Design and Implementation of a 32-bit Single-Cycle RISC-V Processor

Aasheik Saran S
CeNSE
IISc Bangalore
Bangalore, India
aasheiks@iisc.ac.in

Hariram P
DESE
IISc Bangalore
Bangalore, India
hariramp@iisc.ac.in

***Abstract*—This project presents the design and implementation of a 32-bit single-cycle RISC-V processor, compliant with the RV32I instruction set architecture. The processor executes each instruction in a single clock cycle, ensuring minimal control complexity and predictable performance. The design includes key components such as the ALU, register file, immediate generator, instruction and data memory units, and a control unit orchestrating datapath operation. Verilog HDL is used for modeling, and implementation is carried out on the Digilent BASYS3 FPGA board featuring the Xilinx Artix-7 (XC7A35T-CPG236C) chip. The processor implements a sub set of total RISC-V processor. Functional correctness is validated through simulation and on-chip testing(Xilinix Design Suite) using custom programs written in C, compiled with the RISC-V GCC toolchain, and converted into memory initialization files. The implementation showcases the practical feasibility of a simple yet functional processor core for educational and embedded applications.**

***Index Terms*—RISC-V,Cache,ALU,CPU**

I. Introduction

The **RISC-V architecture** has emerged as a revolutionary open-source **instruction set architecture (ISA)**, offering a compelling alternative to proprietary ISAs like ARM, x86, or MIPS. Designed at the University of California, Berkeley, RISC-V is grounded in the principles of simplicity, modularity, and extensibility. Its open and license-free nature has accelerated adoption across both academic research and industrial product development. As described by Patterson and Waterman (2017), RISC-V is suitable not only for teaching but also for real-world implementations ranging from microcontrollers to high-performance processors [1].

In the context of computer architecture education and early-stage system design, **single-cycle processor architectures** play a pivotal role. In such designs, each instruction is completed in a single clock cycle, simplifying timing and control logic. While this simplicity limits achievable clock frequency, it offers an ideal platform for learning, functional validation, and performance profiling. Harris and Harris (2015) emphasize that single-cycle designs help students grasp the entire instruction execution process without the complexity of pipelining or hazard resolution [2].

This project focuses on the design and implementation of a **32-bit single-cycle RISC-V processor** that supports a subset of the RV32I instruction set. The instruction set includes basic arithmetic, logical, load/store, branch, and LUI instructions. However, instructions such as JALR, AUIPC, and system-level instructions like ECALL are deliberately excluded to simplify the control path and datapath design.

To enhance performance without adding significant architectural complexity, this processor integrates a direct-mapped instruction cache. As noted in classic architectural texts by Hennessy and Patterson (2017), cache memories are essential for mitigating latency issues inherent in memory access. A **direct-mapped cache**—the simplest cache organization—assigns each memory block to exactly one cache line, enabling fast lookup and reduced hardware overhead [4]. The implemented instruction cache uses a **4-word block size** to improve spatial locality during instruction fetches.

Previous research and open-source hardware projects serve as important benchmarks. The PicoRV32 core is an example of a compact, resource-efficient RISC-V core with flexible configurability, though it typically adopts multi-cycle or finite-state execution strategies. The VScale processor, developed by UC Berkeley, provides a single-cycle implementation but includes **broader instruction support**. These efforts demonstrate the feasibility of implementing RISC-V processors on resource-constrained platforms like FPGAs.

Further, Lopes et al. (2020) present a single-cycle RISC-V processor tailored for educational use and implemented on an FPGA. Their work emphasizes the trade-offs between **clock frequency**, **area usage**, and instruction support in minimalist RISC-V designs [5]. This project builds upon such insights, emphasizing **FPGA efficiency** and **architectural clarity** while adding cache support—an often-overlooked yet vital component for practical performance in real-time applications.

The processor is modeled using *Verilog HDL* and implemented on a Digilent BASYS3 FPGA development board, which houses the Xilinx Artix-7 XC7A35T-CPG236C. Programs written in C are compiled using the **RISC-V GCC toolchain**, translated into binary, and stored in instruction memory for execution. Simulation and on-board validation verify functionality, performance, and instruction compliance.

Overall, this work demonstrates the feasibility and advantages of a cache-enabled **single-cycle RISC-V processor** tailored for both educational and embedded system applications.

It serves as a foundational platform for future enhancements such as pipelining, branch prediction, forwarding logic, and multi-level cache hierarchies.

II. Motivation

The driving need to create and deploy a **32-bit single-cycle RISC-V processor** is derived from the increasing demand for open-source, customizable, and pedagogically friendly processor architectures. **RISC-V** is unique because it has a **modular architecture** and open licensing, meaning developers and researchers are free to experiment and implement processor cores without either legal entanglements or financial considerations. This openness renders it extremely appropriate in academic settings, embedded systems, and research-oriented development. A **one-cycle design**, while constraining in performance, is a great learning base because it makes the process of executing instruction and the logic of control easier, which makes the inner working of a CPU more understandable to students and programmers alike. Adding a **direct-mapped instruction cache** in the processor design helps close this learning gap between theoretical education and actual performance, bringing in ideas of memory hierarchy without increasing the level of complexity. In addition, the utilization of FPGA platforms such as the **BASYS3** facilitates learning by doing and quick prototyping, promoting greater insight into digital systems and hardware-software co-design. In total, this project is a learning aid as well as a stepping stone toward future processor architectures.

III. Background Study

Processor architecture has seen considerable development, with milestones in the evolution of **RISC (Reduced Instruction Set Computing)** and **CISC (Complex Instruction Set Computing)** systems. RISC architectures such as RISC-V, emphasize simplifying the instruction set to enable faster execution and pipelining in more complicated designs. This simplicity is in contrast to the more complex CISC designs, employing a wider instruction set, yet at times being more flexible but more difficult to optimize for performance. RISC-V, as an open-standard and extensible architecture, has been found to be broadly used in both academia and industry. Its design encourages ease of implementation, with a modular instruction set that facilitates customization depending on the particular application needs.

