

## Article

# A Cross-Platform Toolchain for Migrating Software to an OpenRISC-Based FPGA SoC

Roland Szabo 

Faculty of Electronics, Telecommunications and Information Technologies, Politehnica University Timisoara, Vasile Parvan Av., No. 2, 300223 Timisoara, Romania; roland.szabo@upt.ro; Tel.: +40-256-40-3351

## Abstract

This paper describes the development of several software-based games using a high-level programming language (C in our case), designed so that they can be ported to a Field-Programmable Gate Array (FPGA). It also outlines the mathematical foundations underlying these games. Making executables portable in this way can simplify running applications on FPGA platforms. Porting a game to an FPGA serves as evidence that arbitrary executables can be migrated to such hardware. The complete workflow for creating the game, along with the final game outcomes, is detailed in this paper. In addition, statistical analyses of these games were conducted. The proposed approach relies on graphics and character-handling libraries typically available in advanced programming languages. The background of this work is that a microcontroller architecture which can easily be run on a Spartan-6 FPGA was needed. The innovative point of this paper is that it created the cross-compilation toolchain on an uncommon microcontroller architecture, like the OpenRISC.

**Keywords:** artificial intelligence; character creation; field-programmable gate arrays; operating system design; software development environments; serious gaming applications

## 1. Introduction

Hardware designers can now create specialized computing platforms that are optimized for particular performance power or security constraints thanks to the development of custom microprocessor architectures on Field-Programmable Gate Arrays (FPGAs). Nevertheless the software's accessibility frequently restricts the usefulness of such custom architectures. Developing a retargetable cross-compilation toolchain is one way that this paper attempts to solve the crucial problem of software enablement for a new non-standard FPGA-based processor. To prove this unconventional architecture's feasibility and usefulness, it is essential to be able to transfer current applications and games to it. The goal of this work is to validate the architecture's potential for practical deployment by bridging the gap between custom hardware implementation and a working software ecosystem through the adaptation of the GNU (GNU's Not Unix!) Compiler Collection (GCC) and related binutils.

This paper describes the development of several games within a software environment that use a high-level programming language [1]. These games are designed so that they can be easily transferred to a Field-Programmable Gate Array (FPGA) on which a microcontroller architecture has been implemented [2–4].

This paper further examines the mathematical foundations of these games and concludes with a statistical analysis of them.



Academic Editor: Chen Yang

Received: 21 January 2026

Revised: 19 February 2026

Accepted: 26 February 2026

Published: 3 March 2026

**Copyright:** © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons](https://creativecommons.org/licenses/by/4.0/)

[Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

The primary aim of this paper is to develop a cross-compilation toolchain for the FPGA, enabling any application to be ported to it.

The purpose of this project was to remake a selection of classic computer games [5]. One of the first video games was a Pong-style game developed by William Higinbotham in 1958 on an oscilloscope, originally titled Tennis for Two [6]. During that period, he created only two paddles (for Tennis for Two) and represented the ball with a simple spot. His creation became one of the first computer games and significantly influenced the lives of people in the modern era [7]. Following their invention, the major boom in computer games began, leading to the situation we see today. Today, computer games can be found on almost every device equipped with a screen and a microcontroller. They are available on PCs, laptops, tablets, mobile phones, and many other electronic devices [8].

The game industry developed rapidly following the widespread adoption of personal computers by users [9]. One of the contributions in this article is the addition of several games to the extensive catalog of computer games. The paper describes the stages through which the development of the computer games presented were planned and executed [10].

The conceptuality of this paper is to present the creation of three simple games which can then be ported on an uncommon microprocessor architecture created on an FPGA.

Project-based game development is an effective motivational tool and the pedagogical value of FPGAs in teaching complex digital design concepts is well established (Liu et al. (2018)) [11]. Higher-level specifications have replaced low-level hardware description languages as research has progressed methods for configuring these devices. Geier et al. (2019) demonstrated this in the Syntroids project which used Temporal Stream Logic to synthesize and formally specify an arcade-style game on an FPGA [12]. In addition Van Acker et al. (2017) demonstrated how the platform could create intricate dynamic virtual worlds through simulations of cellular automata [13]. Together these studies highlight how the FPGA has developed from a purely educational tool to a flexible platform that can support complex interactive virtual worlds.

Liu et al. (2018) report that, in academic contexts, integrating FPGA-based game development into coursework is an effective way to teach complex hardware concepts [11]. In one digital system design course, for example, students worked on FPGA projects to build games such as a maze and Tetris, which improved their grasp of hardware description languages and the overall design workflow [11].

Geier et al. (2019) reported that advances in hardware synthesis have enabled automatic generation of FPGA configurations directly from high-level specifications [12]. In order to illustrate the potential of formal methods for scalable and effective hardware design, the Syntroids project uses Temporal Stream Logic (TSL) to describe and synthesize a space shooter game on an FPGA [12].

Geier et al. (2019) added that the Syntroids project serves as an example of how to use Temporal Stream Logic (TSL) to formally specify and implement an FPGA space shooter game [12]. Their methodology demonstrates how formal methods can facilitate scalable and effective hardware design [12].

The Van Acker et al. (2017) studies have looked into using cellular automata simulations on FPGAs to create visual depictions of virtual worlds [13]. These simulations can be used to train neural networks in gaming contexts, underscoring the significant role of FPGAs in creating complex and dynamic virtual worlds [13].

This paper's contribution lies in implementing an OpenRISC (Open source Reduced Instruction Set Computer) microprocessor architecture on a Spartan-6 FPGA using the ATLYS board, enabling users to cross-compile and execute arbitrary applications. This is a way to create a SoC (System on a Chip) on an FPGA. The cross-compilation capability was validated using several games developed by the author. These games are relatively simple

and served solely as proof-of-concept examples to show that any application can be ported and cross-compiled on the proposed FPGA platform.

Unfortunately, only a limited number of studies specifically address deploying applications on FPGA-based microprocessors. The works identified in the literature are therefore considered the most closely related to the approach presented here.

Investigating the cross-compilation of applications and games for the OpenRISC platform on FPGAs advances the electronics domain by demonstrating how soft-core processors can be practically used in FPGA-based environments for embedded applications. It highlights the effective deployment of open-source RISC architectures on FPGAs, thereby bridging the adaptability of software development with the performance advantages of hardware.

Table 1 presents a comparative evaluation of the design in terms of speed and resource utilization.

**Table 1.** Comparative evaluation of the design in terms of speed and resource utilization.

| Feature                     | Liu et al. (2018) [11]                  | Geier et al. (2019) [12]           | Van Acker et al. (2017) [13]                       | Current Paper                            |
|-----------------------------|-----------------------------------------|------------------------------------|----------------------------------------------------|------------------------------------------|
| <b>Speed</b>                | Tailored for instant gaming experiences | Quick owing to immediate synthesis | Extremely rapid as a result of parallel processing | Intermediate range (100–200 MHz)         |
| <b>Parallelism</b>          | Medium (somewhat accelerated pace)      | High (custom logic)                | Very high (extensively parallel)                   | Low (executing in a sequential manner)   |
| <b>Resource Utilization</b> | Effective for gaming logic              | Game synthesis optimization        | Efficient for simulations on a large scale         | Significant (comparable to CPU overhead) |
| <b>Best Use Case</b>        | Game mechanics and visuals              | Game synthesis                     | Extensive simulations                              | General-purpose computation              |

Table 2 provides a comparison between this work and prior studies.

By facilitating more seamless software–hardware integration improving embedded gaming experiences and promoting effective use of computational resources, this work advances digital systems’ embedded computing and game development.

This original research on cross-compiling applications and games with OpenRISC on FPGA centers on adapting and enhancing software to run efficiently on an OpenRISC soft-core processor implemented on an FPGA. Building a cross-compilation toolchain for porting games and applications that already exist, optimizing the hardware–software interaction, and benchmarking performance evaluation are important elements. This work lays the groundwork for low-cost, open source platforms for embedded applications and games, thereby promoting broader adoption of FPGAs in educational, research, and real-time computing environments.

The trade-off of this method is that, while it offers flexibility and makes it straightforward to port an application to an FPGA using a microcontroller-style architecture on top of it, dedicated hardware logic written in VHDL/Verilog delivers superior performance and efficiency, since it runs entirely in hardware rather than software, and hardware is inherently faster. This entire strategy is aimed at simplifying application portability, so there is no need to rewrite every application in a hardware description language.

The games transferred to the microcontroller implemented on the FPGA platform will be: a Pong clone, a fighting game, and a chip placement game based on backtracking. All of these titles were entirely developed from the ground up by the author.

Table 3 provides a comparison of soft-core CPU architectures implemented on FPGA.

This paper focuses on the OpenRISC architecture on the ATLYS board with the Spartan-6 FPGA; this is a different architecture. It was done on a new architecture: the OpenRISC.

The OpenRISC was the architecture that had the possibility to be implemented easier on the Spartan-6 from the ATLYS board. OpenRISC is rarer, it is not so common; this way it was thought it would be interesting to implement.

**Table 2.** Prior work comparison.

| Characteristic            | Liu et al. (2018) [11]                                                                | Geier et al. (2019) [12]                                                              | Van Acker et al. (2017) [13]                                                                            | Current Paper                                                                                                                                  |
|---------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Focus Area</b>         | Utilizing FPGAs in the education of digital system design through gaming applications | The creation of games using FPGAs through temporal logic                              | Simulation of cellular automata using FPGAs for neural network training with virtual environment images | Cross compiling games and applications for the OpenRISC architecture created on the FPGA                                                       |
| <b>Application Domain</b> | Digital design course educational resource                                            | The automated creation of games                                                       | Training neural networks and AI within virtual settings                                                 | Porting games and applications to other architectures                                                                                          |
| <b>FPGA Use</b>           | A learning tool for students to understand hardware design                            | Produces game logic that is compatible with FPGAs                                     | Applies cellular automata models to FPGA for artificial intelligence training                           | Generating the OpenRISC microprocessor architecture                                                                                            |
| <b>Methodology</b>        | Incorporates FPGAs into coursework through interactive gaming projects                | Structured development of gameplay mechanics derived from formal logical descriptions | Utilizes cellular automata to mimic virtual environments for AI training                                | Using FPGA to create OpenRISC microprocessor to run Linux operating system                                                                     |
| <b>Key Contribution</b>   | Incorporating gaming projects within FPGA-driven educational experiences              | Illustrates the structured creation of a game through temporal logic                  | Simulation utilizing FPGAs to produce training datasets for neural networks                             | Creating OpenRISC microprocessor architecture from the Spartan-6 FPGA and cross-compiling personal games and applications to this architecture |
| <b>Strengths</b>          | Facilitates education by using participatory and captivating projects                 | Guarantees the accuracy of the developed game mechanics                               | AI training harnessing FPGA technology through real-time simulation                                     | Made an OpenRISC architecture and demonstrated that any application or game can be cross-compiled to this architecture                         |
| <b>Limitations</b>        | Restricted ability to scale outside of educational environments                       | The intricacies of constraints in formal logic                                        | Potential constraints may arise in applications involving large-scale neural networks                   | The source code of every application or game is needed in order to cross-compile it to the OpenRISC architecture                               |

This work is different from standard embedded-Linux-on-FPGA workflows, because it uses an uncommon FPGA architecture which is the OpenRISC architecture. The main contribution is educational, to show how to port applications on an uncommon microprocessor architecture.

The main contribution of this article is to create a cross-compilation toolchain on an uncommon microprocessor architecture, which is the OpenRISC.

The innovative point of the paper is that the cross-compilation toolchain was created on an uncommon microcontroller architecture, like the OpenRISC.

**Table 3.** Soft-core CPU architecture comparison on FPGA.

| Parameter             | OpenRISC  | RISC-V    | ARM Cortex-M | MicroBlaze  |
|-----------------------|-----------|-----------|--------------|-------------|
| ISA Type              | RISC      | RISC      | RISC         | RISC        |
| ISA License           | Open      | Open      | Proprietary  | Proprietary |
| Core License          | Open      | Open      | Proprietary  | Proprietary |
| Typical Freq. (MHz)   | 50–200    | 100–300   | 100–400      | 100–450     |
| Min. LUTs (Spartan-6) | 1500–2500 | 1200–2200 | 1800–3000    | 900–1800    |
| Min. FFs (Spartan-6)  | 1300–2200 | 1000–2000 | 1600–2800    | 800–1600    |
| BRAM (blocks)         | 4–8       | 3–8       | 6–12         | 2–6         |
| DSP Slices            | 0–4       | 0–8       | 4–16         | 0–8         |
| DMIPS/MHz             | 0.7–1.2   | 1.1–1.8   | 1.2–2.0      | 0.9–1.6     |
| CoreMark/MHz          | 2.1–3.2   | 3.1–4.5   | 3.4–4.8      | 2.8–4.2     |
| Pipeline Stages       | 5         | 3–5       | 3            | 3–8         |
| Cache Support         | Yes       | Yes       | Limited      | Yes         |
| MMU Support           | Yes       | Yes       | Limited      | Yes         |
| Debug Interface       | JTAG      | JTAG      | JTAG/SWD     | JTAG        |
| Toolchain             | GCC       | GCC/LLVM  | ARM GCC      | Vitis       |
| Ecosystem Maturity    | Medium    | High      | Very High    | High        |

Note: The values are approximations that rely on the target FPGA synthesis options and particular configuration. For uniformity, measurements are based on Spartan-6/LX45.

## 2. Problem Formulation

The assignment was to develop several software-based games. The game engines also encapsulate the computer’s intelligence, enabling the user to play against the computer, that is, against an AI. The games must also support multiplayer functionality so that users can play against each other. Building the entire system presented numerous challenges, which made the game development process particularly exciting. The games use a  $640 \times 480$  screen resolution, whereas on the FPGA we work with a VGA display. In this setup, the effective screen size is  $80 \times 25$  characters.

The ultimate aim of this paper is to develop a platform that enables porting or cross-compiling arbitrary applications to an FPGA-based Linux system running on the OpenRISC architecture.

## 3. Problem Solution

### 3.1. Mathematical Background of the Pong Game

The following section will present the mathematical model and simulation of a Pong game implemented on a microcontroller architecture realized on an FPGA [14–17]. By employing differential and integral equations, we are able to model the ball’s motion, the paddles’ movements, and their interactions. In order to calculate the total distance the ball travels during the game, this work will look at the routes taken by the paddles and the ball.

Let  $\vec{r}(t) = [x(t), y(t)]$  indicate the location of the ball at a given time  $t$ , and let  $\vec{v}(t) = [v_x(t), v_y(t)]$  represent its velocity vector. These are the equations that explain how the ball moves:

$$\vec{v}(t) = \frac{d\vec{r}(t)}{dt} \quad (1)$$

When the ball strikes the walls and paddles it undergoes elastic collisions that change the direction of its velocity as explained below:

$$v_x(t) = -v_x(t) \quad (\text{on horizontal walls}) \quad (2)$$

$$v_y(t) = -v_y(t) \quad (\text{on paddles or vertical walls}) \quad (3)$$

The velocity change when colliding with walls and paddles is specified by

$$v_x(t) = -v_x(t) \quad (\text{on right or left walls}) \quad (4)$$

$$v_y(t) = -v_y(t) \quad (\text{on top or bottom walls}) \quad (5)$$

Let  $y_{\text{paddle}}(t)$  denote the position of the paddle at time  $t$ . The following formula can be used to describe the paddle's motion if it moves at a constant speed:

$$v_{\text{paddle}} = \frac{dy_{\text{paddle}}(t)}{dt} \quad (6)$$

If the paddle undergoes uniform acceleration  $a_{\text{paddle}}$ , then

$$a_{\text{paddle}} = \frac{d^2 y_{\text{paddle}}(t)}{dt^2} \quad (7)$$

By integrating the ball's speed we can find the total distance it has traveled for a duration of  $t$ :

$$\text{Distance}(t) = \int_0^t \|\vec{v}(\tau)\| d\tau \quad (8)$$

where  $\|\vec{v}(\tau)\|$  represents the magnitude of the velocity vector.

The following equations control the ball's dynamics:

$$\begin{aligned} v_x(t) &= \frac{dx(t)}{dt}, \\ v_y(t) &= \frac{dy(t)}{dt}, \\ F_{\text{collision}_x}(t) &= \frac{dv_x(t)}{dt}, \quad F_{\text{collision}_x}(t) = \text{Force applied during x-axis collision}, \\ F_{\text{collision}_y}(t) &= \frac{dv_y(t)}{dt}, \quad F_{\text{collision}_y}(t) = \text{Force applied during y-axis collision}. \end{aligned} \quad (9)$$

We describe the motion of the ball in a Pong game using this set of differential equations. The ball travels inside the lines set by the paddle and the walls and each time it strikes one of them its speed varies.

Figure 1 shows the ball's trajectory. Every time the ball hits the walls or the paddle it changes course in accordance with the predetermined motion rules. The coordinate system values displayed are not the same as those used in the game itself; they are merely for illustration.

