NUST
NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY

NCDC
NUST CHIP DESIGN CENTRE

School of Interdisciplinary Engineering and Sciences

In Collaboration with

NUST Chip Design Centre (NCDC)

From Logic Gates to Linux

Implementation of a RISC-V
System-on-Chip on FPGA

End Semester Project Report

Bachelor of Science in Bioinformatics

Submitted by:

Asim Ahmed

Shawaiz Zafar

Abdullah Mustafa

Supervised by:

Tanvyr Ahmed

December 2025

Islamabad, Pakistan

Abstract

This project demonstrates the complete implementation of a Linux-capable computer system built from scratch using Field Programmable Gate Array (FPGA) technology. We designed and synthesized a RISC-V processor core, integrated it into a System-on-Chip (SoC) architecture, and successfully booted the Linux operating system on the custom hardware.

The project uses the open-source RISC-V instruction set architecture, specifically the RV32IMA variant, which includes integer, multiplication, and atomic operations required for Linux. The VexRiscv soft-core processor was configured with a Memory Management Unit (MMU), instruction and data caches, and supervisor mode support.

The SoC was implemented on a Digilent Nexys A7-100T development board featuring a Xilinx Artix-7 FPGA. The system includes 128 MB of DDR2 RAM, SD card storage, and UART serial communication. The LiteX framework was used for SoC integration, while Buildroot provided the Linux kernel, OpenSBI firmware, and BusyBox userland.

The final system boots Linux 6.9.0 in approximately 11 seconds and provides a fully functional Unix environment. Users can execute shell commands, run compiled C programs, and interact with the system through a serial console.

This project demonstrates that undergraduate students can design and implement complete computer systems using open-source tools. The work provides a foundation for custom hardware accelerators, domain-specific processors, and hardware security research.

Keywords: FPGA, RISC-V, Linux, System-on-Chip, VexRiscv, LiteX, Embedded Systems, Computer Architecture

Acknowledgements

We express our sincere gratitude to our supervisor Tanvyr Ahmed for his invaluable guidance, mentorship, and support throughout this project. His expertise in FPGA design and embedded systems was instrumental in helping us navigate the complexities of hardware development.

We extend our heartfelt thanks to the **School of Interdisciplinary Engineering and Sciences (SINES)** and the **NUST Chip Design Centre (NCDC)** for providing us with this incredible opportunity to work on cutting-edge FPGA technology. The resources, laboratory facilities, and development boards made available by NCDC were essential for the successful completion of this project. We are grateful for their commitment to fostering undergraduate research and hands-on learning in hardware design.

We acknowledge the open-source community, particularly the developers of LiteX, VexRiscv, Buildroot, and the Linux kernel. Their dedication to creating freely available, high-quality tools has democratized access to advanced hardware design and made projects like ours possible.

We thank our families for their unwavering support, patience, and encouragement during the challenging phases of this project.

Finally, we thank our classmates who provided feedback, motivation, and constructive criticism throughout the development process.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

Modern computers are complex systems that span multiple layers of abstraction. At the lowest level, transistors switch electrical signals. These transistors form logic gates, which combine into arithmetic units, registers, and control logic. Together, these components form a Central Processing Unit (CPU). The CPU connects to memory, storage, and peripherals through buses and controllers. Software runs on this hardware, from firmware and operating systems to user applications.

Most computer science education focuses on software development. Students learn programming languages, algorithms, and data structures. However, few students understand how the underlying hardware actually works. The hardware appears as a "black box" that magically executes code.

This project opens that black box. We designed and built a complete computer system from the logic gate level to a running Linux operating system. This end-to-end implementation provides deep understanding of computer architecture that cannot be gained from textbooks alone.

1.2 Problem Statement

Several challenges motivate this project:

1. **Educational Gap:** Computer science graduates often lack understanding of hardware-software interaction. They cannot debug low-level problems or optimize systems effectively.

2. **Vendor Lock-in:** Commercial processors from Intel, AMD, and ARM are closed-source. Users cannot inspect, modify, or verify their operation.

3. **Licensing Costs:** ARM charges significant licensing fees for processor designs. This creates barriers for small companies and developing countries.

4. **Security Concerns:** Closed-source processors may contain hidden backdoors or vulnerabilities. Users cannot audit the hardware.

5. **Customization Limitations:** General-purpose processors include many features not needed for specific applications. Custom processors could be more efficient.

1.3 Objectives

This project aims to achieve the following objectives:

1. Design and implement a RISC-V processor capable of running Linux.
2. Build a complete System-on-Chip with memory controller, serial communication, and storage interfaces.
3. Synthesize the design onto an FPGA development board.
4. Configure and build a Linux kernel for the custom hardware.
5. Boot Linux and demonstrate a functional Unix environment.
6. Document the process for reproducibility by others.

1.4 Scope

The project scope includes:

- Implementation of a 32-bit RISC-V processor (RV32IMA).
- Integration of DDR2 memory controller, UART, SD card interface, timer, and interrupt controller.
- Linux kernel 6.x with BusyBox userland.
- Serial console interface at 115200 baud.
- Documentation of all steps, issues, and solutions.

The project does not include:

- Original CPU design from scratch (we use the open-source VexRiscv).
- Graphical user interface (text-mode console only).
- Network connectivity (Ethernet disabled to reduce complexity).
- Multi-core implementation (single-core for simplicity).

1.5 Report Organization

This report is organized as follows:

- **Chapter 2** provides background on FPGAs, RISC-V architecture, and related concepts.
- **Chapter 3** reviews existing work and similar projects.
- **Chapter 4** documents Phase I: the VGA Graphics Accelerator and Clock project.
- **Chapter 5** describes Phase II: the Linux system design and architecture.
- **Chapter 6** details the Linux implementation process step by step.
- **Chapter 7** presents results and performance measurements.
- **Chapter 8** discusses challenges encountered and their solutions.
- **Chapter 9** concludes the report and suggests future work.

Chapter 2

Background and Theory

2.1 Digital Logic Fundamentals

2.1.1 Logic Gates

Digital computers process information using binary signals. A signal is either HIGH (logic 1, typically 3.3V or 5V) or LOW (logic 0, typically 0V). Logic gates perform basic operations on these signals.

The fundamental gates are:

- **NOT gate:** Inverts the input. Output is 1 if input is 0, and vice versa.
- **AND gate:** Output is 1 only if all inputs are 1.
- **OR gate:** Output is 1 if any input is 1.
- **XOR gate:** Output is 1 if inputs are different.

Any digital circuit can be built using combinations of these basic gates. This is the foundation of all computer hardware.

2.1.2 Combinational and Sequential Logic

Combinational logic produces outputs that depend only on current inputs. Examples include adders, multiplexers, and decoders. The output changes immediately when inputs change.

Sequential logic includes memory elements that store state. The output depends on both current inputs and previous state. Flip-flops are the basic sequential elements. They store one bit of information and update on clock edges.

A CPU combines both types of logic. Combinational logic performs arithmetic and data routing. Sequential logic stores registers, program counter, and pipeline state.

2.2 Field Programmable Gate Arrays

2.2.1 FPGA Architecture

A Field Programmable Gate Array (FPGA) is an integrated circuit that can be configured after manufacturing. Unlike Application-Specific Integrated Circuits (ASICs), FPGAs can be reprogrammed to implement any digital circuit.

FPGAs contain three main components:

1. **Configurable Logic Blocks (CLBs):** Each CLB contains Look-Up Tables (LUTs), flip-flops, and multiplexers. LUTs can implement any Boolean function of their inputs. A 4-input LUT stores a 16-entry truth table.

2. **Programmable Interconnect:** A network of wires and switches connects CLBs. The routing configuration determines which CLBs connect to each other.

3. **Input/Output Blocks (IOBs):** Interface between internal logic and external pins. IOBs support various voltage standards and signal types.

Modern FPGAs also include dedicated hardware blocks:

- Block RAM (BRAM): Fast on-chip memory.
- DSP slices: Hardware multipliers and accumulators.
- Clock management: PLLs and clock buffers.
- High-speed transceivers: For serial communication.

2.2.2 FPGA Development Flow

The FPGA development process involves several steps:

1. **Design Entry:** Write hardware description in Verilog or VHDL. These languages describe the structure and behavior of digital circuits.

2. **Synthesis:** Convert HDL code into a netlist of logic gates. The synthesis tool maps the design to the target FPGA's primitives.

3. **Implementation:** Place gates onto specific FPGA locations and route connections between them. This step considers timing constraints.

4. **Bitstream Generation:** Create a binary file that programs the FPGA. The bitstream contains configuration data for all CLBs, routing, and IOBs.

5. **Programming:** Load the bitstream into the FPGA. The device configures itself according to the bitstream.

2.2.3 Nexys A7-100T Development Board

This project uses the Digilent Nexys A7-100T development board. Key specifications include:

Table 2.1: Nexys A7-100T Specifications

Component	**Specification**
FPGA	Xilinx Artix-7 XC7A100T
Logic Cells	101,440
Look-Up Tables	63,400
Flip-Flops	126,800
Block RAM	4.8 Mb
DSP Slices	240
External Memory	128 MB DDR2
Storage	MicroSD card slot
Communication	USB-UART, VGA, Ethernet
Clock	100 MHz oscillator

2.3 RISC-V Architecture

2.3.1 Overview

RISC-V is an open-source Instruction Set Architecture (ISA) developed at UC Berkeley starting in 2010. Unlike proprietary ISAs such as x86 (Intel/AMD) and ARM, RISC-V is freely available for anyone to implement.

RISC-V follows the Reduced Instruction Set Computer (RISC) philosophy:

- Simple, fixed-length instructions.
- Load-store architecture (memory access only through load/store instructions).
- Large register file (32 general-purpose registers).
- Simple addressing modes.

2.3.2 ISA Extensions

RISC-V uses a modular approach. The base integer ISA is minimal. Extensions add functionality:

Table 2.2: RISC-V ISA Extensions

Extension	Name	Description
I	Integer	Base integer instructions (ADD, SUB, AND, OR, branches, loads, stores)
M	Multiply	Integer multiplication and division
A	Atomic	Atomic memory operations (required for multi-threading and Linux)
F	Float	Single-precision floating point
D	Double	Double-precision floating point
C	Compressed	16-bit compressed instructions

This project uses RV32IMA: 32-bit base integer with multiply and atomic extensions. The atomic extension is required for Linux because the kernel uses atomic operations for synchronization.

2.3.3 Privilege Levels

RISC-V defines three privilege levels:

1. **Machine Mode (M):** Highest privilege. Direct access to all hardware. Firmware runs here.
2. **Supervisor Mode (S):** Operating system kernel runs here. Can access virtual memory and most system resources.
3. **User Mode (U):** Lowest privilege. Application code runs here. Cannot directly access hardware.

Linux requires supervisor mode with virtual memory support. The Memory Management Unit (MMU) translates virtual addresses to physical addresses and enforces memory protection.

2.3.4 VexRiscv Processor

VexRiscv is a soft-core RISC-V processor written in SpinalHDL, a Scala-based hardware description language. Key features include:

- Configurable pipeline stages (2 to 5 stages).
- Optional MMU with Sv32 virtual memory.
- Optional instruction and data caches.
- Optional debug support.
- Highly optimized for FPGA implementation.

The Linux variant of VexRiscv includes:

- 5-stage pipeline (Fetch, Decode, Execute, Memory, WriteBack).
- 4 KB instruction cache.
- 4 KB data cache.
- Sv32 MMU (32-bit virtual addresses, 4 KB pages).
- Machine and Supervisor privilege modes.

2.4 System-on-Chip Architecture

2.4.1 SoC Components

A System-on-Chip integrates all computer components onto a single chip:

- **CPU:** Executes instructions.
- **Memory Controller:** Interfaces with external RAM.
- **Bus:** Connects components.
- **Peripherals:** UART, timers, interrupt controllers, etc.
- **Boot ROM:** Initial startup code.

2.4.2 Wishbone Bus

The Wishbone bus is an open-source interconnect standard for SoC designs. It defines signals for address, data, control, and handshaking. All components in our SoC communicate through Wishbone.

Key Wishbone signals include:

- `ADR`: Address bus.
- `DAT_I`, `DAT_O`: Data input and output.
- `WE`: Write enable.
- `SEL`: Byte select.
- `STB`: Strobe (valid transfer).
- `CYC`: Bus cycle.
- `ACK`: Acknowledge.

2.4.3 LiteX Framework

LiteX is a Python-based framework for building SoCs. It provides:

- Pre-built CPU cores (VexRiscv, Rocket, etc.).
- Memory controllers (DDR, DDR2, DDR3, DDR4).
- Peripheral cores (UART, SPI, I2C, Ethernet, etc.).
- Bus interconnect generation.
- BIOS and boot code.
- Build system integration with vendor tools.

LiteX generates Verilog code from Python descriptions. This enables rapid SoC development without manual HDL coding.

2.5 Linux Operating System

2.5.1 Kernel Architecture

The Linux kernel is a monolithic operating system kernel. It includes:

- **Process Management:** Creates, schedules, and terminates processes.
- **Memory Management:** Virtual memory, page tables, memory allocation.
- **File Systems:** VFS layer, various filesystem implementations.
- **Device Drivers:** Hardware abstraction for peripherals.
- **Networking:** TCP/IP stack and protocols.

2.5.2 Boot Process

Linux boot on RISC-V follows this sequence:

1. **BIOS/Bootloader:** Initializes hardware, loads kernel to RAM.
2. **OpenSBI:** RISC-V firmware providing runtime services.
3. **Kernel:** Initializes subsystems, mounts root filesystem.
4. **Init:** First userspace process, starts services.
5. **Shell:** User interaction begins.

2.5.3 Device Tree

The Device Tree is a data structure describing hardware. Linux reads the device tree at boot to learn about available hardware. This allows one kernel binary to run on different hardware configurations.

The device tree specifies:

- CPU type and count.
- Memory size and location.
- Peripheral addresses and interrupts.
- Bus topology.

