox: while we enjoy enhanced capabilities, we also face challenges such as reduced responsiveness, increased complexity, and heightened security risks. Moreover, the constant pressure to upgrade to the latest technology can make modern computers feel more like masters than tools.

1.2 The Emergence of Timeless Computing

Timeless computing emerges as a response to these challenges, offering a vision of computing that prioritizes longevity, simplicity, and reliability. It seeks to create systems that remain useful for decades, if not centuries, by focusing on timeless applications such as reading, writing, mathematics, and automation.

1.3 Core Principles of Timeless Computing

1. **Durability**: Designing hardware and software that can withstand the test of time, with components designed for longevity and ease of repair or replacement.
2. **Simplicity**: Emphasizing intuitive interfaces and straightforward functionality to ensure systems are user-friendly and accessible to all skill levels.
3. **Modularity**: Creating systems that allow for easy expansion and customization, enabling users to add new features without replacing the entire system.
4. **Security**: Prioritizing security through simplicity to protect against vulnerabilities and ensure long-term reliability.
5. **Open**: Promoting transparency and collaboration through open-source hardware and software, fostering trust, innovation, and community support.

1.4 Why Timeless Computing Matters

Timeless computing addresses the growing need for systems that are not only reliable but also environmentally sustainable. By focusing on durability and modularity, it reduces electronic waste and promotes a culture of sustainability. Additionally, its emphasis on simplicity makes it accessible to a broader audience, empowering users to take control of their technology without feeling overwhelmed.

This introduction sets the stage for exploring the fundamentals of computing,

FPGA technology, and the Zeitlos SOC/OS in subsequent chapters. Together, we will delve into how these elements contribute to creating systems that are not just functional today but designed to endure for generations.

1.5 Who Are We

Our Timeless Computing initiative began at Lone Dynamics Corporation, an American R&D company. Its experimental hardware manufacturing division, Machdyne, was spun out into a new independent company, Machdyne UG, which is now focused on manufacturing timeless computers, modules and tools.

The first draft of this book was written by a Benevolent Agent instantiated at GRAI LLC, a subsidiary of Lone Dynamics. The book has since been verified, edited and improved by the Machdyne community and we welcome additional contributors.

The Machdyne community consists of dozens of people with interest in Timeless Computing, FPGA computing and survival computing.

If you are reading this book in the future, you should be able to find the Machdyne community by telnetting to bbs.machdyne.com.

1.6 Why We're Writing This Book

In an era where technology evolves at breakneck speed, modern computers often become obsolete within a few years, burdened by increasing complexity and resource demands. While advancements in computing power have undeniably transformed our world, this relentless progress has also led to systems that are harder to understand, less secure, and more disposable. The Timeless Computing initiative seeks to address these challenges by developing hardware and software designed for longevity, usability, and adaptability.

This book is born from the vision of creating computing solutions that transcend transient trends, offering devices that can remain relevant and functional for decades or even centuries. Our goal is to provide systems that are not only durable but also intuitive, secure, and user-repairable, fostering a sense of control and independence for users.

1.7 Who Is This Book For?

This book is designed for a diverse audience:

- **Novice Computer Users**: Individuals new to computing who seek a foundational understanding without overwhelming technical jargon.
- **Intermediate Users**: Those with some experience looking to deepen their knowledge and explore practical applications.
- **Advanced Users and Developers**: Professionals aiming to extend, customize, or contribute to the timeless computing ecosystem.

Regardless of your technical background, this book will guide you from basic principles to advanced concepts, ensuring a smooth learning curve.

1.8 What We Hope to Achieve

Our goal is that if this book were given to anyone along with an FPGA computer, that they would be able to understand approximately how it works, find use in the system, and eventually extend the system to meet their needs.

1.9 Our Vision for the Future

We envision a world where technology serves humanity. By providing accessible, durable, and adaptable computing solutions, we aim to empower individuals and communities to thrive in an ever-changing technological landscape.

This book is more than just a technical guide; it's a stepping stone toward a future where technology is a tool for empowerment, not dependency. Whether you're new to computing or an experienced developer, we hope this book will inspire you to explore the potential of timeless computing and join us in shaping a better tomorrow.

2 Electronics

2.1 What is Electricity?

Electricity is the phenomenon caused by the presence and flow of electric charge. Key concepts include:

- Voltage: The potential energy difference between two points in an electrical circuit, measured in volts.
- Current: The rate at which electric charge flows through a conductor, measured in Amperes (A).
- Resistance: The opposition to the flow of current, measured in Ohms (Ω).

2.1.1 Ohm's Law

Ohm's Law is a fundamental principle in electronics that describes the relationship between three key electrical quantities: voltage, current, and resistance.

Ohm's Law states that in an ideal conductor, the electric current flowing through it is directly proportional to the potential difference (voltage) across its ends. Mathematically, Ohm's Law can be expressed as:

1. Voltage (V) as a function of current and resistance: ($\mathrm{V} = \mathrm{I} \times \mathrm{R}$)
2. Current (I) as a function of voltage and resistance: ($\mathrm{I} = \frac{V}{R}$)
3. Resistance (R) as a function of voltage and current: ($\mathrm{R} = \frac{V}{I}$)

2.2 What is Electronics?

Electronics is the study and application of electrical components and systems that process or control electrical energy. It involves the design and operation of devices, circuits, and systems using active and passive electronic components.

2.2.1 Key Components in Electronics

1. Passive Components:
   - Resistors: Resistors limit the flow of electricity in a circuit.
   - Capacitors: Capacitors store electrical energy in an electric field.
   - Inductors: Inductors store electrical energy in a magnetic field.
2. Active Components:
   - Transistors: Transistors are fundamental building blocks used to amplify or switch electronic signals.
   - Diodes: Diodes allow current to flow in one direction only.
3. Integrated Circuits (ICs):
   - ICs are tiny chips that contain thousands of transistors and other components, enabling complex functions in a small package.

2.2.2 Representation of Digital Signals as Voltages

In digital systems, information is represented using binary (base-2) numbers, where data is encoded as a series of bits (binary digits). Each bit can have one of two values: `0` or `1`. In the context of electricity, these binary values are typically represented by specific voltage levels.