The mathematical underpinnings of modeling the Pong game implemented on an FPGA platform are then examined incorporating AI functionality for the computer-controlled paddle. In order to capture the underlying mechanics of the game, the AI's decision-making processes, and the corresponding optimization strategies, a number of sophisticated differential integral and matrix equations must be applied. In-depth tables and graphs are also provided to show overall game behavior and performance indicators.

One of the first and most basic video games was Pong. Hardware-level optimization is made possible by implementing Pong on an FPGA, but adding AI capabilities to capture dynamic behavior adds complexity. In this paper, we describe integrating AI learning techniques utilizing matrix operations to effectively manage FPGA resource allocation and representing the motion of the ball and paddle using differential equations.

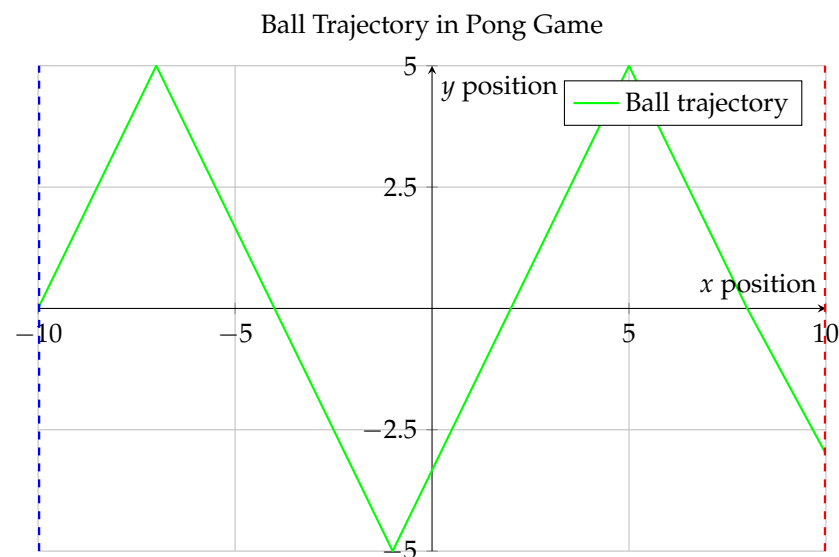
To describe the motion of the ball a system of second-order differential equations can be used:

$$\frac{d^2x(t)}{dt^2} = -\mu \frac{dx(t)}{dt} + F_x(t) \quad (10)$$

$$\frac{d^2y(t)}{dt^2} = -\mu \frac{dy(t)}{dt} + F_y(t) \quad (11)$$

where

- $x(t)$  and  $y(t)$  are the ball's location at a given time  $t$ ;
- $\mu$  represents the drag coefficient;
- $F_x(t)$  and  $F_y(t)$  are the forces exerted on the ball.



**Figure 1.** The ball's path in a simulation of the Pong game. The blue dashed line represents the wall, while the red dashed line shows the paddle's trajectory rather than the paddle itself.

The current network topology is the recurrent neural network (RNN), where the number of epochs is 75 and the running time is 65 s.

The movement of the computer-controlled paddle is characterized by first-order differential equations enhanced with an AI control algorithm:

$$\frac{dp(t)}{dt} = \alpha \cdot (y(t) - p(t)) + \beta \cdot \frac{dy(t)}{dt} \quad (12)$$

where

- $p(t)$  is the position of the paddle;
- $y(t)$  is given by the position of the ball;
- $\alpha$  and  $\beta$  are AI control parameters adjusted using reinforcement learning techniques.

The AI system that controls the movement of the computer paddle utilizes a neural network with assigned weights  $W \in \mathbb{R}^{n \times m}$ , where  $n$  represents the size of the input and  $m$  denotes the size of the output. The network employs this loss function for training:

$$L(W) = \frac{1}{2} \sum_{i=1}^N (\hat{p}(t_i) - p(t_i))^2 + \lambda \|W\|_2^2 \quad (13)$$

where  $\hat{p}(t_i)$  represents the predicted position of the paddle,  $p(t_i)$  denotes the actual position of the paddle and  $\lambda$  serves as a regularization parameter.

A matrix represents the FPGA resources  $R \in \mathbb{Z}^{k \times k}$ , each element represents a logic block used for various computations:

$$R = \begin{bmatrix} r_{11} & r_{12} & \dots & r_{1k} \\ r_{21} & r_{22} & \dots & r_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ r_{k1} & r_{k2} & \dots & r_{kk} \end{bmatrix} \quad (14)$$

Every  $r_{ij}$  denotes a hardware function which may include an arithmetic logic unit (ALU), a multiplier, or a memory component. The  $k \times k$  is the number of resources in a specific FPGA. The optimization problem aimed at reducing the use of FPGA resources is as follows.

$$\min_R \sum_{i,j} r_{ij} \quad \text{depending on performance limitations} \quad (15)$$

This was not implemented in the actual static design of the OpenRISC core; it is a theoretical aside, not an implemented feature.

The AI of the paddle operates using reinforcement learning. Its goal is to maximize the total accumulated reward, with the reward  $R_t$  at time  $t$  defined as follows:

$$R_t = \delta(t - t_{\text{hit}}) - \gamma \cdot \|p(t) - y(t)\|^2 \quad (16)$$

where  $t_{\text{hit}}$  represents the duration of ball contact,  $\delta(\cdot)$  denotes the Dirac delta function, and  $\gamma$  is a penalty factor for any discrepancy between the paddle and ball positions.

We assess the game's performance using several metrics, such as response time, ball velocity, paddle accuracy, and the AI's learning rate. Table 4 shows the comparison between a human player and an AI player. The human player's response time is slightly slower at 50 ms, while the AI's response time is faster at 35 ms. The human player achieves an accuracy of 85%, while the AI reaches 93% accuracy. Here, accuracy refers to how often the player successfully hits the ball with the paddle instead of missing it. In general, the AI is significantly faster and more accurate than the human player. In this game, the ball travels at 2.5 m/s, and the AI player uses a learning rate of 0.005.

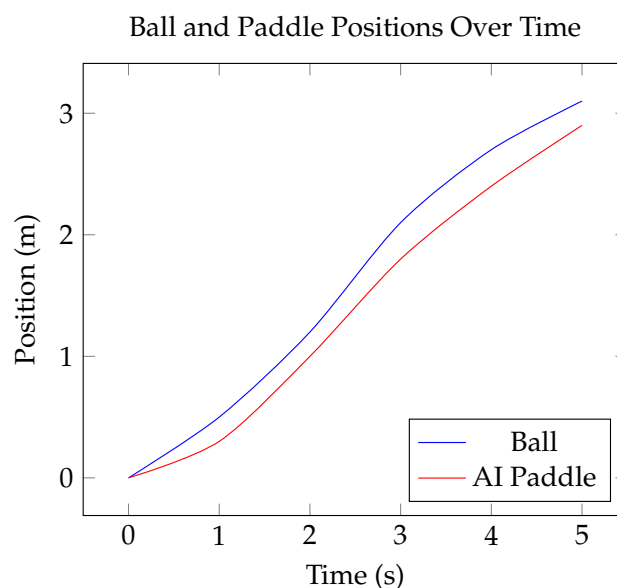
**Table 4.** Evaluation of player and AI paddle movement metrics.

| Metric                   | Player Paddle | AI Paddle | Ball Speed (m/s) |
|--------------------------|---------------|-----------|------------------|
| Response Time (ms)       | 50            | 35        | -                |
| Accuracy (%)             | 85            | 93        | -                |
| Ball Speed Deviation (%) | -             | -         | 2.5              |
| Learning Rate            | -             | 0.005     | -                |

The movement of the ball and paddle positions can be illustrated in the following way. The AI paddle tries to follow the position of the ball, it moves after the position of the ball (Figure 2).

An FPGA was used to implement a complex mathematical representation of the Pong game that included sophisticated AI paddle control techniques. This model provides deep insight into the dynamics of the game by using differential and matrix equations and the AI-driven paddle actions show how learning algorithms and hardware can be seamlessly integrated.

In order to enhance a Pong game built on an AI-based FPGA platform, this study investigated the use of sophisticated mathematical frameworks such as partial differential equations (PDEs), Jacobian matrices and higher-order matrix operations. Through complex modeling, these formulations capture the behavior of the ball and paddle, the distribution of computational resources and the AI's decision-making process. To show how the system changes over time, graphic plots are used.



**Figure 2.** Ball and AI paddle positions over time.

Pong is traditionally built on a straightforward operational framework, but adding AI features and deploying it on an FPGA adds a great deal of complexity. This paper expands on the basic models by describing and analyzing the dynamics of the game using partial differential equations, matrix calculus and optimization techniques.

Partial differential equations that take into consideration time-dependent positional changes brought on by external factors like gravity, air resistance and collisions with paddles are used to model the motion of the ball in a two-dimensional plane. This is the equation that controls the ball's velocity:

$$\frac{\partial^2 \mathbf{r}(x, y, t)}{\partial t^2} = -\mu \frac{\partial \mathbf{r}(x, y, t)}{\partial t} + \mathbf{F}(x, y, t) \quad (17)$$

where

- $\mathbf{r}(x, y, t) = [x(t), y(t)]$  is the ball's position vector at time  $t$ ;
- $\mu$  represents the drag coefficient;
- $\mathbf{F}(x, y, t)$  account for the external forces that influence the ball.

Additionally boundary conditions for interactions with the paddle and the wall are established:

$$\mathbf{r}(x, y, t) \cdot \hat{n} = 0 \quad \text{on the surfaces of boundaries} \quad (18)$$

where the boundary surface's unit normal vector is represented by  $\hat{n}$ .

Jacobian matrices describe the paddle's behavior and are used to calculate the system's gradient in relation to the state variables. The Jacobian matrix  $J$  for the AI-controlled paddle is provided and the system's state is denoted by  $\mathbf{p}(t)$ :

$$J = \frac{\partial \mathbf{p}(t)}{\partial \mathbf{u}(t)} = \begin{bmatrix} \frac{\partial p_x}{\partial u_1} & \frac{\partial p_x}{\partial u_2} & \cdots & \frac{\partial p_x}{\partial u_n} \\ \frac{\partial p_y}{\partial u_1} & \frac{\partial p_y}{\partial u_2} & \cdots & \frac{\partial p_y}{\partial u_n} \end{bmatrix} \quad (19)$$

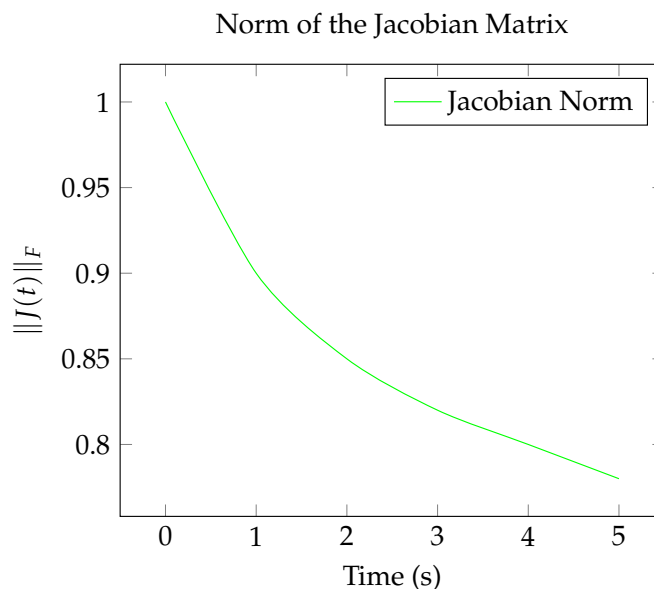
where the Jacobian shows how sensitive the paddle's position is to these control inputs and  $\mathbf{u}(t)$  represents the AI's control inputs.

Reducing the following cost function is the goal of the AI optimization process:

$$C(\mathbf{p}, \mathbf{u}) = \int_0^T \|\mathbf{p}(t) - \mathbf{r}(x, y, t)\|^2 dt + \lambda \|\mathbf{J}\|_F^2 \quad (20)$$

where  $\lambda$  acts as a regularization factor and  $\|\cdot\|_F$  represents the Frobenius norm of the Jacobian matrix.

One important measure of the system's stability is the norm of the Jacobian matrix which represents how strongly the paddle's position responds to the AI control input (Figure 3).



**Figure 3.** The Frobenius norm of the Jacobian matrix evolves over time.

In FPGA design, the allocation of hardware resources is represented by matrix calculus. Let  $R(t) \in \mathbb{Z}^{k \times k}$  represent the resource allocation matrix at time  $t$ , where each entry signifies a particular logic block. The following describes how resource usage has changed over time:

$$\frac{dR(t)}{dt} = AR(t) + Bu(t) \quad (21)$$

The input matrix that controls FPGA operations is denoted by  $B$  and the system matrix is represented by  $A$ .

Reducing total resource consumption is the aim of the optimization problem:

$$\min_{R(t)} \int_0^T \|R(t)\|_F^2 dt \quad (22)$$

The deep neural network used by the paddle's AI is represented as a matrix with weight  $W$ :

$$\mathbf{p}(t) = \sigma(W \cdot \mathbf{x}(t)) \quad (23)$$

The input vector that represents the game's state including the ball's position and velocity is provided by  $\mathbf{x}(t)$ . The activation function is represented by  $\sigma$ .

Backpropagation regulates the neural network training process by adhering to the following update rule:

$$W_{t+1} = W_t - \eta \frac{\partial L(W)}{\partial W_t} \quad (24)$$

In this case  $L(W)$  stands for the loss function which has the following definition and  $\eta$  for the learning rate:

$$L(W) = \frac{1}{2} \sum_{i=1}^N (\hat{p}_i(t) - p_i(t))^2 + \lambda \|W\|_F^2 \quad (25)$$

In order to control AI behavior and distribute hardware resources, a sophisticated mathematical framework for a Pong game on an FPGA was presented, utilizing partial differential equations, Jacobian matrices and matrix calculus. These exacting formulations facilitate the optimization of gameplay performance and FPGA resource utilization while offering a comprehensive understanding of the dynamics of the system.

### 3.2. Mathematical Background of the Fight Game

To compute the position, velocity and acceleration, each player  $p \in \{1, 2\}$  has the following equations:

The position equation is

$$\mathbf{r}_p(t) = \begin{bmatrix} x_p(t) \\ y_p(t) \end{bmatrix} \quad (26)$$

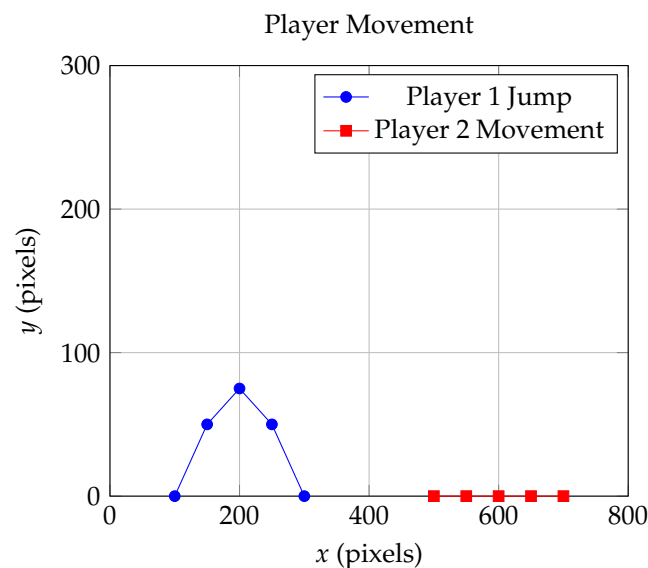
The velocity equation is

$$\mathbf{v}_p(t) = \begin{bmatrix} v_{x,p}(t) \\ v_{y,p}(t) \end{bmatrix} \quad (27)$$

The acceleration equation is

$$\mathbf{a}_p(t) = \begin{bmatrix} a_{x,p}(t) \\ a_{y,p}(t) \end{bmatrix} \quad (28)$$

Figure 4 shows the movement of the player during the game play.



**Figure 4.** Movement of the two players in the fight game.

The equations of motion are the following:

For horizontal motion:

$$v_{x,p}(t+1) = v_{x,p}(t) + a_{x,p}(t)\Delta t \quad (29)$$

For vertical motion:

$$v_{y,p}(t+1) = v_{y,p}(t) + (a_{y,p}(t) - g)\Delta t \quad (30)$$

For the position update:

$$\mathbf{r}_p(t+1) = \mathbf{r}_p(t) + \mathbf{v}_p(t)\Delta t \quad (31)$$

where

- $g = 9.8$  (pixels/frame<sup>2</sup>);
- $\Delta t = 1$  (fixed timestamp).

