**VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY**

**UNIVERSITY OF TECHNOLOGY FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING**

**DEPARTMENT OF ELECTRONICS**

🙞∙∙∙☼∙∙∙🙜



**Capstone Project 1**

**RISC-V Processor with branch comparator**

**another model of ADDER**

**HK242**

**Date: 14/4/2025**

**Supervisor: Dr.Tran Hoang Linh**

|  |  |
| --- | --- |
| **Full Name** | **Student’s ID** |
| Đào Nguyên Khôi | 2151215 |

**Hồ Chí Minh City – 2025**

**Table of Contents**

[INTRODUCTION 7](#_Toc198637753)

[CHAPTER 1. OVERVIEW OF THE TOPIC 8](#_Toc198637754)

[1.1. Reasons for Choosing the Topic: 8](#_Toc198637755)

[1.2. Design Theory: 8](#_Toc198637756)

[CHAPTER 2. SYSTEM ARCHITECTURE AND I/O DEFINITION 9](#_Toc198637757)

[2.1. High-Level Block Diagram: 9](#_Toc198637758)

[2.2. I/O Definition: 10](#_Toc198637759)

[CHAPTER 3. DESIGN MAIN OPERATIONS 11](#_Toc198637760)

[3.1. Flowchart: 11](#_Toc198637761)

[3.2. Control Unit Decode Table: 12](#_Toc198637762)

[CHAPTER 4. DETAILED FUNCTIONS DESCRIPTION 16](#_Toc198637763)

[4.1. Control Unit: 16](#_Toc198637764)

[4.1.1. Block Diagram and I/O Definition: 16](#_Toc198637765)

[4.1.2. Operation Algorithm: 17](#_Toc198637766)

[4.2. Register File: 18](#_Toc198637767)

[4.2.1. Block Diagram and I/O Definition: 18](#_Toc198637768)

[4.2.2. Operation Algorithm: 19](#_Toc198637769)

[4.3. Extend Unit: 20](#_Toc198637770)

[4.3.1. I/O Definition: 20](#_Toc198637771)

[4.3.2. Operation Algorithm: 20](#_Toc198637772)

[4.4. Branch Comparator: 21](#_Toc198637773)

[4.4.1. Block Diagram and I/O Definition: 21](#_Toc198637774)

[4.4.2. Operation Algorithm: 22](#_Toc198637775)

[4.5. ALU: 24](#_Toc198637776)

[4.5.1. Block Diagram and I/O Definition: 24](#_Toc198637777)

[4.5.2. Operation Algorithm: 25](#_Toc198637778)

[4.6. Data Memory and I/O Block: 29](#_Toc198637779)

[4.6.1. Block Diagram and I/O Definition: 29](#_Toc198637780)

[4.6.2. Operation Algorithm: 30](#_Toc198637781)

[CHAPTER 5. RTL STIMULATION RESULT 33](#_Toc198637782)

[5.1. Synthesis. 33](#_Toc198637783)

[5.2. Testbench/environment setup for components. 34](#_Toc198637784)

[5.2.1. Kogge-Stone Adder. 34](#_Toc198637785)

[5.2.2. Branch Comparator Unit. 34](#_Toc198637786)

[5.3. Simulation Verification. 35](#_Toc198637787)

[5.3.1. Instruction Code for Testing. 35](#_Toc198637788)

[5.3.2. Results. 36](#_Toc198637789)

[CHAPTER 6. FPGA PROVEN 37](#_Toc198637790)

[6.1. Intruction Code for Testing. 37](#_Toc198637791)

[6.2. Results. 38](#_Toc198637792)

**Table of Figures**

[Figure 2.1 Top-Level Diagram of RISC-V 9](#_Toc198637793)

[Figure 3.1 Flowchart of Main Operations 11](#_Toc198637794)

[Figure 3.2 Main Decoder - Part 2 14](#_Toc198637795)

[Figure 3.3 ALU Decoder 15](#_Toc198637796)

[Figure 4.1 Control Unit Diagram 16](#_Toc198637797)

[Figure 4.2 RISC-V Base Instruction Formats 17](#_Toc198637798)

[Figure 4.3 Register File Block Diagram 18](#_Toc198637799)

[Figure 4.4 Types of Immediate produced by RISC-V Intructions 20](#_Toc198637800)

[Figure 4.5 Branch Comparator Block Diagram 21](#_Toc198637801)

[Figure 4.6 2-Bits Comparator 22](#_Toc198637802)

[Figure 4.7 4-Bits Comparator 22](#_Toc198637803)

[Figure 4.8 10-Bits Comparator 23](#_Toc198637804)

[Figure 4.9 32-Bits Comparator 23](#_Toc198637805)

[Figure 4.10 ALU Block Diagram 24](#_Toc198637806)

[Figure 4.11 Black and Gray Cells Diagram 26](#_Toc198637807)

[Figure 4.12 32-Bits Kogge-Stone Block Diagram 27](#_Toc198637808)

[Figure 4.13 Barrel Shifter Block Diagram 29](#_Toc198637809)

[Figure 4.14 I/O Block and Data Memory Block Diagram 29](#_Toc198637810)

[Figure 4.15 I/O Addressing Map 31](#_Toc198637811)

[Figure 5.1 Resource Usage Results 33](#_Toc198637812)

[Figure 5.2 Timming Result 33](#_Toc198637813)

[Figure 5.3 Kogge-Stone Testbench Results 34](#_Toc198637814)

[Figure 5.4 Branch Comparator Testbench Results 34](#_Toc198637815)

[Figure 5.5 Functional Stimulation Results 36](#_Toc198637816)

[Figure 6.1 FPGA Test Results 39](#_Toc198637817)

**Table of Tables**

[Table 2.1 Top-Level I/O Description 10](#_Toc198637818)

[Table 3.1 Main Decoder - Part 1 13](#_Toc198637819)

[Table 4.1 Control Unit I/O Definition 17](#_Toc198637820)

[Table 4.2 Register File I/O Definition 19](#_Toc198637821)

[Table 4.3 Extend Unit I/O Definition 20](#_Toc198637822)

[Table 4.4 Branch Comparator I/O Definition 22](#_Toc198637823)

[Table 4.5 ALU I/O Definition 25](#_Toc198637824)

[Table 4.6 ALU Operation Control 25](#_Toc198637825)

[Table 4.7 I/O Block I/O Definition 30](#_Toc198637826)

[Table 4.8 Data Memory I/O Definition 30](#_Toc198637827)

[Table 5.1 Functional Test Instructions 36](#_Toc198637828)

[Table 6.1 Fibonannci Calculation Instructions 37](#_Toc198637829)

# **INTRODUCTION**

In recent years, the RISC-V instruction set architecture (ISA) has emerged as a promising open standard for processor design. Unlike proprietary architectures, RISC-V is open-source, allowing researchers, students, and engineers to explore CPU design without licensing constraints. Its simplicity, modular design, and support for customization make it an ideal choice for academic projects and educational purposes.

This project focuses on re-implementing a single-cycle RISC-V CPU using Verilog on the Quartus development environment. The main goal is to understand and apply core computer architecture concepts, including the design of the datapath, control unit, and ALU. By constructing these components from the ground up, the project reinforces theoretical knowledge through practical implementation.

The design includes essential features of the RISC-V base integer instruction set (RV32I), supporting operations such as arithmetic computation, logic operations, data transfer, and branching. Each instruction is fetched, decoded, and executed within a single clock cycle, reflecting a simplified yet instructive architecture model.

To verify the functionality of the CPU, testbenches are developed for simulation, allowing step-by-step observation of instruction execution. This ensures correctness in logic and timing before synthesis on FPGA. The Quartus platform is used to write, simulate, and potentially deploy the design on actual hardware, although the focus remains on correctness and simulation-level validation.

Overall, the project offers a valuable opportunity to explore digital system design, improve HDL programming skills, and gain insights into the internal workings of modern processors through the lens of the RISC-V architecture.

# **OVERVIEW OF THE TOPIC**

* 1. **Reasons for Choosing the Topic:**

The research focus centers on understanding the RISC-V RV32I processor’s operation, particularly its instruction execution and interaction with a simple GPIO interface for input and output operations. Integrating a Kogge-Stone Adder into the ALU and designing GPIO-integrated components aim to enhance arithmetic efficiency and system usability in embedded applications..

* 1. **Design Theory:**

The RISC-V RV32I processor design leverages the open-source RISC-V ISA, chosen for its modularity and 32-bit integer operation support. A Kogge-Stone Adder, optimized for low-latency arithmetic, is integrated into the ALU for efficient addition, balancing speed and area. The design uses a modular RTL approach in Verilog, ensuring synthesizability and portability across ASIC and FPGA platforms.

The single-cycle architecture executes each instruction in one clock cycle, prioritizing simplicity and predictable timing. A simple GPIO interface handles input and output operations, enabling basic external device interaction. Verification employs testbenches to confirm RV32I instruction execution, adder performance, and GPIO functionality, targeting correct data flow.