In processor design, the performance of a system relies significantly on how well it can **access and process instructions and data**. **A single-cycle processor processes all phases** of instruction processing, including **fetch, decode, execute, access to memory and write-back**, within a single clock cycle. This kind of design makes the control logic simpler and easier to verify, although it places constraints on the clock speed and performance efficiency that can be achieved. Although more sophisticated processor architectures tend to implement multi-cycle or pipelined designs, single-cycle processors allow a basic grasp of CPU operation fundamentals.

To enhance instruction-fetching efficiency, **instruction caches** are widely used in contemporary processors. Instructions most often accessed are cached near the processor to reduce the time required to fetch instructions from main memory by a large factor. **Direct-mapped cache** is one of the easiest forms of cache, whereby every block of memory directly corresponds to a unique cache line. This technique is easy to implement while still ensuring significant performance boosts over non-cache systems, particularly in terms of access time and performance.

Over the past few years, **FPGAs (Field Programmable Gate Arrays)** have gained increased popularity as a platform to implement processor cores, especially for educational and prototyping reasons. **FPGAs** enable dynamic hardware designs that can be adjusted and tuned in real time, allowing designers to rapidly test and experiment with different processor configurations. FPGA-based systems in processor design offer a low-cost, highly programmable environment to learn the complexity of **CPU architectures** and conduct hardware-software co-design.

The embedding of **RISC-V processors** in FPGA platforms enables students, engineers, and researchers to investigate processor design, try out various memory systems, and analyze the complexity-performance-resource trade-offs. This background provides the context for the design and implementation of a **32-bit single-cycle RISC-V processor** with on-chip instruction cache with the goal of achieving both educational simplicity and realistic performance.

IV. Design Space Exploration

Design space exploration (DSE) is an essential step in the development of any hardware system, but especially for **processor architectures**. DSE entails exploring systematically various design alternatives and trading off among performance, resource consumption, power dissipation, and other factors to arrive at the best possible design solution. In the case of a **32-bit single-cycle RISC-V processor**, DSE assists in deciding the optimal configuration of the processor's **datapath, control unit, memory system**, and other critical components. This facilitates the designer to realize the effect of **different architectural choices** and to select the most appropriate according to the objectives of the project.

A. Instruction Set Architecture(ISA)

The first step in the design space exploration for a **RISC-V processor** involves determining the subset of the **RISC-V ISA** to implement. The RISC-V instruction set is modular, with different base and optional extensions. In this project, the base RV32I instruction set has been chosen, which includes 32-bit instructions for integer arithmetic, logical operations, and control flow. A careful selection of the instruction set is critical to ensure that the processor meets the performance and resource requirements while maintaining simplicity for educational purposes. Excluding certain instructions such as **JALR (Jump and Link Register) and AUIPC (Add Upper Immediate to PC)** reduces complexity and streamlines the

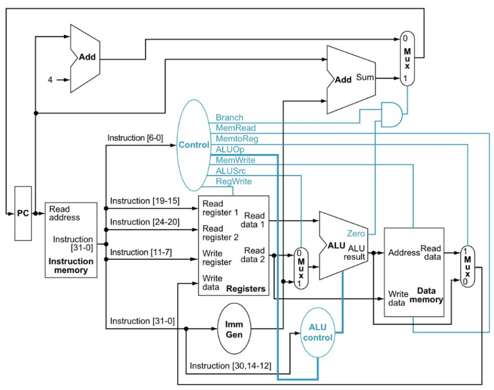
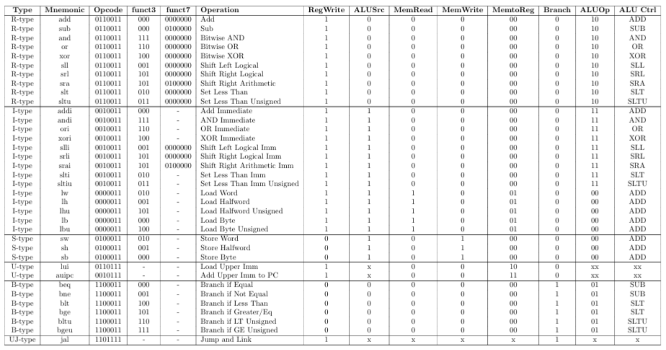