For hit box collision, axis-aligned bounding boxes (AABB) were used as follows:

For the player hit box:

$$\mathcal{B}_p = [x_p - w_p, x_p + w_p] \times [y_p, y_p + h_p] \quad (32)$$

For the punch box:

$$\mathcal{P}_p = [x_p + d_p \ell_p, x_p + d_p (\ell_p + L_p)] \times [y_p + \delta_y, y_p + \delta_y + H_p] \quad (33)$$

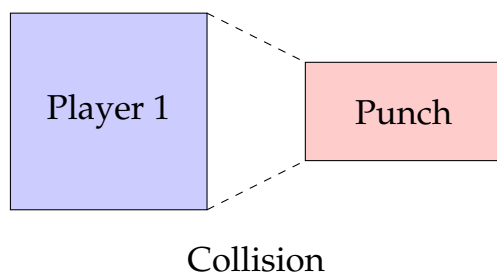
where

- $w_p, h_p$ : player width/height;
- $L_p, H_p$ : punch length/height;
- $d_p \in \{-1, 1\}$ : facing direction.

The collision condition is the following:

$$\mathcal{P}_1 \cap \mathcal{B}_2 \neq \emptyset \iff \begin{cases} \max(x_1 + d_1 \ell_1, x_2 - w_2) \leq \min(x_1 + d_1 (\ell_1 + L_1), x_2 + w_2) \\ \max(y_1 + \delta_y, y_2) \leq \min(y_1 + \delta_y + H_1, y_2 + h_2) \end{cases} \quad (34)$$

Figure 5 shows when the hit boxes overlap.



**Figure 5.** When hit boxes overlap.

For combat mechanics, the health and damage model is as follows:

$$h_p(t+1) = \max(0, h_p(t) - D(s_q)) \quad (35)$$

where

$$D(s_q) = \begin{cases} 10 & \text{(jab)} \\ 20 & \text{(hook)} \\ 0 & \text{(block)} \end{cases}$$

Figure 6 shows healthy decay.

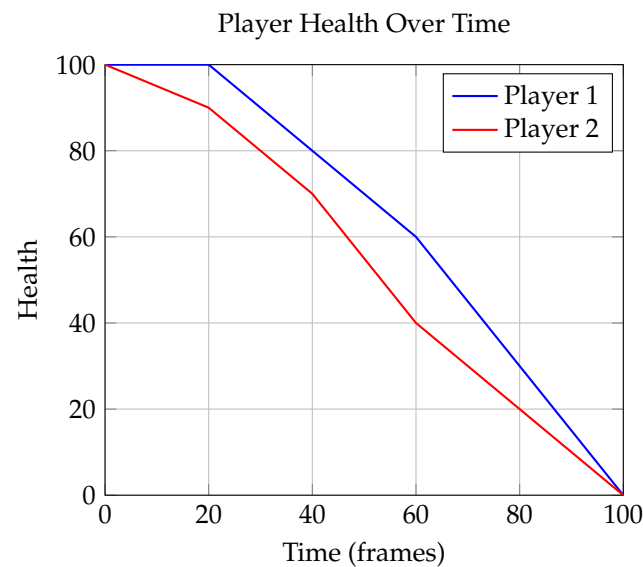
If we analyze impulse mechanics in the knock back effect, then

$$\mathbf{v}_2(t+1) = \mathbf{v}_2(t) + \begin{bmatrix} d_1 \cdot k_x \\ k_y \end{bmatrix} \quad (36)$$

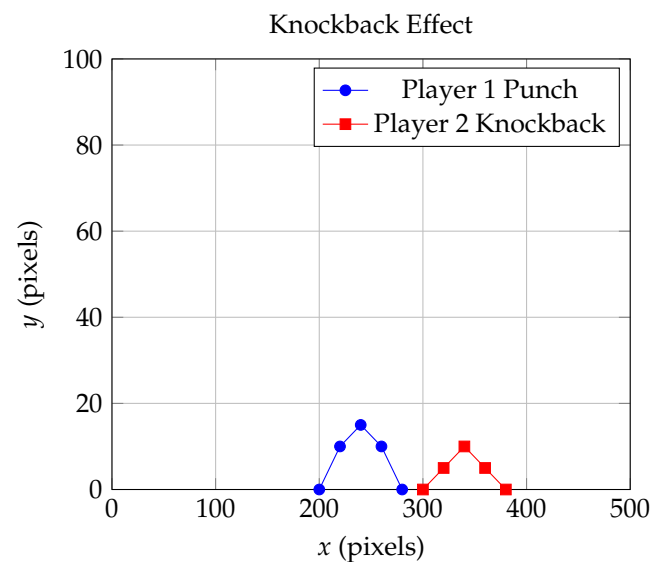
where

- $k_x = 5, k_y = 3$  (knockback coefficients).

Figure 7 shows how the plot will be for the knockback trajectory.



**Figure 6.** Healthy decay of the player over time in a fight game.



**Figure 7.** Knockback effect trajectory in a fight game.

For the game state, we are using a finite state machine (FSM):

$$s_p \in \{\text{idle}, \text{punch}, \text{block}, \text{staggered}\} \quad (37)$$

For the win condition, we have the following equation:

$$\text{Win}(p) \iff h_q \leq 0 \quad \text{or} \quad t \geq t_{\max} \quad (38)$$

### 3.3. Mathematical Background of the Chips Placement Game with Backtracking

Let  $C$  be an  $n \times n$  chessboard. Let  $x_i$  denote the column position of the chip in row  $i$ .  
Row constraint: only one chip per row.

$$\forall i \in \{1, \dots, n\}, \exists! x_i \in \{1, \dots, n\} \quad (39)$$

Column constraint: only one chip per column.

$$\forall j \in \{1, \dots, n\}, \exists! i \text{ such that } x_i = j \quad (40)$$

The method for this implementation is to use the recursive backtracking algorithm.

Base case: if all rows are filled ( $i = n + 1$ ), return the solution.

Recursive step:

- For each column  $j$ , check if placing a chip in  $(i, j)$  is valid,
- If valid, place the chip and repeat for the row  $i + 1$ ,
- If no valid placement exists, backtrack.

The mathematical representation is presented as follows:

$$\text{PlaceChip}(i, C) = \begin{cases} \text{Return } C & \text{if } i = n + 1 \\ \bigcup_{j=1}^n \text{PlaceChip}(i + 1, C \cup \{(i, j)\}) & \text{if Valid}(i, j, C) \\ \emptyset & \text{otherwise (backtrack)} \end{cases} \quad (41)$$

where

- $\text{Valid}(i, j, C) \iff \forall (k, x_k) \in C, x_k \neq j$ .

Next, the implementation of a  $4 \times 4$  chessboard will be presented.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (42)$$

Figure 8 shows the chessboard with the placed chips.

|    |   |   |   |   |
|----|---|---|---|---|
| R1 | 1 | 2 | 3 | 4 |
| R2 |   |   |   |   |
| R3 |   |   |   |   |
| R4 |   |   |   |   |

Figure 8. The chessboard with the placed chips in the N-Queens game.

The interpretation of the matrix is as follows:

- Row 1: chip in column 2;
- Row 2: chip in column 4;
- Row 3: chip in column 1;
- Row 4: chip in column 3.

Figure 9 shows the (partial) backtracking tree for the search space, where  $n = 3$ .

Where

- Green nodes: valid solutions;
- Red nodes: invalid placements (backtracking points).

For the complexity analysis, also represented in Figure 10:

- Time complexity:  $O(n!)$  (worst case, exhaustive search);
- Space Complexity:  $O(n)$  (depth of the recursion stack).

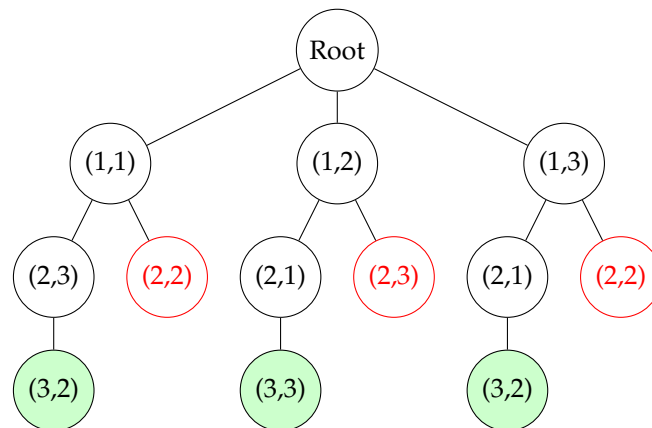


Figure 9. (Partial) backtracking tree for  $n = 3$ .

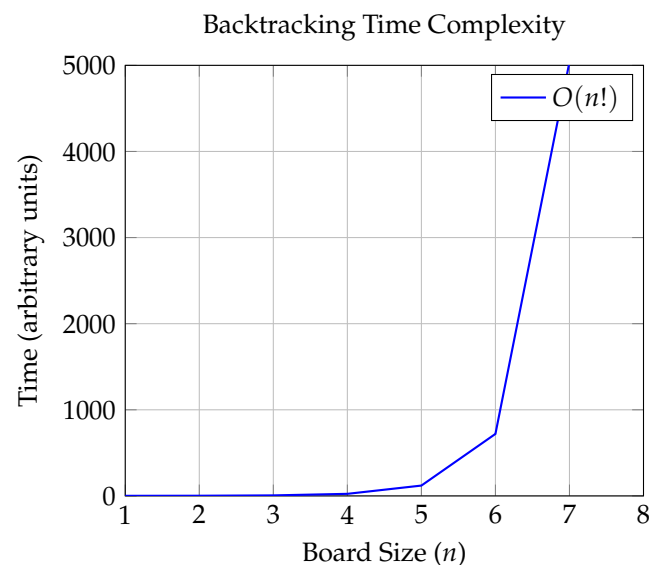


Figure 10. Backtracking efficiency in the N-Queens game.

### 3.4. Practical Implementation of the Games

The decision was made to develop the games in software using a high-level programming language due to its ease of use.

In the Pong game, the two paddles were assigned blue and red colors, while the ball was colored yellow. The character unit of the programming language was used instead of a dedicated graphics unit, since the simple visual elements do not require advanced graphical tools. All objects can be rendered by inserting the ASCII codes of specific special characters directly in the program. The paddles are drawn using ASCII code 219, which represents a solid filled square; by placing several of these characters next to each other, a bar-shaped figure is formed that resembles a paddle and can also be colored as desired. The ball uses ASCII code 2, the STX (start of text) control character, which appears as a round, smiley-face symbol in command-line mode, making it suitable to represent the ball in the Pong game.

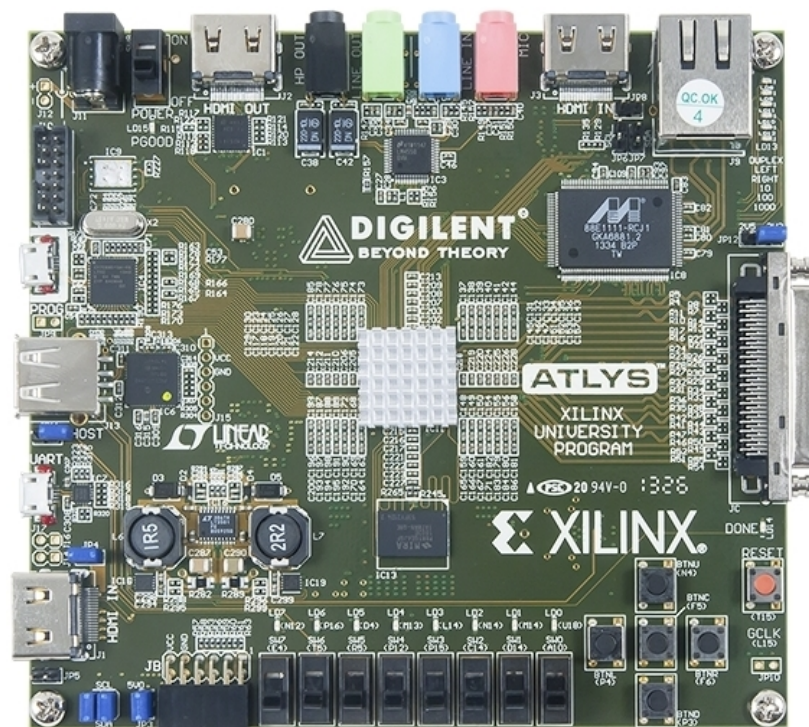
From this point on, you need to define and apply the rules. The first rule is that the ball should bounce off the side walls and the paddles, but if it is not hit by a paddle, it must be allowed to leave the screen. In that case, the player who failed to hit the ball with their paddle concedes a goal. After a goal, the ball always starts its movement from the side of the player who was scored against and travels in the opposite direction. This is the typical rule in many Pong games: the ball is restarted by moving away from the side where the point was scored. The paddles must remain on the screen; they can only

move vertically, either from top to bottom or from bottom to top. There also needs to be a mode in which one player can compete against the computer, which means the game must include computer-controlled behavior. The paddles are controlled by human players or, in single-player mode, by the computer, but the ball is always controlled by the computer, with only its direction influenced by the players [18].

In the fighting game, two characters are used: one wears a white kimono and the other a gray kimono, with one having a red belt and the other a blue belt. Both fighters have yellow skin. The body parts of the characters were more difficult to draw because they are constructed from squares, circles, parallelograms, trapezoids, and various lines. To move a character or animate its arm for a punch, a black box is drawn over the part that is in motion. Each fighter's health is displayed at the top of the screen. When a player is hit, their health decreases, and if it reaches zero, that player loses the match. The most challenging aspect of creating this game is coding the drawing of the characters; the rest is relatively simple.

In the chip placement game using backtracking, or in the N-Queens problem, a chessboard is drawn with alternating blue and white squares, and the chips are shown in red. The objective is to position at most one chip in each row and each column and to determine all possible valid arrangements. The most suitable approach to solve this is to use the backtracking algorithm.

The ATLYS board is the development platform that is being used (Figure 11).



**Figure 11.** The Digilent ATLYS development board [19].

The goal is to implement the games in a high-level programming language (for example, C) and execute it on an FPGA. To achieve this, a microprocessor architecture must first be set up on the FPGA, and a Linux operating system must be installed on that architecture. Once this is done, the games need to be cross-compiled for that specific microprocessor architecture, then the resulting binary must be transferred and executed on the microprocessor running on the FPGA. If these steps are successful, all other tools can likewise be run on the FPGA; they simply need to be cross-compiled for the microprocessor architecture implemented on the FPGA.

The observed system performance while running a 2D game is as follows: CPU load: 40%, frame rate: 60 FPS (VGA standard with hardware synchronization), power consumption: 4 Watts (FPGA static plus I/O power, with game logic contribution being negligible), and latency: 33 ms (due to the frame buffer architecture and polling pulse of 1–2 frames).

In Figure 12, the Spartan-6 FPGA can be seen.

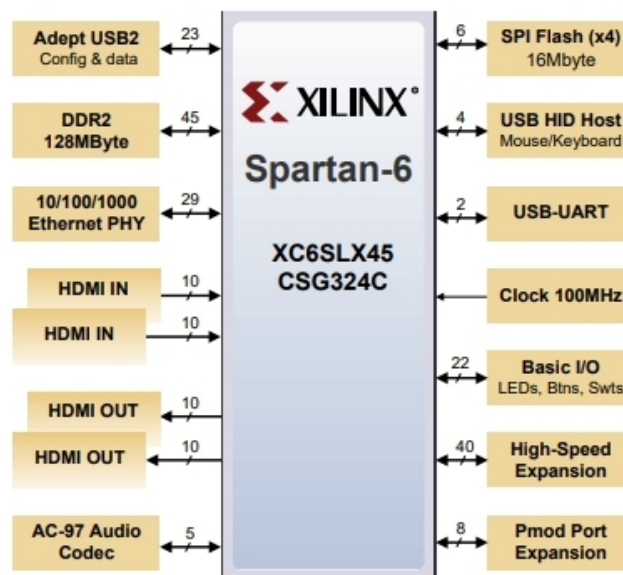


Figure 12. The Spartan-6 FPGA [19].

In Figure 13 can be seen the configuration of the Spartan-6 FPGA.

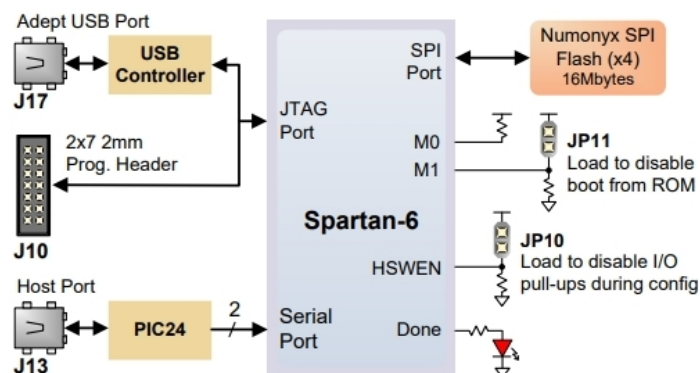


Figure 13. Configuration of the Spartan-6 FPGA [19].

In Figure 14 can be seen how to start the Atlys FPGA development board.



Figure 14. Starting up the Atlys FPGA development board [19].

In Figure 15 can be seen the HDMI port of the Atlys FPGA development board.