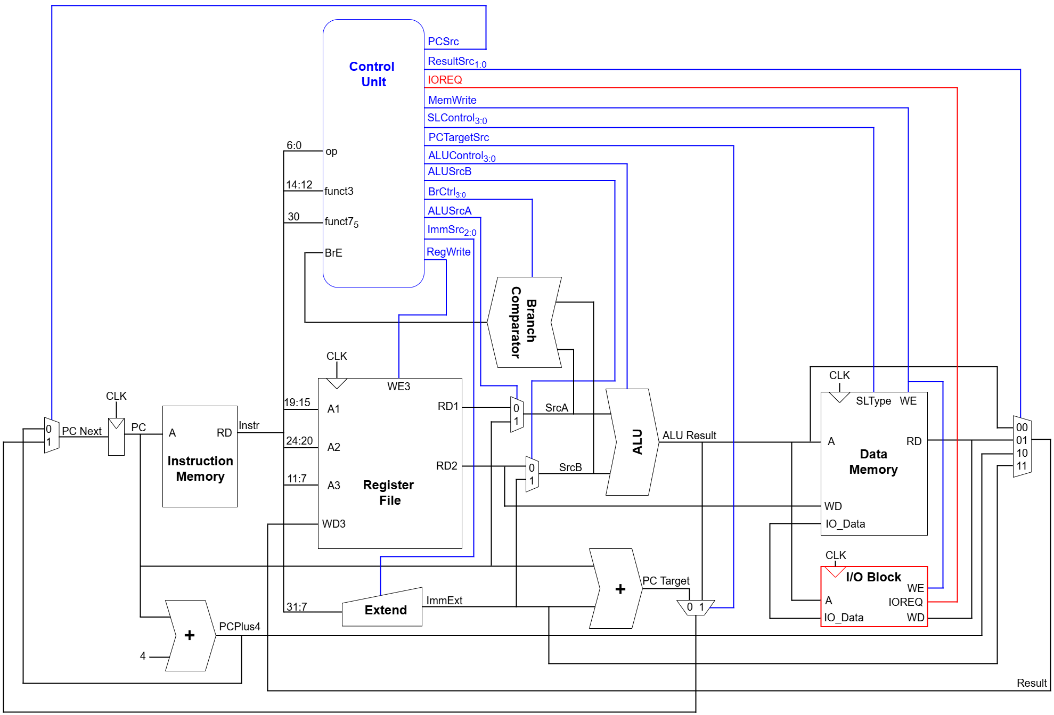
The methodology focuses on synthesis optimization, minimizing gate count and power consumption. The GPIO’s simplicity aids prototyping, especially for FPGA, while the modular design supports future enhancements. The Kogge-Stone Adder’s implementation provides insights into high-performance arithmetic for embedded systems.

# **SYSTEM ARCHITECTURE AND I/O DEFINITION**

* 1. **High-Level Block Diagram:**

The single-cycle RISC-V RV32I processor architecture is adapted from the foundational design presented in “*Digital Design and Computer Architecture – RISC-V Edition”* by Harris and Harris, which outlines the core modules and their interconnections for a basic RISC-V CPU. To create a functional, detailed implementation, the original design has been extended with new components, including a Kogge-Stone Adder within the ALU for efficient arithmetic, a Branch Comparator for conditional jumps, and an I/O Block implementing a simple GPIO interface for input and output operations.

The block diagram below illustrates the hierarchy and connections among these subblocks for instruction fetch, decode, execution, memory access, and I/O interfacing, showcasing the single-cycle datapath and the integration of the components, with control signals coordinating operations across the processor.



**Figure 2.1 Top-Level Diagram of RISC-V**

The Instruction Fetch unit, with the Program Counter and Instruction Memory, retrieves 32-bit instructions, incrementing the PC by 4 each cycle. The Decode unit, including the Control Unit and Register File, interprets instructions, generates control signals, and provides operands. The Execution unit features the ALU, integrating a Kogge-Stone Adder for efficient arithmetic, alongside a Branch Comparator for conditional jumps, with inputs selected between registers, immediates, and PC values. The Data Memory handles load and store operations, while the I/O Block, implementing a simple GPIO interface, manages input and output operations with external devices, controlled by the IOREQ signal. All subblocks connect via a unified data path, completing each instruction in one clock cycle, with the Control Unit coordinating operations through signals like PCSrc, ALUSrc, and RegWrite.

* 1. **I/O Definition:**

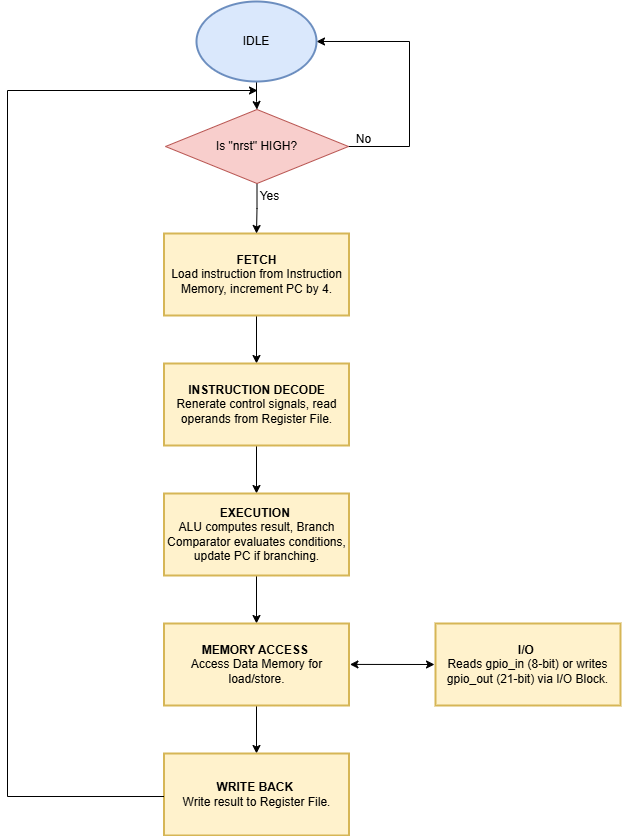
The top-level I/O defines the external interface for synchronous operation and data exchange with external devices via a simple GPIO interface. While the block diagram primarily highlights the clock signal distribution, this section details the confirmed top-level ports, including clock, reset, and GPIO signals. The instruction input, potentially managed internally through the Instruction Memory, is omitted pending further design clarification, with internal subblock interconnections deferred to the detailed functions description for comprehensive analysis.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit Width | Direction | Function Description |
| clk | 1 | Input | Clock signal for synchronous operations |
| nrst | 1 | Input | Negative edge resets. nrst=0 will to its initial state |
| gpio\_in | 8 | Output | Reads 8-bit external input data for processing |
| gpio\_out | 21 | Output | Outputs 21-bit data to drive three 7-bit 7SegLED displays |

**Table 2.1 Top-Level I/O Description**

# **DESIGN MAIN OPERATIONS**

* 1. **Flowchart:**



**Figure 3.1 Flowchart of Main Operations**

The process begins by checking the reset signal, if nrst is low, the processor initializes, setting the Program Counter and registers to their default states. If nrst is high, the processor fetches a 32-bit instruction from the Instruction Memory using the Program Counter, incrementing the Program Counter by 4.

The Control Unit decodes the instruction, generating control signals, while the Register File reads operands.

The ALU, with an integrated Kogge-Stone Adder, performs arithmetic or logic operations, and the Branch Comparator evaluates conditions for jumps, updating the Program Counter if needed.

The Data Memory handles load or store operations, the I/O Block, implementing a simple GPIO interface, reads 8-bit input data or writes 21-bit output data for three 7-segment LED displays, and the Register File updates with the result, all in a single cycle.

* 1. **Control Unit Decode Table:**

The single-cycle RISC-V processor relies on a Control Unit composed of a Main Decoder and an ALU Decoder to generate control signals in one clock cycle, eliminating the need for a traditional state machine.

The Main Decoder interprets the 7-bit opcode to produce high-level control signals, use to build 37 instructions, such as RegWrite, ALUSrcA, ALUSrcB, MemWrite, IOREQ, and Branch, etc; while the ALU Decoder uses the opcode, funct3, and funct7 fields to determine the ALUControl signal for operations like addition, subtraction, or comparison, supporting the ALU with its Kogge-Stone Adder and the Branch Comparator.

The tables below detail the decoding logic for common RV32I instruction types, ensuring efficient control of the processor’s datapath.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Op** | **RegWrite** | **ImmSrc** | **ALUSrcB** | **ALUSrcA** | **MemWrite** | **IOREQ** |
| lb | 0000011 | 1 | 000 | 1 | 0 | 0 | 1 |
| lh | 0000011 | 1 | 000 | 1 | 0 | 0 | 1 |
| lw | 0000011 | 1 | 000 | 1 | 0 | 0 | 1 |
| lbu | 0000011 | 1 | 000 | 1 | 0 | 0 | 1 |
| lhu | 0000011 | 1 | 000 | 1 | 0 | 0 | 1 |
| addi | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| slli | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| slti | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| sltiu | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| xori | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| srli | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| srai | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| ori | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| andi | 0010011 | 1 | 000 | 1 | 0 | 0 | 0 |
| auipc | 0010111 | 1 | 100 | 1 | 1 | 0 | 0 |
| sb | 0100011 | 0 | 001 | 1 | 0 | 1 | 1 |
| sh | 0100011 | 0 | 001 | 1 | 0 | 1 | 1 |
| sw | 0100011 | 0 | 001 | 1 | 0 | 1 | 1 |
| add | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| sub | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| sll | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| slt | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| sltu | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| xor | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| srli | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| sra | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| or | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| and | 0110011 | 1 | xxx | 0 | 0 | 0 | 0 |
| lui | 0110111 | 1 | 100 | 1 | 0 | 0 | 0 |
| beq | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| bne | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| blt | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| bge | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| bltu | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| bgeu | 1100011 | 0 | 010 | 0 | 0 | 0 | 0 |
| jalr | 1100111 | 1 | 000 | 1 | 0 | 0 | 0 |
| jal | 1101111 | 1 | 011 | x | 0 | 0 | 0 |