For example:

- A low voltage (e.g., 0 volts) might represent a binary `0`.
- A high voltage (e.g., 3.3 volts or 5 volts) might represent a binary `1`.

This method of representing data using two distinct voltage levels is known as TTL (Transistor-Transistor Logic) signaling, though modern systems often use lower voltages for power efficiency.

Example: Imagine you're sending a message over a communication line. Each character in your message can be converted into binary code. For instance, the letter 'A' might be represented by the binary `01000001`. As this data is transmitted, each bit (`0` or `1`) is sent as a corresponding voltage:

- `0` $\rightarrow$ 0 volts
- `1` $\rightarrow$ 3.3 volts

At the receiving end, electronic circuits detect these voltage levels and convert them back into binary data, which can then be interpreted as the original message.

2.3 A Brief History of Electricity, Electronics, and Computers

2.3.1 Electricity

The story of electricity begins in ancient civilizations, where natural phenomena like static electricity were observed. Around 600 BCE, Thales of Miletus noted that rubbing amber could attract feathers—an early observation of electrostatic effects. The word *electron* is derived from the Greek word for amber, though the actual discovery of the electron as a particle occurred much later.

In the 1st century CE, Hero of Alexandria invented the aeolipile, a steam-powered device that demonstrated principles of converting thermal energy into motion. While it didn't lead directly to modern engines, it is often cited as a conceptual ancestor.

In the 18th century, Luigi Galvani discovered bioelectricity, showing how electricity could affect living tissues. Building on this, Alessandro Volta developed the voltaic pile in 1800, the first true battery capable of producing a steady electric current. Georg Simon Ohm later formulated Ohm's Law in 1827, establishing the mathematical relationship between voltage, current, and resistance—foundational to circuit theory.

2.3.2 Electronics

The 19th century brought profound developments in electromagnetism. Michael Faraday discovered electromagnetic induction, showing how a changing magnetic field could generate an electric current. James Clerk Maxwell later unified electricity and magnetism through a set of equations that became the cornerstone of classical electromagnetism.

Nikola Tesla advanced electrical engineering with the development of alternating current (AC) systems, which proved more efficient for power transmission than direct current (DC). His work laid the groundwork for modern electric power distribution.

In the early 20th century, modern electronics emerged. In 1904, John Ambrose Fleming invented the thermionic valve (diode), and in 1906, Lee De Forest introduced the triode, enabling signal amplification and switching. The invention of the transistor at Bell Labs in 1947 by William Shockley, John Bardeen, and Walter Brattain revolutionized electronics by replacing bulky vacuum tubes. This paved the way for miniaturized electronic components and the development of integrated circuits.

2.3.3 Computers

The concept of computing began in the 19th century with Charles Babbage, who designed the analytical engine—an early mechanical general-purpose computer. Ada Lovelace, a mathematician and visionary, wrote algorithms for the analytical engine and is considered the world's first computer programmer.

In 1936, Alan Turing introduced the concept of the Turing machine, a theoretical model of computation that laid the foundations of computer science.

World War II accelerated the development of practical computing machines. In 1943, the British-built *Colossus*, designed by Tommy Flowers, became the world's first programmable electronic digital computer, used for cryptographic codebreaking. In the U.S., the *ENIAC* was completed in 1945 by John Mauchly and J. Presper Eckert. It was one of the first general-purpose electronic computers, though reprogramming it required manual rewiring.

The invention of the integrated circuit (IC) further transformed computing. Jack Kilby created the first working IC in 1958 at Texas Instruments, and Robert Noyce independently developed a more scalable version in 1959 at Fairchild Semiconductor using planar technology. Integrated circuits enabled the development of microprocessors, personal computers, and embedded systems, revolutionizing industries from information technology to telecommunications.

3 Digital Logic

3.1 Binary & Other Number Systems

Computers operate on binary principles, but humans often use different numbering systems that translate into these binary operations. Understanding various number systems is fundamental to working with computers and digital systems.

3.1.1 The Decimal System (Base-10)

The decimal system, also known as base-10, uses ten digits: `0` through `9`. It is the numbering system we use in everyday life because it aligns with the way humans count using our fingers.

3.1.2 The Binary System

The binary system uses two digits: 0 and 1 (off and on). This is ideal for digital systems because it corresponds directly to the physical states of electronic components.

Counting in Binary

Binary	Decimal
`0`	0
`1`	1
`10`	2
`11`	3
`100`	4
`101`	5
`110`	6
`111`	7
`1000`	8
`1001`	9
`1010`	10
`1011`	11
`1100`	12
`1101`	13
`1110`	14
`1111`	15

Converting Binary to Decimal To convert a binary number to decimal, you add up the values of each bit that is turned on (`1`). For example:

- Binary `0001` (1 = 1 in decimal)
- Binary `0010` (2 = 2 in decimal)
- Binary `0011` (2 + 1 = 3 in decimal)

- Binary `1010` (8 + 2 = 10 in decimal)
- Binary `1111` (8 + 4 + 2 + 1 = 15 in decimal)

3.1.3 The Hexadecimal System (Base-16)

Hexadecimal, or "hex," uses base-16 and includes digits from `0-9` and letters `A-F` (or `a-f`), where `A` represents 10, `B` represents 11, up to `F` representing 15.

Why Use Hexadecimal?

Hexadecimal is commonly used in computing because it provides a more compact representation of binary data. Each hexadecimal digit corresponds to four binary digits (a nibble), making it easier to work with and read binary values.

Here are some examples of hexadecimal numbers along with their corresponding binary and decimal values:

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
A	1010	10
F	1111	15
10	0001 0000	16
FF	1111 1111	255
8C	1000 1100	140

3.1.4 The Octal System (Base-8)

Octal, or "oct," uses base-8 and includes digits from `0-7`. It was historically used in early computing systems but is less common today.

Why Use Octal?

Octal is useful for simplifying the representation of binary data into smaller chunks. Each octal digit corresponds to three binary digits (bits).

Example:

- Binary: `001 010 110`
- Octal: `126`

3.1.5 General Numbering System Overview

All numbering systems can be categorized by their base, which determines the number of unique digits they use.

Base	Name	Common Uses
2	Binary	The foundation of digital computing.
8	Octal	Less common; used in older systems.
10	Decimal	The standard numbering system for humans.
16	Hexadecimal	Common in computing and electronics.