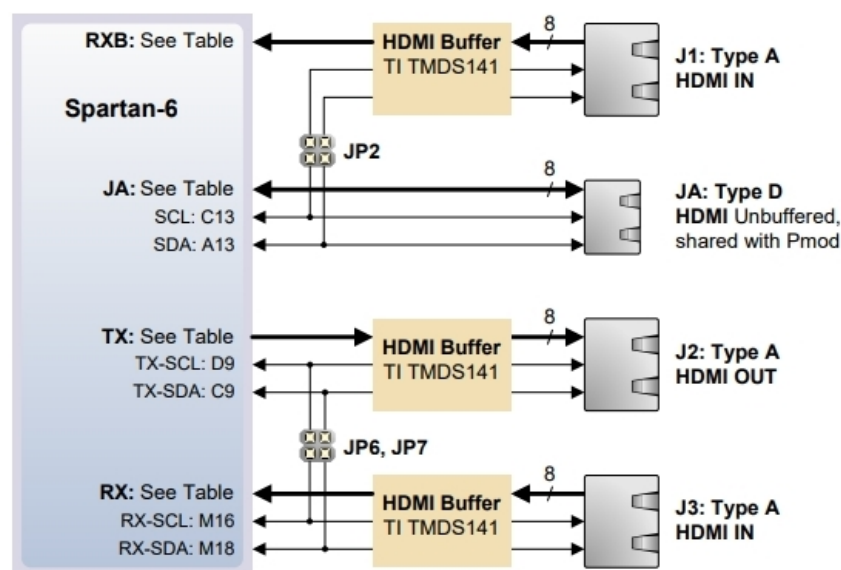


Figure 15. HDMI port of the Atlys FPGA development board [19].

In Table 5, the HDMI configuration table of the Atlys FPGA development board can be seen.

Table 5. HDMI configuration table of the Atlys FPGA development board [19].

| HDMI Type A Connectors     |            |            |                                             | HDMI Type D |          |
|----------------------------|------------|------------|---------------------------------------------|-------------|----------|
| Pin/Signal                 | J1: IN     | J2: Out    | J3: IN                                      | Pin/Signal  | JA: BiDi |
| 1: D2+                     | B12        | B8         | J16                                         | 1: HPD      | JP3*     |
| 2: D2_S                    | GND        | GND        | GND                                         | 2: RES      | VCCB2    |
| 3: D2−                     | A12        | A8         | J18                                         | 3: D2+      | N5       |
| 4: D1+                     | B11        | C7         | L17                                         | 4: D2_S     | GND      |
| 5: D1_S                    | GND        | GND        | GND                                         | 5: D2−      | P6       |
| 6: D1−                     | A11        | A7         | L18                                         | 6: D1+      | T4       |
| 7: D0+                     | G9         | D8         | K17                                         | 7: D1_S     | GND      |
| 8: D0_S                    | GND        | GND        | GND                                         | 8: D1−      | V4       |
| 9: D0−                     | F9         | C8         | K18                                         | 9: D0+      | R3       |
| 10: Clk+                   | D11        | B6         | H17                                         | 10: D0_S    | GND      |
| 11: Clk_S                  | GND        | GND        | GND                                         | 11: D0−     | T3       |
| 12: Clk−                   | C11        | A6         | H18                                         | 12: Clk+    | T9       |
| 13: CEC                    | NC         | 0 K to Gnd | NC                                          | 13: Clk_S   | GND      |
| 14: RES                    | NC         | NC         | NC                                          | 14: Clk−    | V9       |
| 15: SCL                    | C13        | D9         | M16                                         | 15: CEC     | VCCB2    |
| 16: SDA                    | A13        | C9         | M18                                         | 16: Gnd     | GND      |
| 17: Gnd                    | GND        | GND        | GND                                         | 17: SCL     | C13**    |
| 18: 5 V                    | JP4        | 5 V        | JP8                                         | 18: SCA     | A13**    |
| 19: HPD                    | 1 K to 5 V | NC         | 1 K to 5 V                                  | 19: 5 V     | JP3      |
| *jumper can disconnect Vdd |            |            | **shared with J1 I2C signals via jumper JP2 |             |          |

In Figure 16, the USB port of the Atlys FPGA development board can be seen.

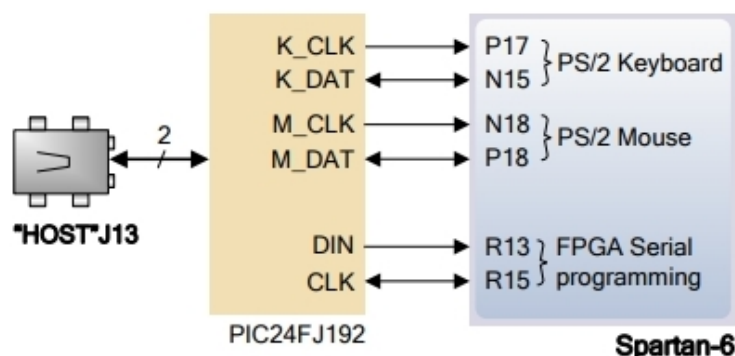


Figure 16. USB port of the Atlys FPGA development board [19].

In Figure 17 can be seen the USB configuration of the Atlys FPGA development board.

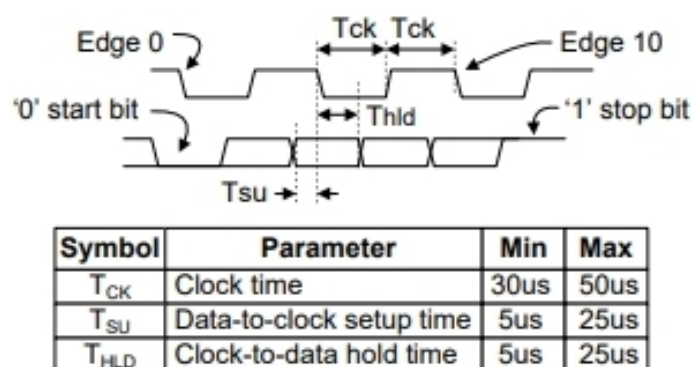


Figure 17. USB configuration of the Atlys FPGA development board [19].

In Figure 18 can be seen the keyboard connection to the Atlys FPGA development board.

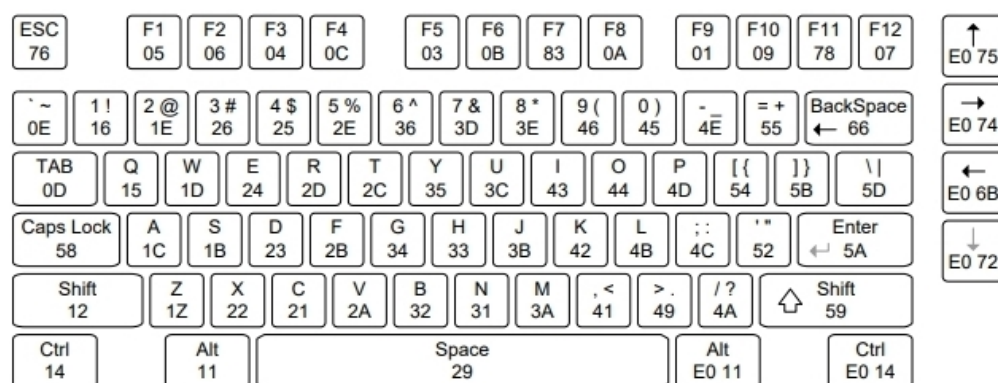


Figure 18. Keyboard connection to the Atlys FPGA development board [19].

In Figure 19 can be seen the mouse connection to the Atlys FPGA development board.

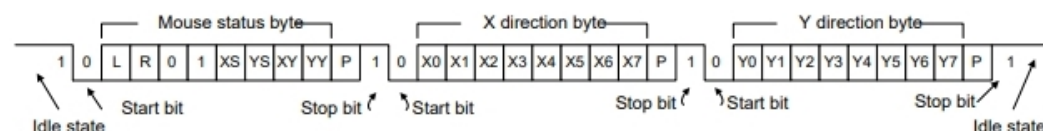
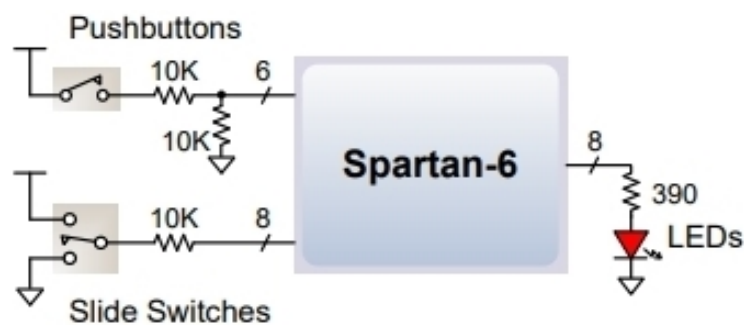


Figure 19. Mouse connection to the Atlys FPGA development board [19].

In Figure 20 can be seen the push buttons and side switches of the Atlys FPGA development board.



**Figure 20.** Push buttons and side switches of the Atlys FPGA development board [19].

In Table 6, the configuration table of push buttons and side switches of the Atlys FPGA development board can be seen.

**Table 6.** Push buttons and side switches configuration table of the Atlys FPGA development board [19].

| Push Buttons | Slide Switches | LEDs    |
|--------------|----------------|---------|
| BTNU:N4      | SW0:A10        | LD0:U18 |
| BTNC:F5      | SW1:D14        | LD1:M14 |
| BTNR:F6      | SW2:C14        | LD2:N14 |
| BTNL:P4      | SW3:P15        | LD3:L14 |
| BTND:P3      | SW4:P12        | LD4:M13 |
| BRST:T15     | SW5:R5         | LD5:D4  |
|              | SW6:T5         | LD6:P16 |
|              | SW7:E4         | LD7:N12 |

The main purpose of running Linux on an FPGA is to enable the operating system to execute any application or game once its source code has been cross-compiled for the OpenRISC platform. Although it would be much simpler to describe the motion of the table tennis paddle and ball directly in VHDL/Verilog, this particular game was developed solely to demonstrate how to cross-compile an application or game for OpenRISC and to show that virtually any software can be ported. The objective is not to use Linux just to run Pong; rather, it is to run Linux so that a wide range of applications and games can be executed on the FPGA platform. Linux is more suitable for running general-purpose software, while a MicroBlaze soft-core processor is more than adequate for a simple Pong game, but may fall short for more complex applications. Pong was ported to the OpenRISC platform only to illustrate how straightforward the process becomes once the cross-compilation toolchain has been set up.

In Figure 21, there is the block diagram of the Pong game.

Figure 22 shows the OpenRISC 1200-based architecture on the Spartan-6 FPGA.

One of the most difficult tasks can be getting Linux to run on an FPGA, since you must first implement a microprocessor architecture on the FPGA itself.

The microprocessor architecture selected was the 32-bit OpenRISC processor. This open-source and freely available design enables straightforward deployment across a wide range of devices. After integrating the microprocessor, a Linux distribution had to be installed on the system. The chosen option was a standard Linux kernel version 2.6 configured to run in command-line mode. To keep the system footprint small and reduce hardware requirements, the Linux installation had to be heavily stripped down, excluding the X-Server. To fulfill these constraints, Linux was compiled from source, including only the essential modules.

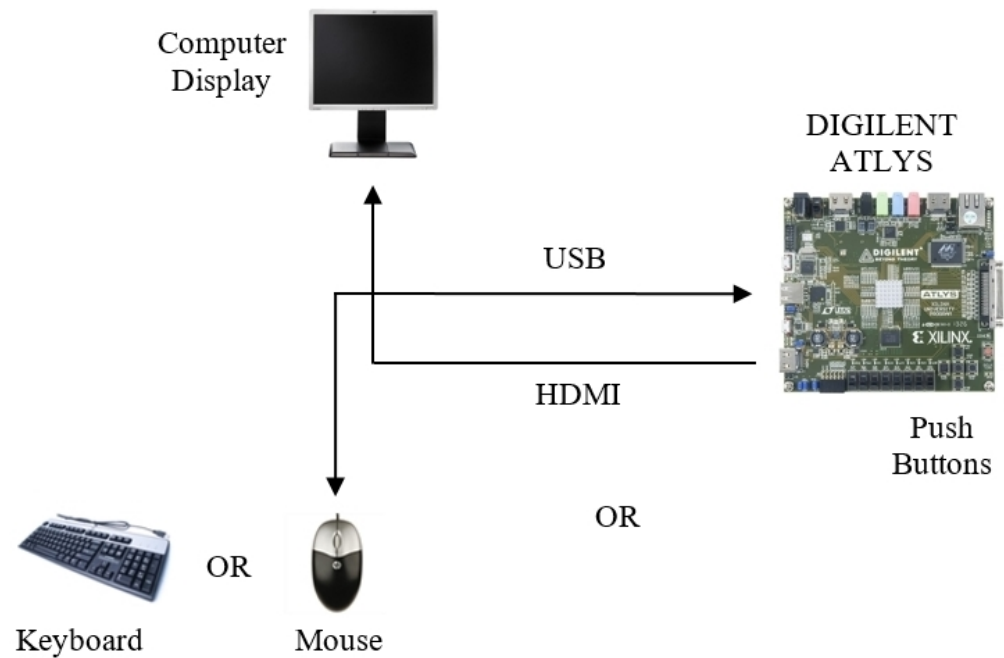


Figure 21. Block diagram of the Pong game.

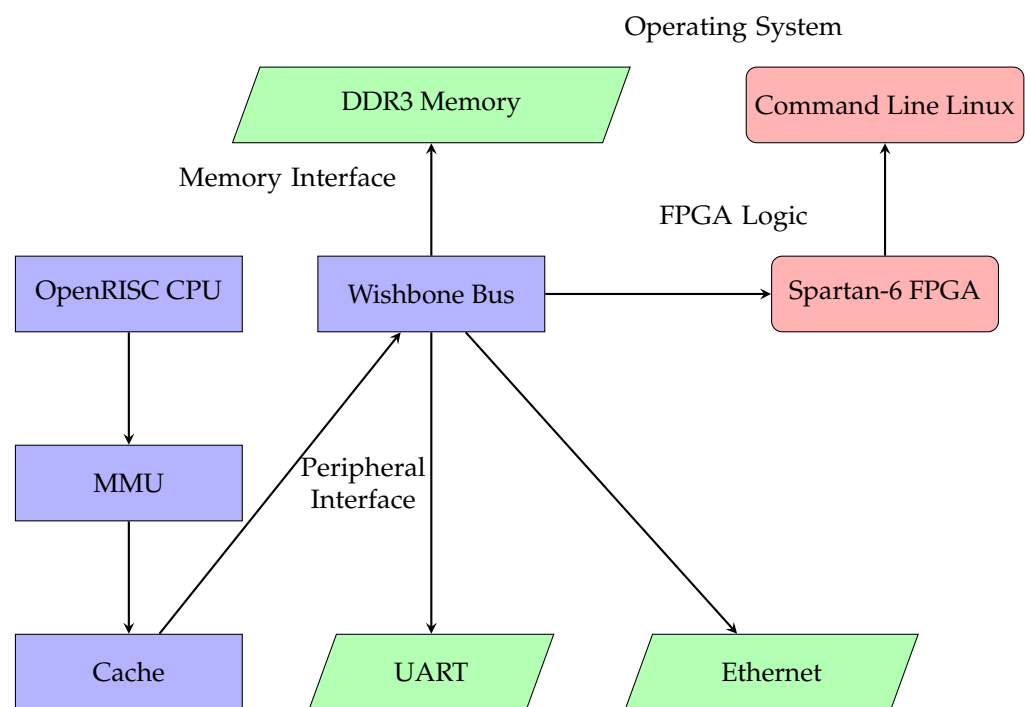


Figure 22. OpenRISC 1200-based architecture on the Spartan-6 FPGA.

Setting up a cross-compilation toolchain on a Linux PC for creating applications aimed at the embedded Linux platform was the next step. In order to enable testing applications before deploying them to the embedded device, this toolchain had to supply a compiler for the OpenRISC processor and ideally a Linux emulator for OpenRISC.

Selecting the platform that the system will operate on is the first step in the implementation process. The Spartan-6 FPGA-equipped Digilent ATLYS board was chosen for this.

The first step in deploying a complex system such as Linux on a relatively uncommon FPGA-based processor is to emulate it on a more powerful machine. In practice, this means installing a Linux system on a desktop PC and configuring the toolchain needed for

compiling and emulating the smaller Linux environment. This target system is a command line-only Linux installation that runs on the OpenRISC processor implemented in the FPGA. A text-based system without an X-Server was chosen due to tight hardware constraints. The ATLYS board, which uses a Spartan-6 FPGA, includes only a 100 MHz quartz oscillator, with clock speeds boosted up to 500 MHz via a PLL. It also provides 16 MByte  $\times$  4 SPI flash memory, so the Linux image has to be small enough to fit into this limited storage.

The chosen system that operates the lightweight Linux emulator is built on the Ubuntu Linux distribution.

In Figure 23, it is shown that the Ubuntu Linux operating system can run the OpenRISC emulator.