**Table 3.1 Main Decoder - Part 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Op** | **ResultSrc** | **Branch** | **Jump** | **PCTargetSrc** | **ALUOp** |
| lb | 0000011 | 01 | 0 | 0 | 0 | 00 |
| lh | 0000011 | 01 | 0 | 0 | 0 | 00 |
| lw | 0000011 | 01 | 0 | 0 | 0 | 00 |
| lbu | 0000011 | 01 | 0 | 0 | 0 | 00 |
| lhu | 0000011 | 01 | 0 | 0 | 0 | 00 |
| addi | 0010011 | 00 | 0 | 0 | 0 | 10 |
| slli | 0010011 | 00 | 0 | 0 | 0 | 10 |
| slti | 0010011 | 00 | 0 | 0 | 0 | 10 |
| sltiu | 0010011 | 00 | 0 | 0 | 0 | 10 |
| xori | 0010011 | 00 | 0 | 0 | 0 | 10 |
| srli | 0010011 | 00 | 0 | 0 | 0 | 10 |
| srai | 0010011 | 00 | 0 | 0 | 0 | 10 |
| ori | 0010011 | 00 | 0 | 0 | 0 | 10 |
| andi | 0010011 | 00 | 0 | 0 | 0 | 10 |
| auipc | 0010111 | 00 | 0 | 0 | 0 | 11 |
| sb | 0100011 | x | 0 | 0 | 0 | 00 |
| sh | 0100011 | x | 0 | 0 | 0 | 00 |
| sw | 0100011 | x | 0 | 0 | 0 | 00 |
| add | 0110011 | 00 | 0 | 0 | 0 | 10 |
| sub | 0110011 | 00 | 0 | 0 | 0 | 10 |
| sll | 0110011 | 00 | 0 | 0 | 0 | 10 |
| slt | 0110011 | 00 | 0 | 0 | 0 | 10 |
| sltu | 0110011 | 00 | 0 | 0 | 0 | 10 |
| xor | 0110011 | 00 | 0 | 0 | 0 | 10 |
| srli | 0110011 | 00 | 0 | 0 | 0 | 10 |
| sra | 0110011 | 00 | 0 | 0 | 0 | 10 |
| or | 0110011 | 00 | 0 | 0 | 0 | 10 |
| and | 0110011 | 00 | 0 | 0 | 0 | 10 |
| lui | 0110111 | 11 | 0 | 0 | 0 | 11 |
| beq | 1100011 | x | 1 | 0 | 0 | 01 |
| bne | 1100011 | x | 1 | 0 | 0 | 01 |
| blt | 1100011 | x | 1 | 0 | 0 | 01 |
| bge | 1100011 | x | 1 | 0 | 0 | 01 |
| bltu | 1100011 | x | 1 | 0 | 0 | 01 |
| bgeu | 1100011 | x | 1 | 0 | 0 | 01 |
| jalr | 1100111 | 10 | 0 | 1 | 1 | 01 |
| jal | 1101111 | 10 | 0 | 1 | 0 | x |

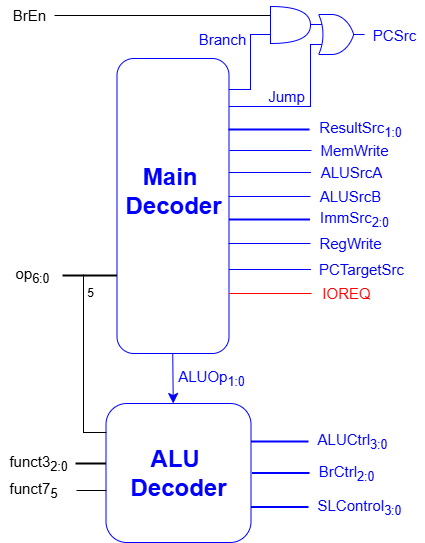
**Figure 3.2 Main Decoder - Part 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ALUOp** | **funct3** | **{op5,funct75}** | **ALUControl** | **BrCtrl** | **SLType** | **Instruction** |
| 00 | 000 | 0x | 0000 (add) | x | 0000 | lb |
|  | 001 | 0x | 0000 (add) | x | 0001 | lh |
|  | 010 | 0x | 0000 (add) | x | 0010 | lw |
|  | 100 | 0x | 0000 (add) | x | 0100 | lbu |
|  | 101 | 0x | 0000 (add) | x | 0101 | lhu |
|  | 000 | 1x | 0000 (add) | x | 1000 | sb |
|  | 001 | 1x | 0000 (add) | x | 1001 | sh |
|  | 010 | 1x | 0000 (add) | x | 1010 | sw |
| 01 | 000 | x | x | 000 | x | beq |
|  | 001 | x | x | 001 | x | bne |
|  | 100 | x | x | 100 | x | blt |
|  | 101 | x | x | 101 | x | bge |
|  | 110 | x | x | 110 | x | bltu |
|  | 111 | x | x | 111 | x | bgeu |
| 10 | 000 | 00, 01, 10 | 0000 | x | x | add, addi |
|  | 000 | 11 | 0001 | x | x | sub |
|  | 001 | 00, 10 | 1101 | x | x | sll, slli |
|  | 010 | 00, 01, 10 | 0101 | x | x | slt, slti |
|  | 011 | 00, 01, 10 | 0100 | x | x | sltu, sltiu |
|  | 100 | 00, 01, 10 | 0111 | x | x | xor, xori |
|  | 101 | 00, 10 | 1001 | x | x | srl, srli |
|  | 101 | 01, 11 | 1011 | x | x | sra, srai |
|  | 110 | 00, 01, 10 | 0011 | x | x | or, ori |
|  | 111 | 00, 01, 10 | 0010 | x | x | and, andi |
| 11 | x | x | 0000 (add) | x | x | auipc, lui |

**Figure 3.3 ALU Decoder**

# **DETAILED FUNCTIONS DESCRIPTION**

* 1. **Control Unit:**
     1. **Block Diagram and I/O Definition:**

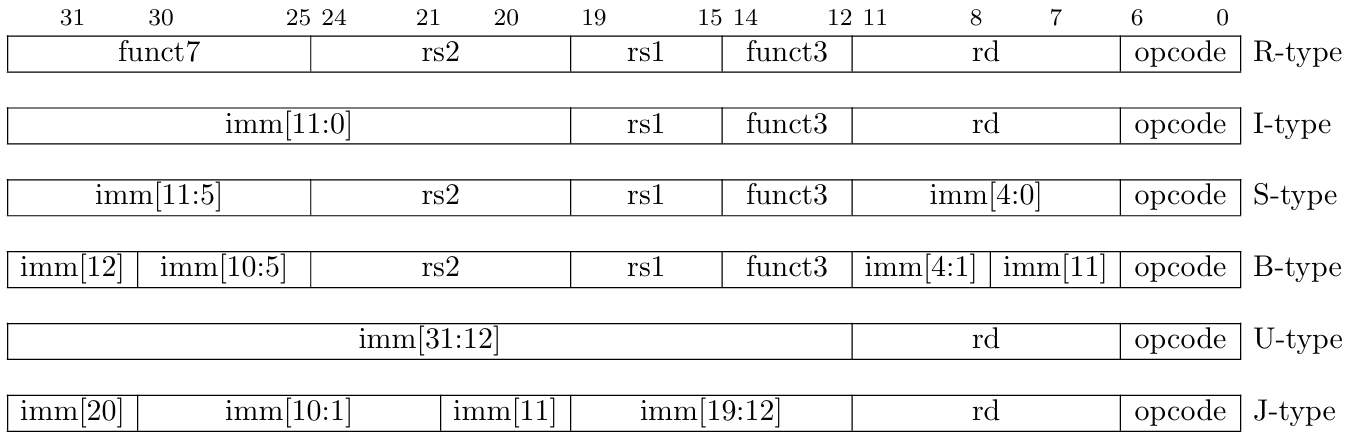
****

**Figure 4.1 Control Unit Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| op6:0 | 7 | Input | Defines the 7-bit opcode of the instruction |
| funct32:0 | 3 | Input | Specifies the 3-bit  funct3 field |
| funct75 | 1 | Input | Indicates the 1-bit  funct7 field |
| BrEn | 1 | Input | Enables branch condition evaluation from  Branch Comparator |
| PCSrc | 1 | Output | Selects between PC+4 (0) and PC target (1) |
| ResultSrc1:0 | 2 | Output | Determines the result source |
| MemWrite | 1 | Output | Controls memory write(1) or read(0) |
| ALUSrcA | 1 | Output | Selects first ALU  operand source |
| ALUSrcB | 1 | Output | Selects second ALU operand source |
| ImmSrc2:0 | 3 | Output | Defines immediate field selection |
| RegWrite | 1 | Output | Enables register  write-back |
| PCTargetSrc | 1 | Output | Selects PC target source for jumps |
| IOREQ | 1 | Output | Triggers I/O operations via I/O Block |
| ALUCtrl3:0 | 4 | Output | Sets 4-bit ALU control signal |
| BrCtrl2:0 | 3 | Output | Configures 3-bit branch control |
| SLControl3:0 | 4 | Output | Defines 4-bit shift/load control |

**Table 4.1 Control Unit I/O Definition**

* + 1. **Operation Algorithm:**



**Figure 4.2 RISC-V Base Instruction Formats**

The control unit of the processor is a sophisticated combination of a Main Decoder and an ALU Decoder, orchestrating the execution of each instruction within one clock cycle. Initially, the Main Decoder receives the 7-bit opcode signal, decodes it to identify the instruction type, such as R-type, I-type, or custom I/O operations, and generates a suite of control signals, including RegWrite, ALUSrcA, ALUSrcB, and IOREQ, which are dispatched to components like the Register File, ALU, and I/O Block.

Subsequently, the ALU Decoder takes these inputs, along with op5, funct3, funct7, and ALUOp from the Main Decoder, to determine precise ALU operations, such as addition and subtraction via the Kogge-Stone Adder, the Branch Comparator, or Store/Load type within the system, ensuring seamless integration with the processor’s datapath and external interfacing capabilities.

* 1. **Register File:**
     1. **Block Diagram and I/O Definition:**

A computer circuit diagram with many lines

AI-generated content may be incorrect.