Fig. 1: Block Diagram



Type	Mnemonic	Opcode	funct3	funct7	Operation	RegWrite	ALUSrc	MemRead	MemWrite	MemtoReg	Branch	ALUOp	ALU Ctrl
R-type	add	0110011	000	0000000	Add	1	0	0	0	00	0	10	ADD
R-type	sub	0110011	000	0100000	Sub	1	0	0	0	00	0	10	SUB
R-type	and	0110011	111	0000000	Bitwise AND	1	0	0	0	00	0	10	AND
R-type	or	0110011	110	0000000	Bitwise OR	1	0	0	0	00	0	10	OR
R-type	xor	0110011	100	0000000	Bitwise XOR	1	0	0	0	00	0	10	XOR
R-type	sll	0110011	001	0000000	Shift Left Logical	1	0	0	0	00	0	10	SLL
R-type	srl	0110011	101	0000000	Shift Right Logical	1	0	0	0	00	0	10	SRL
R-type	sra	0110011	101	0100000	Shift Right Arithmetic	1	0	0	0	00	0	10	SRA
R-type	slt	0110011	010	0000000	Set Less Than	1	0	0	0	00	0	10	SLT
R-type	sltu	0110011	011	0000000	Set Less Than Unsigned	1	0	0	0	00	0	10	SLTU
I-type	addi	0010011	000	-	Add Immediate	1	1	0	0	00	0	11	ADD
I-type	andi	0010011	111	-	AND Immediate	1	1	0	0	00	0	11	AND
I-type	ori	0010011	110	-	OR Immediate	1	1	0	0	00	0	11	OR
I-type	xori	0010011	100	-	XOR Immediate	1	1	0	0	00	0	11	XOR
I-type	slli	0010011	001	0000000	Shift Left Logical Imm	1	1	0	0	00	0	11	SLL
I-type	srli	0010011	101	0000000	Shift Right Logical Imm	1	1	0	0	00	0	11	SRL
I-type	srai	0010011	101	0100000	Shift Right Arithmetic Imm	1	1	0	0	00	0	11	SRA
I-type	slti	0010011	010	-	Set Less Than Imm	1	1	0	0	00	0	11	SLT
I-type	sltiu	0010011	011	-	Set Less Than Imm Unsigned	1	1	0	0	00	0	11	SLTU
I-type	lw	0000011	010	-	Load Word	1	1	1	0	01	0	00	ADD
I-type	lh	0000011	001	-	Load Halfword	1	1	1	0	01	0	00	ADD
I-type	lhu	0000011	101	-	Load Halfword Unsigned	1	1	1	0	01	0	00	ADD
I-type	lb	0000011	000	-	Load Byte	1	1	1	0	01	0	00	ADD
I-type	lbu	0000011	100	-	Load Byte Unsigned	1	1	1	0	01	0	00	ADD
S-type	sw	0100011	010	-	Store Word	0	1	0	1	00	0	00	ADD
S-type	sh	0100011	001	-	Store Halfword	0	1	0	1	00	0	00	ADD
S-type	sb	0100011	000	-	Store Byte	0	1	0	1	00	0	00	ADD
U-type	lui	0110111	-	-	Load Upper Imm	1	x	0	0	10	0	xx	xx
U-type	auipc	0010111	-	-	Add Upper Imm to PC	1	x	0	0	11	0	xx	xx
B-type	beq	1100011	000	-	Branch if Equal	0	0	0	0	00	1	01	SUB
B-type	bne	1100011	001	-	Branch if Not Equal	0	0	0	0	00	1	01	SUB
B-type	blt	1100011	100	-	Branch if Less Than	0	0	0	0	00	1	01	SLT
B-type	bge	1100011	101	-	Branch if Greater/Eq	0	0	0	0	00	1	01	SLT
B-type	bltu	1100011	110	-	Branch if LT Unsigned	0	0	0	0	00	1	01	SLTU
B-type	bgeu	1100011	111	-	Branch if GE Unsigned	0	0	0	0	00	1	01	SLTU
UJ-type	jal	1101111	-	-	Jump and Link	1	x	x	x	x	1	x	x

Fig. 2: Control Unit

implementation of the processor, allowing for easier debugging and performance evaluation.

B. Data-Path Design

The **32-bit single-cycle RISC-V processor** follows a simplified **datapath design**, where each instruction is executed in a **single clock cycle**. This design integrates several core components, including the **Program Counter (PC), Instruction Memory, Register File, Arithmetic Logic Unit (ALU), Immediate Generator, and Data Memory**.

Key Components of the Datapath:

1) ***Program Counter (PC):*** The PC holds the address of the next instruction to be fetched from the instruction memory.

2) ***Instruction Memory:*** This memory module stores the instructions to be executed. Register File: The processor includes a register file with **32 registers (x0 to x31)**.

3) ***Arithmetic Logic Unit(ALU):*** The ALU performs arithmetic and logical operations such as addition, subtraction, AND, OR, and comparisons.

4) ***Immediate Value generator:*** For I-type and S-type instructions, the immediate generator extracts the immediate values from the instruction and extends them to 32 bits.

5) ***Data Memory:*** : The processor includes a data memory module for load and store operations.

Single-Cycle Datapath Execution In this design, all operations—**fetching, decoding, execution, memory access, and write-back** are completed in one clock cycle. Although this simplifies control, it means that the critical path must be optimized.

C. Control Unit

This diagram[1] is a **RISC-V Instruction Table** with Control Signals. It's used to guide the implementation of a RISC-V processor, especially the control unit and ALU control logic.

- Instruction type (R, I, S, B, U, UJ)
- Opcode, funct3, funct7 fields (used in instruction decoding)
- The operation name (e.g., add, sub, and)
- Control signals needed to execute that instruction properly:
  - **RegWrite**: Write to register file
  - **ALUSrc**: Choose ALU second operand from register or immediate.
  - **MemRead**: Read from memory.
  - **MemWrite**: Write to memory.
  - **MemtoReg**: Choose data to write to register from ALU or memory.
  - **Branch**: Is it a branch instruction.
  - **ALUOp**: 2-bit code passed to ALU control unit.
  - **ALU Ctrl**: Final ALU operation selected

D. Instruction Set

1) ***R-type Instruction:***

- Opcode: 0110011
- Uses both funct3 and funct7
- RegWrite = 1, ALUSrc = 0 (use both registers), no memory access

2) ***I-type Instruction:***

- Opcode varies depending on the instruction
- ALUSrc = 1 (immediate operand)
- MemRead = 1 for load instructions, MemtoReg = 01 to write from memory

3) ***S-type Instruction:***

- Opcode: 0100011
- MemWrite = 1, RegWrite = 0
- ALUSrc = 1 (address = base + offset), MemRead = 0

4) ***B-type Instruction:***

- Opcode: 1100011.
- Uses register comparisons.
- Branch = 1, RegWrite = 0, ALUOp = 01

5) ***U-type (upper immediate):***

- Opcode: 0110111 for LUI, 0010111 for AUIPC
- LUI: load upper 20 bits; AUIPC: PC + immediate.
- AUIPC sets ALUOp = 00 (adds PC + immediate), LUI uses MemtoReg = 10.