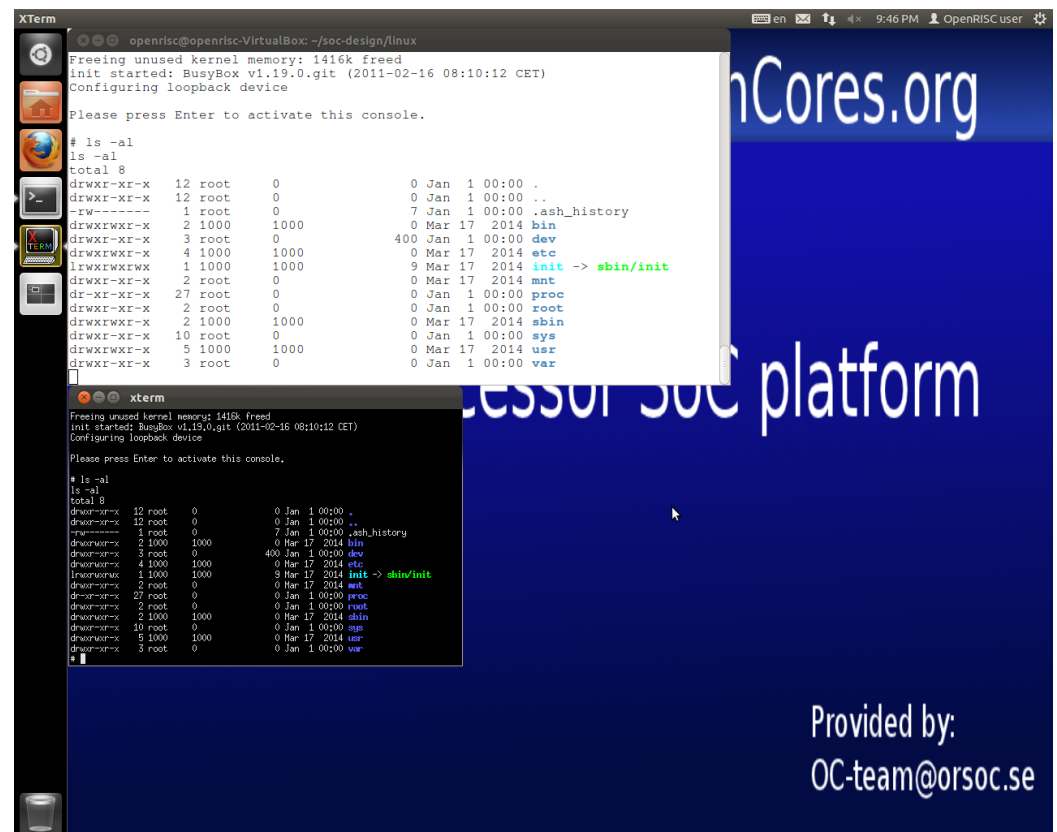


Figure 23. Ubuntu Linux operating system which can run the OpenRISC emulator.

In Figure 24, there is an embedded Linux system running within an emulator under the Ubuntu Linux distribution.

Evidence for the construction of the cross-compilation toolchain will now be shown using screenshots of the Linux operating system and the Pong game running on the OpenRISC platform. The generated binaries are not tailored to leverage any particular FPGA device; instead, they are intended for execution on standard microcontroller-based systems. The underlying concept is to enable the porting of any application or game to any FPGA platform that incorporates the OpenRISC microprocessor. The key contribution of this paper lies in cross-compiling applications and games for the OpenRISC microprocessor, developing the corresponding cross-compilation toolchain, implementing the OpenRISC microprocessor on the FPGA, and subsequently installing a Linux operating system on this target platform.

```

openrisc@openrisc-VirtualBox: ~/soc-design/linux
Freeing unused kernel memory: 1416k freed
init started: BusyBox v1.19.0.git (2011-02-16 08:10:12 CET)
Configuring loopback device

Please press Enter to activate this console.

# ls -al
ls -al
total 8
drwxr-xr-x 12 root 0 0 Jan 1 00:00 .
drwxr-xr-x 12 root 0 0 Jan 1 00:00 ..
-rw----- 1 root 0 7 Jan 1 00:00 .ash_history
drwxrwxr-x 2 1000 1000 0 Mar 17 2014 bin
drwxr-xr-x 3 root 0 400 Jan 1 00:00 dev
drwxrwxr-x 4 1000 1000 0 Mar 17 2014 etc
lrwxrwxrwx 1 1000 1000 9 Mar 17 2014 init -> sbin/init
drwxr-xr-x 2 root 0 0 Mar 17 2014 mnt
dr-xr-xr-x 27 root 0 0 Jan 1 00:00 proc
drwxr-xr-x 2 root 0 0 Jan 1 00:00 root
drwxrwxr-x 2 1000 1000 0 Mar 17 2014 sbin
drwxr-xr-x 10 root 0 0 Jan 1 00:00 sys
drwxrwxr-x 5 1000 1000 0 Mar 17 2014 usr
drwxr-xr-x 3 root 0 0 Jan 1 00:00 var

```

**Figure 24.** An embedded Linux emulator designed for the Ubuntu Linux distribution, specifically intended to execute on an FPGA-implemented OpenRISC processor [20].

In Figure 25 is the output displayed in the terminal of the Ubuntu Linux distribution when the compact Linux emulator for the OpenRISC processor on the FPGA is run. This Linux environment is intended to function on an OpenRISC processor realized on an FPGA. Ignore the displayed time, as it is not configured; it is reset whenever power is lost because the FPGA board lacks an onboard battery for the CMOS clock.

```

xterm
Freeing unused kernel memory: 1416k freed
init started: BusyBox v1.19.0.git (2011-02-16 08:10:12 CET)
Configuring loopback device

Please press Enter to activate this console.

# ls -al
ls -al
total 8
drwxr-xr-x 12 root 0 0 Jan 1 00:00 .
drwxr-xr-x 12 root 0 0 Jan 1 00:00 ..
-rw----- 1 root 0 7 Jan 1 00:00 .ash_history
drwxrwxr-x 2 1000 1000 0 Mar 17 2014 bin
drwxr-xr-x 3 root 0 400 Jan 1 00:00 dev
drwxrwxr-x 4 1000 1000 0 Mar 17 2014 etc
lrwxrwxrwx 1 1000 1000 9 Mar 17 2014 init -> sbin/init
drwxr-xr-x 2 root 0 0 Mar 17 2014 mnt
dr-xr-xr-x 27 root 0 0 Jan 1 00:00 proc
drwxr-xr-x 2 root 0 0 Jan 1 00:00 root
drwxrwxr-x 2 1000 1000 0 Mar 17 2014 sbin
drwxr-xr-x 10 root 0 0 Jan 1 00:00 sys
drwxrwxr-x 5 1000 1000 0 Mar 17 2014 usr
drwxr-xr-x 3 root 0 0 Jan 1 00:00 var
#

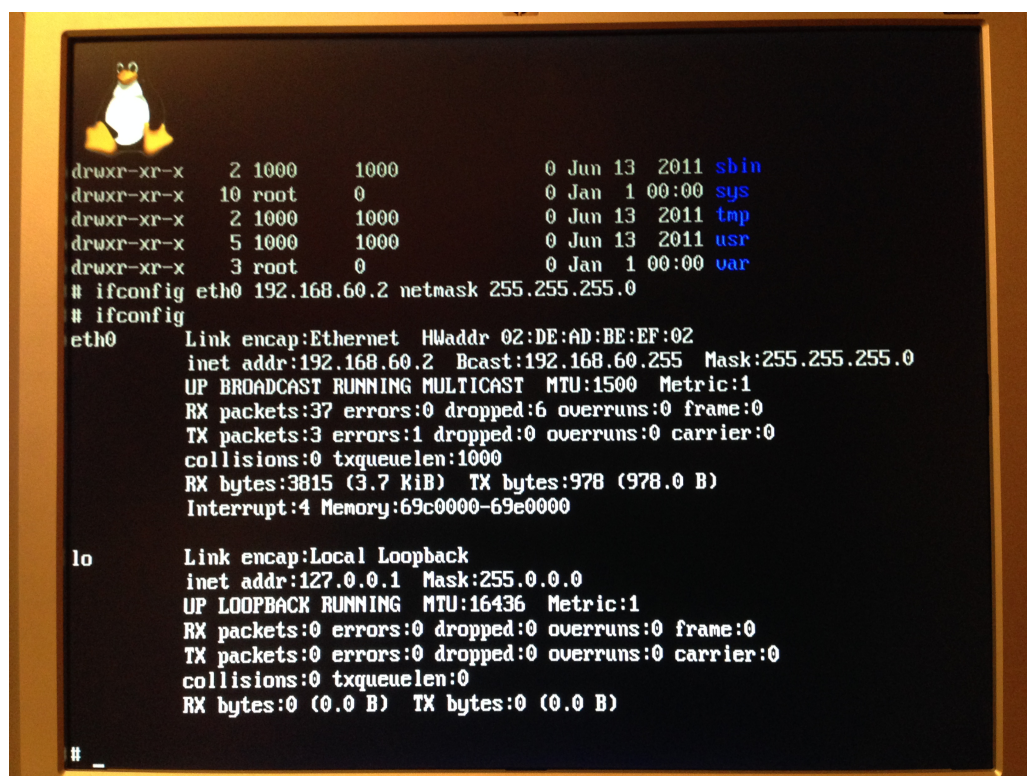
```

**Figure 25.** Output shown in the terminal after executing the Linux emulator designed for an OpenRISC processor implemented on an FPGA [20].

The initial objective was to design the processor core for the 32-bit OpenRISC processor adapted to the ATLYS board. The next step was to load the command-line Linux image into the ATLYS board's flash memory. Because the command-line Linux system had to fit

within the ATLYS board's 16 MB  $\times$  4 SPI flash capacity, it had to be manually recompiled to substantially reduce its size.

In Figure 26, the image shows Linux running on an ATLYS FPGA board, configured to use its Ethernet interface for data transfer. The network setup consists of a PC assigned the IP address 192.168.60.1 and an FPGA board assigned 192.168.60.2. Both devices use the 255.255.255.0 subnet mask, meaning they reside on the same subnet and can communicate smoothly via a switch. An important detail is that the ATLYS FPGA board needs a cross-over Ethernet cable because its port does not support Auto-MDIX, so the RX and TX lines must be swapped manually in the cable.



```

drwxr-xr-x  2 1000    1000          0 Jun 13  2011 sbin
drwxr-xr-x 10 root      0          0 Jan  1  00:00 sys
drwxr-xr-x  2 1000    1000          0 Jun 13  2011 tmp
drwxr-xr-x  5 1000    1000          0 Jun 13  2011 usr
drwxr-xr-x  3 root      0          0 Jan  1  00:00 var
# ifconfig eth0 192.168.60.2 netmask 255.255.255.0
# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:DE:AD:BE:EF:02
          inet addr:192.168.60.2  Bcast:192.168.60.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:37 errors:0 dropped:6 overruns:0 frame:0
          TX packets:3 errors:1 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3815 (3.7 KiB)  TX bytes:978 (978.0 B)
          Interrupt:4 Memory:69c0000-69e0000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

# _

```

**Figure 26.** Linux operating on the OpenRISC processor built on the FPGA board, managing the Ethernet port configuration [20].

Data communication between the PC and the ATLYS FPGA board is made possible by the Ethernet interface. The Linux system runs an FTP server that allows straightforward data transfer across the network [20].

In Figure 27, the diagram shows the code structure for the Pong game. It is divided into three initialization steps and three steps that handle movement during gameplay. The initialization consists of drawing the yellow ball, drawing the blue paddle, and drawing the red paddle. The movement phase then involves updating the position of the yellow ball, moving the blue paddle using controls, and moving the red paddle using controls. These controls are specific keyboard keys selected by the programmer, which are continuously checked in a loop through the 'readkey()' function.

In Figure 28, there is a print screen of the Pong game.

In Figure 29, there is a picture of the Pong game, made with a photo camera, played on full screen on an LCD monitor.

Figure 25 shows the emulation of OpenRISC with Linux on a standard PC, to test the created toolchain; Figure 26 shows running the Linux operating system on the OpenRISC microprocessor created on the ATLYS development board with the Spartan-6 FPGA; Figure 28 shows a print screen captured of the Pong game running on the Linux operating

system installed on the OpenRISC platform; and Figure 29 shows a picture taken of the LCD monitor with the Pong game running on OpenRISC with Linux.

In Figure 30, there is the code diagram of the fight game.

In Figure 31, there is a print screen of the fight game.

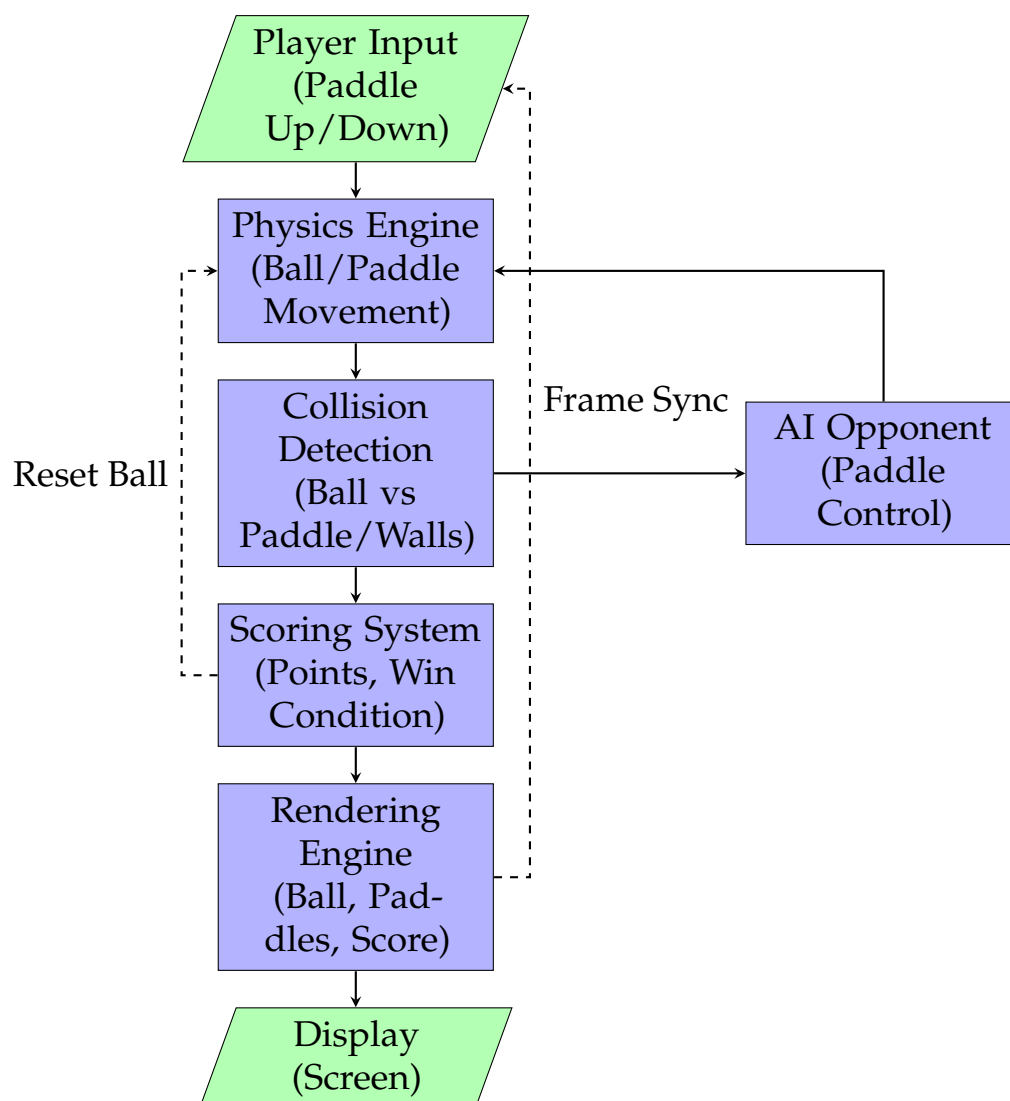


Figure 27. Code diagram of the Pong game.