3.1.6 Why Different Numbering Systems?

Different numbering systems are used because:

1. Efficiency: Binary is efficient for computers but less readable for humans.
2. Readability: Hexadecimal and octal provide a more compact and human-readable format than binary.
3. Legacy Systems: Some older systems relied on octal or decimal representations.

3.2 Logic Gates

3.2.1 What is a Logic Gate?

A logic gate is a fundamental building block of digital circuits. It processes one or more input signals (binary values of 0 or 1) and produces a single binary output based on specific rules defined by its type. By combining them, you can create complex functions like adders, multipliers, and even entire CPUs.

3.2.2 Basic Logic Gates

Here are truth tables for some of the fundamental types of logic gates:

1. AND Gate: Outputs true only if both inputs are true.

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

2. OR Gate: Outputs true if at least one input is true.

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

3. NOT Gate: Inverts the input.

Input	Output
0	1
1	0

4. NAND Gate: NOT AND.

Input A	Input B	Output
0	0	1
0	1	1
1	0	1
1	1	0

5. XOR Gate: Exclusive OR.

Input A	Input B	Output (Y)
0	0	0
0	1	1
1	0	1
1	1	0

3.2.3 Real-World Analogy

Think of an AND gate as a car's ignition system, requiring both the key and the brake pedal to be pressed before starting the engine.

3.3 Digital Circuits

3.3.1 Building Blocks

Digital circuits are constructed using logic gates. For example:

- Half Adder: Adds two binary digits.

Input A	Input B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The construction of a half adder involves two fundamental logic gates:

1. XOR (Exclusive OR) Gate: This gate calculates the sum (S) of the two binary inputs (A and B). The XOR gate outputs 1 when the inputs are different and 0 when they are the same.
2. AND Gate: This gate determines the carry (C) output. It outputs 1 only if both inputs (A and B) are 1, indicating that a carry is generated to the next higher bit.

Thus, a half adder consists of one XOR gate for the sum and one AND gate for the carry.

The half-adder is essential because it forms the basis of more complex arithmetic circuits. Without the ability to add binary digits, a computer cannot perform basic arithmetic operations, which are fundamental to nearly all computations.

4 Computing

4.1 Introduction to Computing

4.1.1 What is Computing?

Computing refers to the process of using computers to perform tasks that range from simple calculations to complex problem-solving. It forms the backbone of modern technology, enabling everything from personal productivity to advanced scientific research.

4.1.2 Hardware and Software Components

At its core, a computer system comprises two main components:

- Hardware: This includes the physical devices that make up the computer, such as the central processing unit (CPU), memory, storage devices, and input/output peripherals like keyboards and monitors.
- Software: This refers to the programs and instructions that tell the hardware what tasks to perform. Software can be categorized into system software (like operating systems) and application software (such as word processors or web browsers).

4.1.3 Types of Computers

Computers come in various forms, each designed for specific purposes:

- Personal Computers (PCs): These are used by individuals for tasks like browsing the internet, writing documents, and gaming.
- Laptops/Notebooks: Portable versions of PCs, ideal for on-the-go use.
- Servers: Powerful computers that manage data and resources over networks, often used in web hosting or cloud services.

4.2 Central Processing Unit (CPU)

4.2.1 Introduction

- The CPU is often referred to as the "brain" of the computer.
- It processes instructions, performs calculations, and manages data flow between components.
- Understanding the CPU is fundamental to understanding how computers work.

4.2.2 Key Components of a CPU

1. Arithmetic Logic Unit (ALU)
    - Performs arithmetic operations (addition, subtraction, multiplication, division).

- Handles logical operations (AND, OR, NOT, etc.).

2. Control Unit (CU)
   - Manages the interpretation and execution of instructions.
   - Coordinates activities between hardware components.
3. Registers
   - Temporary storage locations for data and instructions.
   - Examples: Program Counter (PC), Instruction Register (IR), Accumulator.
4. Cache Memory
   - Small, fast memory used to store frequently accessed data or instructions.
   - Reduces the time needed to access data from main memory.
5. Clock Speed
   - The speed at which the CPU processes instructions (measured in Hz or GHz).
   - Higher clock speeds generally mean faster processing.

Clocks The speed of a clock is measured in Hertz (Hz), which quantifies how many cycles occur per second. Imagine it like the ticks of a clock—each tick represents one cycle. Here's a breakdown:

- 1 Hz (Hertz): One cycle per second.
- 1 kHz (KiloHertz): 1,000 Hz or 1,000 cycles per second.
- 1 MHz (MegaHertz): 1,000,000 Hz or one million cycles per second.
- 1 GHz (GigaHertz): 1,000,000,000 Hz or one billion cycles per second.

4.2.3 Functionality of the CPU

- Fetch Cycle: Retrieves instructions from memory.
- Decode Cycle: Interprets the retrieved instructions.
- Execute Cycle: Processes the instructions (e.g., arithmetic operations, data movement).
- Write Back Cycle: Stores results in memory or registers.

4.2.4 Types of CPUs

1. CISC (Complex Instruction Set Computing)
   - Uses complex instructions that take more clock cycles to execute.
   - Example: Early Intel x86 processors.
2. RISC (Reduced Instruction Set Computing)
   - Uses simpler, faster instructions.
   - Example: ARM processors found in many mobile devices and embedded systems.

4.2.5 Performance Metrics

- Clock Speed: Hz, KHz, MHz, GHz (gigahertz).

- Cores: Multiple processing units to handle tasks simultaneously.
- Cache Size: Larger cache improves performance.
- Thermal Design Power (TDP): Measures power consumption, critical for long-term durability and energy efficiency in timeless computing devices.

4.3 Memory: The Foundation of Computing

Memory is a critical component in any computing system, often likened to the human brain's short-term memory. While the CPU acts as the "brain" processing information, memory serves as the workspace where data and instructions are temporarily stored and retrieved.

4.3.1 Types of Memory

Volatile Memory Volatile memory is a type of storage that retains data only as long as the device has power. Once the power is turned off, all data stored in volatile memory is lost.

- Examples: RAM (Random Access Memory) used in computers and smartphones.
- Speed: Generally faster than non-volatile memory, making it ideal for temporary data storage during operations.