**Figure 4.3 Register File Block Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| clk | 1 | Input | Clock signal for synchronous operations |
| nrst | 1 | Input | Negative edge resets. nrst=0 will reset all value in Register File |
| A1 | 5 | Input | Selects the first register (RS1) to read |
| A2 | 5 | Input | Selects the second register (RS2) to read |
| A3 | 5 | Input | Selects the destination register (RD) to write |
| WD3 | 32 | Input | 32-bit data to write to the selected destination register |
| WE3 | 1 | Input | Enables writing to the destination register |
| RD1 | 32 | Output | Outputs the 32-bit data read from RS1 |
| RD2 | 32 | Output | Outputs the 32-bit data read from RS2 |

**Table 4.2 Register File I/O Definition**

* + 1. **Operation Algorithm:**

The operation of the Register File occurs synchronously within one clock cycle, driven by the clk signal. The process begins by checking the nrst signal, where a low value triggers initialization, clearing all 32 registers (x0–x31) to zero, with x0 being hardwired to zero as per RISC-V specifications, ensuring it remains read-only. Upon a high nrst, the Register File reads operands by accepting 5-bit addresses A1 and A2 from instruction fields rs1 and rs2, outputting the corresponding 32-bit data as RD1 and RD2 to supply the ALU and Branch Comparator, with x0 always returning zero if addressed.

Simultaneously, if the RegWrite signal is active, the Register File prepares for write-back by receiving a 5-bit address A3 from rd, 32-bit write data WD3 from the execution result, and a 1-bit write enable WE3, enabling the update of the selected register; except x0, which remains unaffected; upon the next clk edge. This efficient operation ensures seamless data handling for the processor’s single-cycle execution while maintaining x0’s hardwired zero constraint.

* 1. **Extend Unit:**
     1. **I/O Definition:**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| Instr | 24 | Input | Bits 31:7 of instruction, containing the raw immediate fields |
| ImmSrc | 3 | Input | Selects the immediate format based on instruction type |
| ImmExt | 32 | Output | Outputs 32-bit sign-extended or zero-extended immediate value |

**Table 4.3 Extend Unit I/O Definition**

* + 1. **Operation Algorithm:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 4.4 Types of Immediate produced by RISC-V Intructions**

The Extend Unit transforms raw instruction bits into a 32-bit extended immediate for the datapath. It receives the upper 24 bits of the instruction, Instr31:7, containing the immediate field, and a 3-bit control signal ImmSrc from the Main Decoder to identify the immediate type.

Based on ImmSrc, the unit handles the immediate as follows: for ImmSrc = 000, it extracts bits 31:20 as an I-Type immediate, sign-extending to 32 bits; for ImmSrc = 001, it combines bits 31:25 and 11:7 as an S-Type immediate, sign-extending to 32 bits; for ImmSrc = 010, it assembles bits 31, 7, 30:25, and 11:8 with a 1-bit left shift as a B-Type immediate, sign-extending to 32 bits; for ImmSrc = 011, it constructs bits 31, 19:12, 20, and 30:21 with a 1-bit left shift as a J-Type immediate, sign-extending to 32 bits; and for ImmSrc = 100, it takes bits 31:12, shifts them left by 12 bits, and zero-extends to 32 bits as a U-Type immediate.

* 1. **Branch Comparator:**
     1. **Block Diagram and I/O Definition:**



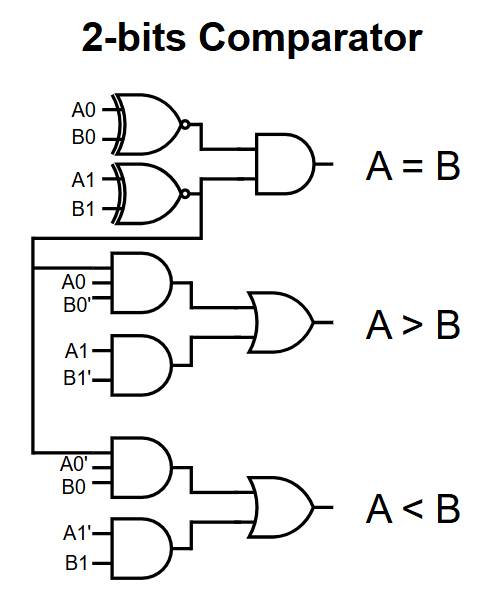
**Figure 4.5 Branch Comparator Block Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| A | 32 | Input | The first operand for comparison (RD1) |
| B | 32 | Input | The second operand for comparison (RD2) |
| BrCtrl | 3 | Input | Specifies the branch comparison type |
| BrOut | 1 | Output | Outputs the branch decision (1 for branch taken, 0 for not taken) |

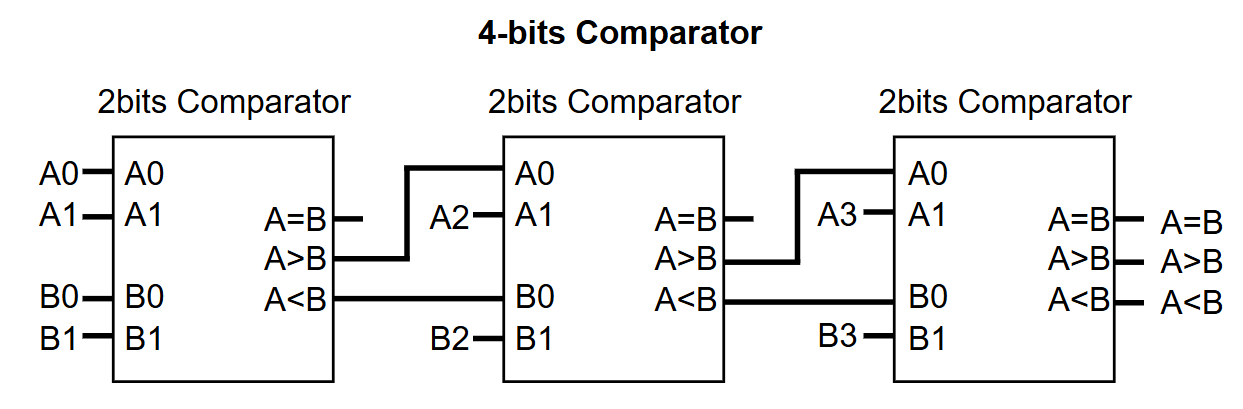
**Table 4.4 Branch Comparator I/O Definition**

* + 1. **Operation Algorithm:**
       1. Unsigned Comparator Operation Algorithm:

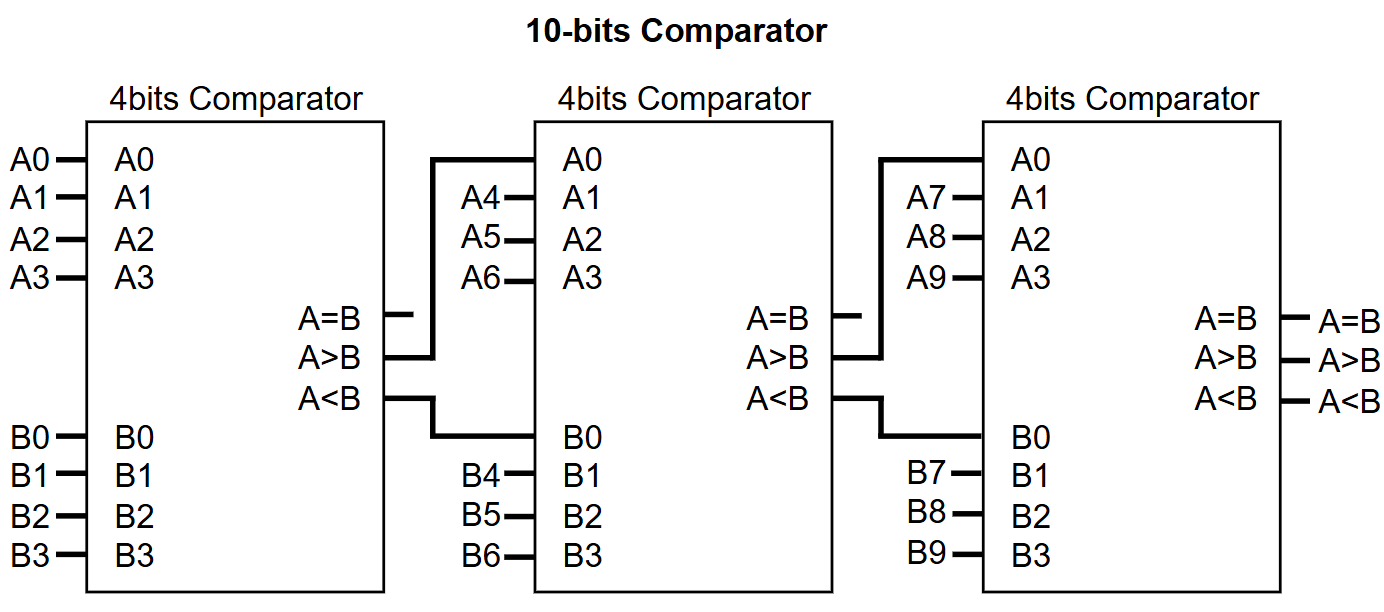
The model of the Unsigned Comparator is constructed using 2-bit comparators as the foundational block, which are cascaded to form 4-bit comparators by rippling less-than and greater-than outputs, as described by **Nagaraju et al., 2020**. These 4-bit comparators are further combined into 10-bit comparators, which, along with additional 4-bit and 2-bit modules, are interconnected to create the final 32-bit Unsigned Comparator, comparing least significant bits first and propagating results to the most significant bits. This hierarchical approach ensures efficient comparison for branch instructions like BLTU and BGEU within the Unsigned Comparator.