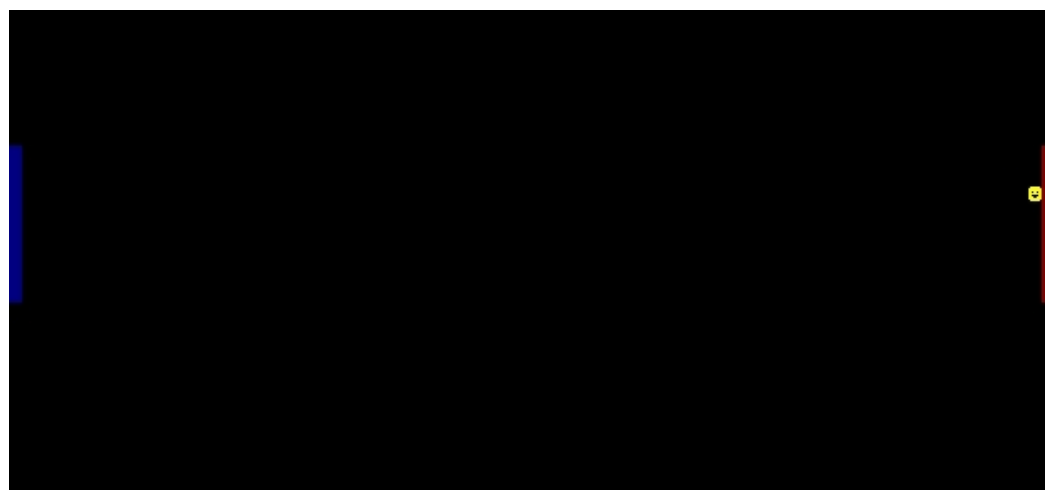


Figure 28. Print screen of the Pong game made in a higher level programming language.



Figure 29. Pong game played on full screen on an LCD monitor.

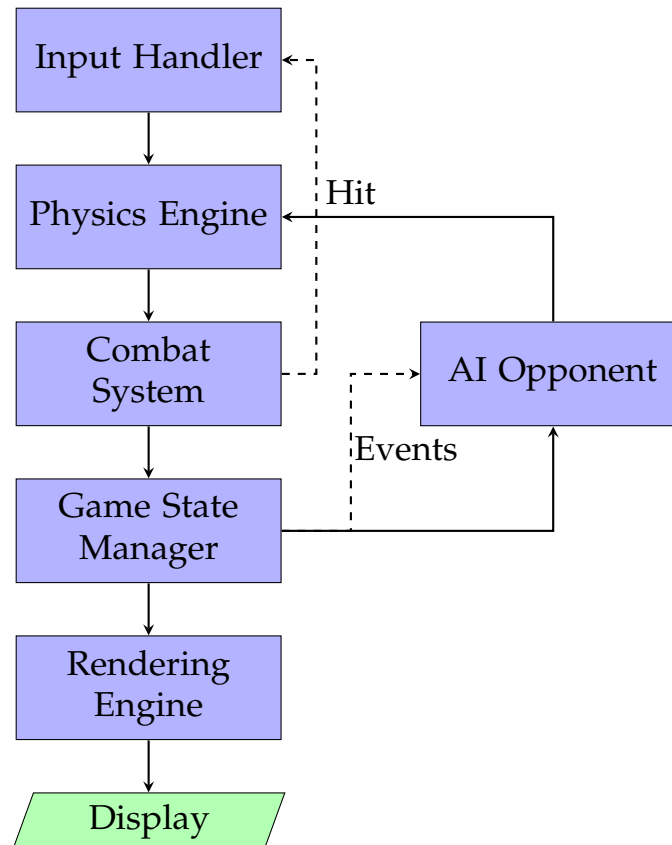
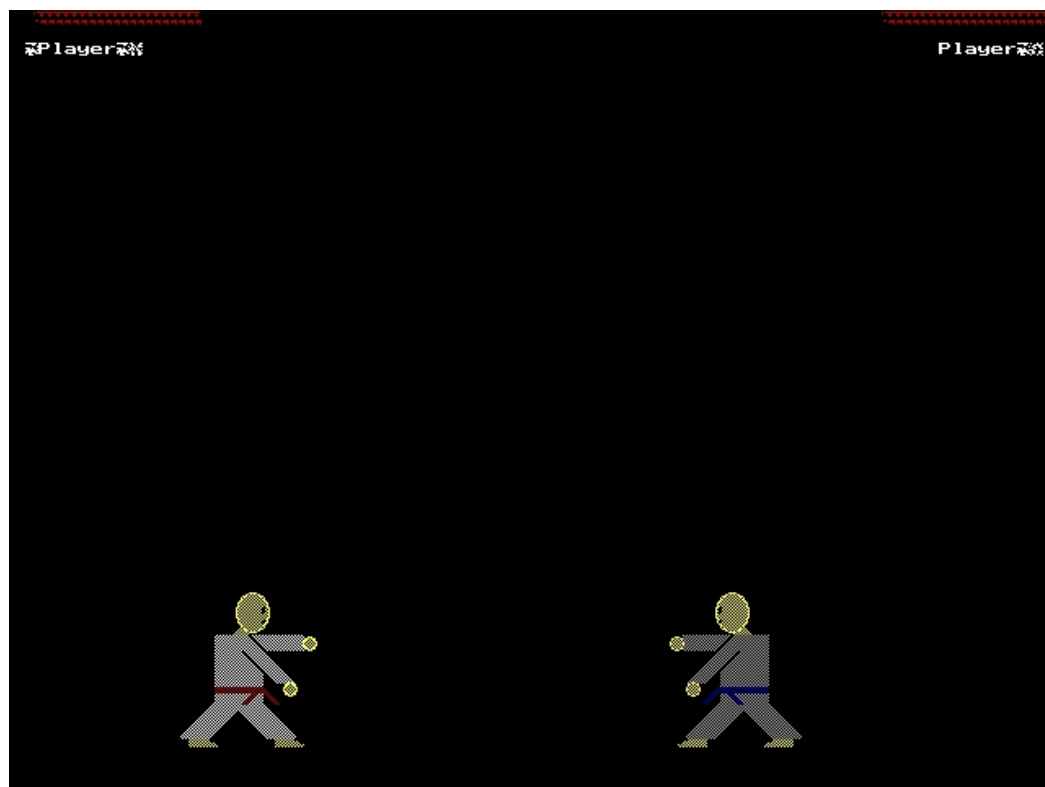
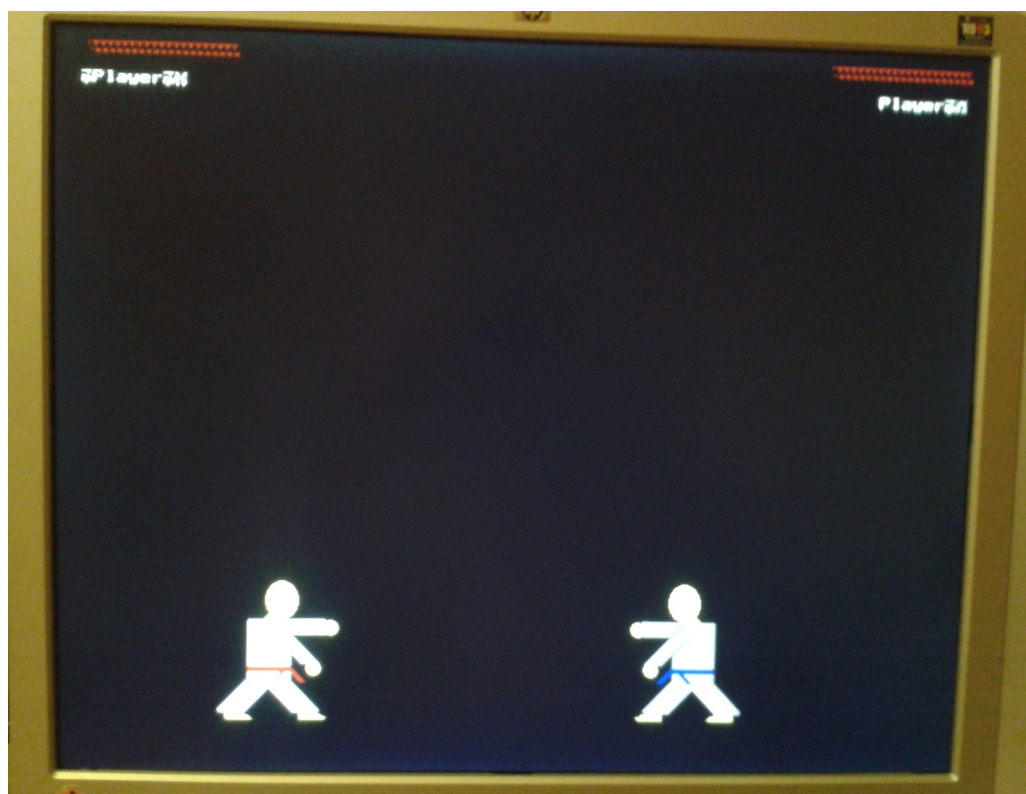


Figure 30. Code diagram of the fight game.



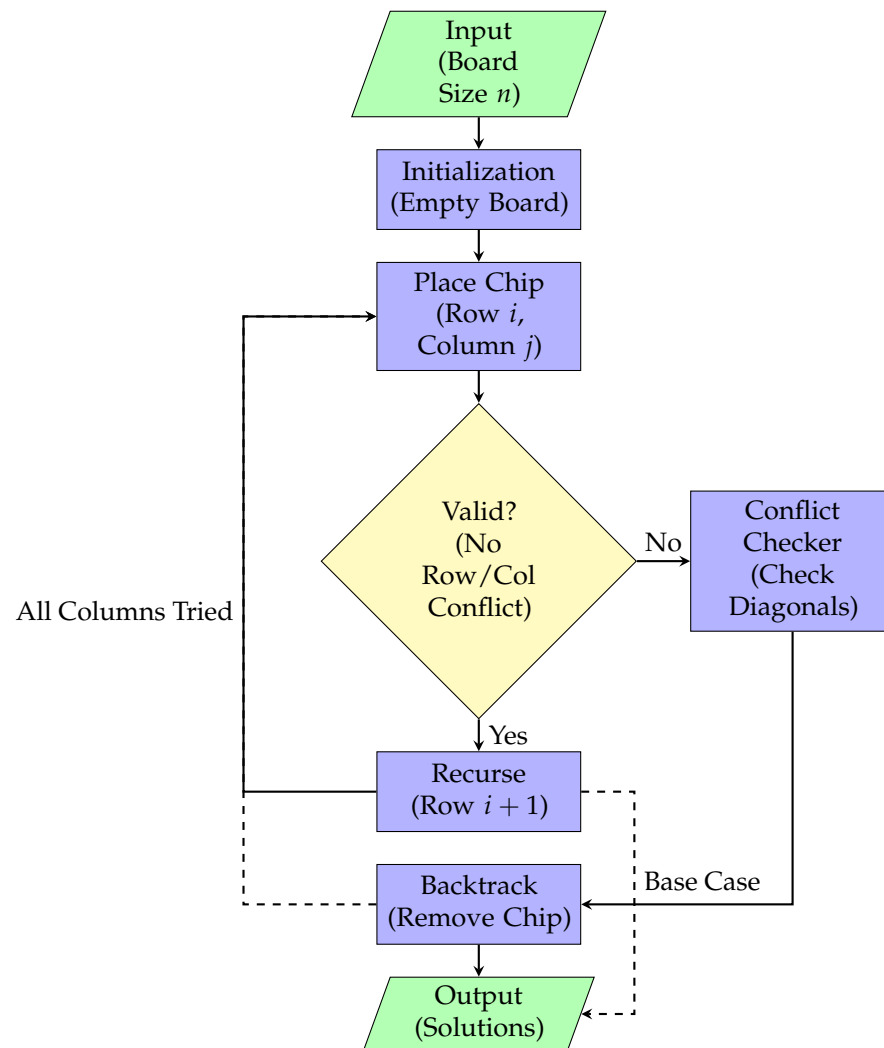
**Figure 31.** Print screen of the fight game made in a higher level programming language [21].

In Figure 32, there is a picture of the fight game, made with a photo camera, played on full screen on an LCD monitor.

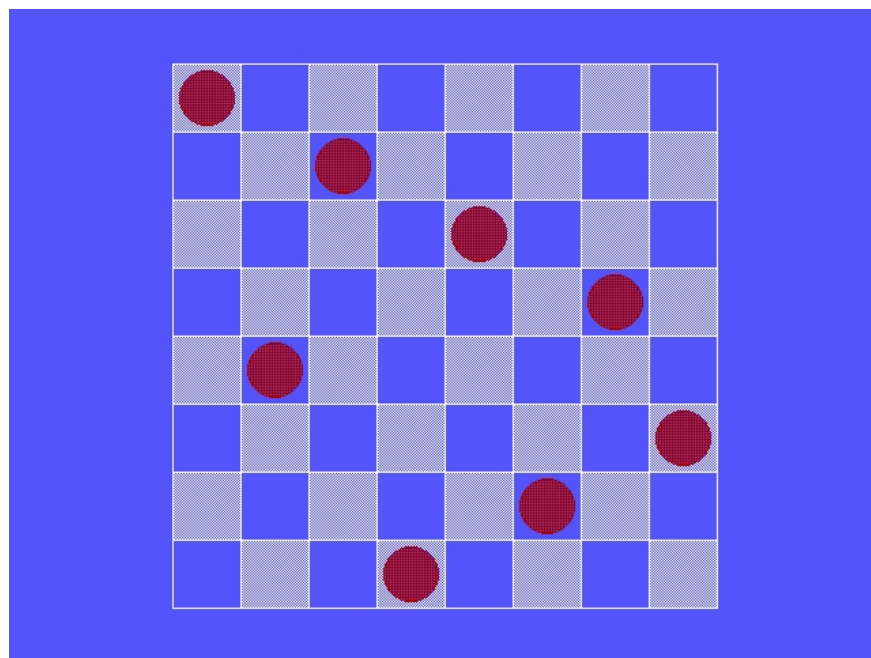


**Figure 32.** Fight game played on full screen on an LCD monitor [21].

In Figure 33, there is the code diagram of the backtracking game of chips placement.  
In Figure 34, there is a print screen of the game of placing the chips with backtracking.

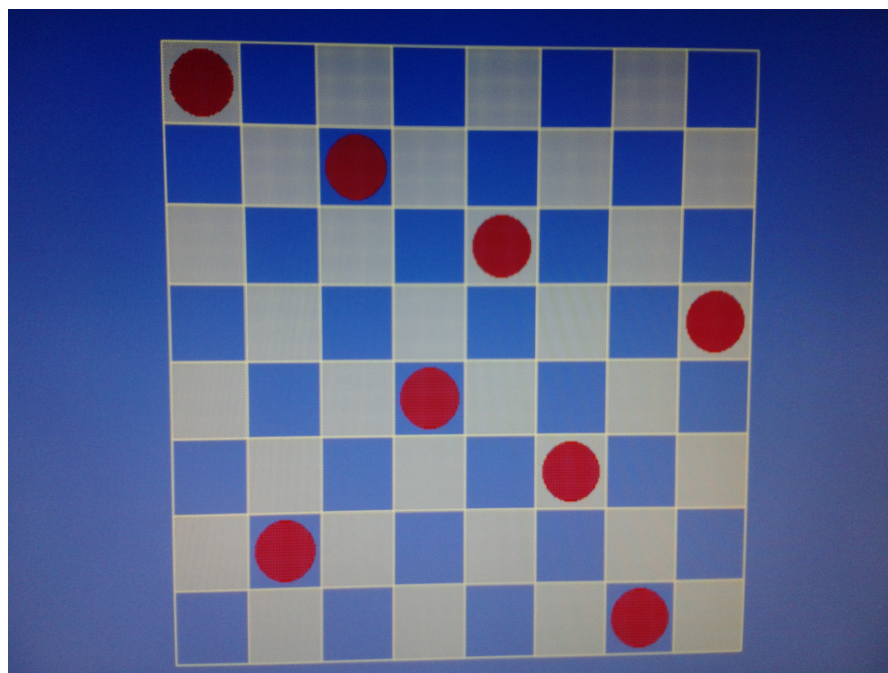


**Figure 33.** Code diagram of chips placement game with backtracking.



**Figure 34.** Print screen of the chips placement game with backtracking made in a higher level programming language.

In Figure 35, there is a picture of the backtracking game of chips placement, made with a photo camera, played on a full screen on an LCD monitor.



**Figure 35.** Chips placement game with backtracking played on full screen on an LCD monitor.

### 3.5. Statistical Analysis of the Pong Game

Next, a statistical analysis of the Pong game implemented on an FPGA will be presented. This analysis involves evaluating key aspects of the game—including the trajectory of the ball, the frequency of collisions between the ball and the paddle, and the system latency—to characterize the expected behavior using statistical techniques.

The location of the ball at time  $t$  is represented by  $\vec{r}(t) = [x(t), y(t)]$  and its velocity by  $\vec{v}(t) = [v_x(t), v_y(t)]$ . Assuming that the ball begins from a uniformly distributed position within the game area, its position over time will maintain a uniform distribution across the game field.

The position of the ball is described by the probability density function (PDF), which is written as follows:

$$f(x, y) = \frac{1}{A} \quad \text{for} \quad -W \leq x \leq W, -H \leq y \leq H \quad (43)$$

The area of the playing field with width  $2W$  and height  $2H$  is represented by  $A = 4WH$ .

The average position of the ball at any moment is

$$\mathbb{E}[\vec{r}(t)] = [\mathbb{E}[x(t)], \mathbb{E}[y(t)]] = [0, 0] \quad (44)$$

as the ball travels symmetrically around the center of the game.