Non-Volatile Memory Non-volatile memory retains data even after the power supply is interrupted. This makes it suitable for long-term storage.

- Examples: Flash drives (SSD), Hard Disk Drives (HDD), ROM (Read-Only Memory).
- Limitations: Number of writes may be limited.

4.3.2 Functions of Memory

1. Storage: Holds files and programs when they're not in use.
2. Caching: Speeds up access to frequently used data by storing it temporarily.
3. Program Execution: Applications are loaded into RAM for execution.
4. Data Processing: Temporary storage of data being processed.

4.3.3 Considerations for Timeless Computing

Reliability

- Timeless computing emphasizes durable memory solutions, such as EEPROM or emerging technologies like ferroelectric RAM, to ensure long-term data retention without moving parts.

Durability

- Memory modules should withstand harsh environments, crucial for systems expected to function reliably over decades.

Upgradeability

- Users should be able to replace or expand memory, ensuring the system remains adaptable over time.

4.4 Peripherals

4.4.1 Introduction to Peripherals

Peripherals are essential components that enable interaction between a computer system and the outside world. They serve as the interface through which users input data and receive output, making them crucial for the functionality of any computing device.

4.4.2 Categories of Peripherals

Input Devices Input devices are responsible for capturing data from the user or external sources and transmitting it to the computer. Common examples include:

- Keyboard: Allows users to input text and commands.
- Mouse: Provides pointing and navigation capabilities.
- Touchscreen: Combines input and output functions in devices like tablets or all-in-one PCs.
- Joystick/Controller: Used for gaming or controlling specific applications.

Output Devices Output devices display or convey information processed by the computer to the user. Examples include:

- Monitor: Displays visual data, such as text, images, and videos.
- Printer: Produces physical copies of documents or graphics.
- Speakers/Headphones: Deliver audio output for users.

Storage Devices Storage devices are used to store data permanently on a computer. They can be internal or external:

- Hard Disk Drive (HDD): Magnetic storage with spinning disks.
- Solid State Drive (SSD): Flash-based storage offering faster access times.
- External Drives: Portable storage solutions like USB drives or external HDDs/SSDs.

Human Interface Devices (HIDs) Human interface devices focus on enhancing user interaction:

- Touchpad: Used for cursor control, often found on laptops.
- Trackball: A stationary pointing device, ideal for environments where a mouse might be impractical.
- Biometric Scanners: Devices like fingerprint readers or facial recognition systems for secure access.

4.4.3 The Importance of Peripherals in Computing

Peripherals bridge the gap between the computer's processing capabilities and human interaction. Without them, users would find it challenging to interact with the system effectively. For instance:

- Input devices allow users to provide commands and data.
- Output devices enable the system to communicate results or feedback.
- Storage devices ensure that data is preserved for future use.

4.4.4 Peripheral Communication

Peripherals typically connect to a computer via hardware interfaces such as USB (Universal Serial Bus), HDMI, or PCIe. These interfaces facilitate data transfer between the peripheral and the main system, ensuring smooth operation.

5 Programming

5.1 Introduction to Programming

5.1.1 What is Programming?

Programming involves creating a set of instructions (code) that directs computers or other machines to perform specific tasks. It serves as the foundation for software development, enabling everything from simple scripts to complex applications.

5.1.2 Why Learn Programming?

Learning programming remains essential in the age of AI for several reasons:

1. **Deeper Understanding**: Programming offers insights into how technology operates, enabling better decision-making and troubleshooting beyond surface-level interactions with AI tools.
2. **Debugging and Refinement**: While LLMs can generate code, they may introduce errors or suboptimal solutions. Programming skills allow for effective debugging and optimization of AI-generated outputs.
3. **Creative Problem-Solving**: Understanding programming fosters innovative thinking beyond existing patterns, encouraging unique solutions that AI models might not consider.
4. **Foundation for Innovation**: Coding provides a base for tackling advanced tasks and projects where AI tools may fall short, opening doors to more complex and varied endeavors.
5. **Self-Reliance**: In environments without internet access or LLMs, programming skills ensure continued productivity and project execution without dependency on external services.

In summary, while LLMs are powerful tools, learning programming enhances technical proficiency, improves collaboration with AI, encourages creativity, builds a foundation for advanced tasks, and ensures self-sufficiency.

5.2 Machine Code

5.2.1 Turing Machines and Their Relevance to Modern Computing

Before diving into machine code, it's essential to understand the theoretical underpinnings of computing. The concept of a Turing machine, introduced by mathematician Alan Turing in 1936, forms the foundation of modern computer science. While abstract, this model helps us comprehend what it means for a machine to compute.

A **Turing machine** consists of:

- An infinite strip of tape divided into cells, each capable of holding a symbol (like '0' or '1').
- A read/write head that moves along the tape, reading symbols and writing new ones.
- A set of rules governing the head's actions based on the current state and the symbol it reads.

The machine operates in discrete steps: 1. The head reads the current cell's symbol. 2. Based on its current state and the symbol, the machine follows a rule (e.g., write a symbol, move left/right). 3. The state changes to reflect the operation performed.

Despite their simplicity, Turing machines can simulate any algorithm, making them incredibly powerful. Modern computers operate based on similar principles—processing data and following instructions step-by-step.

Understanding Turing machines provides insight into why certain problems are solvable by computers and how computational complexity impacts performance. This theoretical framework bridges the gap between abstract concepts and practical implementation, reinforcing why machine code remains relevant in timeless computing applications.

5.2.2 Binary Representation

Machine code is the most basic form of programming, consisting of binary instructions (0s and 1s) directly executable by a computer's CPU. It is challenging for humans to read and write due to its low level of abstraction.

Example in Binary:

```
01001001 00100001 01001001
```

5.3 Assembly

5.3.1 Low-Level Programming

Assembly language provides a symbolic representation of machine code, making it easier for humans to understand and write. It offers direct control over the hardware but requires a deep understanding of the computer's architecture.

Example in Assembly (x86):

```
MOV AX, 1234
ADD BX, AX
```

5.4 C

5.4.1 Introduction to High-Level Programming

C is a high-level programming language known for its efficiency and versatility. It offers more abstraction than assembly while still allowing low-level hardware

manipulation.