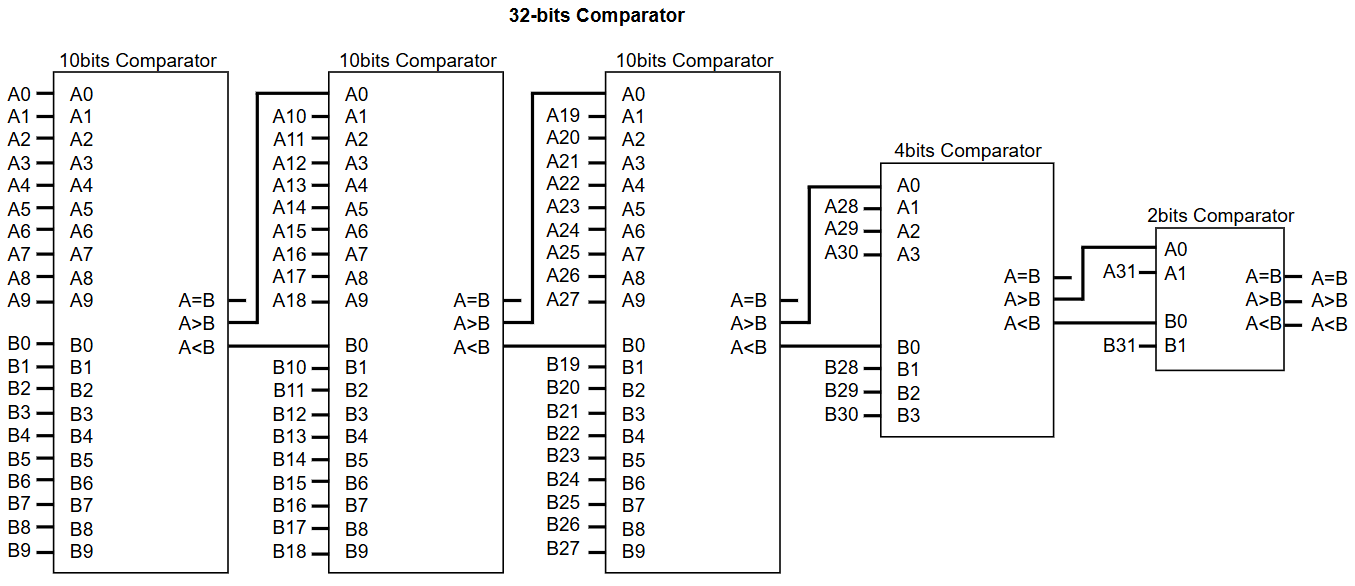
**Figure 4.6 2-Bits Comparator**



**Figure 4.7 4-Bits Comparator**



**Figure 4.8 10-Bits Comparator**



**Figure 4.9 32-Bits Comparator**

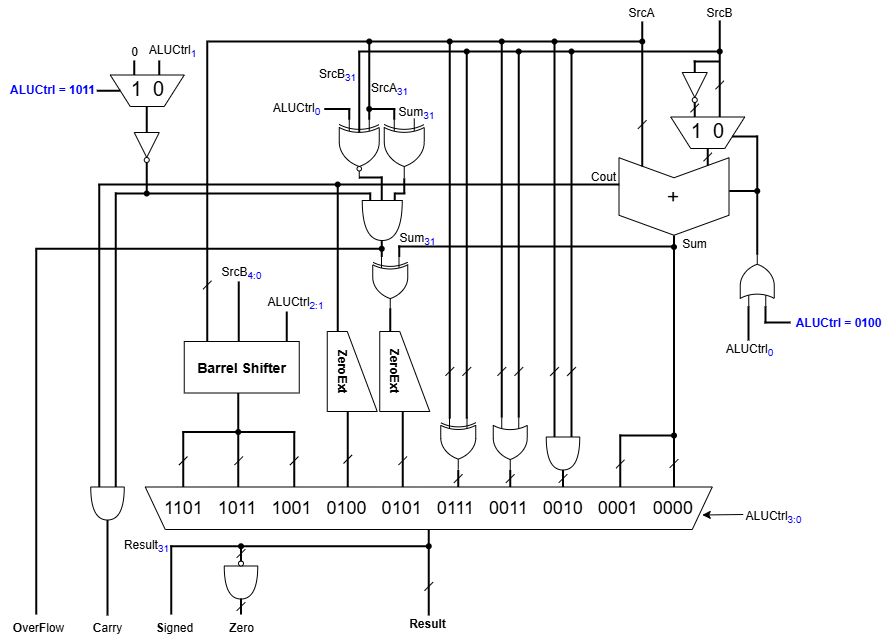
* + - 1. Branch Comparator Operation Algorithm:

The unit first performs an initial comparison to check if A equals B, yielding results for Equal and Not Equal conditions, which are identical for both signed and unsigned numbers, supporting instructions like BEQ and BNE. For Less Than and Greater or Equal comparisons, such as BLT and BGE, the unit adapts to handle both signed and unsigned operations, using BrCtrl[1] to select the comparison mode, and ultimately outputs BrOut through a 4x1 MUX based on {BrCtrl[2], BrCtrl[0]}, mapping to 00 for Equal, 01 for Not Equal, 10 for Less Than, and 11 for Greater or Equal.

For unsigned comparisons, activated when BrCtrl[1] = 0, the Branch Comparator employs an Unsigned Comparator to directly compare A and B, treating both operands as unsigned integers. This straightforward comparison determines if A is less than B for Less Than operations, or if A is greater than or equal to B for Greater or Equal operations, corresponding to instructions like BLTU and BGEU. The MUX select signal, sel, is set to 0, ensuring the output of the Unsigned Comparator is directly passed through the MUX.

For signed comparisons, triggered when BrCtrl[1] = 1, the Branch Comparator evaluates A and B as signed integers, requiring a more detailed approach due to 2’s complement representation, supporting instructions like BLT and BGE. The unit first examines the most significant bits, A[31] and B[31], to determine the signs of the operands: if the signs are the same, the Unsigned Comparator is reused to compare the numbers directly, leveraging 2’s complement properties for correct ordering; however, if the signs differ, the unit computes the MUX select signal sel as A[31] XOR B[31], and for Less Than, a sel of 1 (indicating A[31] = 1 and B[31] = 0, meaning A is negative and B is positive) sets the output to 1, confirming A is less than B. This signed comparison result, along with Equal and Not Equal outcomes, feeds into the 4x1 MUX to produce BrOut.

* 1. **ALU:**
     1. **Block Diagram and I/O Definition:**



**Figure 4.10 ALU Block Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| SrcA | 32 | Input | The first operand, selected between Register File RD1 or PC |
| SrcB | 32 | Input | The second operand, selected between Register File RD2 or ImmExt |
| ALUCtrl | 4 | Input | Specifies the operation to perform |
| Result | 32 | Output | Outputs the 32-bit result of the ALU operation |

**Table 4.5 ALU I/O Definition**

* + 1. **Operation Algorithm:**
       1. ALU:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | ALUCtrl3 | ALUCtrl2 | ALUCtrl1 | ALUCtrl0 |
| ADD | 0 | 0 | 0 | 0 |
| SUB | 0 | 0 | 0 | 1 |
| AND | 0 | 0 | 1 | 0 |
| OR | 0 | 0 | 1 | 1 |
| SLTU | 0 | 1 | 0 | 0 |
| SLT | 0 | 1 | 0 | 1 |
| XOR | 0 | 1 | 1 | 1 |
| SRL | 1 | 0 | 0 | 1 |
| SAR | 1 | 0 | 1 | 1 |
| SLL | 1 | 1 | 0 | 1 |

**Table 4.6 ALU Operation Control**

The ALU using a 4-bit ALUCtrl signal to execute various operations, producing a 32-bit Result along with flags for Overflow, Carry, Signed, and Zero. The ALU integrates components like the Kogge-Stone Adder for arithmetic operations and the Barrel Shifter for shift operations, ensuring efficient single-cycle execution.

The operation starts with the ALUCtrl signal, which selects the ALU’s mode based on the operation table mention above. This control signal directs the ALU to perform arithmetic, logical, or shift operations by routing inputs to the appropriate internal modules. For arithmetic operations, the Kogge-Stone Adder computes the Sum, while the Barrel Shifter handles shift operations, with the final Result chosen via a multiplexer based on ALUCtrl.

For the MUX selection and Cin of the Adder, specific logic is applied where ALUCtrl0 = 1 or ALUCtrl=0100 is used, to perform subtraction and generate the Carry flag for SLT, ensuring all cases are covered. Additionally, a NOT of the MUX output is implemented when ALUCtrl=1011 or when selecting "0" to compute accurate flags for future usage.

* + - 1. Kogge-Stone ADDER:

A diagram of a diagram

AI-generated content may be incorrect.

**Figure 4.11 Black and Gray Cells Diagram**

A black and white grid with squares and lines

AI-generated content may be incorrect.

**Figure 4.12 32-Bits Kogge-Stone Block Diagram**

The Kogge-Stone Adder performs 32-bit addition on operands SrcA and SrcB in three distinct stages, achieving a logarithmic delay of O(log₂(32)) to support high-speed operations like ADD and ADDI.

In the Pre-processing stage, the adder computes initial generate signals Gi = SrcAi  & SrcBi and propagate signals Pi = SrcAi ⊕ SrcBi for each bit position i from 0 to 31, preparing the groundwork for carry computation across all bits. These signals are derived directly from the input operands, enabling the subsequent stages to efficiently handle carry propagation.

In the Prefix-Computation stage, the Kogge-Stone Adder constructs a tree structure using black and grey cells to compute group generate Gi:k and group propagate Pi:k signals. This stage, spanning log₂(32) = 5 levels for a 32-bit adder, recursively combines signals across bit positions, utilizing the low fan-out and optimal depth of the Kogge-Stone structure to ensure rapid carry propagation with a computational complexity of 32 × log₂(32) - 32 + 1 nodes. The black cells compute both G and P signals, while grey cells compute only G signals needed for the final stage, ensuring efficient parallel processing.