*6) **UJ-type**:*

- Opcode: 1101111
- Writes PC + 4 to rd, jumps to PC + offset
- RegWrite = 1, uses PC logic, control signals: x or custom logic

E. ALUOp and ALU Ctrl

- The ALUOp signal (from control unit) tells the ALU Control Unit what type of instruction this is.
- The ALU Ctrl is the final operation passed to the **ALU:ADD, SUB, AND, OR, XOR, SLT**, etc.Determined using funct3, funct7, and ALUOp.

F. Data Path Schematics

1) Program Counter :

- PC holds the address of the current instruction.
- It sends this address to the Instruction Memory.
- After fetching an instruction, the PC is updated (typically PC + 4 or to a branch/jump target).

2) Instruction Memory:

- Input: PC address. Output: The 32-bit instruction.
- The instruction is split into fields: [31-26] – opcode, [25-21] – rs, [20-16] – rt, [15-11] – rd,[15-0] – immediate value (for I-type),[25-0] – address (for J-type)

3) Control Unit:

- Takes the opcode Instruction[31:26] as input.
- Generates control signals.

4) Register File:

- Reads two registers: rs and rt.
- Outputs Read data 1 and Read data 2.
- These are sent to the ALU and/or Data Memory depending on the instruction.

5) Immediate Generator:

- Extends the 16-bit immediate value to 32 bits (Sign/Zero extension depending on instruction).

6) ALU and ALU Control:

- ALU Control takes ALUOp (from Control Unit) and funct (Instruction[5:0]) to decide the operation.
- ALU performs operations (add, sub, and, or, slt, etc.) between:
  - Read data 1
  - Either Read data 2 or the Immediate (depending on ALUSrc)

7) Data Memory:

- Address from ALU result.
- If MemRead is enabled, reads data from memory.
- If MemWrite is enabled, writes Read data 2 to memory.
- Output sent to Write Data MUX (MemtoReg).

8) Jump Logic:

- Upper 4 bits of PC+4
- 26-bit address from instruction shifted left by 2.

9) PC Update Logic:

- PC + 4 (default)
- Branch target (if Branch is taken)
- Jump target (if Jump)

10) Branch Logic:

- Checks the Zero output from ALU and Branch signal.
- If both are 1, enables branching.
- Target address is calculated by adding sign-extended offset (shifted left 2) to PC+4.

V. Design Strategies

Loop Unrolling in a one-cycle RISC-V processor is achieved by expanding the loop body manually and decreasing the number of iterations. For instance, if a loop is used to process an array(bubble sort array), the loop body is repeated many times to process multiple elements within one iteration. This decreases the number of loop iterations and the branch instructions that go with them. To execute it, the assembly code for loading and storing array elements is replicated for several array elements within a single iteration of the loop. The loop counter is incremented correspondingly, advancing over several elements simultaneously (e.g., incrementing by 4 for unrolling by a factor of 4). The **register file of the processor** should be large enough to store the intermediate values of each unrolled iteration. Once unrolled, the number of iterations is less, and there are fewer branches, enhancing performance by reducing loop control overhead. But the increase in code size and register usage must be controlled carefully, lest too much unrolling would result in diminishing returns.

VI. Instruction Cache Design

1. Parameters

Let the cache design parameters be:

- C: Total cache size in bytes
- B: Block size in bytes (number of bytes per cache block)
- $N = \frac{C}{B}$: Number of cache lines (blocks)
- A: Address size in bits (e.g., 32 bits)
- $M = \frac{2^A}{B}$: Number of blocks in memory

2. Cache Address

Each memory address is divided into three fields in a direct-mapped cache:

$$\text{Memory Address} = \underbrace{\text{Tag}}_{\text{A - Index - Offset}} \quad \underbrace{\text{Index}}_{\log_2(N)} \quad \underbrace{\text{Offset}}_{\log_2(B)}$$

- **Offset**: Identifies a byte within the block
- **Index**: Selects a cache line
- **Tag**: Used to verify if the correct block is stored

3. Cache Access Process

On accessing an instruction at address X:

1) Extract the **Index** to select the cache line.
2) Extract and compare the **Tag** with the stored tag at that line.
3) If they match and the valid bit is set $\Rightarrow$ **Cache Hit**.
4) Else $\Rightarrow$ **Cache Miss**, and the block is fetched from main memory.

4. Calculation

Assume the following:

- $A = 32$ bits (address size)
- $C = 1024$ bytes (cache size)
- $B = 16$ bytes (block size)

Then,

$$N = \frac{1024}{16} = 64 \quad \Rightarrow \quad \log_2(N) = 6 \text{ bits}$$

$$\log_2(B) = \log_2(16) = 4 \text{ bits}$$

$$\text{Tag bits} = 32 - 6 - 4 = 22 \text{ bits}$$

So the 32-bit address is split as:

$$[\text{Tag (22 bits)}] \mid [\text{Index (6 bits)}] \mid [\text{Offset (4 bits)}]$$

5. Hit/Miss Logic

- **Hit**: If Cache[Index].Tag == Address.Tag and Valid bit is set.
- **Miss**: Otherwise, load the memory block into that cache line.

6. Performance Metrics

Average Memory Access Time (AMAT) is given by:

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

7. Design Rationale

Direct-mapped caches are ideal for instruction fetches due to:

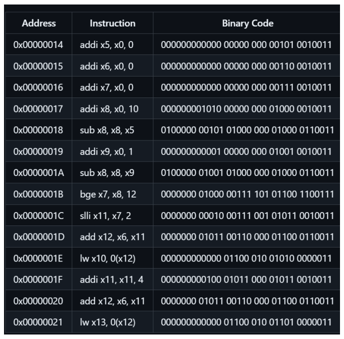
- Simplicity in address mapping
- Fast access due to single comparison
- Low hardware cost

VII. TESTING

- Writing Fibonacci Series Algorithm in C
- Converting C code to RISC-V Assembly code.
- The assembly code was then converted into binary instructions and loaded into the instruction memory.

