Journal of Theoretical and Applied Information Technology
15th July 2025. Vol.103. No.13

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REFINEMENT OF A SOFT-CORE PROCESSOR IMPLEMENTATION ON FPGA

GYOO SOO CHAE[1]

[1]Professor, Baekseok University, Division of Advanced IT, Cheonan-City, South Korea

E-mail: [1]gschae@bu.ac.kr

ABSTRACT

CPUs with soft cores represent a category of microprocessors whose architecture and functionality can be delineated entirely utilizing languages used to describe hardware, such as Verilog or VHDL. These processors offer a high degree of customization tailored to specific applications and are deployable on platforms for reconfigurable hardware, including Field Programmable Gate Arrays (FPGAs). After the design is successfully implemented, the potential exists for developing Application Specific Integrated Circuits (ASICs) for large-scale production. This study entails designing, simulating, and validating an 8-bit processor utilizing VHDL. The envisaged processors are intended for application in control systems characterized by modest to moderate complexity. Moreover, the 8-bit soft processor design lays the groundwork for potential ASIC development. The simulation outcomes, facilitated by a commercial software (ModelSim), are comprehensively illustrated via timing diagrams. Additionally, the practical outcomes of this design's execution on the VIRTEX-5 are meticulously presented through laboratory experimentation. The primary contribution of this study is the development of a compact, application-focused soft-core processor architecture that emphasizes predictable execution, minimal response time, and efficient use of hardware resources—features that are crucial for embedded control system applications. The simulation and practical implementation results offer insights into the performance and feasibility of deploying such processors in real-world scenarios, paving the way for potential ASIC development and broader industrial application.

Keywords: *Soft-core processor, FPGA, VHDL, Verilog, Hardware description language, ASIC, customization, Reconfigurable hardware, Control applications, Simulation software(ModelSim), implementation, Xilinx Spartan-3, Laboratory experimentation*

1. INTRODUCTION

The development of soft-core processors can be traced back to the early 1990s when researchers began exploring synthesizable processor cores for FPGA platforms. Please complete the modifications and submit the updated document as soon as possible forms. Initially conceived as educational tools, soft-core processors have evolved significantly to support complex introduction sets, multithreading, and hardware accelerators. The need for specialized processing units tailored to specific requirements has become increasingly apparent in control applications. Eight-bit processors remain highly relevant in embedded systems and industrial control applications due to their simplicity, low power consumption, and cost-effectiveness. While general-purpose processors can handle various tasks, they may offer a lower level of performance or customization needed for control-oriented applications. As a result, there has been a growing interest in developing dedicated CPUs optimized for control tasks, leveraging the flexibility and configurability offered by soft-core processors on FPGA platforms. The utilization of soft-core Central Processing Units (CPUs) as Intellectual Property (IP) cores compatible with Field Programmable Gate Array (FPGA) devices [1-8] has garnered significant attention. These processors offer various advantages, including accelerated time to market, seamless integration with FPGA fabric, reduced obsolescence, and customizable performance and pricing [9]. Modern FPGA architectures integrate hard-core and soft-core processing units onto a single chip, providing extensive configurability for application developers [10]. Researchers have explored applying a dataflow-based design technique to develop soft-core processors for FPGA platforms, mainly focusing on picture manipulation applications [10-