In the Post-processing stage, the adder computes the final carry bits Ci = Gi:0 for each bit position i, where C-1 is typically 0 for addition without a carry-in, and then calculates the sum bits Si = Pi ⊕ Ci-1 for i from 0 to 31, producing the 32-bit Result output. This stage finalizes the addition by combining the propagate signals with the computed carries, ensuring the correct sum is delivered to the ALU’s output.

* + - 1. Barrel Shifter:

The Barrel Shifter uses a 2-bit control signal Sel to determine its operation mode, where Sel[0] selects between logical (0) and arithmetic (1) operations, deciding whether zeros or the sign bit A[31] fills shifted positions, and Sel[1] sets the direction, with 0 for right and 1 for left shifts. The operating modes:

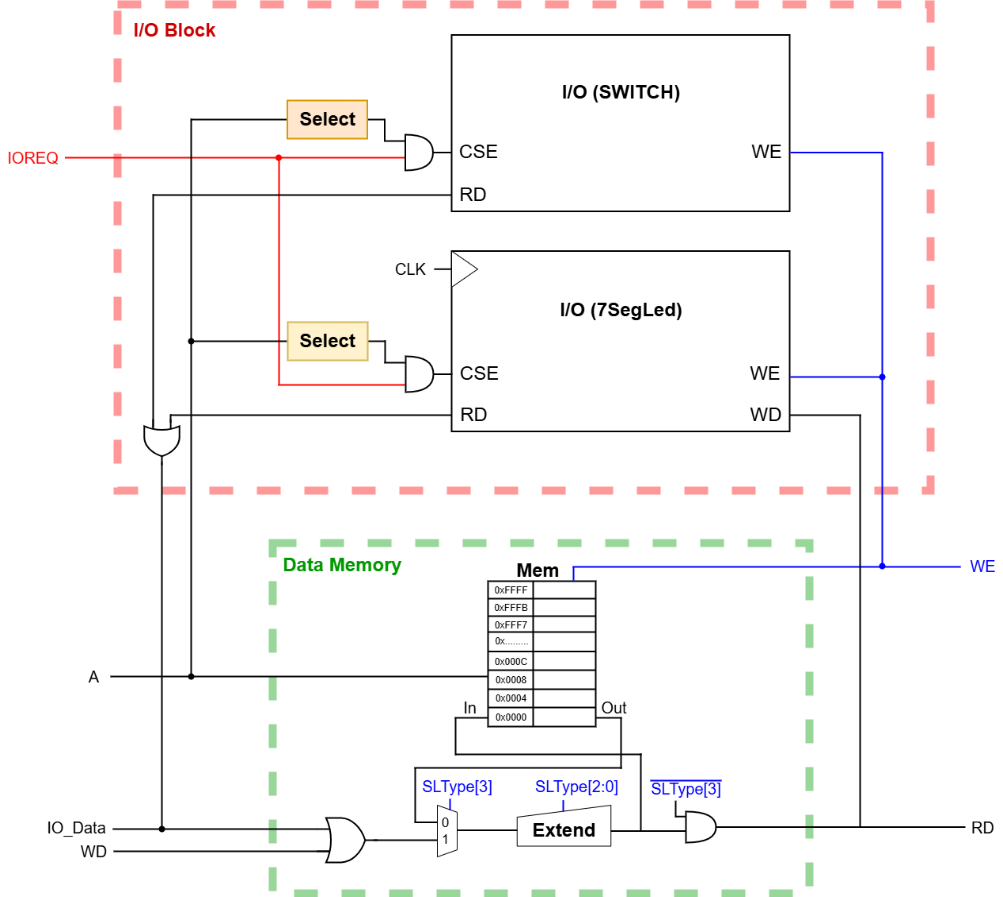
* Sel[1]=0 and Sel[0]=0, it performs a shift right logical, filling upper bits with zeros.
* Sel[1]=0 and Sel[0]=1, it performs an arithmetic shift right (ASR), filling with the sign bit.
* Sel[1]=1 and Sel[0]=0, it performs a shift left logical, filling lower bits with zeros.
* Sel[1]=1 and Sel[0]=1, it performs an arithmetic shift left (ASL), also filling lower bits with zeros while preserving the sign.

These operations are executed across five stages, controlled by the shift amount B[4:0], to produce the output Y in the ALU, supporting instructions like SRL, SRA, SLL, and SLA.Below is an example of an 8-bit Barrel Shifter to illustrate how it works:



**Figure 4.13 Barrel Shifter Block Diagram**

* 1. **Data Memory and I/O Block:**
     1. **Block Diagram and I/O Definition:**



**Figure 4.14 I/O Block and Data Memory Block Diagram**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| clk | 1 | Input | Clock signal for synchronous operations |
| nrst | 1 | Input | Negative edge resets. nrst=0 will clear I/O Register |
| IOREQ | 1 | Input | Request signal to initiate I/O operation |
| A | 32 | Input | Address for I/O operation |
| WD | 32 | Input | Write data to external devices |
| IO\_Data | 32 | Output | Data output to Memory |

**Table 4.7 I/O Block I/O Definition**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bit-Width | Direction | Function Description |
| clk | 1 | Input | Clock signal for synchronous operations |
| A | 32 | Input | Address for Memory operation |
| IO\_Data | 32 | Input | Data input from external devices |
| WD | 32 | Input | Write data to Memory from system |
| SLType | 4 | Input | Selects Store or Load operation type |
| RD | 32 | Output | Read data returned to the processor |

**Table 4.8 Data Memory I/O Definition**

* + 1. **Operation Algorithm:**

The I/O Blockoperates using the IOREQ signal as the main way to address I/O devices, defining how addresses are assigned and distinguished from memory locations in a manner inspired by standard IO techniques. When IOREQ=1 and the address matches either 0xFF0 for switches (SW) or 0xFF8 for the 7-segment LED (7SegLed), the I/O Block activates to process Store or Load commands, while all other addresses are treated as memory locations, creating disjoint address sets without an additional signal like MREQ.

A diagram of a computer chip

AI-generated content may be incorrect.

**Figure 4.15 I/O Addressing Map**

Unlike traditional standard IO, which uses separate address spaces with explicit signals such as MREQ for memory and IOREQ for I/O, or memory-mapped IO, which integrates I/O and memory into the same address space using address values (e.g., the last 4 bits as 1111 for I/O) and selection circuitry, this design relies on IOREQ with fixed addresses to select I/O devices, ensuring clear separation from memory.

For a Store instruction, the processor directs the Data Memory to store a value from the I/O, where the 32-bit WD (write data) is OR with IO\_Data to form the stored value, using the address provided by the 32-bit A input.The Load instruction reverses this process, transferring data from the Data Memory to the I/O register based on the same address conditions. The Data Memory includes a 4-bit SLType to define the operation mode, supporting load and store variants:

* 0000 for lb (load byte).
* 0001 for lh (load halfword).
* 0010 for lw (load word).
* 0100 for lbu (load byte unsigned).
* 0110 for lhu (load halfword unsigned).
* 1000 for sb (store byte).
* 1001 for sh (store halfword).
* 1010 for sw (store word).

To ensure data integrity, IO\_Data is masked to zero when the address does not match 0xFF0 or 0xFF8, preventing incorrect data from being loaded into the Memory, thus maintaining the I/O Block’s reliability within the single-cycle datapath.

# **RTL STIMULATION RESULT**

* 1. **Synthesis.**

The design was successfully synthesized using Quartus, confirming the functionality. The synthesis process completed without errors, and the resource usage results are presented below to evaluate the design’s hardware implementation.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 5.1 Resource Usage Results**

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 5.2 Timming Result**

The design uses 1,761 ALMs (4%), 2726 registers, with no DSP blocks, indicating efficient FPGA utilization suitable for the ALU, Barrel Shifter, and I/O Block. The low usage leaves room for optimization or additional features, though reducing I/O Block logic could further enhance performance if needed.

* 1. **Testbench/environment setup for components.**
     1. **Kogge-Stone Adder.**

A computer screen shot of a program

AI-generated content may be incorrect.

**Figure 5.3 Kogge-Stone Testbench Results**

The Kogge-Stone Adder testbench successfully verified its functionality within the ALU, confirming accuracy of. Its parallel prefix architecture ensured efficient flag generation for Carry and Overflow, supporting instructions like SLT and SLTU.

* + 1. **Branch Comparator Unit.**

**A computer screen shot of a computer

AI-generated content may be incorrect.**

**Figure 5.4 Branch Comparator Testbench Results**

The Branch Comparator Unit (BCU), implemented as a separate block from the ALU, passed its testbench verification by directly comparing register values from the Register File to evaluate branch conditions like BEQ, BNE, BLT, and BGE, ensuring accurate branch handling.