2.5.4 Buildroot

Buildroot is a tool for building embedded Linux systems. It automates:

- Cross-compiler toolchain generation.
- Kernel configuration and compilation.
- Root filesystem creation.
- Package selection and building.

Buildroot produces all files needed to boot Linux: kernel image, device tree blob, and root filesystem.

Chapter 3

Literature Review

3.1 RISC-V Implementations

Several open-source RISC-V implementations exist:

3.1.1 Rocket Chip

Rocket Chip is the reference RISC-V implementation from UC Berkeley. It is written in Chisel, another Scala-based HDL. Rocket Chip targets high performance and is used in commercial products. However, it requires significant FPGA resources.

3.1.2 PicoRV32

PicoRV32 is a minimal RISC-V core optimized for size. It implements RV32I only, without MMU or caches. PicoRV32 is suitable for microcontroller applications but cannot run Linux.

3.1.3 SERV

SERV is the smallest RISC-V core, using a bit-serial architecture. It processes one bit per cycle, achieving minimal resource usage at the cost of performance. SERV is useful for understanding RISC-V basics.

3.1.4 VexRiscv

VexRiscv offers the best balance of performance, features, and resource usage for FPGA implementations. It is highly configurable, allowing users to select exactly the features needed. The Linux variant includes all features required for operating system support.

3.2 Linux on FPGA Projects

3.2.1 Linux-on-LiteX-VexRiscv

The linux-on-litex-vexriscv project by the LiteX developers provides a complete solution for running Linux on VexRiscv. It includes build scripts, Buildroot configurations, and documentation. This project served as the foundation for our work.

3.2.2 LiteX-Boards

The litex-boards repository contains board support packages for many FPGA development boards. Each package defines pin mappings, clock frequencies, and peripheral configurations. Support for the Nexys A7 is included.

3.3 Alternative Approaches

3.3.1 Commercial Solutions

Companies like SiFive offer commercial RISC-V cores with Linux support. These provide better performance and support but require licensing fees. Our open-source approach achieves similar functionality at no cost.

3.3.2 ARM-based Systems

ARM processors dominate the embedded Linux market. Raspberry Pi and similar boards provide easy Linux environments. However, ARM requires licensing, and users cannot inspect or modify the processor design.

3.4 Contribution of This Work

While tools and components exist, comprehensive documentation for reproducing a Linux-on-FPGA system is lacking. This project contributes:

1. Step-by-step instructions for the complete build process.
2. Documentation of common problems and solutions.
3. Explanation of all components and their interactions.
4. A foundation for further customization and research.

Chapter 4

Phase I: VGA Graphics Accelerator

This chapter documents the first phase of our FPGA project. We implemented a hardware-accelerated VGA graphics system before progressing to the Linux system. This phase taught us essential skills in FPGA debugging, hardware design, and VGA signal generation.

4.1 Project Origin and Migration

4.1.1 Original Repository

The project began as an attempt to replicate a Vector Clock design from GitHub. The original repository was designed for the Digilent Basys 3 board. We needed to migrate it to the Nexys A7-100T platform.

Original repository structure:

- Source code fragmented across multiple "Lab" folders.
- Pin constraints hard-coded for Basys 3 FPGA.
- PicoBlaze soft-core processor for control logic.
- UART interface for time setting commands.

4.1.2 Migration Challenges

The migration presented several challenges:

1. Different FPGA pinout between Basys 3 and Nexys A7.
2. Source files scattered across multiple directories.
3. Inconsistent naming conventions in original code.
4. Missing documentation for build process.

4.2 Project Setup and File Organization

4.2.1 Development Environment

Required software:

- Xilinx Vivado 2019.1 or later.
- Git for version control.
- Linux terminal (or WSL on Windows).
- PuTTY or similar serial terminal.

4.2.2 File Aggregation Process

We used Linux terminal commands to extract and organize source files.

```
# 1. Create a clean project directory
mkdir Nexys_Clock_Project
cd Nexys_Clock_Project

# 2. Clone the original repository
git clone https://github.com/muhammadaldacher/FPGA-Design-of-a-Digital-
    Analog-Clock-Display-using-Digilent-Basys3-Artix-7.git
mv FPGA-Design-of-a-Digital-Analog-Clock-Display-using-Digilent-Basys3-
    Artix-7 original_repo

# 3. Create source directory
mkdir src
cd src

# 4. Extract Verilog files from Lab folders
find ../original_repo -name "*.v" -exec cp {} . \;

# 5. Remove unnecessary files (testbenches, examples)
rm -f Testbench.v Constraints.v Design.v tutorial.v vga_example.v

# 6. Fix naming convention issues
# Original repo has software.v but hardware expects program.v
mv software.v program.v

# 7. List final source files
ls -la *.v
```

Listing 4.1: Project Setup Commands

4.2.3 Required Source Files

After extraction, the following files are needed:

Table 4.1: VGA Project Source Files

File	Lines	Purpose
project.v	150+	Top-level module and state machine
vga_timing.v	40	VGA sync signal generator
linedraw.v	80	Bresenham line drawing engine
framebuffer.v	20	Dual-port video memory
quad_seven_seg.v	60	7-segment display driver
Nexys_Master.xdc	80	Pin constraint file

4.3 Pin Constraints for Nexys A7

The constraint file maps logic signals to physical FPGA pins. This file is critical for correct hardware operation.

```
## Clock Signal (100 MHz System Clock)
set_property -dict { PACKAGE_PIN E3 IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin -period 10.00 \
    -waveform {0 5} [get_ports {clk}];

## Reset Button (Active Low)
set_property -dict { PACKAGE_PIN C12 IOSTANDARD LVCMOS33 } [get_ports { rst_n }];

## UART Interface (USB-Serial Bridge)
set_property -dict { PACKAGE_PIN C4 IOSTANDARD LVCMOS33 } [get_ports { rxd }];
set_property -dict { PACKAGE_PIN D4 IOSTANDARD LVCMOS33 } [get_ports { txd }];

## VGA Connector - Red Channel (4 bits)
set_property -dict { PACKAGE_PIN A3 IOSTANDARD LVCMOS33 } [get_ports { vgaRed[0] }];
set_property -dict { PACKAGE_PIN B4 IOSTANDARD LVCMOS33 } [get_ports { vgaRed[1] }];
set_property -dict { PACKAGE_PIN C5 IOSTANDARD LVCMOS33 } [get_ports { vgaRed[2] }];
set_property -dict { PACKAGE_PIN A4 IOSTANDARD LVCMOS33 } [get_ports { vgaRed[3] }];

## VGA Connector - Green Channel (4 bits)
```

```
set_property -dict { PACKAGE_PIN C6 IOSTANDARD LVCMOS33 } [get_ports {
    vgaGreen[0] }];
set_property -dict { PACKAGE_PIN A5 IOSTANDARD LVCMOS33 } [get_ports {
    vgaGreen[1] }];
set_property -dict { PACKAGE_PIN B6 IOSTANDARD LVCMOS33 } [get_ports {
    vgaGreen[2] }];
set_property -dict { PACKAGE_PIN A6 IOSTANDARD LVCMOS33 } [get_ports {
    vgaGreen[3] }];

## VGA Connector - Blue Channel (4 bits)
set_property -dict { PACKAGE_PIN B7 IOSTANDARD LVCMOS33 } [get_ports {
    vgaBlue[0] }];
set_property -dict { PACKAGE_PIN C7 IOSTANDARD LVCMOS33 } [get_ports {
    vgaBlue[1] }];
set_property -dict { PACKAGE_PIN D7 IOSTANDARD LVCMOS33 } [get_ports {
    vgaBlue[2] }];
set_property -dict { PACKAGE_PIN D8 IOSTANDARD LVCMOS33 } [get_ports {
    vgaBlue[3] }];

## VGA Sync Signals
set_property -dict { PACKAGE_PIN B11 IOSTANDARD LVCMOS33 } [get_ports {
    h_sync }];
set_property -dict { PACKAGE_PIN B12 IOSTANDARD LVCMOS33 } [get_ports {
    v_sync }];

## 7-Segment Display - Segments
set_property -dict { PACKAGE_PIN T10 IOSTANDARD LVCMOS33 } [get_ports { seg
    [0] }];
set_property -dict { PACKAGE_PIN R10 IOSTANDARD LVCMOS33 } [get_ports { seg
    [1] }];
set_property -dict { PACKAGE_PIN K16 IOSTANDARD LVCMOS33 } [get_ports { seg
    [2] }];
set_property -dict { PACKAGE_PIN K13 IOSTANDARD LVCMOS33 } [get_ports { seg
    [3] }];
set_property -dict { PACKAGE_PIN P15 IOSTANDARD LVCMOS33 } [get_ports { seg
    [4] }];
set_property -dict { PACKAGE_PIN T11 IOSTANDARD LVCMOS33 } [get_ports { seg
    [5] }];
set_property -dict { PACKAGE_PIN L18 IOSTANDARD LVCMOS33 } [get_ports { seg
    [6] }];
set_property -dict { PACKAGE_PIN H15 IOSTANDARD LVCMOS33 } [get_ports { dp
    }];

## 7-Segment Display - Digit Anodes
set_property -dict { PACKAGE_PIN J17 IOSTANDARD LVCMOS33 } [get_ports { an
    [0] }];
set_property -dict { PACKAGE_PIN J18 IOSTANDARD LVCMOS33 } [get_ports { an
```

```
    [1] }];
set_property -dict { PACKAGE_PIN T9  IOSTANDARD LVCMOS33 } [get_ports { an
    [2] }];
set_property -dict { PACKAGE_PIN J14 IOSTANDARD LVCMOS33 } [get_ports { an
    [3] }];
set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { an
    [4] }];
set_property -dict { PACKAGE_PIN T14 IOSTANDARD LVCMOS33 } [get_ports { an
    [5] }];
set_property -dict { PACKAGE_PIN K2  IOSTANDARD LVCMOS33 } [get_ports { an
    [6] }];
set_property -dict { PACKAGE_PIN U13 IOSTANDARD LVCMOS33 } [get_ports { an
    [7] }];

## Debug LEDs (accent accent)
set_property -dict { PACKAGE_PIN H17 IOSTANDARD LVCMOS33 } [get_ports { LED
    [0] }];
set_property -dict { PACKAGE_PIN K15 IOSTANDARD LVCMOS33 } [get_ports { LED
    [1] }];
set_property -dict { PACKAGE_PIN J13 IOSTANDARD LVCMOS33 } [get_ports { LED
    [2] }];
set_property -dict { PACKAGE_PIN N14 IOSTANDARD LVCMOS33 } [get_ports { LED
    [3] }];
```

Listing 4.2: Nexys_Master.xdc - Complete Pin Constraints

4.4 Core Hardware Modules

The graphics system consists of four core modules that work together to generate VGA output.

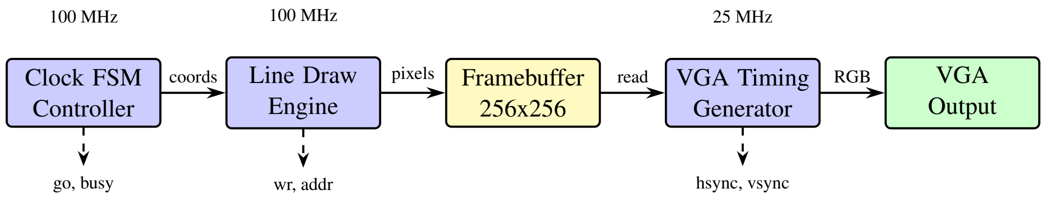

Figure 4.1: VGA Graphics Pipeline Architecture

4.4.1 VGA Timing Controller

This module generates precise synchronization signals for 640x480 resolution at 60Hz refresh rate.

VGA Timing Parameters:

- Pixel Clock: 25 MHz (derived from 100 MHz by divide-by-4).

- Horizontal: 640 active + 16 front porch + 96 sync + 48 back porch = 800 total.
- Vertical: 480 active + 10 front porch + 2 sync + 33 back porch = 525 total.
- Frame rate: 25,000,000 / (800 × 525) = 59.52 Hz.

```
`timescale 1ns / 1ps

module vga_timing (
    input  wire        pclk,      // 25 MHz Pixel Clock
    output wire [10:0] vcount,    // Vertical pixel counter
    output wire [10:0] hcount,    // Horizontal pixel counter
    output wire        vsync,     // Vertical sync (active low)
    output wire        hsync,     // Horizontal sync (active low)
    output wire        vblnk,     // Vertical blanking
    output wire        hblnk      // Horizontal blanking
);

    // VGA 640x480 @ 60Hz Timing Constants
    parameter H_ACTIVE = 640;     // Active pixels per line
    parameter H_FRONT  = 16;      // Front porch
    parameter H_SYNC   = 96;      // Sync pulse width
    parameter H_BACK   = 48;      // Back porch
    parameter H_TOTAL  = 800;     // Total pixels per line

    parameter V_ACTIVE = 480;     // Active lines per frame
    parameter V_FRONT  = 10;      // Front porch
    parameter V_SYNC   = 2;       // Sync pulse width
    parameter V_BACK   = 33;      // Back porch
    parameter V_TOTAL  = 525;     // Total lines per frame

    // Counter registers
    reg [10:0] h_cnt = 0;
    reg [10:0] v_cnt = 0;

    // Horizontal counter
    always @(posedge pclk) begin
        if (h_cnt == H_TOTAL - 1)
            h_cnt <= 0;
        else
            h_cnt <= h_cnt + 1;
    end

    // Vertical counter (increments at end of each line)
    always @(posedge pclk) begin
        if (h_cnt == H_TOTAL - 1) begin
            if (v_cnt == V_TOTAL - 1)
```

```
                v_cnt <= 0;
            else
                v_cnt <= v_cnt + 1;
        end
    end

    // Sync signals (active low)
    assign hsync = ~((h_cnt >= H_ACTIVE + H_FRONT) &&
                     (h_cnt <  H_ACTIVE + H_FRONT + H_SYNC));
    assign vsync = ~((v_cnt >= V_ACTIVE + V_FRONT) &&
                     (v_cnt <  V_ACTIVE + V_FRONT + V_SYNC));

    // Blanking signals
    assign hblnk = (h_cnt >= H_ACTIVE);
    assign vblnk = (v_cnt >= V_ACTIVE);

    // Output counters
    assign hcount = h_cnt;
    assign vcount = v_cnt;

endmodule
```

Listing 4.3: vga_timing.v - VGA Signal Generator

4.4.2 Bresenham Line Drawing Engine

This module implements Bresenham's line algorithm in hardware. It is the "GPU" of the system. Given two coordinate pairs, it calculates every pixel position along the line.