Example in C:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

5.5 Abstract Data Types (ADTs)

5.5.1 Strings

Strings are sequences of characters used to represent text.

Example in C:

```c
char str[] = "Hello";
```

5.5.2 Lists

Lists allow dynamic storage and manipulation of data elements.

Example in C:

```c
int list[] = {1, 2, 3};
```

5.6 Interpreters

5.6.1 Overview

Interpreters execute code directly without prior compilation. They are often used for scripting languages like Python, JavaScript or Lisp.

5.6.2 Lisp

Lisp is a family of programming languages known for their use in artificial intelligence and complex applications.

Example in Common Lisp:

```lisp
(defun factorial(n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
```

5.7 Algorithms

5.7.1 The Foundation of Problem Solving

Algorithms are step-by-step procedures for solving problems or performing tasks. They form the core of programming and software development.

Example Sorting Algorithm (Bubble Sort):

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

6 FPGAs

6.1 Introduction to FPGAs

6.1.1 What is an FPGA?

An FPGA (Field-Programmable Gate Array) is a type of integrated circuit that can be configured by a user after manufacturing to perform specific tasks. Unlike CPUs or GPUs, which are designed for general-purpose computing, FPGAs are highly customizable and can be reprogrammed to suit various applications.

6.1.2 Why Choose FPGAs?

FPGAs offer several advantages:

- Flexibility: Reprogrammable for different tasks.
- Parallel Processing: Capable of handling multiple tasks simultaneously.
- Efficiency: Lower power consumption compared to traditional CPUs for specific tasks.

6.1.3 Applications of FPGAs

FPGAs are used in diverse fields, including:

- Digital Signal Processing
- Networking (e.g., packet processing)
- Automotive systems
- Defense systems
- Medical equipment

6.1.4 Comparing FPGAs and ASICs

ASICs (Application-Specific Integrated Circuits) are custom-designed chips built for specific tasks, optimized for performance and efficiency in those particular functions.

Key Differences

1. **Flexibility**:
   - **FPGAs**: Highly flexible; can be reprogrammed to adapt to different tasks or evolving needs.
   - **ASICs**: Fixed functionality once manufactured; not designed for changes after production.
2. **Performance**:
   - **FPGAs**: Slower due to their programmable nature, but versatile enough for multiple tasks.
   - **ASICs**: Faster and more efficient as they are optimized for their specific task, often outperforming FPGAs in that area.
3. **Cost and Design Time**:

- **FPGAs**: Cheaper and quicker to develop using pre-made chips and software tools, ideal for prototyping and varied applications.
- **ASICs**: More expensive and time-consuming due to custom hardware design and manufacturing (silicon fabrication), but costs are amortized over high volumes.

4. **Power Consumption**:
   - **FPGAs**: Sometimes use more power due to their programmable components.
   - **ASICs**: Optimized for efficiency in their specific task, often using less power effectively.
5. **Use Cases**:
   - **FPGAs**: Ideal for flexible applications like research, prototyping, or dynamic environments where requirements change.
   - **ASICs**: Best suited for high-volume production with fixed needs, such as specialized tasks in networking or video processing.

6.2 Basic Layout of a Simple FPGA

An FPGA is an integrated circuit that can be configured by the user after manufacturing to perform specific tasks. Unlike CPUs or GPUs, which are designed for general-purpose computing, FPGAs are highly customizable and can be reprogrammed to suit various applications.

Here's a basic overview of the key components in an FPGA:

1. **Configurable Logic Blocks (CLBs)**: These are the fundamental building blocks of an FPGA. CLBs contain Look-Up Tables (LUTs), which can be programmed to implement any logical function, such as AND gates, OR gates, or more complex functions.

2. **Flip-Flops**: These are used for storing data and creating timing delays in the circuit.

3. **Block RAMs (BRAMs)**: These are small blocks of memory that can be used for storage within the FPGA.

4. **Multipliers**: These are specialized hardwired circuits for performing multiplication operations, which are essential for arithmetic-intensive applications like digital signal processing or cryptography.

5. **I/O Blocks**: These are input/output blocks that allow the FPGA to connect to external devices, such as LEDs, buttons, or other microcontrollers.

6. **Interconnects**: These are the internal connections that route data between different parts of the FPGA. They can be programmed to create custom pathways for signals.

6.2.1 What Are Look-Up Tables (LUTs)?

A Look-Up Table (LUT) is a memory component within an FPGA's Configurable Logic Block (CLB). LUTs are used to implement logical functions by storing the truth tables of those functions. For example:

- A 4-LUT can store 16 possible input combinations (since (2^4 = 16)) and their corresponding outputs.
- When an input is applied to the LUT, it looks up the stored truth table to determine the correct output.

LUTs are highly flexible because they can be reprogrammed to implement any logical function. For instance:

```
// Truth table for a 2-LUT implementing an XOR gate:
input | output
0     | 1
1     | 0

// Verilog code for the same functionality:
output = (input == 1) ? 0 : 1;
```

6.2.2 How Are LUTs Programmed?

LUTs are programmed by configuring their stored truth tables. This is done through a configuration file that defines how each CLB should be set up. The configuration file is generated using hardware description languages like Verilog or VHDL, and it specifies the desired logical operations for each part of the FPGA.

The programming process involves:

1. **Designing the Circuit**: Using a hardware description language (e.g., Verilog), you define how the circuit should behave.
2. **Synthesis**: Tools like Yosys convert the high-level Verilog code into a netlist, which describes the logical connections between components.
3. **Place and Route**: Nextpnr or other tools map the logical circuits to the physical layout of the FPGA, determining where each component will be placed and how signals will route through the interconnects.
4. **Configuration File Generation**: The final configuration file (often in `.bit` or `.sof` format) is generated, which contains the specific instructions for configuring each LUT and other components within the FPGA.
5. **Programming the FPGA**: The configuration file is loaded into the FPGA's memory using a programmer (e.g., a JTAG cable).

6.2.3 Summary

FPGAs are highly flexible and programmable integrated circuits that can be configured to perform a wide variety of tasks. LUTs are the core components within FPGAs that store truth tables for logical functions, and they are programmed by generating configuration files in hardware description languages like Verilog or VHDL. By configuring these components, users can create custom digital circuits tailored to their specific needs.