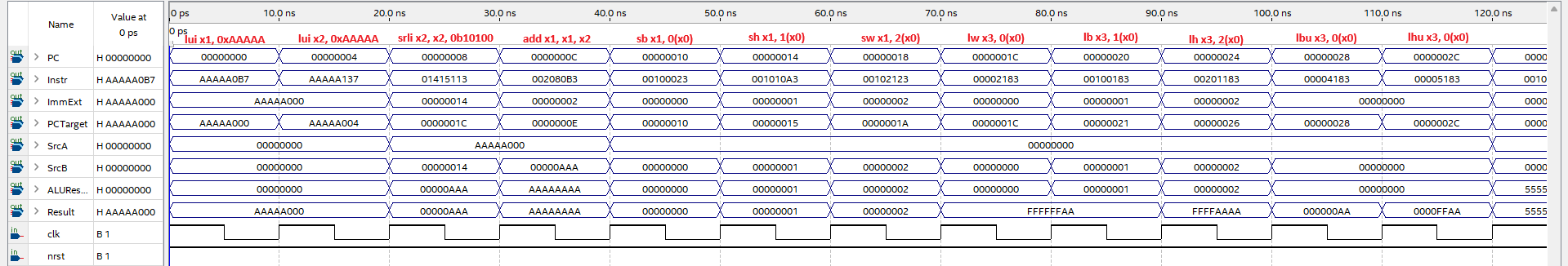
* 1. **Simulation Verification.**
     1. **Instruction Code for Testing.**

|  |  |  |
| --- | --- | --- |
| Address | Assembly Code | Comment |
| 0x0000 | lui r1, 0xAAAAA | % r1 = 0xAAAA A000 |
| 0x0004 | lui r2, 0xAAAAA | % r2 = 0xAAAA A000 |
| 0x0008 | srli r2, r2, 0b10100 | % r2 = 0x0000 0AAA |
| 0x000C | add r1, r1, r2 | % r1 = 0xAAAA AAAA |
| 0x0010 | sb r1, 0(r0) | % [0x0000] = 0xFFFF FFAA |
| 0x0014 | sh r1, 1(r0) | % [0x0001] = 0xFFFF AAAA |
| 0x0018 | sw r1,2(r0) | % [0x0002] = 0xAAAA AAAA |
| 0x001C | lw r3, 0(r0) | % r3 = 0xFFFF FFAA |
| 0x0020 | lb r3, 1(r0) | % r3 = 0xFFFF FFAA |
| 0x0024 | lh r3, 2(r0) | % r3 = 0xFFFF AAAA |
| 0x0028 | lbu r3, 0(r0) | % r3 = 0x0000 00AA |
| 0x002C | lhu r3, 0(r0) | % r3 = 0x0000 FFAA |
| 0x0030 | srli r2, r1, 1 | % r2 = 0x5555 5555 |
| 0x0034 | or r3, r1, r2 | % r3 = 0xFFFF FFFF |
| 0x0038 | and r4, r2, r3 | % r4 = 0x5555 5555 |
| 0x003C | xor r4, r4, r4 | % clear r4 (r4=0) |
| 0x0040 | lui r5, 0x00000 | % r5 = a (ex: a=5) |
| 0x0044 | addi r5, r5, 0x005 |  |
| 0x0048 | lui r6, 0x00000 | % r6 = b (ex: b=5) |
| 0x004C | addi r6, r6,0x005 |  |
| 0x0050 | jal r10, mult | % == Call mult |
| 0x0054: end | auipc r9, 0 | % show PC when finish function |
| 0x0058 | add r7, r7, r0 | % show value on datapath |
| 0x005C | jal r0, end | % junp back to end Label |
| 0x0060: mult | slt r4, r6, r0 | % check if b = 0 |
| 0x0064 | bne r0, r4, end | % if b=0, end funtion |
| 0x0068: | addi r8, r8, 0x001 |  |
| 0x006C: loop | add r7, r7, r5 | % start multiply, result = r7 |
| 0x0070 | sub r6, r6, r8 |  |
| 0x0074 | blt r0, r6, loop |  |
| 0x0078 | add r10, r10, r0 |  |
| 0x007C | jalr r0, r10, 0 | % == Return |

**Table 5.1 Functional Test Instructions**

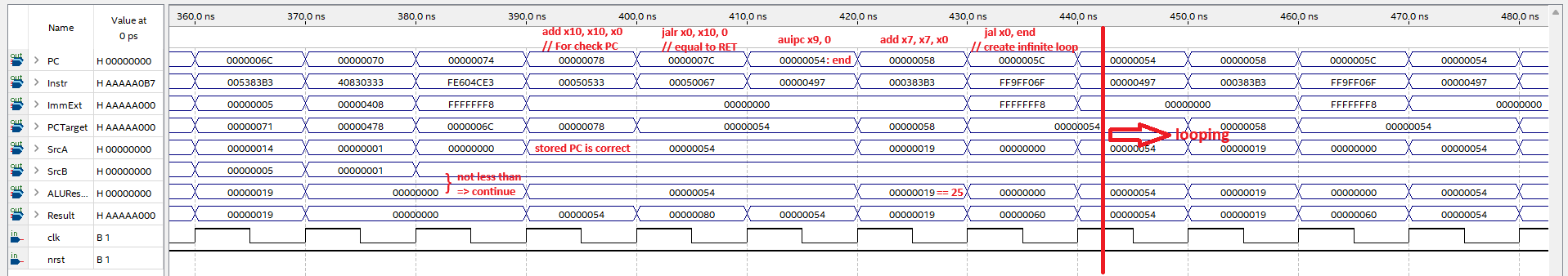
Using the instructions above, the ALU, Barrel Shifter, Kogge-Stone Adder, Register File, and I/O Block are checked for correct operation. This ensures the design performs as intended after synthesis.

* + 1. **Results.**

****

**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.**

**Figure 5.5 Functional Stimulation Results**

# **FPGA PROVEN**

* 1. **Intruction Code for Testing.**

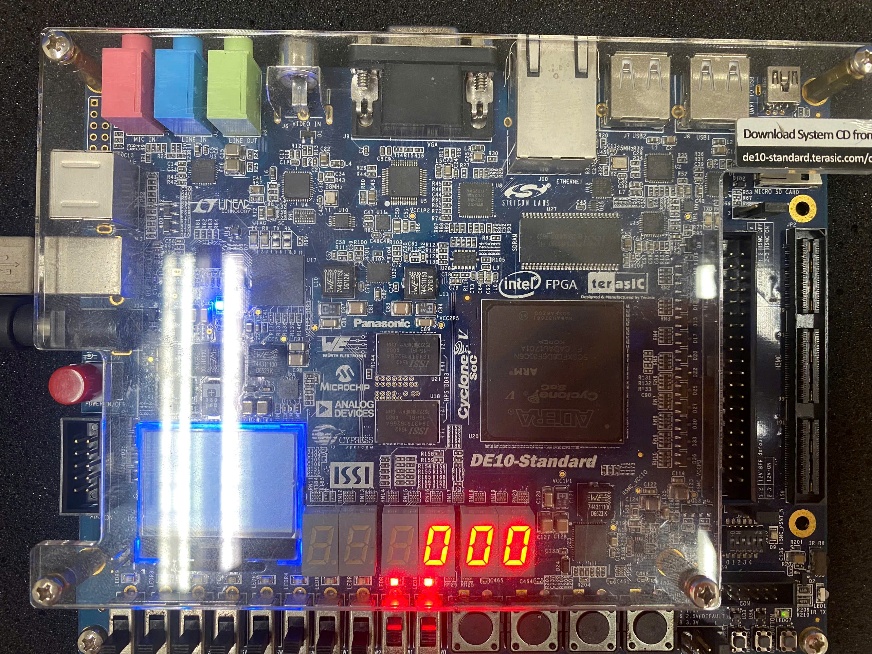
|  |  |  |
| --- | --- | --- |
| Address | Assembly Code | Comment |
| 0x0000: loop | addi x19, x0, -16 | % Switch address = 0xFFFF FFF0 |
| 0x0004 | sw x0, 0(x19) | % Save Switch value to Memory |
| 0x0008 | lw x24, 0(x19) | % Load Switch value to x24 |
| 0x000C | addi x21, x0, 0x000 | % x21 = fibo\_0 |
| 0x0010 | addi x22, x0, 0x001 | % x22 = fibo\_1 |
| 0x0014 | beq x24, x0, fib\_done | % if input\_val == 0, return 0 |
| 0x0018: fib\_loop | add x23, x21, x22 | % fibo\_next = fibo\_0 + fibo\_1 |
| 0x001C | add x21, x0, x22 | % fibo\_0 <= fibo\_1 |
| 0x0020 | add x22, x0, x23 | % fibo\_1 <= fibo\_next |
| 0x0024 | bltu x24, x22, fib\_done | % if fibo\_1 > input\_val, done function |
| 0x0028 | jal x0, fib\_loop | % jump back to fib\_loop |
| 0x002C: fib\_done | add x25, x0, x21 | % results save in x25 = fibo\_0 |
| 0x0030 | addi x20, x0, -8 | % 7SegLed address = 0xFFFF FFF8 |
| 0x0034 | sw x25, 0(x20) | % Store Results to 7SegLed address |
| 0x0038 | add x25, x0, x25 | % Check x25 |
| 0x003C | lw x0, 0(x20) | % Load Results from 7SegLed address to 7SegLed |
| 0x0040 | jal x0, loop | % jump back to loop |

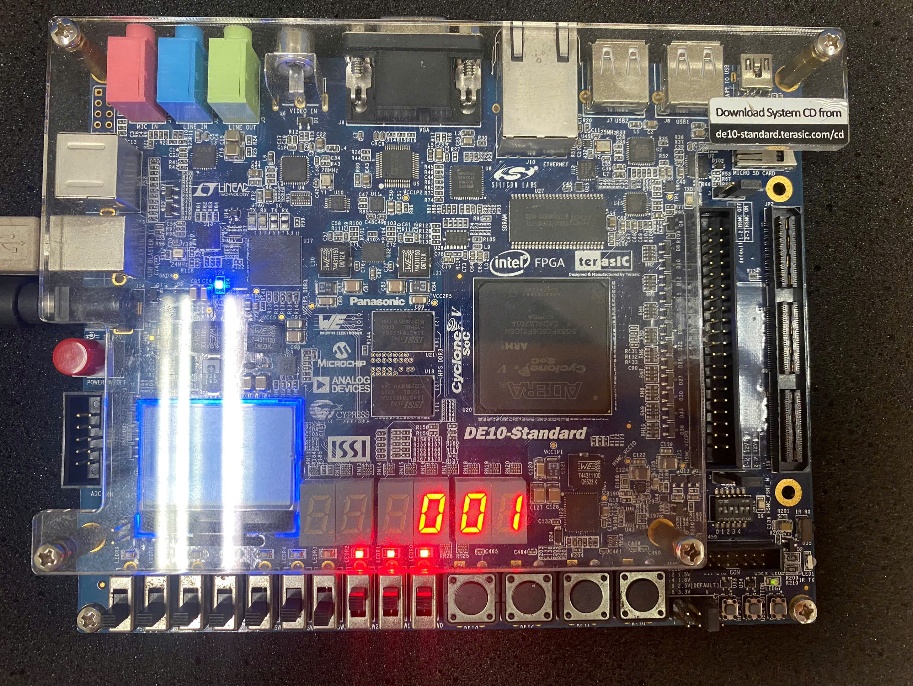
**Table 6.1 Fibonannci Calculation Instructions**