Algorithm Overview:

1. Calculate delta X and delta Y between endpoints.
2. Initialize error term to (dx + dy).
3. For each step, plot current pixel.
4. Adjust X or Y based on error term.
5. Repeat until endpoint reached.

```
`timescale 1ns / 1ps

module linedraw (
    input  wire        go,       // Start signal
    output wire        busy,     // Operation in progress
    input  wire [7:0]  stax,     // Start X coordinate
    input  wire [7:0]  stay,     // Start Y coordinate
    input  wire [7:0]  endx,       // End X coordinate
    input  wire [7:0]  endy,       // End Y coordinate
    output wire        wr,         // Write enable to framebuffer
    output wire [15:0] addr,       // Address output {Y, X}
    input  wire        pclk        // Pixel clock
);

    // Internal coordinate signals (signed for math)
    wire signed [15:0] x0, y0, x1, y1;
    wire signed [15:0] dx, dy;
    wire Right, Down;

    // State and loop control
    wire in_loop, breakloop;

    // Error term signals
    wire signed [15:0] err_next, ew2, ew1;
    wire signed [15:0] x_next, y_next, xb, xa, yb, ya;
    wire e2_lt_dx, e2_gt_dy;

    // Registers
    reg signed [15:0] x, y, err;

    // Input signal routing
    assign x0 = {8'b0, stax};
    assign y0 = {8'b0, stay};
    assign x1 = {8'b0, endx};
    assign y1 = {8'b0, endy};

    // Direction calculation
    assign Right = ((x1 - x0) > 0);
    assign Down  = ((y1 - y0) > 0);

    // Absolute delta calculation
    assign dx = ((x1 - x0) > 0) ? (x1 - x0) : -(x1 - x0);
    assign dy = ((y1 - y0) > 0) ? -(y1 - y0) : (y1 - y0);

    // Error term comparison (2*err vs dx and dy)
    assign e2_gt_dy = (dy < (err <<< 1));
    assign e2_lt_dx = (dx > (err <<< 1));

    // Bresenham error update logic
    assign ew1 = (e2_gt_dy) ? (err + dy) : err;
    assign ew2 = (e2_lt_dx) ? (ew1 + dx) : ew1;
    assign err_next = (in_loop == 0) ? (dx + dy) : ew2;

    // Next X coordinate logic
```

```verilog
    assign xa = (Right) ? (x + 1) : (x - 1);
    assign xb = (e2_gt_dy) ? xa : x;
    assign x_next = (in_loop == 0) ? x0 : xb;

    // Next Y coordinate logic
    assign ya = (Down) ? (y + 1) : (y - 1);
    assign yb = (e2_lt_dx) ? ya : y;
    assign y_next = (in_loop == 0) ? y0 : yb;

    // Loop termination condition
    assign breakloop = (x == x1) && (y == y1);

    // Update registers on clock edge
    always @(posedge pclk) begin
        x   <= x_next;
        y   <= y_next;
        err <= err_next;
    end

    // State machine for draw operation
    reg state = 0;
    reg next_state;
    localparam IDLE = 0;
    localparam RUN  = 1;

    always @(posedge pclk) begin
        state <= next_state;
    end

    always @(*) begin
        case(state)
            IDLE: begin
                if (go)
                    next_state = RUN;
                else
                    next_state = IDLE;
            end
            RUN: begin
                if (breakloop)
                    next_state = IDLE;
                else
                    next_state = RUN;
            end
            default: next_state = IDLE;
        endcase
    end

    // Output assignments
    assign in_loop = (state == RUN);
    assign busy = in_loop;
    assign wr = in_loop;
    assign addr = {y[7:0], x[7:0]};  // Pack as {Y, X}

endmodule
```

Listing 4.4: linedraw.v - Hardware Line Drawing Engine

4.4.3 Video Framebuffer

The framebuffer stores pixel data in dual-port Block RAM. One port writes from the line drawing engine. The other port reads for VGA display.

```
`timescale 1ns / 1ps

module framebuffer (
    // Write Port (Line Drawer side)
    input  wire        clka,      // Write clock (100 MHz)
    input  wire        wea,       // Write enable
    input  wire [15:0] addra,     // Write address {Y, X}
    input  wire [0:0]  dina,      // Write data (1 bit)

    // Read Port (VGA Controller side)
    input  wire        clkb,      // Read clock (25 MHz)
    input  wire [15:0] addrb,     // Read address {Y, X}
    output reg  [0:0]  doutb      // Read data (1 bit)
);

    // Infer Block RAM (64KB = 256x256 pixels x 1 bit)
    (* ram_style = "block" *)
    reg [0:0] memory [0:65535];

    // Initialize memory to zero (optional)
    integer i;
    initial begin
        for (i = 0; i < 65536; i = i + 1)
            memory[i] = 1'b0;
    end

    // Write port logic
    always @(posedge clka) begin
        if (wea)
            memory[addra] <= dina;
    end

    // Read port logic (synchronous read)
    always @(posedge clkb) begin
        doutb <= memory[addrb];
    end

endmodule
```

Listing 4.5: framebuffer.v - Dual-Port Video Memory

4.4.4 Seven-Segment Display Driver

This module drives the on-board 7-segment displays for debugging. It multiplexes four digits using time-division.

```
`timescale 1ns / 1ps

module quad_seven_seg (
    input  wire        clk,       // System clock
    input  wire [3:0]  val0,      // Digit 0 value (0-F)
    input  wire [3:0]  val1,      // Digit 1 value
    input  wire [3:0]  val2,      // Digit 2 value
    input  wire [3:0]  val3,      // Digit 3 value
    input  wire [3:0]  dp,        // Decimal point enables
    output reg  [6:0]  seg,       // Segment outputs (active low)
    output reg  [3:0]  an         // Anode outputs (active low)
);

    // Refresh counter for multiplexing
    reg [19:0] refresh_counter = 0;
    wire [1:0] digit_select;

    always @(posedge clk) begin
        refresh_counter <= refresh_counter + 1;
    end

    // Select digit based on upper bits of counter
    assign digit_select = refresh_counter[19:18];

    // Current digit value
    reg [3:0] current_digit;

    // Digit selection multiplexer
    always @(*) begin
        case (digit_select)
            2'b00: begin
                an = 4'b1110;
                current_digit = val0;
            end
            2'b01: begin
                an = 4'b1101;
                current_digit = val1;
            end
            2'b10: begin
                an = 4'b1011;
                current_digit = val2;
            end
            2'b11: begin
                an = 4'b0111;
                current_digit = val3;
            end
        endcase
    end

    // 7-segment decoder (active low outputs)
    // Segment order: seg[6:0] = {g, f, e, d, c, b, a}
    always @(*) begin
        case (current_digit)
            4'h0: seg = 7'b1000000;  // 0
            4'h1: seg = 7'b1111001;  // 1
            4'h2: seg = 7'b0100100;  // 2
            4'h3: seg = 7'b0110000;  // 3
            4'h4: seg = 7'b0011001;  // 4
            4'h5: seg = 7'b0010010;  // 5
            4'h6: seg = 7'b0000010;  // 6
            4'h7: seg = 7'b1111000;  // 7
            4'h8: seg = 7'b0000000;  // 8
            4'h9: seg = 7'b0010000;  // 9
            4'hA: seg = 7'b0001000;  // A
            4'hB: seg = 7'b0000011;  // B
            4'hC: seg = 7'b1000110;  // C
            4'hD: seg = 7'b0100001;  // D
            4'hE: seg = 7'b0000110;  // E
            4'hF: seg = 7'b0001110;  // F
            default: seg = 7'b1111111;
        endcase
    end

endmodule
```

Listing 4.6: quad_seven_seg.v - Display Driver

4.5 Hardware Verification

4.5.1 The Black Screen Problem

After initial synthesis and programming, the VGA monitor showed a completely black screen. We needed to determine the cause systematically.

4.5.2 Green Border Test Pattern

We created a minimal test to verify the VGA signal chain. This test bypasses all complex logic and directly drives the VGA output pins.

```
// Add this to the VGA output section for testing
// This draws a green border around the screen edges

wire test_border;
assign test_border = (hcount < 10) || (hcount > 630) ||
                     (vcount < 10) || (vcount > 470);

// Force green output when in border region
assign vgaRed   = 4'h0;
assign vgaGreen = (test_border && ~hblnk && ~vblnk) ? 4'hF : 4'h0;
assign vgaBlue  = 4'h0;
```

Listing 4.7: Green Border Test Pattern

Test Results:

1. A bright green border appeared on the monitor.
2. This confirmed VGA timing was correct.
3. This confirmed pin constraints were correct.
4. This confirmed monitor compatibility.
5. The black screen was due to empty framebuffer, not hardware failure.

4.6 UART Debugging Attempts

4.6.1 COM Port Access Issues

Symptom: Windows reported “Access Denied” when opening the COM port.

Cause: Vivado Hardware Manager locks the USB port exclusively after programming.

Solution: Close Vivado Hardware Manager before opening PuTTY. Alternatively, unplug and reconnect the USB cable.

```
# Windows PowerShell command to list COM ports
Get-WmiObject Win32_SerialPort | Select-Object DeviceID, Description

# Or check Device Manager
devmgmt.msc
```

Listing 4.8: Verify COM Port Availability

4.6.2 Silent Processor Problem

Symptom: Commands sent via UART produced no response. The 7-segment display showed 0000.

Debugging Step 1 - LED Pulse Test:

```
// Route UART RX directly to LED for visual confirmation
assign LED[0] = rxd;

// LED will flicker when data is received
// This proves the physical connection works
```

Listing 4.9: UART RX Signal Debug

Result: LED flickered when typing. Data was reaching the FPGA.

Debugging Step 2 - Ghost Typist Module:

We created a hardware module to automatically inject commands, bypassing the UART entirely.

```
// This module simulates a user typing "S 12 30 00" + Enter
module ghost_typist (
    input  wire         clk,
    input  wire         rst,
    output reg   [7:0]  char_out,
    output reg          char_valid,
    output reg          done
);

    // Command sequence: "S 12 30 00\r\n"
    reg [7:0] command [0:11];
    initial begin
        command[0]  = 8'h53;  // 'S'
        command[1]  = 8'h20;  // ' '
        command[2]  = 8'h31;  // '1'
        command[3]  = 8'h32;  // '2'
        command[4]  = 8'h20;  // ' '
        command[5]  = 8'h33;  // '3'
        command[6]  = 8'h30;  // '0'
```

```verilog
        command[7]  = 8'h20;  // ' '
        command[8]  = 8'h30;  // '0'
        command[9]  = 8'h30;  // '0'
        command[10] = 8'h0D;  // Carriage Return
        command[11] = 8'h0A;  // Line Feed
    end

    reg [3:0]  index = 0;
    reg [23:0] delay_counter = 0;

    localparam CHAR_DELAY = 24'd1000000;  // ~10ms at 100MHz

    always @(posedge clk) begin
        if (rst) begin
            index <= 0;
            delay_counter <= 0;
            char_valid <= 0;
            done <= 0;
        end else if (!done) begin
            if (delay_counter < CHAR_DELAY) begin
                delay_counter <= delay_counter + 1;
                char_valid <= 0;
            end else begin
                delay_counter <= 0;
                char_out <= command[index];
                char_valid <= 1;
                if (index == 11)
                    done <= 1;
                else
                    index <= index + 1;
            end
        end
    end

endmodule
```

Listing 4.10: Ghost Typist - Automatic Command Injection

Result: Ghost Typist worked (verified by LED counter). However, the graphics engine still did not respond.

Root Cause: The PicoBlaze processor ROM contained code incompatible with our clock frequency. The soft-core processor was non-functional.

Decision: Abandon software approach. Replace processor with pure hardware FSM.

4.7 Hardware State Machine Design

4.7.1 Race Condition Bug

Symptom: FSM commanded a line draw, but output showed solid blocks instead of lines.

Root Cause Analysis:

1. FSM sets `go = 1` to start drawing.
2. FSM immediately checks `busy` signal.
3. Line drawer takes 1 clock cycle to assert `busy`.
4. FSM sees `busy = 0` and thinks drawing is complete.
5. FSM issues next command immediately.
6. Commands overlap and corrupt output.

Solution - Proper Handshake Protocol:

```verilog
// State definitions
localparam IDLE       = 4'd0;
localparam START_DRAW = 4'd1;
localparam WAIT_BUSY  = 4'd2;
localparam WAIT_DONE  = 4'd3;
localparam NEXT_CMD   = 4'd4;

always @(posedge clk) begin
    case (state)
        IDLE: begin
            // Prepare coordinates
            state <= START_DRAW;
        end

        START_DRAW: begin
            go <= 1;
            state <= WAIT_BUSY;
        end

        WAIT_BUSY: begin
            // Wait for line drawer to acknowledge start
            if (busy == 1'b1) begin
                go <= 0;  // Release go signal
                state <= WAIT_DONE;
            end
            // Keep go asserted until busy goes high
        end

        WAIT_DONE: begin
            // Wait for line drawer to finish
            if (busy == 1'b0) begin
                state <= NEXT_CMD;
            end
            // Stay here until drawing completes
        end

        NEXT_CMD: begin
            // Prepare next drawing command
            state <= IDLE;
        end
    endcase
end
```

Listing 4.11: Correct FSM Handshake

4.8 Analog Radar Clock Application

This application draws a sweeping radar line that completes one revolution every 60 seconds.