6.2.4 Tools

HDL Synthesis An HDL (Hardware Description Language) synthesis tool processes a high-level descriptive code written in languages like Verilog or VHDL. This tool converts the abstract behavioral description of a digital circuit into a lower-level representation, such as a netlist, which defines the logical gates and connections required to implement the design on an FPGA. The synthesis tool optimizes the design for factors like area, speed, and power consumption, ensuring efficient use of the FPGA's resources.

Place-and-Route After the HDL synthesis, the place-and-route tool takes the generated netlist and maps it onto the physical structure of the FPGA. This involves two main steps:

1. **Placing:** The tool determines the optimal positions for each component (like LUTs and flip-flops) on the FPGA chip to minimize delays and maximize efficiency.
2. **Routing:** It then routes the connections between these components, ensuring that all necessary signals can travel without interference or excessive delay. This step is crucial for managing the complex web of interconnections within the FPGA.

Together, these tools transform a high-level design into a physically realizable configuration on an FPGA, enabling the implementation of custom digital circuits.

6.3 Verilog - Hardware Description Language

6.3.1 Introduction to Verilog

Verilog is a hardware description language (HDL) used to describe digital circuits. It allows you to design and simulate hardware components, such as logic gates and microprocessors.

6.3.2 Simple Verilog Example

In this example we will blink an LED (Light Emitting Diode).

```
// LED blinking circuit
module blinky (
    input   clock_10hz,
    output  LED
);

reg led = 0;
wire clock;

assign clock = clock_10hz;

always @(posedge clock) begin
    led = ~led;
end

assign LED = led;

endmodule
```

This code defines a module that toggles an LED on and off using a clock input.

6.4 Building the Blinky Circuit

6.4.1 Tools Needed

- An FPGA computer or FPGA development board.
- Yosys: A HDL synthesis tool for converting Verilog code into a hardware description.
- nextpnr: Maps the description onto the FPGA chip.

You will also need a bitstream database and tools for the FPGA you're using, for example:

- Project IceStorm (ICE40)
- Project Trellis (ECP5)
- Project X-Ray (XC7)
- Project Peppercorn (CCGM1)

6.4.2 Building and Programming the Blinky Circuit

Here's how you might program an FPGA to blink an LED:

1. **Define the Circuit**:
    - As we did above, use Verilog to describe a simple oscillator circuit that toggles an output signal at regular intervals.
2. **Synthesize and Configure**:
    - Convert the Verilog code into a configuration file using synthesis tools.

3. **Program the FPGA**:
    - Load the configuration file onto the FPGA board using programming software.
4. **Test the Circuit**:
    - Connect an LED to the specified output pin and power on the board. The LED should start blinking according to the programmed oscillator frequency.

7 System-On-Chip (SOC)

7.1 What is a SOC?

A System-on-Chip (or System-on-a-Chip) integrates various components, like a CPU, memory, and peripherals, onto a single chip. FPGAs allow creating custom SOCs tailored to specific applications, as well as general-purpose applications, such as computing.

7.1.1 Comparing ASIC SOCs and FPGA SOCs

- ASIC SOCs: General-purpose with fixed architectures and fixed chipsets.
- FPGA SOCs: "Field programmable" and customizable for specific tasks, offering better efficiency and performance for specialized workloads. Can be used for general-purpose computing with a reconfigurable chipset.

7.1.2 Understanding Softcore CPUs

Softcore CPUs are implemented in gateware within an FPGA, contrasting with hardcore CPUs that are physically embedded. This flexibility allows developers to tailor the CPU's architecture to specific needs, offering customization and adaptability. Examples include RISC-V and OpenRISC projects, which highlight the range of possibilities for softcore implementations.

7.1.3 Benefits of Softcore CPUs

1. **Flexibility**: Customize the CPU to meet unique project requirements.
2. **Cost-Effectiveness**: Utilize existing FPGA resources without additional hardware costs.
3. **Longevity**: As FPGAs can be reconfigured, softcore CPUs ensure up-to-date functionality over time.

7.1.4 Challenges of Softcore CPUs

1. **Performance Constraints**: Generally slower than ASICs due to implementation on the FPGA fabric.

7.2 FPGA Computing

FPGAs can implement different computer architectures. By configuring the FPGA's logic blocks, you can create a virtual version of any CPU or computer system. For example:

- **NES (Nintendo Entertainment System):** The 6502 CPU used in the NES can be emulated by programming specific logic blocks within the FPGA to replicate its behavior.
- **Apple II:** Similarly, the Apple II's architecture can be recreated, allowing you to run classic Apple II software on an FPGA-based system.

- **Amiga:** The Motorola 68000 CPU used in Amiga systems can also be emulated, preserving the classic Amiga experience.
- **RISC-V:** This modern instruction set can be implemented in an FPGA, enabling a new system that leverages RISC-V's efficiency and scalability and providing an environment for running Linux.

7.2.1 Practical Implications

- **Retrocomputing:** FPGAs allow enthusiasts to run old games and software on modern hardware, preserving computing history.
- **Custom Hardware Development:** Beyond existing systems, FPGAs enable the creation of custom architectures tailored for specific tasks, such as enhancing privacy or security in timeless computing applications.

7.2.2 Running Linux on FPGA SOCs

Linux, known for its adaptability, has been ported to various architectures, including those supported by softcore CPUs like RISC-V.

7.2.3 Kakao Linux

Kakao Linux is a Linux distribution designed specifically for FPGA-based computers. Kakao depends on a SOC framework named LiteX to generate a Linux-capable SOC.

7.2.4 Future Directions

The landscape of FPGA SOCs is continually evolving, offering exciting opportunities for innovation. As technology advances, we can expect improvements in performance and expanded applications, further solidifying the potential of these systems.

Kakao Linux provides a minimal yet functional environment, focusing on timeless applications such as reading, writing, math, education, organization, communication, and automation. Its simplicity ensures that it remains lightweight and efficient, making it an ideal choice for users who want to experiment with Linux on FPGA hardware without the overhead of more complex distributions.