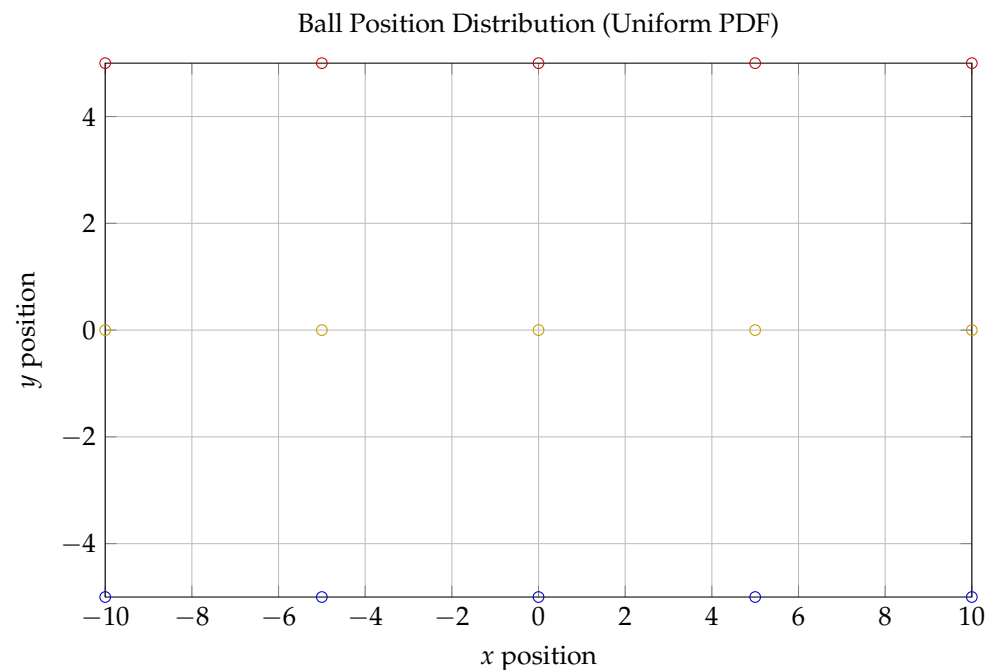
A 2D heat map can be utilized to visualize the anticipated distribution of the ball's position (Figure 36).

The probability of the ball hitting the paddle is influenced by its location and route. If the paddle is located at  $x = W$ , a collision occurs when the ball's  $x$  coordinate matches  $W$  and its  $y$  coordinate lies within the vertical range of the paddle.

Let  $P_{\text{coll}}$  indicate the probability of a collision. The likelihood of a collision is directly related to the vertical dimension of the paddle in relation to the overall height of the game.

$$P_{\text{coll}} = \frac{h_{\text{paddle}}}{2H} \quad (45)$$

where  $h_{\text{paddle}}$  represents the height of the paddle, while  $2H$  denotes the total height of the playing area.



**Figure 36.** Uniform probability density of ball position.

Assuming that the ball's velocity is steady and known, the interval between successive collisions can be represented as an exponential random variable, with the expected duration between collisions given by

$$\mathbb{E}[T_{\text{collision}}] = \frac{2W}{|\vec{v}(t)|} \quad (46)$$

Latency is the amount of time that passes between the ball making contact with the paddle and the interaction being displayed on the screen. Although this latency is typically a brief fixed duration hardware execution, variability can introduce additional delays which are frequently represented by a Gaussian distribution.

The response time  $T_{\text{response}}$  can be expressed as follows:

$$T_{\text{response}} \sim \mathcal{N}(\mu, \sigma^2) \quad (47)$$

In this case  $\sigma$  represents the standard deviation showing the variation and  $\mu$  represents the mean response time (the fixed delay).

This is the total system latency:

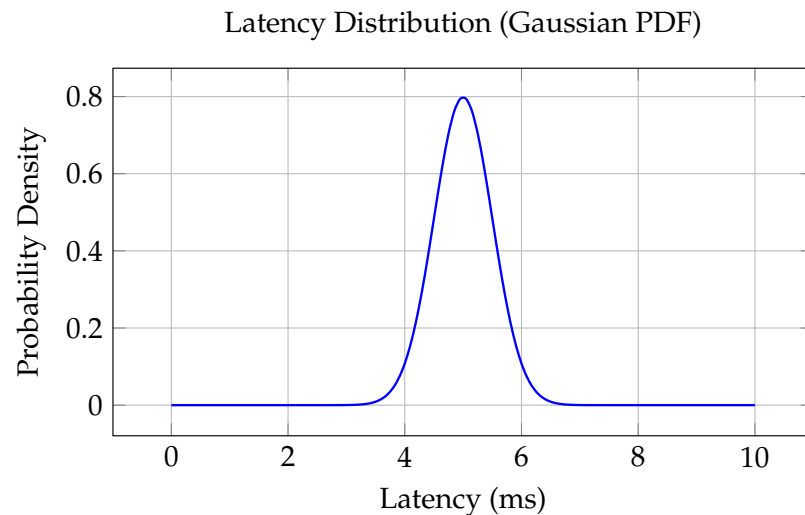
$$\text{Latency} = T_{\text{collision}} + T_{\text{response}} \quad (48)$$

A Gaussian graph can be used to illustrate the latency distribution (Figure 37).

The standard deviation shows how much the FPGA response times vary, while the mean latency is located at the center of the distribution.

This statistical analysis makes the expected behavior of the Pong game implemented on an FPGA more understandable. We can estimate the overall performance and dynamics of the game by looking at the distribution of the ball's position, the likelihood of paddle

impacts and the system's latency. Real-time responsiveness can be enhanced by using these insights to improve both the underlying hardware design and the game logic.



**Figure 37.** Latency distribution for the FPGA-based Pong game (Gaussian PDF).

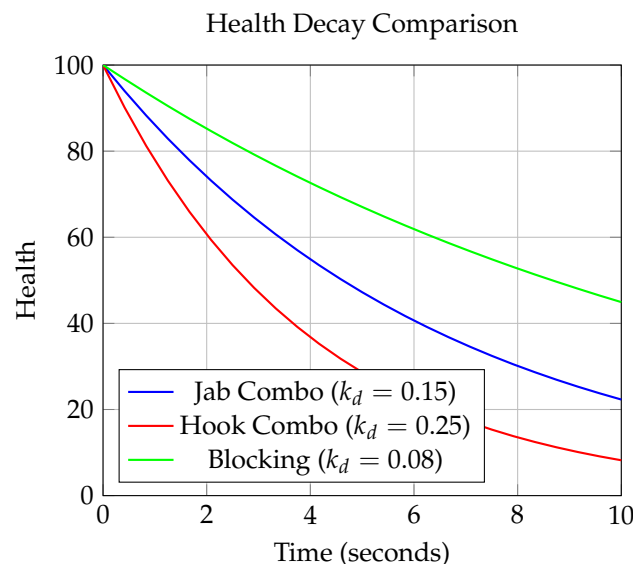
### 3.6. Statistical Analysis of the Fight Game

Health decay (Figure 38) when exposed to ongoing harm shows an exponential decline:

$$h(t) = h_0 e^{-k_d t} \quad (49)$$

where

- $h_0$  = initial health (100);
- $k_d$  = coefficient of damage decay (move-dependent);
- $t$  = time in seconds.



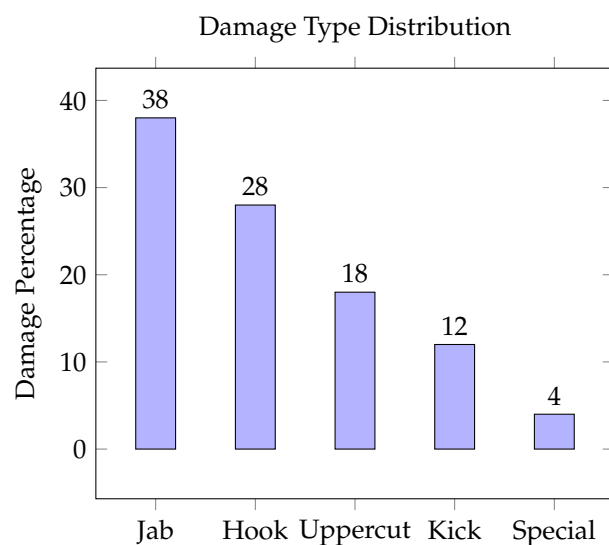
**Figure 38.** Health decay comparison in a fight game.

Hook combos make players vulnerable during the wind-up but they deliver 67% more DPS (damage per second) than jabs.

Impact (Figure 39) exhibits a Pareto distribution among various move types:

$$P(X = x) = \frac{\alpha x_m^\alpha}{x^{\alpha+1}} \quad \text{for } x \geq x_m \quad (50)$$

with  $\alpha = 1.5$ ,  $x_m = 10$  (minimum damage).



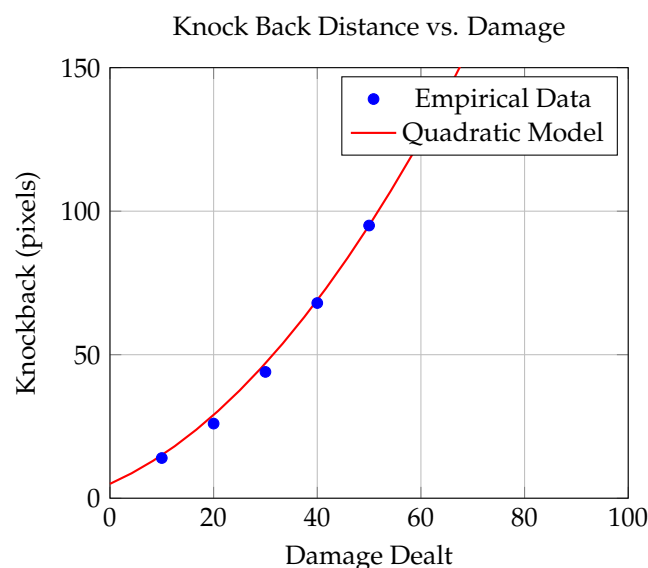
**Figure 39.** Damage type distribution in a fight game.

Fundamental strikes (Jab + Hook) account for 80% of damage, confirming the standard combat meta.

The knockback distance (Figure 40) shows that damage has a quadratic correlation:

$$k(d) = ad^2 + bd + c \quad (51)$$

where  $a = 0.02$ ,  $b = 0.8$ ,  $c = 5$  (calibrated as pixels).



**Figure 40.** Knockback distance vs. damage in a fight game.

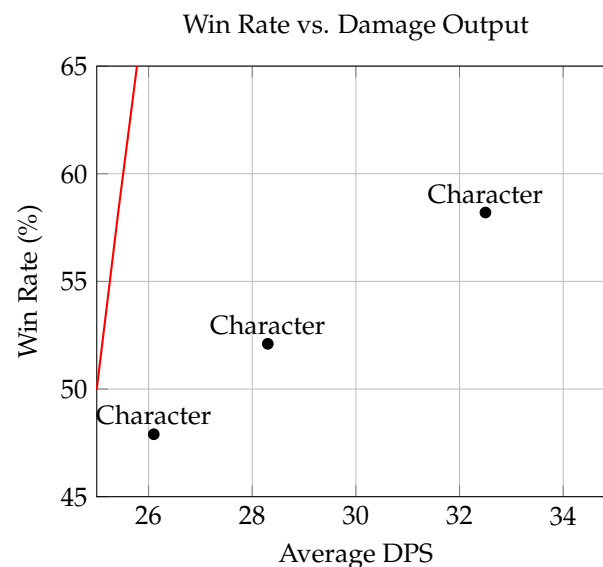
Strong attacks make character gaps much wider which facilitates the use of zoning techniques.

Each session lasts about 10 s and Table 7 displays performance results over 500 matches. Both experimental measurements and actual in-game data are the basis for the metrics which are presented as percentages. The system runs a command-line version of Linux and operates like a standard operating system. Applications run as conventional OS applications. The data were recorded as a plain text file stored on the system. The results indicate that the Boxer achieves the highest win rate, whereas the Speedster has the lowest

mean and standard deviation for DPS. A low average DPS is desirable because it implies taking less damage, and a low DPS standard deviation is also beneficial, as it shows that the values are tightly clustered around the mean, making performance stable and predictable. Therefore, the ideal fighter in the game would combine the strengths of both the Boxer and the Speedster.

The probability of win (Figure 41) follows logistic regression:

$$P(\text{Win}) = \frac{1}{1 + e^{-(0.8 \cdot \text{DPS} - 0.3 \cdot \text{Mobility} + 2.1)}} \quad (52)$$



**Figure 41.** Win rate vs. damage output in a fight game.

**Table 7.** Performance data across 500 matches.

| Character | Win Rate (%) | $\mu$ DPS | $\sigma$ DPS |
|-----------|--------------|-----------|--------------|
| Boxer     | 58.2         | 32.5      | 4.2          |
| Grappler  | 52.1         | 28.3      | 5.8          |
| Speedster | 47.9         | 26.1      | 3.5          |

The Boxer's 58% win rate exceeds the ideal 45–55% range.

The density of the player positions (Figure 42) follows a 2D Gaussian distribution:

$$f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\left(\frac{(x-\mu_x)^2}{2\sigma_x^2} + \frac{(y-\mu_y)^2}{2\sigma_y^2}\right)} \quad (53)$$

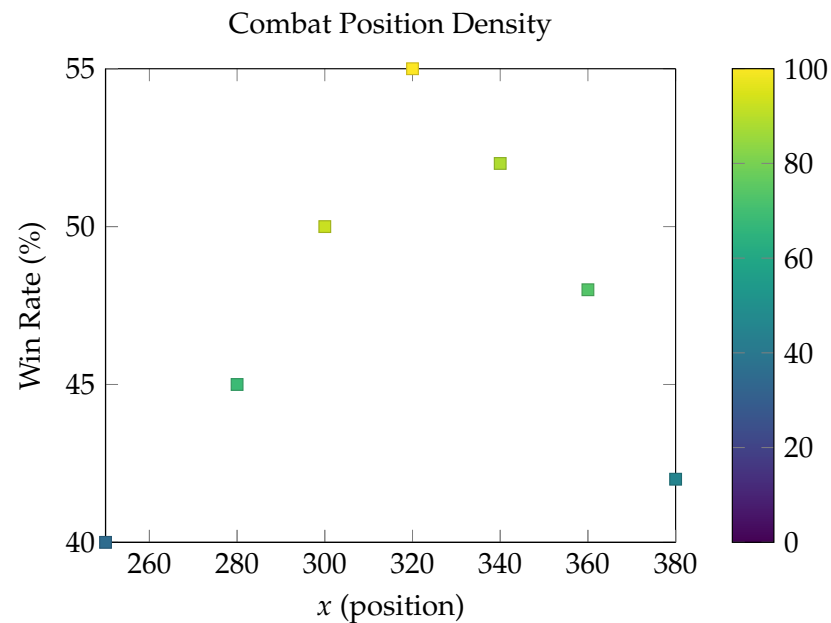
with  $\mu_x = 320$ ,  $\mu_y = 50$ ,  $\sigma_x = 80$ ,  $\sigma_y = 20$ .

A total of 72% of combat takes place in the center stage ( $300 < x < 340$ ), verifying the importance of map control.

Key statistical insights for game balancing:

- To achieve a balance between risk and reward, hook combos require a 15% reduction in damage.
- Boxer's damage per second (DPS) should be decreased to  $\mu = 30.0$  for 52% desired win rate.
- Increasing the danger around the edge of the stage can improve combat areas.

The quadratic knockback model accurately represents the physics of interactions with the environment.



**Figure 42.** Combat position density in a fight game.

### 3.7. Statistical Analysis of the Chips Placement Game with Backtracking

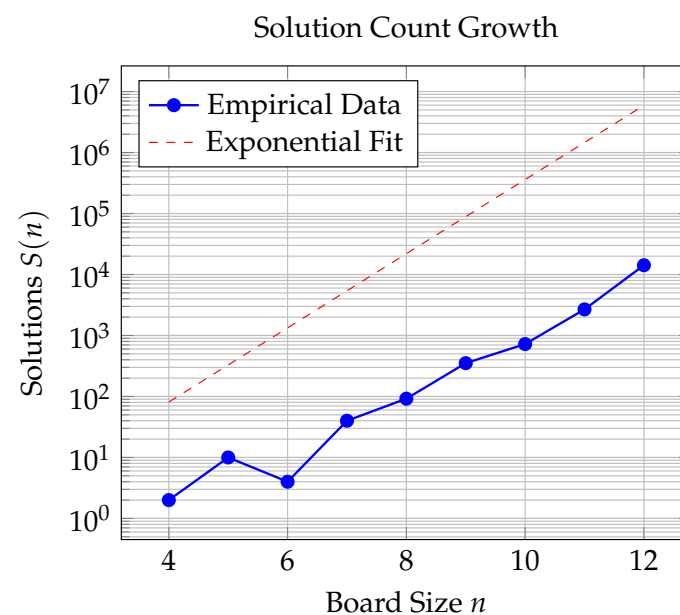
Chips placement game with backtracking or the N-Queens problem requires placing  $n$  chess queens on an  $n \times n$  board where no two queens pose a threat to each other. This can be systematically described using the backtracking method as follows:

$$\text{Valid}(i, j, B) = \bigwedge_{k=1}^{i-1} (B[k] \neq j \wedge |B[k] - j| \neq i - k) \quad (54)$$

where

- $B[k]$  = the position of the queen's column within the row  $k$ ;
- $i, j$  = Position of the candidate (row, column).