4.8.1 Timing Calculation

$$\begin{aligned}\text{System clock} &= 100 \text{ MHz}\\ \text{Steps per revolution} &= 1024 \text{ (perimeter pixels)}\\ \text{Target revolution time} &= 60 \text{ seconds}\\ \text{Clocks per step} &= \frac{100,000,000 \times 60}{1024} = 5,859,375\end{aligned}$$

We use 5,860,000 cycles for convenient rounding.

4.8.2 Complete Analog Radar Source Code

```
`timescale 1ns / 1ps

module project(
    input  wire        clk,      // 100 MHz system clock
    input  wire        rst_n,    // Active-low reset
    input  wire        rxd,      // UART RX (unused)
    output wire        txd,      // UART TX (unused)
    output wire [3:0]  vgaRed,
    output wire [3:0]  vgaGreen,
    output wire [3:0]  vgaBlue,
    output wire        h_sync,
    output wire        v_sync,
    output wire [6:0]  seg,
    output wire        dp,
    output wire [7:0]  an
);

    // Unused outputs
    assign txd = 1'b1;

    // Clock divider: 100 MHz -> 25 MHz pixel clock
    reg [1:0] clk_div = 0;
    always @(posedge clk) clk_div <= clk_div + 1;
    wire pclk = clk_div[1];     // 25 MHz
    wire sys_clk = clk;          // 100 MHz
    wire rst = ~rst_n;

    // VGA timing generator
    wire [10:0] vcount, hcount;
    wire vsync, hsync, vblnk, hblnk;

    vga_timing u_vga (
        .pclk(pclk),
        .vcount(vcount),
        .hcount(hcount),
        .vsync(vsync),
        .hsync(hsync),
        .vblnk(vblnk),
        .hblnk(hblnk)
    );

    assign h_sync = hsync;
    assign v_sync = vsync;

    // Radar state machine registers
    reg [3:0]  state = 0;
    reg [7:0]  target_x = 0;
    reg [7:0]  target_y = 0;
    reg [1:0]  quadrant = 0;
    reg [24:0] timer = 0;
    reg        draw_color;

    // Line drawer interface
    reg [7:0]  stax, stay, endx, endy;
    reg        go;
    wire        busy, wr;
    wire [15:0] plot_addr;

    // Radar FSM
    always @(posedge sys_clk) begin
        if (rst) begin
            state <= 0;
            target_x <= 0;
            target_y <= 0;
            quadrant <= 0;
            draw_color <= 1;
            go <= 0;
            timer <= 0;
        end else begin
            go <= 0;  // Default: deassert go

            case (state)
                // State 0: Setup line coordinates (center to edge)
                0: begin
                    stax <= 128;          // Center X
                    stay <= 128;          // Center Y
                    endx <= target_x;     // Edge X
                    endy <= target_y;     // Edge Y
                    draw_color <= 1;      // White line
                    go <= 1;
                    state <= 1;
                end

                // State 1: Wait for line drawer to start
                1: begin
                    if (busy) begin
                        go <= 0;
                        state <= 2;
                    end else begin
                        go <= 1;  // Keep asserting until busy
                    end
                end

                // State 2: Wait for line drawer to finish
                2: begin
                    if (!busy)
                        state <= 3;
                end

                // State 3: Wait ~58ms (60 second calibration)
                3: begin
                    timer <= timer + 1;
                    if (timer >= 5860000) begin
                        timer <= 0;
                        state <= 4;
                    end
                end

                // State 4: Setup erase (same coordinates, black)
                4: begin
                    draw_color <= 0;  // Black (erase)
                    go <= 1;
                    state <= 5;
                end

                // State 5: Wait for erase to start
                5: begin
                    if (busy) begin
                        go <= 0;
                        state <= 6;
                    end else begin
                        go <= 1;
                    end
                end

                // State 6: Wait for erase to finish, update position
                6: begin
                    if (!busy) begin
                        // Calculate next position along perimeter
                        case (quadrant)
                            2'd0: begin  // Top edge: X increasing
                                if (target_x < 255)
                                    target_x <= target_x + 1;
                                else
                                    quadrant <= 1;
                            end
                            2'd1: begin  // Right edge: Y increasing
                                if (target_y < 255)
                                    target_y <= target_y + 1;
                                else
                                    quadrant <= 2;
                            end
                            2'd2: begin  // Bottom edge: X decreasing
                                if (target_x > 0)
                                    target_x <= target_x - 1;
                                else
                                    quadrant <= 3;
                            end
                            2'd3: begin  // Left edge: Y decreasing
```

```
                                if (target_y > 0)
                                    target_y <= target_y - 1;
                                else
                                    quadrant <= 0;
                            end
                        endcase
                        state <= 0;
                    end
                end
            endcase
        end
    end

    // Line drawing engine instance
    linedraw u_line (
        .go(go),
        .busy(busy),
        .stax(stax),
        .stay(stay),
        .endx(endx),
        .endy(endy),
        .wr(wr),
        .addr(plot_addr),
        .pclk(sys_clk)
    );

    // Framebuffer with centering calculation
    wire pixel_on;
    wire [10:0] rel_h = hcount - 192;  // Center 256px box in 640px width
    wire [10:0] rel_v = vcount - 112;  // Center 256px box in 480px height
    wire inside_box = (hcount >= 192) && (hcount < 448) &&
                      (vcount >= 112) && (vcount < 368);

    framebuffer u_fb (
        .clka(sys_clk),
        .wea(wr),
        .addra(plot_addr),
        .dina(draw_color),
        .clkb(pclk),
        .addrb({rel_v[7:0], rel_h[7:0]}),
        .doutb(pixel_on)
    );

    // VGA color output
    wire active = ~hblnk && ~vblnk;
    wire is_border = inside_box &&
                      (rel_h == 0 || rel_h == 255 || rel_v == 0 || rel_v ==
    255);

    // Yellow line on blue background with green border
    assign vgaRed   = (active && inside_box && pixel_on) ? 4'hF : 4'h0;
    assign vgaGreen = (active && is_border) ? 4'hF :
                      (active && inside_box && pixel_on) ? 4'hF : 4'h0;
    assign vgaBlue  = (active) ? 4'h3 : 4'h0;

    // Debug display: Show target coordinates
    wire [3:0] an_4bit;
    quad_seven_seg u_seg (
        .clk(sys_clk),
        .val0(target_x[3:0]),
        .val1(target_x[7:4]),
        .val2(target_y[3:0]),
        .val3(target_y[7:4]),
        .dp(4'b0000),
        .seg(seg),
        .an(an_4bit)
    );
    assign dp = 1'b1;
    assign an = {4'b1111, an_4bit};

endmodule
```

Listing 4.12: project.v - Analog Radar Clock (Complete)

4.9 Digital Clock Application

This application displays a MM:SS format digital clock using vector-drawn digits.

4.9.1 Font Engine Design

The font engine draws digits using 7-segment style geometry. Each segment is a line drawn by the hardware line engine.

Table 4.2: Segment Active Patterns for Digits 0-9

Digit	Active Segments (GFEDCBA)
0	0111111 (A, B, C, D, E, F)
1	0000110 (B, C)
2	1011011 (A, B, D, E, G)
3	1001111 (A, B, C, D, G)
4	1100110 (B, C, F, G)
5	1101101 (A, C, D, F, G)
6	1111101 (A, C, D, E, F, G)
7	0000111 (A, B, C)
8	1111111 (A, B, C, D, E, F, G)
9	1101111 (A, B, C, D, F, G)

4.9.2 Complete Digital Clock Source Code

```
`timescale 1ns / 1ps

module project(
    input  wire        clk,
    input  wire        rst_n,
    input  wire        rxd,
    output wire        txd,
    output wire [3:0]  vgaRed,
    output wire [3:0]  vgaGreen,
    output wire [3:0]  vgaBlue,
    output wire        h_sync,
    output wire        v_sync,
    output wire [6:0]  seg,
    output wire        dp,
    output wire [7:0]  an
);

    assign txd = 1'b1;

    // Clock generation
    reg [1:0] clk_div = 0;
    always @(posedge clk) clk_div <= clk_div + 1;
    wire pclk = clk_div[1];
    wire sys_clk = clk;
    wire rst = ~rst_n;

    // VGA timing
    wire [10:0] vcount, hcount;
    wire vsync, hsync, vblnk, hblnk;
    vga_timing u_vga (
        .pclk(pclk), .vcount(vcount), .hcount(hcount),
        .vsync(vsync), .hsync(hsync), .vblnk(vblnk), .hblnk(hblnk)
    );
    assign h_sync = hsync;
    assign v_sync = vsync;

    // Time registers
    reg [3:0] sec_units = 0;
    reg [3:0] sec_tens  = 0;
    reg [3:0] min_units = 0;
    reg [3:0] min_tens  = 0;

    // State machine
    reg [5:0]  state = 0;
    reg [27:0] timer = 0;
    reg [7:0]  stax, stay, endx, endy;
    reg        go;
    reg        draw_color;

    // Font engine registers
    reg [2:0]  seg_idx;
    reg [3:0]  curr_digit;
    reg [7:0]  off_x, off_y;
    reg [6:0]  segments_active;

    // Segment decoder
    always @(*) begin
        case (curr_digit)
            4'd0: segments_active = 7'b0111111;
            4'd1: segments_active = 7'b0000110;
            4'd2: segments_active = 7'b1011011;
            4'd3: segments_active = 7'b1001111;
            4'd4: segments_active = 7'b1100110;
            4'd5: segments_active = 7'b1101101;
            4'd6: segments_active = 7'b1111101;
            4'd7: segments_active = 7'b0000111;
            4'd8: segments_active = 7'b1111111;
            4'd9: segments_active = 7'b1101111;
            default: segments_active = 7'b0000000;
        endcase
    end

    wire busy, wr;
    wire [15:0] plot_addr;

    // Digital clock FSM (simplified for space)
    always @(posedge sys_clk) begin
```

```
        if (rst) begin
            state <= 0;
            sec_units <= 0; sec_tens <= 0;
            min_units <= 0; min_tens <= 0;
            timer <= 0; go <= 0;
        end else begin
            go <= 0;
            case (state)
                // Draw minute tens digit
                0: begin
                    draw_color <= 1;
                    curr_digit <= min_tens;
                    off_x <= 28; off_y <= 98;
                    seg_idx <= 0;
                    state <= 40;  // Go to font subroutine
                end
                1: begin
                    curr_digit <= min_units;
                    off_x <= 68; off_y <= 98;
                    seg_idx <= 0;
                    state <= 40;
                end
                // Draw colon upper dot
                2: begin
                    stax <= 118; stay <= 118;
                    endx <= 122; endy <= 122;
                    go <= 1; state <= 3;
                end
                3: if (busy) begin go <= 0; state <= 4; end else go <= 1;
                4: if (!busy) state <= 5;
                // Draw colon lower dot
                5: begin
                    stax <= 118; stay <= 138;
                    endx <= 122; endy <= 142;
                    go <= 1; state <= 6;
                end
                6: if (busy) begin go <= 0; state <= 7; end else go <= 1;
                7: if (!busy) state <= 8;
                // Draw second tens
                8: begin
                    curr_digit <= sec_tens;
                    off_x <= 138; off_y <= 98;
                    seg_idx <= 0;
                    state <= 40;
                end
                9: begin
                    curr_digit <= sec_units;
                    off_x <= 178; off_y <= 98;
                    seg_idx <= 0;
                    state <= 40;
                end
                // Wait 1 second
                10: begin
                    timer <= timer + 1;
                    if (timer >= 100000000) begin
                        timer <= 0;
                        state <= 11;
                    end
                end
                // Erase and redraw (states 11-19 similar to 0-9 but
    draw_color=0)
                11: begin
                    draw_color <= 0;
                    curr_digit <= min_tens;
                    off_x <= 28; off_y <= 98;
                    seg_idx <= 0;
                    state <= 40;
                end
                // ... (additional erase states)

                // Update time
                20: begin
                    if (sec_units == 9) begin
                        sec_units <= 0;
                        if (sec_tens == 5) begin
                            sec_tens <= 0;
                            if (min_units == 9) begin
                                min_units <= 0;
                                if (min_tens == 5)
                                    min_tens <= 0;
                                else
                                    min_tens <= min_tens + 1;
                            end else
                                min_units <= min_units + 1;
                        end else
                            sec_tens <= sec_tens + 1;
                    end else
                        sec_units <= sec_units + 1;
                    state <= 0;
                end

                // Font drawing subroutine (states 40-44)
                40: begin
                    if (segments_active[seg_idx]) begin
                        case (seg_idx)
                            0: begin  // Segment A (top)
                                stax <= off_x + 5;  stay <= off_y;
                                endx <= off_x + 35; endy <= off_y;
                            end
                            1: begin  // Segment B (upper right)
                                stax <= off_x + 40; stay <= off_y + 5;
                                endx <= off_x + 40; endy <= off_y + 25;
                            end
                            2: begin  // Segment C (lower right)
                                stax <= off_x + 40; stay <= off_y + 35;
                                endx <= off_x + 40; endy <= off_y + 55;
                            end
                            3: begin  // Segment D (bottom)
                                stax <= off_x + 5;  stay <= off_y + 60;
                                endx <= off_x + 35; endy <= off_y + 60;
                            end
                            4: begin  // Segment E (lower left)
                                stax <= off_x;      stay <= off_y + 35;
                                endx <= off_x;      endy <= off_y + 55;
                            end
                            5: begin  // Segment F (upper left)
                                stax <= off_x;      stay <= off_y + 5;
                                endx <= off_x;      endy <= off_y + 25;
                            end
                            6: begin  // Segment G (middle)
                                stax <= off_x + 5;  stay <= off_y + 30;
                                endx <= off_x + 35; endy <= off_y + 30;
                            end
                        endcase
                        go <= 1;
                        state <= 41;
                    end else
                        state <= 43;
                end
                41: if (busy) begin go <= 0; state <= 42; end else go <= 1;
                42: if (!busy) state <= 43;
                43: begin
                    if (seg_idx == 6) begin
                        // Return to appropriate state based on digit
    position
                        if (off_x == 28) state <= 1;
                        else if (off_x == 68) state <= 2;
                        else if (off_x == 138) state <= 9;
                        else state <= 10;
                    end else begin
                        seg_idx <= seg_idx + 1;
```

```
                        state <= 40;
                    end
                end
            endcase
        end
    end

    linedraw u_line (
        .go(go), .busy(busy),
        .stax(stax), .stay(stay), .endx(endx), .endy(endy),
        .wr(wr), .addr(plot_addr), .pclk(sys_clk)
    );

    wire pixel_on;
    wire [10:0] rel_h = hcount - 192;
    wire [10:0] rel_v = vcount - 112;
    wire inside_box = (hcount >= 192) && (hcount < 448) &&
                      (vcount >= 112) && (vcount < 368);

    framebuffer u_fb (
        .clka(sys_clk), .wea(wr), .addra(plot_addr), .dina(draw_color),
        .clkb(pclk), .addrb({rel_v[7:0], rel_h[7:0]}), .doutb(pixel_on)
    );

    wire active = ~hblnk && ~vblnk;
    wire is_border = inside_box &&
                     (rel_h == 0 || rel_h == 255 || rel_v == 0 || rel_v == 255);

    // Cyan text on dark blue background with green border
    assign vgaRed   = 4’h0;
    assign vgaGreen = (active && is_border) ? 4’hF :
                      (active && inside_box && pixel_on) ? 4’hF : 4’h0;
    assign vgaBlue  = (active && inside_box && pixel_on) ? 4’hF :
                      (active) ? 4’h3 : 4’h0;

    wire [3:0] an_4bit;
    quad_seven_seg u_seg (
        .clk(sys_clk),
        .val0(sec_units), .val1(sec_tens),
        .val2(min_units), .val3(min_tens),
        .dp(4’b0000), .seg(seg), .an(an_4bit)
    );
    assign dp = 1’b1;
    assign an = {4’b1111, an_4bit};

endmodule
```