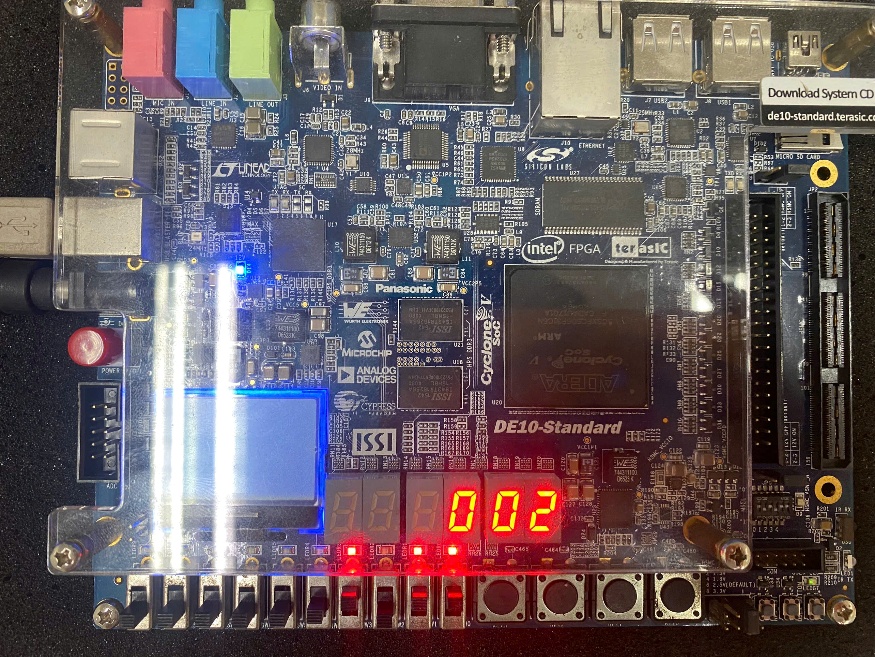
The provided instructions are used to implement a Fibonacci application on the RISC-V RV32I processor, validating its practical functionality. The processor reads input values from the switches (SW) at address 0xFF0, stores them in the system for Fibonacci calculation using the ALU and Register File, and then displays the result on the 7-segment LED (7SegLed) at address 0xFF8 via the I/O Block. This process verifies the processor’s ability to handle I/O operations and arithmetic computations accurately.

* 1. **Results.**

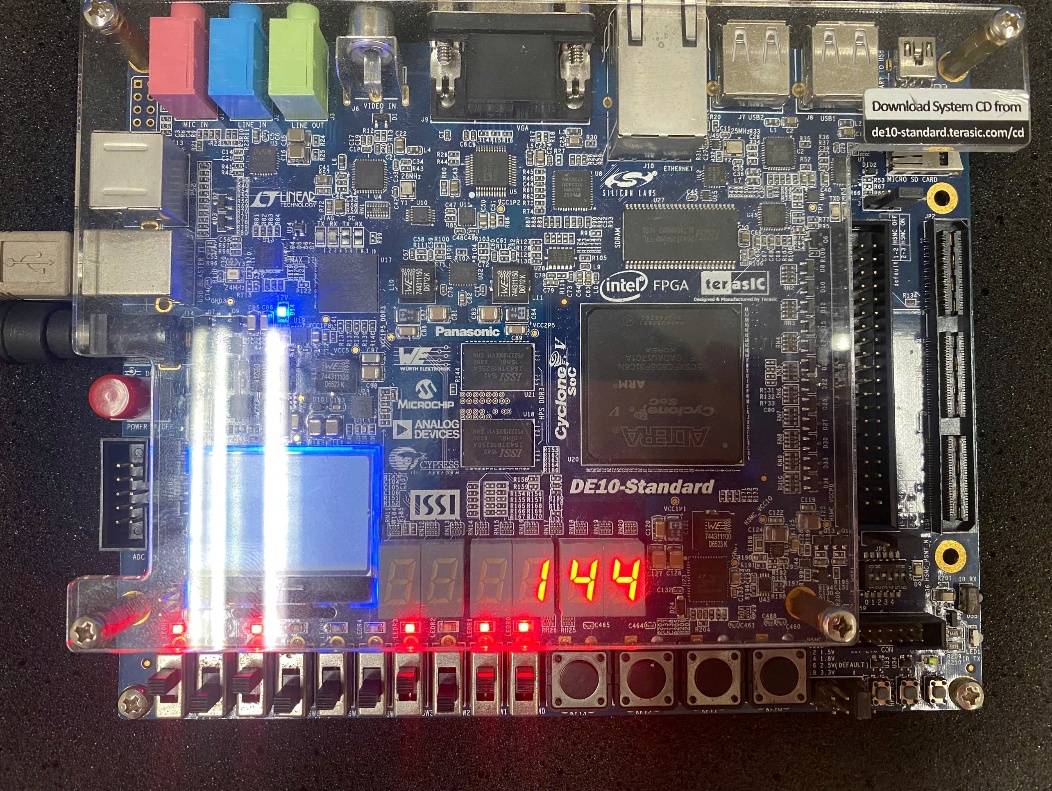
The results images of FPGA test:

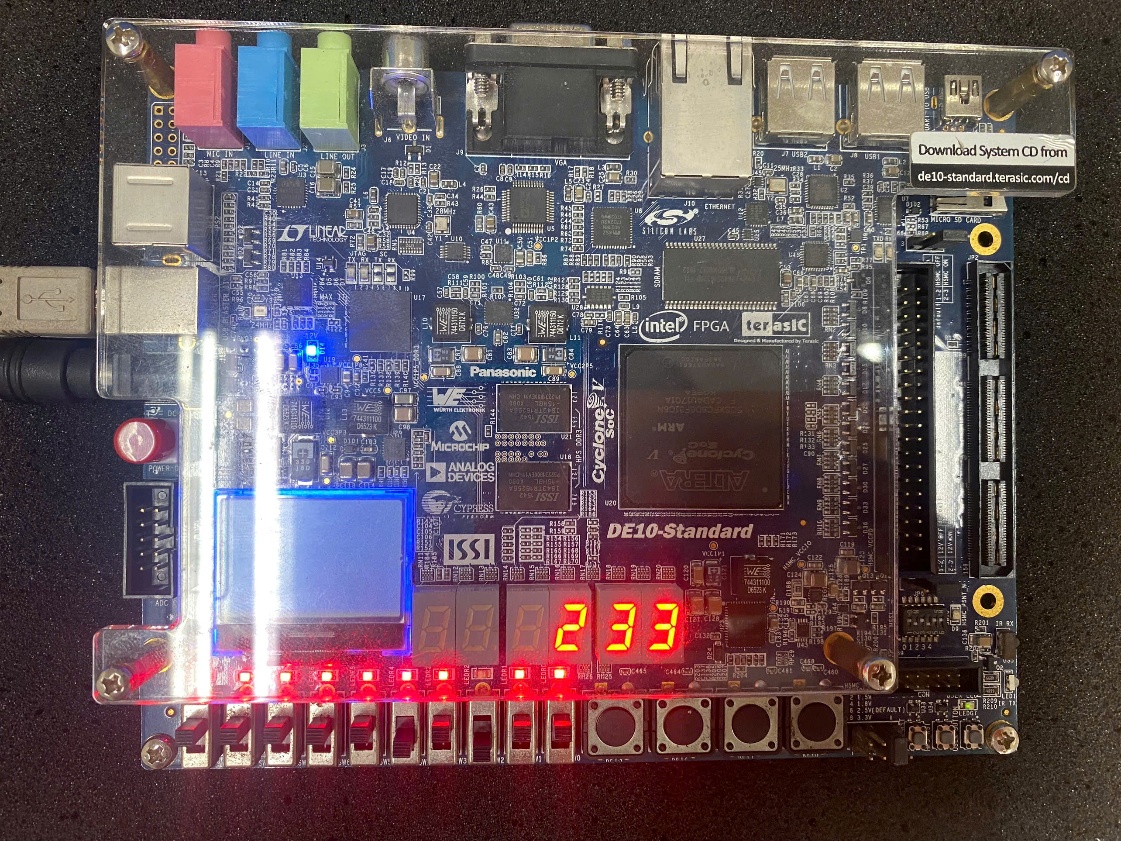












**Figure 6.1 FPGA Test Results**

**Appendix**

|  |  |
| --- | --- |
| [1] | D. H. Sarah L. Harris, Digital Design and Computer Architecture-RISC-V-Edition, 2022. |
| [2] | D. P. &. A. Waterman, The RISC-V Reader: An Open Architecture Atlas, Beta Edition, 0.0.1, October 4, 2017. |
| [3] | "Addressing IO devices," [Online]. Available: https://www.cs.emory.edu/~cheung/Courses/355/Syllabus/6-io/addressing.html. |
| [4] | A. K. A. D. P. G. PRAGATI SACHAN, "BARREL SHIFTER," ISSN: 2348-4098, SEP 23, 2014. |
| [5] | S. G. R. &. R. Kalaimathi, "Design and Analysis of Kogge-Stone and Han-Carlson Adders in 130nm CMOS Technology," e-ISSN: 2348-6848 & p-ISSN: 2348-795X , March 2018. |
| [6] | P. A. B. V. R. R. R. M. V. Siva Nagarajua, "Design and Implementation of Low Power 32-bit Comparator," ICICNIS 2020, 21 Jan 2021. |