A. TESTING DIRECT MEMORY CACHE

We are using Bubble Sort Algorithm to test the functionality of direct memory cache. The figure[2] shows the binary isntruction for testing direct memory cache.



Address	Instruction	Binary Code
0x00000014	addi x5, x0, 0	000000000000 00000 000 00101 0010011
0x00000015	addi x6, x0, 0	000000000000 00000 000 00110 0010011
0x00000016	addi x7, x0, 0	000000000000 00000 000 00111 0010011
0x00000017	addi x8, x0, 10	000000001010 00000 000 01000 0010011
0x00000018	sub x8, x8, x5	0100000 00101 01000 000 01000 0110011
0x00000019	addi x9, x0, 1	000000000001 00000 000 01001 0010011
0x0000001A	sub x8, x8, x9	0100000 01001 01000 000 01000 0110011
0x0000001B	bge x7, x8, 12	0000000 01000 00111 101 01100 1100111
0x0000001C	slli x11, x7, 2	0000000 00010 00111 001 01011 0010011
0x0000001D	add x12, x6, x11	0000000 01011 00110 000 01100 0110011
0x0000001E	lw x10, 0(x12)	000000000000 01100 010 01010 0000011
0x0000001F	addi x11, x11, 4	000000000100 01011 000 01011 0010011
0x00000020	add x12, x6, x11	0000000 01011 00110 000 01100 0110011
0x00000021	lw x13, 0(x12)	000000000000 01100 010 01101 0000011

Fig. 3: Binary instructions for Bubble Sort

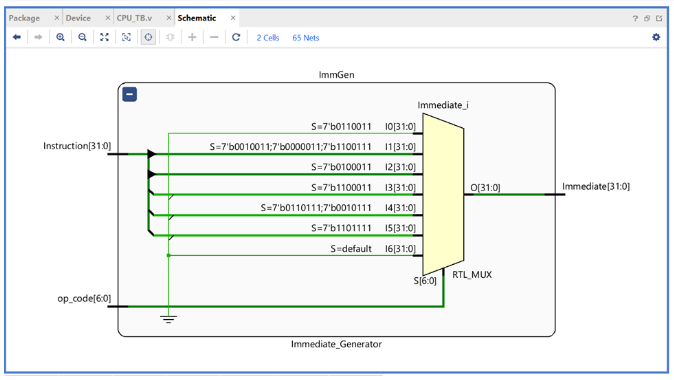


Fig. 4: Schematic for Immediate Value Generator

VIII. IMPLEMENTATION CHALLANGES

- Supporting unaligned access (e.g., lh from an odd address) in a cache with word-aligned blocks was complex.
- **RISC-V instructions** come in different formats (R-type, I-type, S-type, B-type, U-type, J-type). Some caches fetch **32-bit** blocks that might contain partial instructions or misaligned instructions.
- Instructions like **lw** may **become multi-cycle** if the cache misses, breaking assumptions in a single-cycle or simple pipelined processor.
- **Instruction cache** might prefetch ahead of the branch. If the branch is taken, these prefetched instructions are useless.

IX. RESULTS

- Fig 4-13 shows the schematic for RISC-V processor.
- Fig 14-16 shows the timing analysis of Fibnoacci series algorithm.
- Fig 17 shows the timing analysis of bubble sort algorithm to understand use of direct memory cache
- Fig 18 shows the timing analysis for the RISC-V procrssor. The **maximum frequncy** attained is 37.037MHz.
- Fig 20 shows the resource utilization.
- Fig 19 shows the power analysis.
- Fig 21 shows the number of Luts Used.
- **Energy per cycle**