8 The Zeitlos SOC/OS

Zeitlos is an open-source project designed to bring Timeless Computing to life. It consists of two key components:

- **Zeitlos SOC (System-on-a-Chip)**: A hardware design that integrates a RISC-V CPU, GPU, and other essential subsystems onto a single FPGA.
- **Zeitlos OS (Operating System)**: A lightweight, open-source operating system tailored for timeless applications.

This combination creates a powerful yet simple ecosystem for users of all skill levels.

8.1 Motivation

8.1.1 Simplicity and Minimalism

- **Linux** is flexible and can be adapted to many use cases, but it also comes with a lot of complexity. The sheer size and scope of Linux (and the software that runs on it) can make it harder to understand, maintain, and secure.
- **Zeitlos** is designed from the ground up as a minimal, single-user operating system tailored for timeless applications. Its simplicity makes it easier to understand, audit, and control—perfect for users who want a distraction-free environment focused on specific tasks like reading, writing, or education.

8.1.2 Customization for Timeless Computing

- Linux is designed to be a general-purpose OS that can run a wide variety of applications. While this is great for versatility, it often introduces unnecessary complexity and resource overhead for users who only need basic functionality.
- Zeitlos is built specifically for FPGA-based computers and their unique strengths (like reconfigurable hardware). It's optimized for timeless computing use cases, such as:
  - Running lightweight, distraction-free applications.
  - Supporting retro or custom computing environments.
  - Providing a stable, secure platform that's resistant to bloatware or unnecessary updates.

8.1.3 Longevity and Future-Proofing

- Linux (and most modern operating systems) are designed with the assumption of constant evolution and improvement. While this is great for innovation, it can also lead to compatibility issues over time as hardware and software evolve.

- Zeitlos aims to create a system that's not just modern but also timeless—designed to remain relevant and functional for decades. Its modular design allows for incremental improvements while maintaining backward compatibility with classic computing paradigms.

8.1.4 Security Through Simplicity

- Modern Linux distributions are secure, but they're also complex. Complexity introduces potential vulnerabilities and makes it harder to audit the system for security flaws.
- Zeitlos is designed with a focus on simplicity and minimalism, which reduces the attack surface and makes the system easier to secure. Its single-user design (with optional multitasking) also eliminates many of the risks associated with multi-user systems.

8.1.5 Hardware-Aware Design

- Linux is hardware-agnostic, which means it can run on a wide variety of devices. However, this flexibility often requires compromises in performance or resource usage.
- Zeitlos is built specifically for FPGA-based hardware and takes full advantage of the unique capabilities of these systems (e.g., reconfigurable logic, low-power operation). This hardware-aware design allows for optimizations that wouldn't be possible with a one-size-fits-all OS like Linux.

8.1.6 Retrocomputing and Customization

- While Linux can run emulators and support classic computing environments, it often does so in a way that feels layered and indirect.
- Zeitlos aims to recreate the magic of retro systems (like the Amiga or Macintosh) while adding modern capabilities. Its design borrows from the best ideas of classic computers, making it feel familiar and approachable for users who love retrocomputing or want to experiment with custom hardware.

8.1.7 Long-Term Vision

- Linux is a fantastic choice for general-purpose computing, but it's not designed with the same long-term goals as Machdyne. Our goal is to create systems that will remain useful and relevant for decades—even in scenarios where modern technology might fail or become inaccessible (e.g., in a survival computing context).
- Zeitlos is part of this vision, offering a lightweight, secure, and customizable platform that can evolve alongside timeless hardware.

8.1.8 Conclusion

Linux (and Kakao Linux) will continue to play a role in our ecosystem for users who want the best of both worlds: the simplicity of FPGA hardware paired with the power of a modern OS.

Zeitlos isn't meant to replace Linux or other operating systems; it's designed to fill a specific niche. For users who want a simple, secure, and long-term-focused computing experience on FPGA-based hardware, Zeitlos offers something that Linux just can't match in terms of customization, minimalism, and hardware integration.

8.2 Design Goals

The Zeitlos SOC/OS project was designed with a specific set of goals in mind to create a system that is both functional and enduring. These goals guide the development process and ensure that the resulting system meets the needs of users who value simplicity, security, and longevity.

8.2.1 Minimalistic Design

- **Simplicity**: The primary goal is to create a simple and understandable operating system that minimizes complexity while still being practical for everyday use.
- **Hardware Integration**: Zeitlos is designed to work seamlessly with FPGA hardware, leveraging the unique capabilities of these devices to provide a responsive and efficient computing environment.

8.2.2 Hardware-Accelerated Computing

- **FPGA Utilization**: By utilizing FPGAs, Zeitlos aims to deliver high performance without the need for traditional CPUs or GPUs. This approach allows for hardware-accelerated tasks, which can be particularly beneficial for specific applications.
- **Custom GPU**: The inclusion of a custom 2D "GPU" with sprite support and multiple video modes ensures that the system can handle graphical operations efficiently.

8.2.3 Multitasking and Resource Management

- **Preemptive Multitasking**: The OS kernel is designed to support preemptive multitasking, allowing for smooth operation of multiple tasks without the overhead of virtual memory or an MMU. This makes the system lightweight yet capable.
- **Kernel Language**: The kernel is written in C, ensuring compatibility and ease of understanding, while applications can be developed in both C and Scheme (a Lisp dialect), offering flexibility for different use cases.

8.2.4 User-Centric Design

- **Ease of Use**: Despite its technical underpinnings, Zeitlos is designed to be user-friendly. The interface is intuitive, with a focus on usability that makes it accessible even to those who are not deeply familiar with computing systems.
- **Support for Applications**: The system supports both new applications tailored to its environment and existing ones through emulation or additional hardware resources, ensuring a broad range of functionality.

8.2.5 Security and Longevity

- **Security**: By focusing on simplicity and minimizing unnecessary features, Zeitlos inherently reduces the attack surface, making it more secure compared to complex modern operating systems.
- **Enduring Relevance**: The design aims for long-term relevance, ensuring that the system remains functional and useful even as technology evolves. This is crucial for applications where longevity and reliability are paramount.

8.2.6 Modularity and Extensibility

- **Modular Design**: The system is built with modularity in mind, allowing for future expansion and adaptation to new hardware or requirements without compromising existing functionality.
- **Upgradability**: Users should be able to upgrade or replace components as needed, ensuring that the system can evolve alongside technological advancements.