It is demonstrated that the quantity of solutions  $S(n)$  increases at a super-exponential rate, as seen in Figure 43 and in Table 8.



**Figure 43.** Solution count growth in backtracking.

The solution count follows  $S(n) \approx 0.3e^{1.4n}$  for  $n \leq 12$ , illustrating the concept of combinatorial explosion.

**Table 8.** Number of solutions growing exponentially.

| $n$ | Solutions $S(n)$ |
|-----|------------------|
| 4   | 2                |
| 8   | 92               |
| 12  | 14,200           |
| 16  | 14,772,512       |

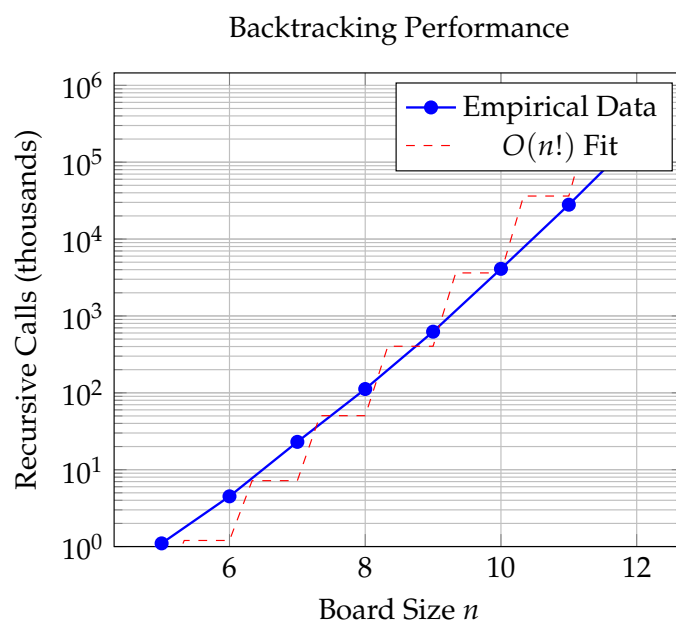
The algorithm, when pruning is considered, exhibits a time complexity of  $O(n!)$  as demonstrated in Figure 44.

$$T(n) = \sum_{i=1}^n (\text{Valid}(i, j) \cdot T(n-1)) + O(n^2) \quad (55)$$

Pruning reduces the number of calls by 40-60%, compared to using brute-force methods  $n^n$ .

For  $n = 8$ , the initial solution was discovered at varying levels of recursion, as illustrated in Figure 45.

At depth, 6.65% of the solutions are clustered, indicating that early pruning is more effective.



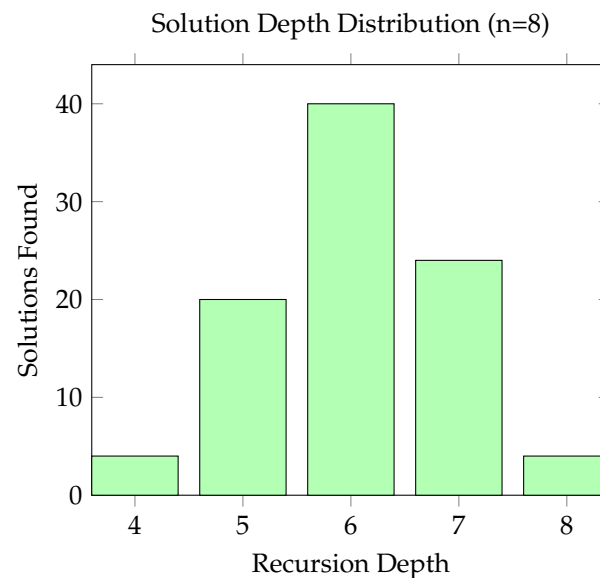
**Figure 44.** Backtracking performance for the N-Queens game.

Table 9 contrasts the usual backtracking approach with forward verification.

$$\text{Improvement Factor} = \frac{T_{\text{backtrack}}}{T_{\text{forward}}} \approx 5 \times \quad (56)$$

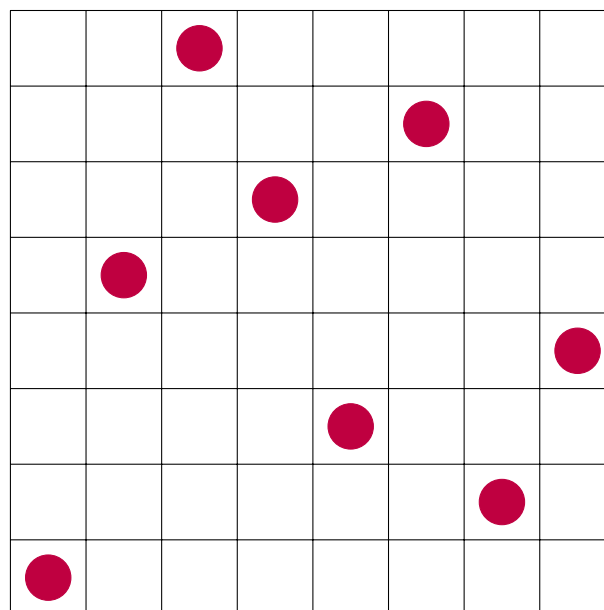
**Table 9.** Backtracking comparison with forward checking.

| Method        | Time (ms) | Calls   |
|---------------|-----------|---------|
| Backtracking  | 420       | 112,000 |
| Forward Check | 85        | 18,500  |



**Figure 45.** Solution depth distribution ( $n = 8$ ) for the N-Queens game.

When considering rotations and reflections for  $n = 8$ , there are precisely 12 distinct solutions available as shown in Figure 46.



**Figure 46.** Unique solution for  $n = 8$  for the N-Queens game.

Checking only  $\lceil n/2 \rceil$  starting from specific positions, the search space is reduced by 87.5%.

Key insights for optimizing the N-Queens problem:

- Using forward checking streamlines the time complexity by a factor of 5;
- 92% of recursive invocations occur at depths  $> \lfloor n/2 \rfloor$ ;
- Breaking symmetries reduces redundant searches by 87.5%;
- The density of the solution reaches its maximum at a recursion depth of  $n - 2$ .

#### 4. Discussion

The experiment successfully met its objective: to demonstrate that the developed games can be easily transferred to an FPGA platform and that any software code written

in a high-level programming language can be readily adapted for implementation on an FPGA.

First, a mathematical study of the Pong game was conducted, and afterwards a statistical analysis was carried out.

The objectives were to implement a microprocessor architecture in the FPGA and to install a Linux operating system on top of it. For our purposes, a command-line-based Linux system was sufficient to run the Pong game. The games we developed were cross-compiled for the Linux environment running on the OpenRISC microprocessor architecture implemented on the FPGA. In this sense, we can state that the games were ultimately ported to the FPGA platform since at the lowest level the system still relies on the FPGA. The main issue was that the selected microprocessor architecture (OpenRISC) is different from the architectures typically used in standard PCs ( $\times 86$  or  $\times 64$ ). Therefore, we cannot simply copy an already compiled executable to the FPGA platform and run it. Instead, we must cross-compile the software for our OpenRISC target and then execute it on the FPGA-based system using the OpenRISC processor and Linux. The innovative aspect lies in the fact that common Linux toolchains provide cross-compilation support for architectures such as ARM, MIPS, PPC, and SPARC, but not for OpenRISC. For OpenRISC, the toolchain must be assembled and configured from various third-party components. As a result, the cross-compiled binaries cannot run on a standard PC, but they will execute correctly on our embedded FPGA system with the OpenRISC processor and Linux. The downside is that we must either have access to the source code or develop the software ourselves for any tools we want to run on the FPGA with OpenRISC and Linux. OpenRISC was selected because of its low resource requirements, making it suitable for implementation on our Spartan-6 FPGA.

These games were selected because the author developed them entirely from scratch, using them as proof-of-concept examples to demonstrate that such software can be ported to an FPGA. Given the described toolchain, I can state that any source code can be cross-compiled and executed on FPGA. The games merely serve as demonstration cases; similarly, any other application can be cross-compiled, and thus deployed on the FPGA platform. The generated binaries specifically target the OpenRISC core and its embedded Linux distribution. With this toolchain, the source code of any application can be cross-compiled for the OpenRISC architecture. These games were ported, even though they could also be implemented directly on a conventional FPGA using VHDL or Verilog, a process that is considerably more difficult than using a high-level programming language. The purpose of this paper is to show that arbitrary applications, including more complex ones that might be prohibitively difficult to implement purely in FPGA hardware, can be ported to this FPGA platform using the OpenRISC microcontroller. The key novelty lies in the ability to port applications and games—even those not developed by me, as long as their source code is available—onto the FPGA via the OpenRISC platform, without redesigning them in hardware. Instead of recreating applications from scratch for FPGA, we can simply cross-compile them. The main objective of this paper was therefore to construct the toolchain and demonstrate its use by porting an application (the three games discussed) to FPGA. In this way, we only need to cross-compile existing applications for this platform rather than rewrite them in a hardware description language such as VHDL or Verilog. Although high-level synthesis tools can be used to mitigate the difficulty of translating code into Hardware Description Languages, this still requires more effort than simply cross-compiling existing applications for the OpenRISC platform running the Linux operating system.

The system has several limitations. It depends on source code for cross-compilation, meaning that if we want to port the application to another architecture—such as RISC-V or MicroBlaze—we must recompile the source code specifically for that target. In the current

setup, the code is compiled for OpenRISC, because the toolchain is available only for this architecture. There are no inherent performance or scalability issues with OpenRISC on a Spartan-6 FPGA, since MicroBlaze and RISC-V can also be implemented on Spartan-6. However, doing so would require substantial effort and the development of a new toolchain. Moreover, highly advanced CPUs like those used in PCs cannot realistically be implemented, because Spartan-6 is effectively limited to low-end CPU designs. As a result, applicability is restricted: on Spartan-6, one can only build CPUs that are capable of running a command-line Linux operating system, without support for graphical interfaces.

## 5. Conclusions

This article presented the development of three games: Pong, a fighting game, and a backtracking-based chip placement game. These games were initially realized in software, which represents only the first stage. The subsequent goal and planned improvement is to implement this software in hardware and migrate the games to an FPGA platform.

There are two approaches: one is to translate the high-level programming language code of the game into VHDL and run it on an FPGA board such as the NEXYS 2 from Digilent; the other is to implement a processor similar to the 8086 and load the Linux game executable onto the FPGA. If the game is ported to the FPGA by converting its high-level source code into VHDL, additional improvements and extensions are possible. Using Mentor Graphics tools, this VHDL description can then be translated into Verilog, and with the same toolchain the physical layout of the chip's silicon die can be generated. With these tools, it is possible to fabricate a standalone ASIC that has our Pong game embedded on it.

The system can be further expanded by implementing additional microprocessor architectures on the FPGA and cross-compiling various tools designed for those architectures to run on the FPGA platform. When using a more advanced FPGA board, such as the Zybo or ZedBoard, a complete SoC is already integrated, allowing these platforms to run a Linux operating system with a graphical user interface. From this point, the options become virtually limitless, as we effectively have a full-fledged computer implemented on an FPGA. If we grow tired of a particular microprocessor, we can simply replace it by reprogramming the FPGA. This means we are not restricted to just reinstalling the operating system; we can also swap out the underlying microprocessor through reconfiguration. Consequently, almost any modern Linux application with a GUI can be executed on our FPGA.

**Funding:** This research was funded by Politehnica University Timisoara, Romania.

**Data Availability Statement:** All the images are available on request from the author.

**Acknowledgments:** The author would like to thank Politehnica University Timisoara, Romania, for the given support.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Salas, J.; Vera, P.; Hernández, A.; Meza-de-Luna, M.E. An Augmented Reality Implementation of the Pong Game. *IEEE Trans. Games* **2021**, *13*, 216–227.
2. Bachiega de Almeida, T.; Carlos Pedrino, E.; Merino Fernandes, M. Complex Morphological Filtering For Serial, Parallel, GPU, SoC, PetaLinux And FPGA Execution. *IEEE Lat. Am. Trans.* **2020**, *18*, 1675–1682.
3. Chen, Y.; Tian, Y.; Zhou, R.; Martínez Castro, D.; Guo, D.; Zhou, Q. NDSTRNG: Non-Deterministic Sampling-Based True Random Number Generator on SoC FPGA Systems. *IEEE Trans. Comput.* **2024**, *73*, 1313–1326.
4. Paludo, R.; Sousa, L. NTT Architecture for a Linux-Ready RISC-V Fully-Homomorphic Encryption Accelerator. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *69*, 2669–2682.
5. Huang, Y.; Xu, D.; Tan, M.; Su, H. Adding Active Learning to LWR for Ping-Pong Playing Robot. *IEEE Trans. Control Syst. Technol.* **2013**, *21*, 1489–1494.

6. Zhu, Y.; Li, W.; Zhao, M.; Hao, J.; Zhao, D. Empirical Policy Optimization for n-Player Markov Games. *IEEE Trans. Cybern.* **2023**, *53*, 6443–6455.
7. Noveletto, F.; Soares, A.V.; Eichinger, F.L.F.; Domenech, S.C.; da S. Hounsell, M.; Bertemes Filho, P. Biomedical Serious Game System for Lower Limb Motor Rehabilitation of Hemiparetic Stroke Patients. *IEEE Trans. Neural Syst. Rehabil. Eng.* **2020**, *28*, 1481–1487.
8. Noveletto, F.; Bertemes Filho, P.; da S. Hounsell, M.; Soares, A.V. Biomedical Control Interface for a Physical Rehabilitation Serious Game. *IEEE Lat. Am. Trans.* **2016**, *14*, 38–44.
9. Livne, D.; Cohen, K. PoPS: Policy Pruning and Shrinking for Deep Reinforcement Learning. *IEEE J. Sel. Top. Signal Process.* **2020**, *14*, 789–801.
10. Chen, Z.; Barrenetxea, G.; Vetterli, M. Distributed Successive Refinement of Multiview Images Using Broadcast Advantage. *IEEE Trans. Image Process.* **2012**, *21*, 4581–4592.
11. Liu, C.C. Use of FPGAs in a Digital System Design Course with Computer Gaming Applications. In Proceedings of the American Society for Engineering Education, Salt Lake City, UT, USA, 23–27 July 2018.
12. Geier, G.; Heim, P.; Klein, F.; Finkbeiner, B. Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications. In Proceedings of the Formal Methods in Computer Aided Design, San Jose, CA, USA, 22–25 October 2019; pp. 138–146.
13. Van Acker, O.; Lachish, O.; Burnett, G. Cellular Automata Simulation on FPGA for Training Neural Networks with Virtual World Imagery. In Proceedings of the IEEE Conference on Computational Intelligence and Games, New York, NY, USA, 22–25 August 2017; pp. 304–305.
14. Kachkachi, N.; Gansmuller, A.; Rabah, H. Design and Experimental Validation of a SoC-FPGA-Based Compact NQR Spectrometer. *IEEE Trans. Instrum. Meas.* **2024**, *73*, 2003012.
15. Cousins, D.B.; Rohloff, K.; Sumorok, D. Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor. *IEEE Trans. Emerg. Top. Comput.* **2017**, *5*, 193–206.
16. Eguzo, C.; Scherer, B.; Keßel, D.; Bekman, I.; Streun, M.; Schlosser, M.; van Waasen, S. On Automating FPGA Design Build Flow Using GitLab CI. *IEEE Embed. Syst. Lett.* **2024**, *16*, 227–230.
17. Ormaetxea, E.; Andreu, J.; Kortabarria, I.; Bidarte, U.; Martínez de Alegría, M.; Ibarra, E.; Olaguenaga, E. Matrix Converter Protection and Computational Capabilities Based on a System on Chip Design with an FPGA. *IEEE Trans. Power Electron.* **2011**, *26*, 272–287.
18. Szabo, R.; Gontean, A. Pong Game on FPGA with CRT or LCD Display and Push Button Controls. In Proceedings of the Federated Conference on Computer Science and Information Systems, Warsaw, Poland, 7–10 September 2014; pp. 735–740.
19. Digilent. Available online: <https://digilent.com/reference/programmable-logic/atlys/start> (accessed on 10 July 2025).
20. Szabo, R.; Gontean, A. Image Acquisition with Linux on FPGA. In Proceedings of the Telecommunications Forum, Belgrade, Serbia, 25–27 November 2014; pp. 1007–1010.
21. Szabo, R.; Gontean, A. Creation of a fight game in Borland Pascal with the possibility to be ported on an FPGA. In Proceedings of the 22nd International Symposium for Design and Technology in Electronic Packaging, Oradea, Romania, 20–23 October 2016; pp. 122–125.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.