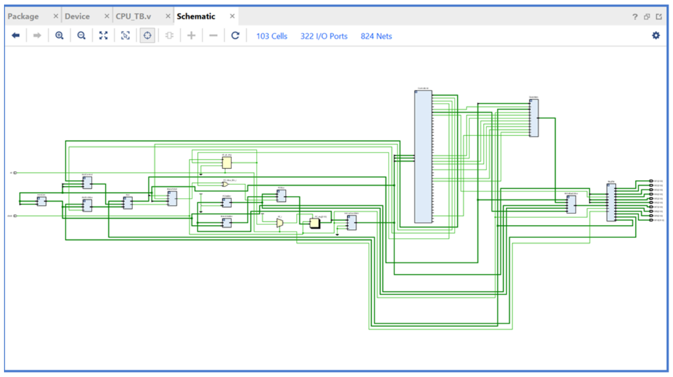


Fig. 5: Overall Schematic

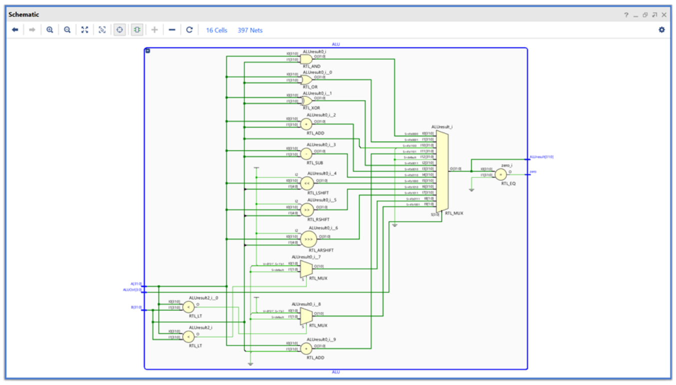


Fig. 8: Schematic for ALUSrc

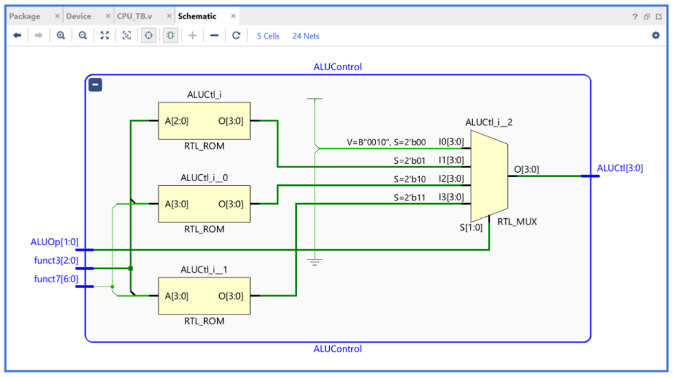


Fig. 6: Schematic for ALUControl

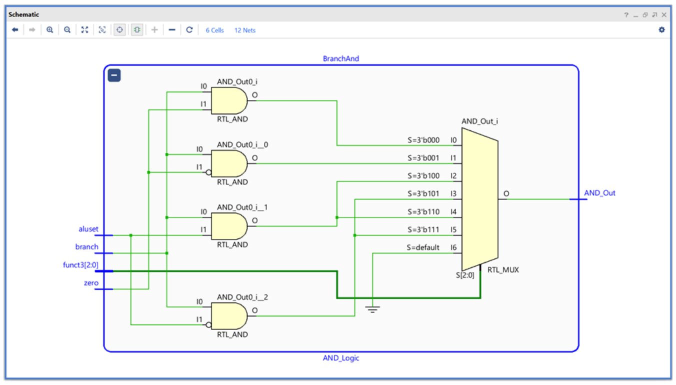


Fig. 9: Schematic for PC controlling MUX

$$E_{\text{cycle}} = P \cdot T$$

Where:

- $P = 0.07\,\text{W}$ (measured or estimated power consumption),
- $T = \alpha = 100\,\text{ns} = 100 \times 10^{-9}\,\text{s}$ (clock period).

Substituting:

$$E_{\text{cycle}} = 0.07 \cdot (26.32 \times 10^{-9}) = 1.8424 \times 10^{-9}\,\text{J} = 1.8424\,\text{nJ}$$

X. CONCLSUION

In this project, a 32-bit single-cycle RISC-V processor was successfully designed and implemented on the Digilent BASYS3 FPGA board. The processor supports a subset of RISC-V instructions, including arithmetic, logical, load/store, and branching operations. The processor was tested using a Fibonacci Series algorithm, which verified its functional correctness. The sorted array was stored in registers, and the processor demonstrated stable performance during execution.The design process involved creating a complete datapath and