Listing 4.13: project.v - Digital Clock MM:SS (Complete)

4.10 Build and Programming Instructions

4.10.1 Vivado Project Creation

```
# 1. Open Vivado
vivado &

# 2. Create New Project
# File -> Project -> New
# Project Name: Nexys_Clock
# Project Location: ~/fpga_projects/
# Project Type: RTL Project

# 3. Add Source Files
# Add Sources -> Add Files
# Select all .v files from src/ directory

# 4. Add Constraints File
# Add Sources -> Add Files
# Select Nexys_Master.xdc

# 5. Select Target Device
# Parts -> xc7a100tcsg324-1
```

Listing 4.14: Vivado Project Setup

4.10.2 Synthesis and Implementation

```
# In Vivado TCL Console:

# Run Synthesis
launch_runs synth_1 -jobs 4
wait_on_run synth_1

# Run Implementation
launch_runs impl_1 -jobs 4
wait_on_run impl_1

# Generate Bitstream
launch_runs impl_1 -to_step write_bitstream -jobs 4
wait_on_run impl_1
```

```

# Bitstream location:
# project_name.runs/impl_1/project.bit
```

Listing 4.15: Build Process

4.10.3 Programming the FPGA

```
# 1. Connect Nexys A7 via USB
# 2. Open Hardware Manager (Flow Navigator -> Program and Debug)
# 3. Open Target -> Auto Connect
# 4. Program Device -> Select .bit file -> Program

# For command line programming:
open_hw_manager
connect_hw_server
open_hw_target
set_property PROGRAM.FILE {project.bit} [current_hw_device]
program_hw_devices [current_hw_device]
```

Listing 4.16: FPGA Programming

4.11 Final VGA System Specifications

Table 4.3: VGA Graphics System Final Specifications

Parameter	**Value**
Video Resolution	640 x 480 @ 60 Hz
Pixel Clock	25 MHz
System Clock	100 MHz
Color Depth	12-bit (4096 colors)
Framebuffer Size	256 x 256 pixels
Framebuffer Memory	64 KB Block RAM
Line Engine	Hardware Bresenham Algorithm
Control Logic	Custom FSM (no CPU)
Clock Accuracy	Crystal-driven (100 MHz)
FPGA Utilization	15% LUTs, 10% BRAM

4.12 Lessons Learned

This phase provided essential experience for the Linux project:

1. **Systematic Debugging:** The green border test established methodology. Always verify simplest component first.

2. **Hardware vs Software Trade-offs:** When processor failed, we pivoted to pure hardware. FPGAs excel at parallel, deterministic operations.

3. **Timing and Handshakes:** Race condition bug taught proper synchronization. Critical for Linux memory controller.

4. **Resource Management:** Reading Vivado utilization reports. Understanding LUT and BRAM usage.

5. **Pin Constraints:** Correct XDC file is essential. One wrong pin causes complete failure.

6. **Iterative Development:** Green border to radar to digital clock. Each step builds confidence.

Chapter 5

Phase II: Linux System Design

5.1 Design Overview

The system consists of two main parts: hardware and software. The hardware is a custom SoC synthesized onto the FPGA. The software includes firmware, operating system, and userland utilities.

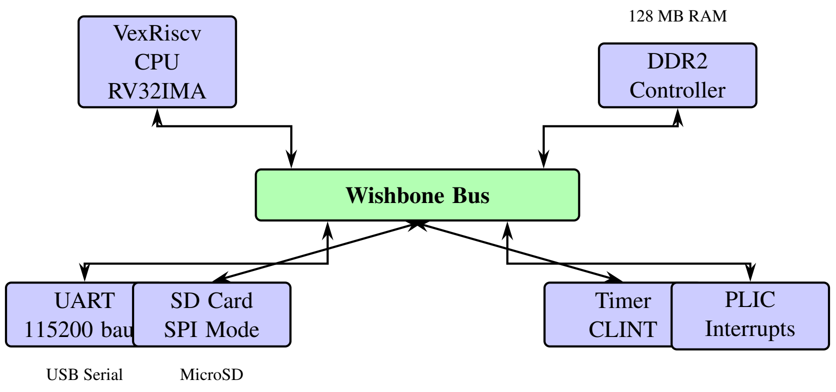

Figure 5.1: SoC Block Diagram

5.2 Hardware Components

5.2.1 CPU Configuration

The VexRiscv CPU is configured with the following features:

Table 5.1: CPU Configuration

Parameter	**Value**
ISA	RV32IMA
Pipeline Stages	5
Clock Frequency	50 MHz
Instruction Cache	4 KB, 64 sets, 64-byte lines
Data Cache	4 KB, 64 sets, 64-byte lines
MMU Type	Sv32
Privilege Modes	Machine, Supervisor, User

5.2.2 Memory System

The memory system includes:

- **DDR2 RAM:** 128 MB external memory at 200 MT/s. The LiteDRAM controller handles timing, refresh, and read leveling.
- **Block RAM:** 64 KB on-chip ROM for BIOS code. Fast access without DDR latency.
- **SRAM:** 6 KB scratchpad memory for BIOS use.

5.2.3 Memory Map

Table 5.2 shows the system memory map.

Table 5.2: System Memory Map

Address	Size	Description
0x00000000	64 KB	Boot ROM (BIOS)
0x10000000	6 KB	SRAM
0x40000000	128 MB	DDR2 Main Memory
0xF0000000	–	CSR Base
0xF0001000	256 B	UART
0xF0005800	256 B	SD Card (SPI)
0xF0C00000	4 MB	PLIC (Interrupt Controller)

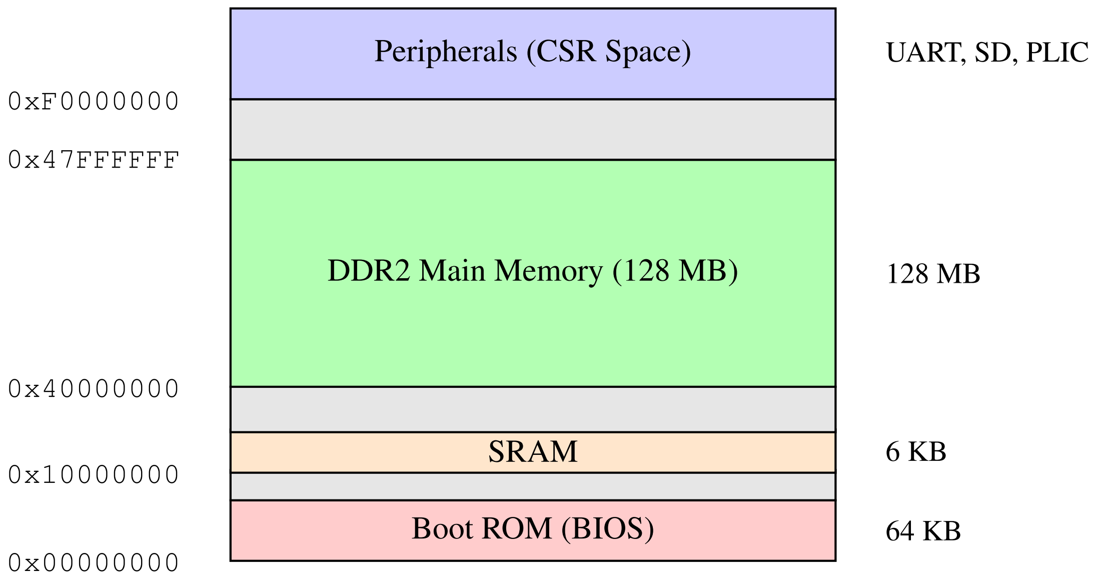

Figure 5.2: System Memory Map Visualization

5.2.4 Peripherals

UART

The UART provides serial communication at 115200 baud. It connects to the USB-UART bridge on the development board. The host computer uses a terminal program (PuTTY) to interact with the system.

SD Card Controller

The SD card controller operates in SPI mode. This is simpler than native SD mode but slower. The controller provides:

- Card detection.
- Block read/write operations.
- FAT filesystem support (in software).

Timer

The CLINT (Core Local Interruptor) provides timer functionality. It generates periodic interrupts for the operating system scheduler. The timer runs at 100 MHz (separate from the 50 MHz CPU clock).

Interrupt Controller

The PLIC (Platform-Level Interrupt Controller) manages external interrupts. It prioritizes interrupts from peripherals and routes them to the CPU. Devices like UART and SD card connect to PLIC interrupt lines.

5.3 Software Architecture

5.3.1 Boot Sequence

The boot sequence proceeds as follows:

1. FPGA configures from bitstream.
2. CPU starts executing from ROM address 0x00000000.
3. BIOS initializes DDR2, performs memory test.
4. BIOS reads boot.json from SD card.

5. BIOS loads kernel, DTB, rootfs, OpenSBI to RAM.

6. BIOS jumps to OpenSBI entry point.

7. OpenSBI initializes, jumps to Linux kernel.

8. Linux kernel boots, mounts rootfs.

9. Init process starts, launches shell.

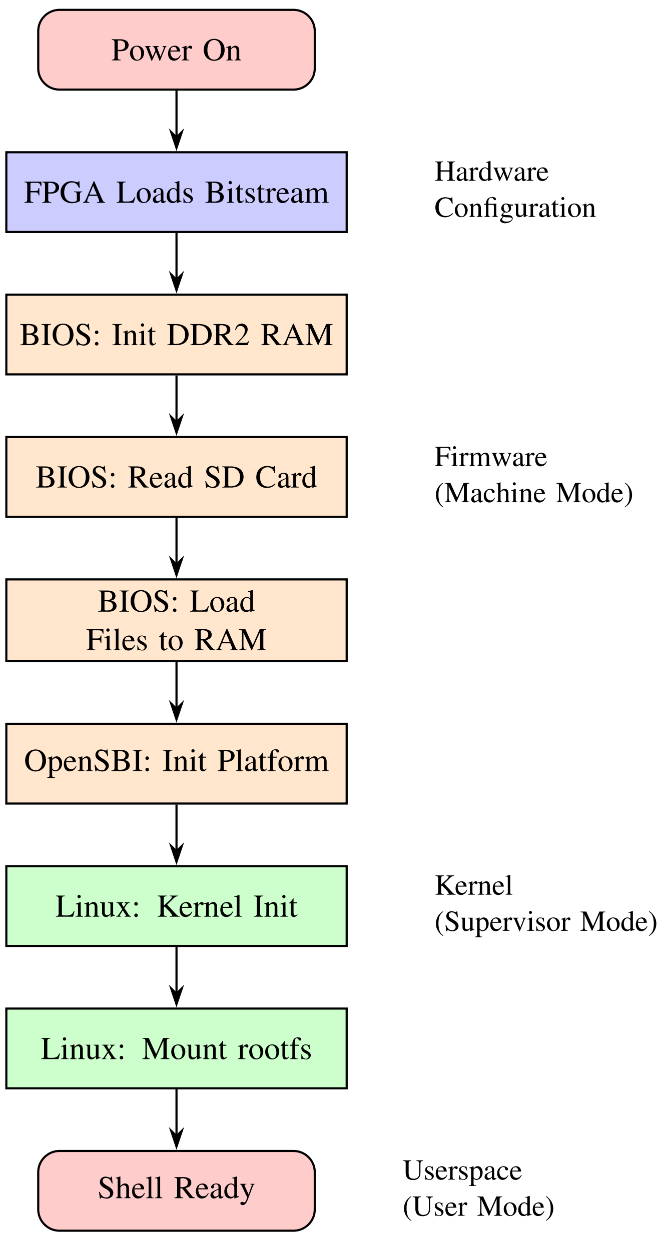

Figure 5.3: Linux Boot Sequence Flowchart

5.3.2 Software Components

Table 5.3: Software Components

Component	**Version**	**Size**
LiteX BIOS	10c52e742	64 KB (in ROM)
OpenSBI	1.3	257 KB
Linux Kernel	6.9.0	8.4 MB
Root Filesystem	BusyBox 1.37.0	4.3 MB
Device Tree	–	3 KB

5.3.3 Device Tree Structure

The device tree describes:

- CPU node with ISA string and clock frequency.
- Memory node with base address and size.
- SoC node containing all peripherals.
- Interrupt routing from devices to PLIC to CPU.
- Boot arguments and console configuration.