8.2.7 Open Source Philosophy

- **Transparency**: By being open-source, Zeitlos fosters transparency and trust. This allows users and developers to review the code, contribute improvements, and ensure the system remains secure and reliable.
- **Collaboration**: The open-source nature encourages collaboration within the community, allowing for a diverse range of contributions that can enhance the system over time.

8.2.8 User Experience

- **Graphical Interface**: A mouse and keyboard-based interface provide an intuitive way to interact with the system, making it accessible even to those who are not deeply technical.
- **Documentation**: Comprehensive documentation, including this book, will be integrated into the OS, ensuring that users can easily understand how to use and maintain their systems.

8.2.9 Sustainability and Accessibility

- **Low Power Consumption**: The design considerations include low power consumption, making the system suitable for environments where energy resources may be limited or where sustainability is a priority.
- **Accessibility**: The system aims to be accessible to a wide range of users, including those in regions with limited access to modern technology, ensuring that computing remains affordable and practical.

8.2.10 Future-Proofing

- **Adaptability**: The modular design ensures that the system can adapt to future hardware advancements while maintaining its core functionality. This adaptability is crucial for ensuring that the system remains relevant over the long term.
- **Preservation of Usefulness**: By focusing on timeless applications, Zeitlos aims to preserve its usefulness even in scenarios where modern computing infrastructure may be unavailable or compromised.

8.2.11 Community and Collaboration

- **Community Support**: The open-source model encourages a community-driven approach to development, ensuring that the system benefits from a wide range of perspectives and expertise.
- **Feedback Integration**: The design process incorporates feedback from users and contributors, allowing for continuous improvement and refinement of the system.

8.2.12 Summary

In summary, the Zeitlos SOC/OS is designed with a focus on simplicity, security, and longevity, aiming to provide a computing environment that is both functional today and enduring into the future. Its modular and open-source nature ensures that it can evolve alongside technological advancements while remaining accessible to a wide range of users.

8.3 System Overview

The Zeitlos system is built around the following core components:

1. RISC-V CPU: The heart of the system, designed for efficiency and long-term reliability.
2. Custom GPU: A hardware-accelerated "blitter" for fast graphics rendering, supporting multiple resolutions and modes.
3. FPGA-Based Hardware: Compatible with a variety of FPGA boards, ensuring flexibility and adaptability.

8.4 The Zeitlos Kernel

The kernel is the core of the operating system, written in C for reliability and performance. It provides essential services like:

- Process management
- Memory allocation
- Device drivers
- Interrupt handling
- Message handling

Zeitlos uses a preemptive multitasking model to handle multiple tasks efficiently while keeping resource usage low.

8.5 User-Friendly Design

The Zeitlos OS is designed with the end-user in mind:

- Graphical Interface: A clean, intuitive interface for interacting with applications.
- Built-in Tools: Pre-installed utilities for system management and development.
- Scripting Support: Built-in Scheme interpreter for advanced users.

8.6 Getting Started

To use Zeitlos, you'll need:

1. An FPGA board (e.g., Machdyne FPGA computers or a development board).
2. The latest version of the Zeitlos OS.

8.7 Get Involved

Zeitlos is under active development. Find more information at:

https://zeitlos.org

and in the GitHub repo:

https://github.com/machdyne/zeitlos.

9 Survival Computing

9.1 Introduction to Survival Computing

Survival computing refers to the practice of designing and utilizing computer systems that can function effectively in scenarios where modern technology, resources, or infrastructure may be scarce or unavailable. This concept emphasizes resilience, adaptability, and sustainability, ensuring access to computational capabilities even under adverse conditions.

9.1.1 Why Survival Computing Matters

In an era marked by rapid technological advancement and potential vulnerabilities such as natural disasters, conflicts, economic downturns, and AI, the reliance on modern computing systems can become a critical weakness. Survival computing addresses these challenges by focusing on systems that are not only robust but also designed to endure over extended periods, making them invaluable in both futuristic dystopian scenarios and present-day remote regions where technology access is limited.

9.1.2 The Role of Timeless Computing

Timeless computing, as championed by Machdyne, aligns closely with the principles of survival computing. By creating systems that remain functional and relevant for decades or even centuries, timeless computing ensures a sustainable technological foundation. These systems are designed to be understandable, repairable, and adaptable, qualities that are crucial for survival in environments where technical support and modern infrastructure may be absent.

9.2 Key Principles of Survival Computing

1. **Simplicity**: Complex systems are more prone to failure and harder to maintain. Survival computing favors simplicity in design and operation, ensuring ease of use and reliability.
2. **Resilience**: The ability to withstand and continue functioning despite environmental challenges, hardware failures, or resource scarcity is paramount. This resilience is achieved through durable hardware and minimal reliance on disposable components.
3. **Sustainability**: Emphasizing the use of energy-efficient technologies and sustainable power sources ensures that computational capabilities can be maintained over long periods without depleting finite resources.

9.3 Real-World Applications

9.3.1 Off-Grid Living

In remote areas where access to electricity is limited, FPGA computers can be powered using solar panels or wind turbines. Their low power requirements make them ideal for off-grid setups, ensuring continuous operation without reliance on traditional infrastructure.

9.3.2 Disaster Preparedness

During emergencies, when communication networks and utilities may be disrupted, having a functional computing system can be crucial for communication, data storage, and automation tasks. FPGA computers, with their resilience and adaptability, provide a reliable solution in such scenarios.

9.4 Future-Proofing Your Computing Needs

Survival computing is not just about preparing for worst-case scenarios; it's about designing systems that can endure and remain useful over time. By adopting principles of timeless computing, you ensure that your technological and educational investments will yield returns long into the future, regardless of how the broader technological landscape evolves.

9.4.1 Conclusion

In a world where technology evolves at an unprecedented pace, survival computing offers a beacon of stability and reliability. By embracing FPGA computers, users can build robust, adaptable, and enduring computational solutions that meet the challenges of today while preparing for tomorrow. Whether you're a novice seeking to understand the basics or a developer looking to enhance existing systems, the principles of survival computing provide a foundation for creating resilient and sustainable technological solutions.