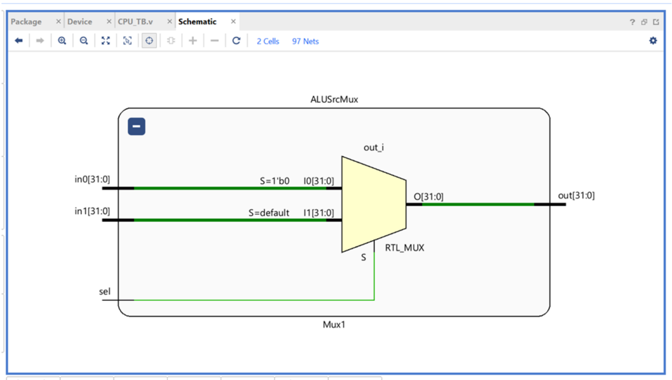


Fig. 7: Schematic for Branch

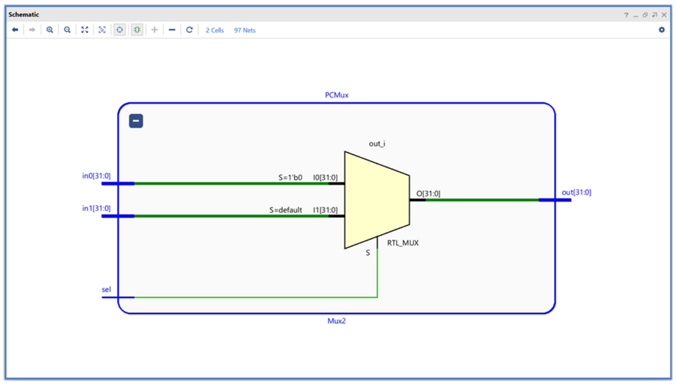


Fig. 10: Schematic for ALU

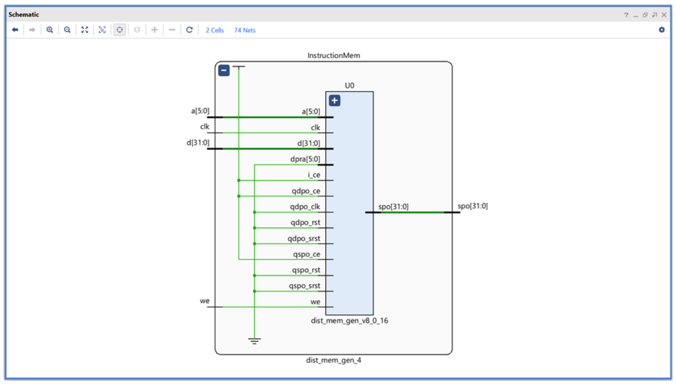


Fig. 11: Schematic for ImmedateMemory

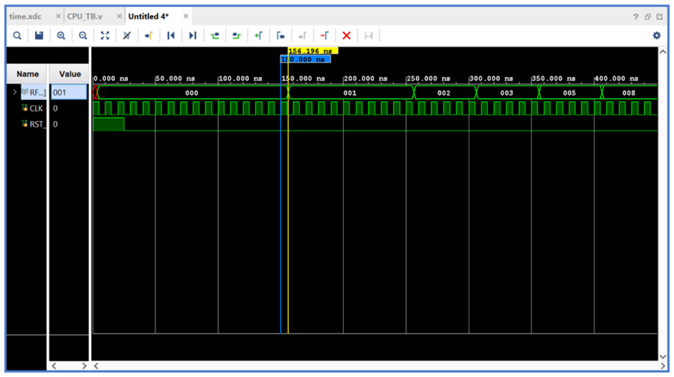


Fig. 14: Behaviour Simulation

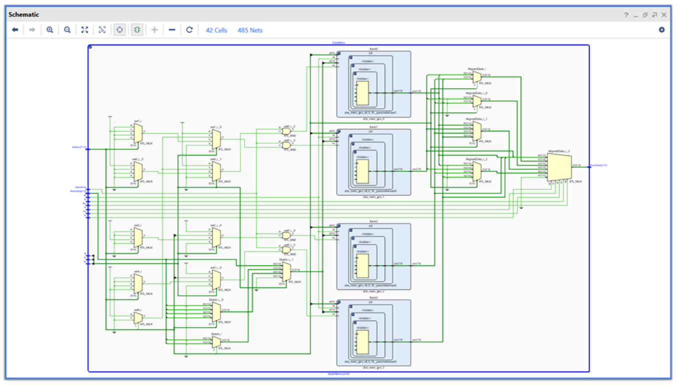


Fig. 12: Schematic for RISC-V processor

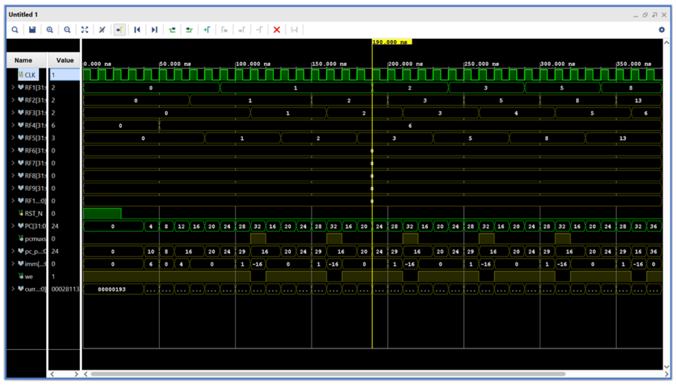


Fig. 15: Behaviour Simulation

control unit, ensuring proper communication between various components such as the ALU, register file, and memory units. The FPGA implementation showed that the processor meets timing requirements with no violations at a 100 ns clock period, and resource utilization was kept within the available limits.Despite the processor's simplicity, it serves as a robust foundation for understanding single-cycle architectures. Future work could expand the instruction set to support more complex operations, such as multiplication and division, and optimize the design for better clock frequency performance.This project provides valuable insights into processor design and highlights

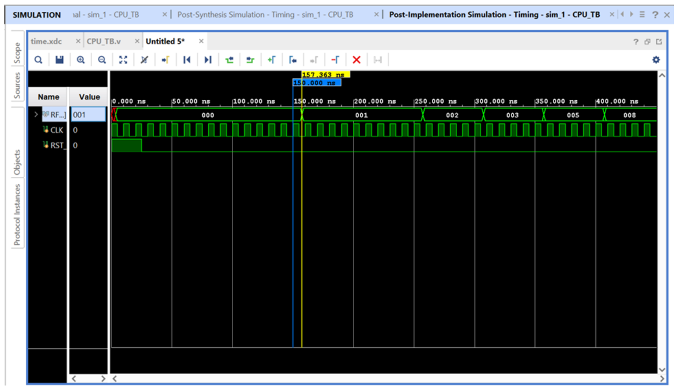


Fig. 16: Post Synthesis Simulation

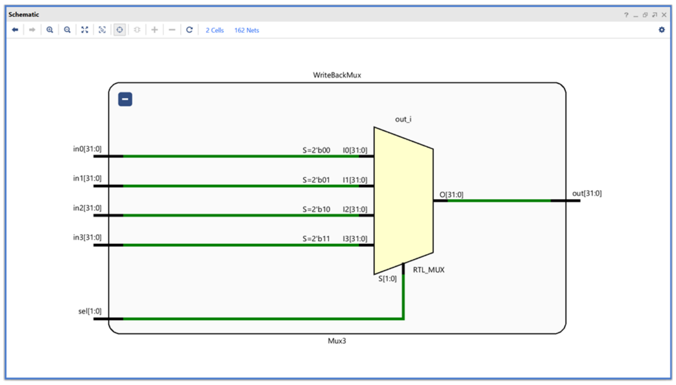


Fig. 13: Schematic for WriteBack multiplexer

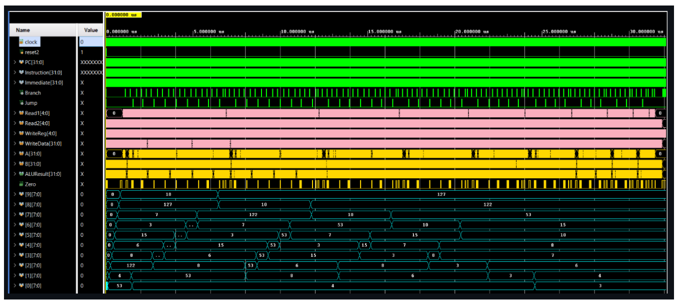
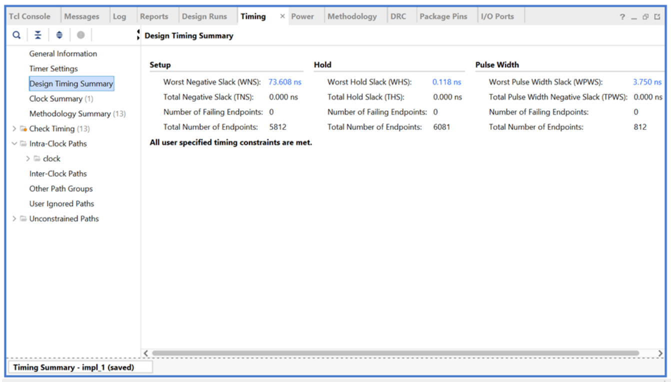