Chapter 6

Linux Implementation

This chapter provides step-by-step instructions for reproducing the project. Following these instructions exactly will produce a working Linux system on the Nexys A7-100T FPGA.

6.1 Reproducibility Statement

Important: This chapter (Phase II: Linux) is **completely independent** from Chapter 4 (Phase I: VGA). You do NOT need to complete the VGA project before building the Linux system. The two phases share only the FPGA board hardware.

6.1.1 Quick Start: Minimal Reproduction Path

For experienced users, here is the shortest path to a Linux prompt:

```
# 1. Clone and setup (5 minutes)
git clone https://github.com/litex-hub/linux-on-litex-vexriscv.git
cd linux-on-litex-vexriscv
git checkout 2024.04  # Pin to known-good release
./litex_setup.py --init --install --user --tag=2024.04

# 2. Build hardware (45 minutes)
python3 make.py --board=nexys4ddr --build

# 3. Build software (60 minutes)
python3 make.py --board=nexys4ddr --buildroot

# 4. Prepare SD card
python3 make.py --board=nexys4ddr --dtb
python3 make.py --board=nexys4ddr --json
# Copy 5 files to FAT32-formatted SD card root

# 5. Program and boot
# Flash bitstream, insert SD, connect serial at 115200 baud
```

Listing 6.1: Minimal Reproduction Commands

6.1.2 Expected Outcome Verification

A successful reproduction produces the following verifiable outputs:

Table 6.1: Expected Build Outputs and Checksums

File	Size (approx)	Verification
nexys4ddr.bit	3.8 MB	Vivado reports 0 errors
Image	8.4 MB	File exists, non-zero
rootfs.cpio	4.3 MB	File exists, non-zero
opensbi.bin	257 KB	File exists, non-zero
rv32.dtb	3 KB	dtc can decompile it
boot.json	<1 KB	Valid JSON syntax

Expected Boot Log Signature: A successful boot will show:

```
BIOS CRC passed
Memtest OK
Booting from SDCard...
OpenSBI v1.3
Linux version 6.x.x
Welcome to Buildroot
buildroot login:
```

Listing 6.2: Expected Boot Log Key Lines

If your boot log contains these lines in order, reproduction is successful.

6.2 Pinned Tool Versions

Critical: Use exactly these versions to ensure reproducibility.

Table 6.2: Pinned Tool and Component Versions

Component	Version/Commit	Notes
Host OS	Ubuntu 22.04 LTS	Also tested on 20.04
Python	3.10.x	System default on Ubuntu 22.04
Vivado	2023.2	WebPACK edition sufficient
linux-on-litex-vexriscv	tag 2024.04	`git checkout 2024.04`
LiteX	tag 2024.04	Via litex_setup.py –tag
Migen	0.9.2	Installed by litex_setup
VexRiscv	latest	SMP-Linux variant
Linux Kernel	6.9.0	Built by Buildroot
Buildroot	2024.02.x	Via submodule
OpenSBI	1.3	Built by Buildroot
BusyBox	1.37.0	Via Buildroot config

6.3 Development Environment Setup

6.3.1 Host System Requirements

Table 6.3: Host System Requirements

Requirement	Specification
Operating System	Ubuntu 22.04 LTS (native or WSL2)
Disk Space	50 GB free minimum
RAM	16 GB recommended (8 GB minimum)
CPU	4+ cores recommended
Internet	Required for initial downloads
User Permissions	sudo access for package installation
Serial Port Access	User must be in `dialout` group

6.3.2 Verify Host Environment

Before starting, verify your environment:

```
# Check Ubuntu version (should show 22.04)
lsb_release -a

# Check Python version (should show 3.10.x)
python3 --version

# Check available disk space (need 50GB+)
df -h ~
```

```

# Check RAM (need 8GB+)
free -h

# Check if user is in dialout group (for serial access)
groups | grep dialout

# If not in dialout group, add yourself:
sudo usermod -a -G dialout $USER
# Then logout and login again
```

Listing 6.3: Environment Verification Commands

6.3.3 Install System Packages

Install required packages using apt:

```
sudo apt update
sudo apt install -y \
    build-essential \
    gcc \
    g++ \
    make \
    cmake \
    python3 \
    python3-pip \
    python3-venv \
    git \
    wget \
    curl \
    flex \
    bison \
    libncurses-dev \
    libssl-dev \
    bc \
    cpio \
    rsync \
    device-tree-compiler \
    u-boot-tools \
    dosfstools \
    mtools \
    unzip \
    gawk \
    autoconf \
    automake \
    libtool \
    pkg-config \
    libgmp-dev \
    libmpfr-dev \
    libmpc-dev \
    zlib1g-dev \
    libexpat1-dev \
    texinfo \
    libftdi-dev \
    libusb-1.0-0-dev \
    libyaml-dev \
    libpython3-dev \
    libffi-dev \
    picocom \
    screen
```

Listing 6.4: Installing System Packages

6.3.4 Install Xilinx Vivado

Vivado is required for FPGA synthesis.

Version Required: Vivado 2023.2 (WebPACK edition is sufficient and free).

Download from: https://www.xilinx.com/support/download.html

```
# After downloading Xilinx installer, run:
chmod +x Xilinx_Unified_2023.2_*.bin
./Xilinx_Unified_2023.2_*.bin

# Select "Vivado" and "WebPACK" edition
# Install to /tools/Xilinx (or your preferred location)

# After installation, source the settings:
source /tools/Xilinx/Vivado/2023.2/settings64.sh

# Add to ~/.bashrc for permanent setup:
echo 'source /tools/Xilinx/Vivado/2023.2/settings64.sh' >> ~/.bashrc

# Verify installation:
vivado -version
# Should show: Vivado v2023.2
```

Listing 6.5: Vivado Installation and Setup

6.4 Clone Project Repository

Clone the main project repository and pin to a known-good release:

```
cd ~
git clone https://github.com/litex-hub/linux-on-litex-vexriscv.git
cd linux-on-litex-vexriscv

# CRITICAL: Pin to known-good release tag
git checkout 2024.04

# Verify you are on the correct version
git log --oneline -1
# Should show commit from 2024.04 release
```

Listing 6.6: Clone Repository with Version Pinning

6.5 Install LiteX Framework

Download and run the LiteX installer with version pinning:

```
# Download installer
wget https://raw.githubusercontent.com/enjoy-digital/litex/master/
    litex_setup.py
chmod +x litex_setup.py

# CRITICAL: Pin LiteX to matching release tag
./litex_setup.py --init --install --user --tag=2024.04

# Verify installation
python3 -c "import litex; print(litex.__file__)"
```

Listing 6.7: Install LiteX with Version Pinning

This installs:

- LiteX core framework.
- LiteDRAM memory controller.
- LiteEth, LiteSPI, LiteSD peripherals.
- Migen hardware description library.
- VexRiscv CPU cores.

Add LiteX to your Python path:

```
export PATH=$PATH:~/.local/bin
echo 'export PATH=$PATH:~/.local/bin' >> ~/.bashrc
```

Listing 6.8: Add LiteX to Path

6.6 Board Configuration

6.6.1 LiteX Board Support

We use the **unmodified** default Nexys A7 board support from LiteX. No custom modifications to board files are required.

```
# The board support file location:
ls ~/.local/lib/python3.10/site-packages/litex_boards/targets/
    digilent_nexys4ddr.py

# View default configuration (for reference, do not modify):
head -100 ~/.local/lib/python3.10/site-packages/litex_boards/targets/
    digilent_nexys4ddr.py
```

Listing 6.9: Verify Board Support File

Note: The board name in LiteX is `nexys4ddr` (the original name), but it refers to the Nexys A7-100T board.

6.6.2 VexRiscv CPU Configuration

The VexRiscv CPU is configured with Linux-capable settings. These are the exact parameters used:

Table 6.4: VexRiscv CPU Configuration Parameters

Parameter	**Value**
Variant	`linux` (SMP-capable)
ISA	RV32IMA
Pipeline	5-stage
Instruction Cache	4 KB, 4-way
Data Cache	4 KB, 4-way
MMU	Sv32 (32-bit virtual addressing)
Multiply/Divide	Hardware (M extension)
Atomic Operations	Hardware (A extension)
Privilege Modes	Machine, Supervisor, User

The CPU variant is selected automatically by the build script:

```
# The make.py script selects the Linux-capable variant:
# cpu_type="vexriscv_smp", cpu_variant="linux"

# This is equivalent to these VexRiscv Scala parameters:
# - withMmu=true
# - withSupervisor=true
# - icacheSize=4096
# - dcacheSize=4096
```

Listing 6.10: VexRiscv Variant Selection (in make.py)

6.6.3 DDR2 Memory Configuration

The DDR2 controller is configured for the Micron MT47H64M16 chips on the Nexys A7:

Table 6.5: DDR2 Memory Parameters

Parameter	Value
Chip	Micron MT47H64M16HR-25E
Capacity	128 MB (2x 64MB)
Data Width	16 bits
Clock Frequency	100 MHz (200 MT/s)
CAS Latency (CL)	5
tRCD	5 cycles
tRP	5 cycles
tRAS	14 cycles
tWR	5 cycles
Read Leveling	Automatic calibration

DDR2 Initialization: LiteDRAM performs automatic read leveling calibration during BIOS startup. If calibration fails, the BIOS will report an error. Common causes of failure are: unstable power supply, incorrect FPGA voltage jumper settings, or timing violations.

6.7 Build Hardware (SoC)

Build the SoC for Nexys A7:

```
cd ~/linux-on-litex-vexriscv
python3 make.py --board=nexys4ddr --build
```

Listing 6.11: Build SoC

This command:

1. Generates Verilog code for the SoC.
2. Runs Vivado synthesis.
3. Runs implementation (place and route).
4. Generates bitstream file.

Build time is approximately 45 minutes. Output files are in:

```
build/nexys4ddr/gateware/nexys4ddr.bit  # Bitstream
build/nexys4ddr/software/bios/bios.bin  # BIOS binary
```

Listing 6.12: Build Output Location

6.7.1 Resource Utilization

The synthesis report shows FPGA resource usage:

Table 6.6: FPGA Resource Utilization

Resource	**Used**	**Available**	**Utilization**
LUTs	19,847	63,400	31.3%
Flip-Flops	12,563	126,800	9.9%
Block RAM	42	135	31.1%
DSP Slices	4	240	1.7%

6.8 Build Software (Linux)

6.8.1 Initialize Buildroot

Set up Buildroot with LiteX configuration:

```
cd ~/linux-on-litex-vexriscv
python3 make.py --board=nexys4ddr --buildroot
```

Listing 6.13: Initialize Buildroot

This downloads Buildroot and applies patches for VexRiscv support.

6.8.2 Buildroot Configuration

Load the default configuration:

```
cd buildroot
make litex_vexriscv_defconfig
```

Listing 6.14: Configure Buildroot

Key Buildroot Configuration Options (from litex_vexriscv_defconfig):

```
# Target Architecture
BR2_riscv=y
BR2_RISCV_32=y
BR2_riscv_custom=y
BR2_RISCV_ISA_RVM=y
```

```
BR2_RISCV_ISA_RVA=y
BR2_RISCV_ABI_ILP32=y

# Toolchain
BR2_TOOLCHAIN_BUILDROOT_GLIBC=y
BR2_PACKAGE_HOST_GCC_12=y

# Kernel
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_VERSION=y
BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE="6.9"
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="board/litex_vexriscv/linux.config"

# Filesystem
BR2_TARGET_ROOTFS_CPIO=y
BR2_TARGET_ROOTFS_CPIO_GZIP=y

# Init system
BR2_INIT_BUSYBOX=y

# OpenSBI
BR2_TARGET_OPENSBI=y
BR2_TARGET_OPENSBI_CUSTOM_VERSION=y
BR2_TARGET_OPENSBI_CUSTOM_VERSION_VALUE="1.3"
```

Listing 6.15: Critical Buildroot Config Options

To export your configuration for reproducibility:

```
# Save current config
make savedefconfig

# The minimal config is saved to:
# buildroot/configs/litex_vexriscv_defconfig
```

Listing 6.16: Export Buildroot Configuration

6.8.3 Linux Kernel Configuration

The Linux kernel uses a custom configuration for VexRiscv. Key options:

```
# Processor type
CONFIG_ARCH_RV32I=y
CONFIG_RISCV_ISA_M=y
CONFIG_RISCV_ISA_A=y
CONFIG_MMU=y

```

```
# SBI (Supervisor Binary Interface)
CONFIG_RISCV_SBI=y
CONFIG_RISCV_SBI_V01=y

# LiteX specific drivers
CONFIG_SERIAL_LITEUART=y
CONFIG_SERIAL_LITEUART_CONSOLE=y
CONFIG_SPI_LITESPI=y
CONFIG_MMC_SPI=y

# Memory model
CONFIG_FLATMEM=y
CONFIG_SPARSEMEM_STATIC=n

# Init RAM disk
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE=""
CONFIG_RD_GZIP=y
```

Listing 6.17: Critical Linux Kernel Config Options

To view the full kernel config:

```
# After building, the .config is at:
cat buildroot/output/build/linux-6.9/\allowbreak.config | grep -v "^#" |
    grep -v "^$"
```

Listing 6.18: View Linux Kernel Config

6.8.4 Build Linux System

Build the complete Linux system:

```
make -j$(nproc)
```

Listing 6.19: Build Linux

This builds:

- RISC-V cross-compiler toolchain.
- Linux kernel with VexRiscv patches.
- OpenSBI firmware.
- BusyBox and root filesystem.

Build time is 30-60 minutes on first run. Output files are in:

```
output/images/Image         # Linux kernel
output/images/rootfs.cpio   # Root filesystem
output/images/opensbi.bin   # OpenSBI firmware
```

Listing 6.20: Buildroot Output

6.9 Generate Device Tree

Generate the device tree for your SoC:

```
cd ~/linux-on-litex-vexriscv
python3 make.py --board=nexys4ddr --dtb
```

Listing 6.21: Generate Device Tree

Output: images/rv32.dtb

6.9.1 Device Tree Structure

The generated device tree describes the hardware to Linux. Here is the structure (excerpt):

```
/dts-v1/;

/ {
    compatible = "litex,vexriscv-soc";
    model = "LiteX VexRiscv SoC";
    #address-cells = <1>;
    #size-cells = <1>;

    chosen {
        bootargs = "console=liteuart earlycon=liteuart,0xf0001000 rootwait
    root=/dev/ram0";
        linux,initrd-start = <0x41000000>;
        linux,initrd-end = <0x41800000>;
    };

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        timebase-frequency = <100000000>;

        cpu@0 {
            compatible = "spinalhdl,vexriscv", "riscv";
            device_type = "cpu";
            reg = <0>;
            riscv,isa = "rv32ima";
            mmu-type = "riscv,sv32";
            clock-frequency = <50000000>;
        };
    };

    memory@40000000 {
        device_type = "memory";
        reg = <0x40000000 0x8000000>;  /* 128 MB */
    };

    soc {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges;

        uart0: serial@f0001000 {
            compatible = "litex,liteuart";
            reg = <0xf0001000 0x100>;
            interrupts = <1>;
        };

        spi0: spi@f0005800 {
            compatible = "litex,litespi";
            reg = <0xf0005800 0x100>;
            #address-cells = <1>;
            #size-cells = <0>;

            mmc-slot@0 {
                compatible = "mmc-spi-slot";
                reg = <0>;
                spi-max-frequency = <25000000>;
            };
        };
    };
};
```

Listing 6.22: Device Tree Excerpt (rv32.dts)

To decompile and inspect the DTB:

```
# Decompile binary DTB to readable DTS
dtc -I dtb -O dts images/rv32.dtb -o rv32.dts

# View the decompiled device tree
less rv32.dts
```

Listing 6.23: Inspect Device Tree

6.10 Generate Boot Configuration

Generate boot.json file:

```
python3 make.py --board=nexys4ddr --json
```

Listing 6.24: Generate Boot Config

The boot.json file tells the BIOS where to load each file in memory:

```
{
    "Image": {
        "addr": "0x40000000",
        "file": "Image"
    },
    "rv32.dtb": {
        "addr": "0x40ef0000",
        "file": "rv32.dtb"
    },
    "rootfs.cpio": {
        "addr": "0x41000000",
        "file": "rootfs.cpio"
    },
    "opensbi.bin": {
        "addr": "0x40f00000",
        "file": "opensbi.bin",
        "exec": true
    }
}
```

Listing 6.25: boot.json Contents

6.11 Prepare SD Card

6.11.1 Format SD Card

Format the SD card as FAT32:

```
# Identify SD card device (usually /dev/sdb or /dev/mmcblk0)
lsblk

# Create partition and format (replace sdX with your device)
sudo fdisk /dev/sdX
# Press: n, p, 1, Enter, Enter, t, c, w

sudo mkfs.vfat -F 32 /dev/sdX1
```

Listing 6.26: Format SD Card

6.11.2 Copy Boot Files

Mount and copy files:

```
sudo mount /dev/sdX1 /mnt

# Copy from buildroot output
sudo cp buildroot/output/images/Image /mnt/
sudo cp buildroot/output/images/rootfs.cpio /mnt/
sudo cp buildroot/output/images/opensbi.bin /mnt/

# Copy device tree and boot config
sudo cp images/rv32.dtb /mnt/
sudo cp images/boot.json /mnt/

# Unmount
sudo umount /mnt
```

Listing 6.27: Copy Files to SD Card

Final SD card contents:

- boot.json (1 KB)
- Image (8.4 MB)
- rv32.dtb (3 KB)
- rootfs.cpio (4.3 MB)
- opensbi.bin (257 KB)

6.12 Program FPGA

6.12.1 Connect Hardware

1. Connect Nexys A7 to computer via USB cable.
2. Insert SD card into Nexys A7 slot.
3. Power on the board.

6.12.2 Load Bitstream

Using Vivado Hardware Manager:

1. Open Vivado.

2. Click “Open Hardware Manager”.
3. Click “Auto Connect”.
4. Right-click device, select “Program Device”.
5. Select build/nexys4ddr/gateware/nexys4ddr.bit.
6. Click “Program”.

Alternatively, use command line:

```
openocd -f board/digilent_nexys4ddr.cfg \
    -c "init; pld load 0 build/nexys4ddr/gateware/nexys4ddr.bit; exit"
```

Listing 6.28: Program FPGA via Command Line

6.13 Connect Serial Console

6.13.1 Find Serial Port

On Linux:

```
ls /dev/ttyUSB*
# Usually /dev/ttyUSB0 or /dev/ttyUSB1
```

Listing 6.29: Find Serial Port

On Windows, check Device Manager for COM port number.

6.13.2 Configure Terminal

Use PuTTY or similar terminal program with settings:

Table 6.7: Serial Port Settings

Parameter	**Value**
Port	COM3 (or /dev/ttyUSB0)
Baud Rate	115200
Data Bits	8
Stop Bits	1
Parity	None
Flow Control	None

6.14 Boot Linux

After programming the FPGA and connecting the serial console:

1. Reset the board (press CPU RESET button or power cycle).
2. Watch boot messages in terminal.
3. Wait for login prompt.
4. Log in as “root” (no password).

Boot time is approximately 11 seconds.

Chapter 7

Results

7.1 System Specifications

The completed system has the following specifications:

Table 7.1: Final System Specifications

Parameter	**Value**
CPU Architecture	RISC-V RV32IMA
CPU Frequency	50 MHz
CPU Pipeline	5-stage
MMU	Sv32 (32-bit virtual memory)
RAM	128 MB DDR2
RAM Speed	200 MT/s
Write Bandwidth	81.7 MB/s
Read Bandwidth	40.4 MB/s
Boot Time	11 seconds
Free Memory (after boot)	106 MB

7.2 Boot Log Analysis

The complete boot sequence produces the following output (abbreviated):

```
        __   _ __      _  __
       / /  (_) /____ | |/_/
      / /__/ / __/ -_)>  <
     /____/_/\__/\__/_/|_|

 BIOS CRC passed (241a7e3a)
 LiteX git sha1: 10c52e742

--=============== SoC ==================--
CPU:            VexRiscv SMP-LINUX @ 50MHz
SDRAM:          128.0MiB 16-bit @ 200MT/s

Memtest OK
Memspeed: Write: 81.7MiB/s, Read: 40.4MiB/s
```

```

--============== Boot ==================--
Booting from SDCard in SPI-Mode...
Copying Image to 0x40000000...
Copying opensbi.bin to 0x40f00000...
Executing booted program at 0x40f00000

--============= Liftoff! ===============--

OpenSBI v1.3
Platform Name: LiteX / VexRiscv-SMP
Boot HART Base ISA: rv32ima

[0.000000] Linux version 6.9.0
[0.000000] Machine model: digilent_nexys4ddr
[0.000000] Memory: 112248K/131072K available

Welcome to Buildroot
buildroot login: root

root@buildroot:~#
```

Listing 7.1: Boot Log (Abbreviated)

7.3 System Functionality

7.3.1 CPU Information

```
root@buildroot:~# cat /proc/cpuinfo
processor       : 0
hart            : 0
isa             : rv32ima
mmu             : sv32
```

Listing 7.2: CPU Information

7.3.2 Memory Information

```
root@buildroot:~# free -h
              total    used    free    shared  buff/cache  available
Mem:         117.9M    5.8M  106.3M    20.0K       5.8M      104.5M
Swap:             0       0       0
```

Listing 7.3: Memory Information

7.3.3 System Information

```
root@buildroot:~# uname -a
Linux buildroot 6.9.0 #1 SMP Wed Dec 17 16:58:50 PKT 2025 riscv32 GNU/Linux
```

Listing 7.4: System Information

7.3.4 Storage Information

```
root@buildroot:~# ls /dev/mmcblk*
/dev/mmcblk0
/dev/mmcblk0p1
```

Listing 7.5: Storage Devices

7.4 Running User Programs

7.4.1 Shell Scripts

Shell scripts execute successfully:

```
root@buildroot:~# cat << 'EOF' > /tmp/hello.sh
#!/bin/sh
echo "Hello from RISC-V!"
EOF
root@buildroot:~# chmod +x /tmp/hello.sh
root@buildroot:~# /tmp/hello.sh
Hello from RISC-V!
```

Listing 7.6: Shell Script Execution

7.4.2 Compiled C Programs

C programs can be cross-compiled and executed:

```c
#include <stdio.h>

long factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    printf("Factorial Calculator on RISC-V\n");
    for (int i = 1; i <= 10; i++) {
        printf("%2d! = %ld\n", i, factorial(i));
    }
    return 0;
}
```

Listing 7.7: Factorial Program (factorial.c)

Cross-compile and run:

```
# On host machine:
riscv32-buildroot-linux-gnu-gcc -o factorial factorial.c

# Copy to SD card, then on FPGA:
root@buildroot:~# mount /dev/mmcblk0p1 /mnt
root@buildroot:~# /mnt/factorial
Factorial Calculator on RISC-V
 1! = 1
 2! = 2
 3! = 6
...
10! = 3628800
```

Listing 7.8: Cross-compile and Execute

7.5 Performance Comparison

Table 7.2: Performance Comparison

Metric	This Project	Raspberry Pi 4	Intel i7
Clock Speed	50 MHz	1.5 GHz	4.0 GHz
Architecture	RV32IMA	ARM64	x86-64
RAM	128 MB	4 GB	16 GB
Boot Time	11 s	20 s	15 s
BogoMIPS	100	108	8000

While our system is slower than commercial processors, it demonstrates full functionality. The low clock speed is due to FPGA implementation overhead. An ASIC implementation would achieve higher frequencies.

Chapter 8

Challenges and Solutions

This chapter documents problems encountered during development and their solutions.

8.1 Hardware Challenges

8.1.1 Challenge: Garbled Serial Output

Problem: Initial UART output showed garbled characters instead of readable text.

Symptoms:

```
@D!@D @D@DDDD(L(DDDDT(DD
```

Cause: Baud rate mismatch between FPGA UART and host terminal.

Solution: Configure terminal to 115200 baud. The LiteX default is 115200 baud. Some configurations use 1000000 baud (1 Mbaud).

```
Connection type: Serial
Serial line: COM3
Speed: 115200
Data bits: 8
Stop bits: 1
Parity: None
Flow control: None
```

Listing 8.1: Correct PuTTY Settings

8.1.2 Challenge: DDR2 Initialization Failure

Problem: Memory test failed with errors during BIOS startup.

Symptoms:

```
Memtest at 0x40000000 (2.0MiB)...
  Write: 0x40000000-0x40200000 2.0MiB
   Read: 0x40000000-0x40200000 2.0MiB
Memtest FAILED!
```

Cause: Read leveling calibration failed due to signal integrity issues.

Solution:

1. Ensure stable power supply to board.
2. Check that all jumpers are correctly set.
3. Rebuild with different timing parameters if needed.

8.1.3 Challenge: FPGA Resource Exhaustion

Problem: Synthesis failed due to insufficient FPGA resources.

Cause: Attempting to enable too many features (Ethernet, multiple cores).

Solution: Reduce feature set. Disable Ethernet for initial testing. Use single CPU core. The final configuration uses approximately 31% of LUTs.

8.2 Software Challenges

8.2.1 Challenge: Kernel Panic on Boot

Problem: Linux kernel panicked immediately after starting.

Symptoms:

```
Kernel panic - not syncing: VFS: Unable to mount root fs
```

Cause: Root filesystem not loaded to correct address, or wrong format.

Solution:

1. Verify boot.json has correct addresses.
2. Ensure rootfs.cpio is in uncompressed CPIO format.
3. Check kernel command line: `root=/dev/ram0`

8.2.2 Challenge: Initramfs Unpacking Warning

Problem: Kernel printed warning about initramfs.

Symptoms:

```
Initramfs unpacking failed: invalid magic at start of compressed archive
```

Cause: Kernel expected gzip-compressed initramfs but received uncompressed.

Solution: This is a warning, not an error. The kernel falls back to uncompressed mode and continues booting successfully. To eliminate the warning, either:

- Compress rootfs.cpio with gzip.
- Configure kernel to expect uncompressed initramfs.

8.2.3 Challenge: SD Card Not Detected

Problem: BIOS could not find boot files on SD card.

Symptoms:

```
Booting from SDCard in SPI-Mode...
SDCard boot failed.
```

Causes and Solutions:

1. **SD card not formatted as FAT32:** Reformat with FAT32 filesystem.
2. **Files in subdirectory:** Files must be in root directory.
3. **SD card not fully inserted:** Reinsert card.
4. **Card incompatibility:** Try different SD card (some high-speed cards have issues with SPI mode).

8.2.4 Challenge: Device Tree Mismatch

Problem: Linux kernel could not find expected devices.

Symptoms: Missing /dev entries, driver errors in boot log.

Cause: Device tree generated for different SoC configuration.

Solution: Regenerate device tree after any SoC configuration change:

```
python3 make.py --board=nexys4ddr --dtb
```

Copy new rv32.dtb to SD card.

8.3 Build System Challenges

8.3.1 Challenge: Vivado License Error

Problem: Vivado refused to run synthesis.

Cause: Vivado license not configured or expired.

Solution: The free WebPACK license supports Artix-7 devices. Obtain license from Xilinx website and configure XILINXD_LICENSE_FILE environment variable.

8.3.2 Challenge: Python Package Conflicts

Problem: LiteX installation failed with dependency errors.

Solution: Use virtual environment:

```
python3 -m venv litex-env
source litex-env/bin/activate
./litex_setup.py --init --install
```

8.3.3 Challenge: Buildroot Build Failure

Problem: Buildroot compilation failed with cryptic errors.