Fig. 17: Bubble Sort Algorithm



Design Timing Summary

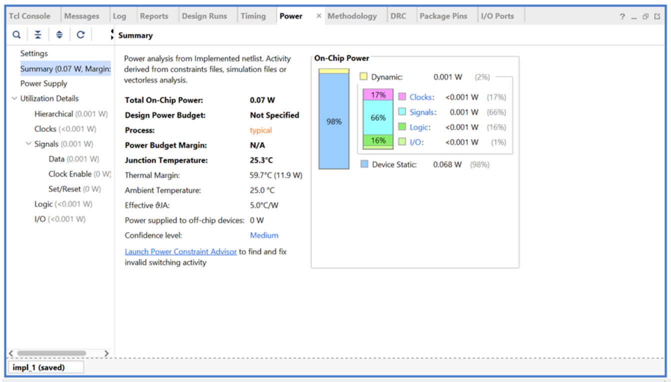
General Information
Timer Settings
Design Timing Summary
Clock Summary (1)
Methodology Summary (13)
Check Timing (13)
Intra-Clock Paths
clock
Inter-Clock Paths
Other Path Groups
User Ignored Paths
Unconstrained Paths

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	73.608 ns	Worst Hold Slack (WHS):	0.118 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5812	Total Number of Endpoints:	6081	Total Number of Endpoints:	812

All user specified timing constraints are met.

Timing Summary - impl_1 (saved)

Fig. 18: Timing Analysis Fmax 37.93MHz



Summary

Settings
Summary (0.07 W, Margin:
Power Supply
Utilization Details
Hierarchical (0.001 W)
Clocks (<0.001 W)
Signals (0.001 W)
Data (0.001 W)
Clock Enable (0 W)
Set/Reset (0 W)
Logic (<0.001 W)
I/O (<0.001 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.07 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.3°C**
Thermal Margin:	59.7°C (11.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.001 W	(2%)
Clocks:	<0.001 W	(17%)
Signals:	0.001 W	(66%)
Logic:	<0.001 W	(16%)
I/O:	<0.001 W	(1%)
Device Static:	0.068 W	(98%)

impl_1 (saved)

Fig. 19: Power Analysis

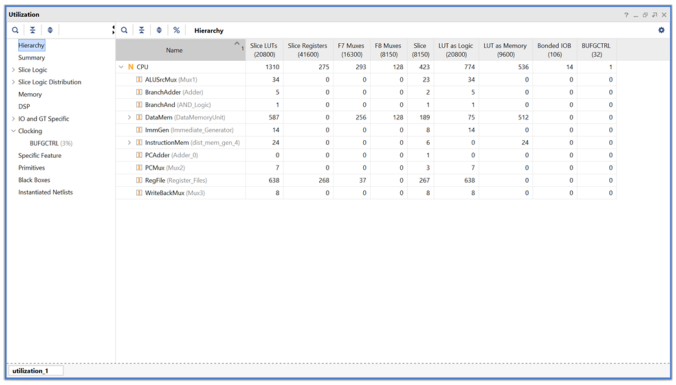
the importance of optimizing control signals and datapath elements for reliable execution on an FPGA platform.

References

[1] Patterson, D., Waterman, A. (2017). The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon LLC.

[2] Harris, D., Harris, S. (2015). Digital Design and Computer Architecture (2nd ed.). Morgan Kaufmann.

[3] Waterman, A., Lee, Y., Patterson, D., Asanović, K. (2014). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. EECS Department, University of California, Berkeley.

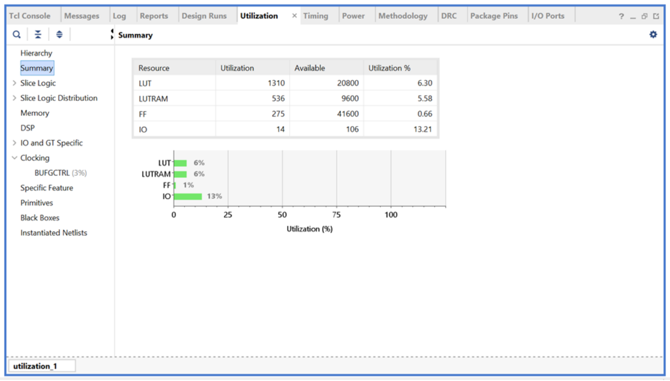


Utilization — Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Bonded IOB (106)	BUFGCTRL (32)
CPU	1310	275	293	128	423	774	536	14	1
ALUSrcMux (Mux1)	34	0	0	0	23	34	0	0	0
BranchAdder (Adder)	5	0	0	0	2	5	0	0	0
BranchAnd (AND_Logic)	1	0	0	0	1	1	0	0	0
DataMem (DataMemoryUnit)	587	0	256	128	189	75	512	0	0
ImmGen (Immediate_Generator)	14	0	0	0	8	14	0	0	0
InstructionMem (dist_mem_gen_4)	24	0	0	0	6	0	24	0	0
PCAdder (Adder_0)	0	0	0	0	1	0	0	0	0
PCMux (Mux2)	7	0	0	0	3	7	0	0	0
RegFile (Register_Files)	638	268	37	0	267	638	0	0	0
WriteBackMux (Mux3)	8	0	0	0	8	8	0	0	0

utilization_1

Fig. 20: Resource Utilization



Summary

Resource	Utilization	Available	Utilization %
LUT	1310	20800	6.30
LUTRAM	536	9600	5.58
FF	275	41600	0.66
IO	14	106	13.21

utilization_1

Fig. 21: LUT Used

[4] Hennessy, J. L., Patterson, D. A. (2017). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.

[5] Lopes, J. P., Silva, F., Monteiro, J. (2020). "A Lightweight Single-Cycle RISC-V Processor for FPGA-Based Educational Environments." IEEE Transactions on Education, 63(2), 103–110.