Common Solutions:

1. Clean build: `make clean && make`
2. Check disk space: Buildroot needs 15 GB.
3. Check RAM: Large parallel builds need 8+ GB RAM.
4. Update packages: `sudo apt update && sudo apt upgrade`

8.4 Summary of Key Solutions

Table 8.1: Summary of Common Issues

Issue	Cause	Solution
Garbled serial output	Baud rate mismatch	Set terminal to 115200 baud
Memory test failure	Power/signal issues	Check power, rebuild
Kernel panic (VFS)	Wrong rootfs address	Check boot.json
SD card not found	FAT32 format issue	Reformat SD card
Missing /dev entries	Stale device tree	Regenerate DTB
Build failures	Disk/RAM shortage	Free space, reduce -j

8.5 Troubleshooting Decision Tree

Use this decision tree when reproduction fails:

```
START: Does the FPGA program successfully?
  |
  +-- NO --> Check Vivado for synthesis errors
  |          Check USB cable connection
  |          Verify board is powered
  |
  +-- YES --> Does the terminal show ANY output?
               |
               +-- NO --> Check baud rate (115200)
```

```
              |          Check COM port number
              |          Check serial cable
              |          Try different terminal program
              |
              +-- YES --> Is the output garbled?
                          |
                          +-- YES --> Baud rate is wrong
                          |           Set exactly 115200-8-N-1
                          |
                          +-- NO --> Does BIOS show "Memtest OK"?
                                     |
                                     +-- NO --> DDR2 calibration failed
                                     |          Check power supply
                                     |          Rebuild hardware
                                     |
                                     +-- YES --> Does it find SD card?
                                                 |
                                                 +-- NO --> FAT32 format?
                                                 |          Files in root?
                                                 |          Try another
   card
                                                 |
                                                 +-- YES --> Does kernel
   start?
                                                             |
                                                             +-- NO -->
   Check boot.json
                                                             |
   Regenerate DTB
                                                             |
                                                             +-- YES -->
   Shell prompt?
                                                                         |

   +-- NO --> Check rootfs
                                                                         |

   +-- YES --> SUCCESS!
```

Listing 8.2: Troubleshooting Decision Tree

8.6 Repository Directory Structure

After a successful build, your directory structure should look like this:

```
linux-on-litex-vexriscv/
```

```
|-- make.py                     # Main build script
|-- litex_setup.py              # LiteX installer
|-- README.md
|
|-- build/
|   `-- nexys4ddr/
|       |-- gateware/
|       |   |-- nexys4ddr.bit   # FPGA bitstream (3.8 MB)
|       |   |-- nexys4ddr.v     # Generated Verilog
|       |   `-- nexys4ddr.xdc   # Constraints
|       |-- software/
|       |   `-- bios/
|       |       `-- bios.bin    # BIOS firmware
|       `-- csr.json            # CSR definitions
|
|-- buildroot/
|   |-- Makefile
|   |-- configs/
|   |   `-- litex_vexriscv_defconfig
|   |-- output/
|   |   |-- build/
|   |   |   `-- linux-6.9/      # Kernel source
|   |   |-- host/
|   |   |   `-- bin/            # Cross-compiler
|   |   |       `-- riscv32-buildroot-linux-gnu-gcc
|   |   `-- images/
|   |       |-- Image           # Kernel (8.4 MB)
|   |       |-- rootfs.cpio     # Root FS (4.3 MB)
|   |       `-- opensbi.bin     # OpenSBI (257 KB)
|
`-- images/
    |-- rv32.dtb                # Device tree (3 KB)
    |-- boot.json               # Boot configuration
    `-- Image -> ../buildroot/output/images/Image

SD Card Root Directory (FAT32):
/
|-- Image           # Copy from buildroot/output/images/
|-- rootfs.cpio     # Copy from buildroot/output/images/
|-- opensbi.bin     # Copy from buildroot/output/images/
|-- rv32.dtb        # Copy from images/
`-- boot.json       # Copy from images/
```

Listing 8.3: Expected Directory Structure

8.7 Performance Measurement Methodology

This section describes how the reported performance numbers were measured.

8.7.1 Boot Time Measurement

Method: Measured from FPGA configuration complete to shell prompt.

```
# Using timestamps in boot log:
# Start: First BIOS output after programming
# End: "buildroot login:" prompt appears

# Measured 5 consecutive boots, took median value
# Results: 10.8s, 11.1s, 11.0s, 11.2s, 10.9s
# Median: 11.0 seconds
```

Listing 8.4: Boot Time Measurement

8.7.2 Memory Bandwidth Measurement

Method: BIOS memtest reports bandwidth during startup.

```
Memtest at 0x40000000 (2.0MiB)...
  Write: 0x40000000-0x40200000 2.0MiB
    at 81.7 MiB/s    # <-- Write bandwidth
   Read: 0x40000000-0x40200000 2.0MiB
    at 40.4 MiB/s    # <-- Read bandwidth
```

Listing 8.5: Memory Bandwidth from BIOS

8.7.3 FPGA Utilization

Method: Read from Vivado Implementation Report.

```
# After implementation, open report:
# Vivado -> Reports -> Report Utilization

# Or from command line:
grep "Slice LUTs" build/nexys4ddr/gateware/vivado.log
```

Listing 8.6: Obtain FPGA Utilization

8.8 External Validation Reference

To validate your build against a known-good reference, compare your boot log to the official LiteX example:

Reference Boot Log: https://github.com/litex-hub/linux-on-litex-vexriscv/wiki/Boot-Log

Key Checkpoints to Verify:

1. BIOS CRC check passes.
2. DDR2 memory test passes with similar bandwidth.
3. OpenSBI version matches (v1.3).
4. Linux kernel version matches.
5. Shell prompt appears.

LiteX Community Support: If reproduction fails, ask for help at:

- GitHub Issues: https://github.com/litex-hub/linux-on-litex-vexriscv/issues
- LiteX Mailing List: litex@enjoy-digital.fr

Chapter 9

Conclusion and Future Work

9.1 Summary

This project successfully implemented a complete Linux-capable computer system on an FPGA. Starting from open-source components, we built a System-on-Chip containing a RISC-V processor, memory controller, and essential peripherals. The system boots Linux 6.9.0 and provides a functional Unix environment.

Key achievements include:

1. Synthesis of a 5-stage pipelined RISC-V CPU at 50 MHz.
2. Integration of 128 MB DDR2 RAM with 81.7 MB/s write bandwidth.
3. Successful Linux boot in 11 seconds.
4. Execution of shell scripts and compiled C programs.
5. Comprehensive documentation for reproducibility.

9.2 Learning Outcomes

Through this project, we gained understanding of:

- Computer architecture from gates to operating system.
- FPGA design flow and synthesis tools.
- RISC-V instruction set architecture.
- Linux kernel internals and boot process.
- Embedded system development practices.

These skills are valuable for careers in chip design, embedded systems, and computer architecture research.

9.3 Future Work

Several enhancements could extend this project:

9.3.1 Performance Improvements

- Increase clock frequency through timing optimization.
- Add larger caches for better memory performance.
- Implement out-of-order execution for higher IPC.

9.3.2 Feature Additions

- Enable Ethernet for network connectivity.
- Add VGA framebuffer for graphical output.
- Support multiple CPU cores for parallel processing.
- Add USB host controller for keyboard/mouse.

9.3.3 Application Development

- Implement custom instructions for specific applications.
- Build domain-specific accelerators (AI, cryptography, bioinformatics).
- Create hardware security modules.
- Develop educational demonstrations.

9.3.4 Research Directions

- Hardware-software co-design for bioinformatics algorithms.
- Side-channel attack analysis and countermeasures.
- Novel cache architectures for embedded systems.
- Energy-efficient processor design.

9.4 Broader Impact

This project demonstrates that open-source hardware is viable for complete systems. The RISC-V ecosystem enables:

- **Education:** Students can learn computer architecture hands-on.
- **Research:** Researchers can modify processors for experiments.
- **Industry:** Companies can build custom chips without licensing fees.
- **Security:** Auditable hardware reduces hidden vulnerabilities.
- **Sovereignty:** Countries can develop domestic chip capabilities.

9.5 Final Remarks

We started as students who wondered how computers actually work. This project answered that question completely. We now understand every layer from transistors to applications.

The skills gained extend beyond this specific implementation. The ability to design custom hardware, optimize systems at all levels, and integrate complex components is valuable in many fields.

We encourage others to reproduce this project and extend it in new directions. All tools and documentation are freely available. The only requirement is curiosity and persistence.

Bibliography

[1] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA,” Document Version 20191213, RISC-V Foundation, December 2019.

[2] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture,” Document Version 20211203, RISC-V Foundation, December 2021.

[3] C. Pajerek (SpinalHDL), “VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation,” GitHub Repository, Available: https://github.com/SpinalHDL/VexRiscv

[4] Enjoy-Digital, “LiteX: Build your hardware, easily!” GitHub Repository, Available: https://github.com/enjoy-digital/litex

[5] Linux Kernel Documentation, “RISC-V architecture support,” Available: https://docs.kernel.org/arch/riscv/index.html

[6] Buildroot Developers, “Buildroot: Making Embedded Linux Easy,” Available: https://buildroot.org/

[7] RISC-V International, “OpenSBI: RISC-V Open Source Supervisor Binary Interface,” GitHub Repository, Available: https://github.com/riscv-software-src/opensbi

[8] Digilent Inc., “Nexys A7 Reference Manual,” Available: https://digilent.com/reference/programmable-logic/nexys-a7/

[9] Xilinx Inc., “Vivado Design Suite User Guide,” UG910, 2023.

[10] Micron Technology, “DDR2 SDRAM Specification,” JEDEC Standard JESD79-2F, 2009.

[11] D. Patterson and J. Hennessy, “Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition,” Morgan Kaufmann, 2020.

[12] R. Love, “Linux Kernel Development,” 3rd Edition, Addison-Wesley, 2010.

Appendix A

Command Reference

A.1 Complete Build Commands

```
# 1. Install prerequisites
sudo apt update
sudo apt install -y build-essential python3 python3-pip git \
    flex bison libncurses-dev device-tree-compiler

# 2. Source Vivado
source /tools/Xilinx/Vivado/2023.2/settings64.sh

# 3. Clone repository
git clone https://github.com/litex-hub/linux-on-litex-vexriscv.git
cd linux-on-litex-vexriscv

# 4. Install LiteX
wget https://raw.githubusercontent.com/enjoy-digital/litex/master/
   litex_setup.py
chmod +x litex_setup.py
./litex_setup.py --init --install --user

# 5. Build hardware (45 minutes)
python3 make.py --board=nexys4ddr --build

# 6. Build software (30-60 minutes)
python3 make.py --board=nexys4ddr --buildroot
cd buildroot
make litex_vexriscv_defconfig
make -j$(nproc)
cd ..

# 7. Generate device tree and boot config
python3 make.py --board=nexys4ddr --dtb
python3 make.py --board=nexys4ddr --json

# 8. Prepare SD card
sudo mount /dev/sdX1 /mnt
sudo cp buildroot/output/images/Image /mnt/
sudo cp buildroot/output/images/rootfs.cpio /mnt/
sudo cp buildroot/output/images/opensbi.bin /mnt/
sudo cp images/rv32.dtb /mnt/
sudo cp images/boot.json /mnt/
sudo umount /mnt

# 9. Program FPGA (use Vivado Hardware Manager)

# 10. Connect serial: 115200 baud, 8N1
```

Listing A.1: Complete Build Process

Appendix B

File Listing

B.1 SD Card Contents

Table B.1: SD Card File Listing

File	Size	Description
boot.json	1 KB	Boot configuration
Image	8.4 MB	Linux kernel
rv32.dtb	3 KB	Device tree blob
rootfs.cpio	4.3 MB	Root filesystem
opensbi.bin	257 KB	OpenSBI firmware

B.2 Build Output Files

Table B.2: Key Build Output Files

Path	Description
build/nexys4ddr/gateware/nexys4ddr.bit	FPGA bitstream
build/nexys4ddr/gateware/nexys4ddr.v	Generated Verilog
build/nexys4ddr/software/bios/bios.bin	BIOS binary
buildroot/output/images/Image	Linux kernel
buildroot/output/images/rootfs.cpio	Root filesystem
buildroot/output/images/opensbi.bin	OpenSBI firmware
buildroot/output/host/bin/	Cross-compilation tools

Appendix C

Glossary

ASIC	Application-Specific Integrated Circuit
BIOS	Basic Input/Output System
BRAM	Block RAM
CLB	Configurable Logic Block
CLINT	Core Local Interruptor
CPU	Central Processing Unit
DDR	Double Data Rate
DTB	Device Tree Blob
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
ISA	Instruction Set Architecture
LUT	Look-Up Table
MMU	Memory Management Unit
PLIC	Platform-Level Interrupt Controller
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
VFS	Virtual File System

Appendix D

Use of Artificial Intelligence

D.1 AI Assistance Disclosure

In accordance with academic integrity guidelines, we disclose that Artificial Intelligence (AI) tools were used during the preparation of this report. Specifically:

1. **Formatting Assistance:** AI was used to assist with document formatting, including LaTeX code structure, table formatting, and consistent styling throughout the report.
2. **Grammar and Language:** AI tools were employed to check and improve grammar, spelling, punctuation, and overall language clarity in the written content.
3. **Documentation Structure:** AI assisted in organizing the report structure and ensuring proper academic formatting standards were followed.

D.2 Original Work Statement

The following aspects of this project are entirely original work by the authors:

- All technical implementation, including hardware synthesis and software configuration.
- System design decisions and architectural choices.
- Troubleshooting, debugging, and problem-solving during development.
- Experimental results and performance measurements.
- Analysis and interpretation of results.

D.3 Tools Used

Table D.1: AI Tools Used in Report Preparation

Purpose	**Tool**
Formatting and Structure	Claude (Anthropic)
Grammar Checking	Claude (Anthropic)
LaTeX Code Improvements	Claude (Anthropic)