15]. Tong et al. compared offerings from open-source and commercial manufacturers in CPU processor design for embedded systems applications [11]. Intellectual Property (IP) cores, also known as soft-core processors, find utility in Field Programmable Gate Array (FPGA) systems [8]. These processors offer benefits such as accelerated time to market, seamless integration with FPGA fabric, reduced obsolescence, and customizable performance and pricing. Modern FPGA designs consolidate hard-core and soft-core processors onto a single chip, providing extensive configurability for application developers [9]. Amiri et al. utilized a dataflow-based design technique to develop CPUs on FPGA devices for image processing applications [10]. Tong et al. conducted a survey comparing soft-core processor offerings from open-source and commercial suppliers for embedded applications [11]. Moreover, a detailed implementation experience with the Xilinx Integrated Software Environment for a soft-core processor is documented in [16]. The research delves into designing an eight-bit CPU tailored for control applications characterized by simple to moderate complexity. The processor's VHDL code is developed entirely using the Xilinx ISE tool and subjected to simulation using the software(ModelSim). The resulting simulation outcomes are meticulously presented and analyzed. Additionally, a summary of digital logic requirements and resource utilization for the target FPGA IC is provided, advocating the efficient implementation of the proposed soft-core processor for control applications. The design of an eight-bit CPU tailored for control applications involves several key considerations, including system architecture, instruction set design, memory hierarchy, and interface compatibility. This section delves into the design methodologies employed in developing the proposed soft-core processor, highlighting the design choices, optimization strategies, and trade-offs made throughout the design process[17-19]. Although extensive research has been conducted on soft-core processor design for image processing [10–15] and general-purpose embedded systems [11], there is a notable lack of focus in existing literature on developing soft-core CPUs specifically optimized for control applications with limited complexity and stringent efficiency demands. Much of the current work prioritizes performance improvements for data-heavy applications, often overlooking the unique requirements of control systems, which rely heavily on simplified architectures, predictable execution, and minimal hardware resource usage. One of the fundamental design considerations is selecting the instruction set architecture (ISA) tailored to the requirements of control applications. The ISA defines the set of instructions the CPU supports and determines its capabilities and performance characteristics. In the context of control applications, the ISA should prioritize efficiency, simplicity, and determinism, enabling the CPU to execute control algorithms with minimal overhead. Another critical aspect of CPU design is the organization of the register array, which serves as the primary storage for intermediate data and operands during computation. The register file architecture significantly impacts the CPU's performance and efficiency, including the number of registers, their width, and the addressing modes supported. In control applications, where real-time responsiveness is crucial, optimizing the register file design to minimize access latency and maximize throughput is essential.

Furthermore, the design of the control unit, which is responsible for sequencing the execution of instructions and coordinating data movement within the CPU, plays a vital role in determining the overall performance and functionality of the processor. The control unit design involves defining the instruction fetch, decode, execute, and write-back stages and implementing control logic for handling interrupts, exceptions, and other system-level events. In control applications, where precise timing and synchronization are critical, designing a robust and efficient control unit is paramount to ensuring the reliable operation of the CPU. Additionally, the design of specialized functional units, such as comparators and shifters, tailored to the requirements of control algorithms further enhances the CPU's performance and efficiency. These units are optimized for specific operations commonly used in control applications, such as comparison, shifting, and bit manipulation, enabling the CPU to execute control algorithms with higher throughput and lower latency.
Moreover, the design and implementation of the Arithmetic Logic Unit (ALU), which is responsible for performing arithmetic and logical operations, are crucial in determining the CPU's computational capabilities. In control applications, where mathematical operations are prevalent, optimizing the ALU design to support a wide range of operations with minimal latency and area overhead is essential.

1.1 FPGA Implementation

Once the design of the soft-core CPU is finalized, the next step is to implement it on an

FPGA platform and evaluate its performance and functionality. This section discusses the FPGA implementation process, including hardware synthesis, place-and-route, timing analysis, and simulation and verification techniques used to validate the design's correctness and robustness. Implementing the soft-core CPU begins with synthesizing the hardware description written in VHDL or Verilog into a netlist of logical gates and interconnections. The synthesis process optimizes the design for area, speed, and power consumption, translating the CPU's high-level description into a low-level representation suitable for FPGA implementation. Following synthesis, the netlist is mapped onto the FPGA fabric using place-and-route tools, determining the physical placement of logic elements and routing resources on the FPGA chip. The place-and-route process aims to minimize signal delays and routing congestion while meeting timing constraints specified by the design. Once the design is mapped onto the FPGA fabric, timing analysis ensures that the critical paths meet the required timing constraints and that the design operates reliably at the target clock frequency. Timing closure, achieved through iterative optimization and refinement, is essential to guarantee the performance and reliability of the CPU in real-world applications. In parallel with hardware implementation, functional verification is conducted using simulation tools to verify the correctness and functionality of the CPU design. Simulation tests, including unit tests, integration tests, and system-level tests, validate the behavior of the CPU under different operating conditions and input stimuli, ensuring that it operates as expected according to the design specifications. Furthermore, FPGA-based prototyping allows for rapid iteration and testing of the CPU design in a real-world environment, enabling designers to validate performance, functionality, and compatibility with target applications. By leveraging FPGA platforms, designers can quickly identify and address design issues, refine implementation strategies, and optimize the performance of the CPU for specific application requirements.

1.2 Performance Evaluation

Once the soft-core CPU is implemented on the FPGA platform, performance evaluation is conducted to assess its suitability for control applications. This section discusses the performance metrics used to evaluate the CPU, including throughput, latency, power consumption, resource utilization, and benchmarking techniques used to compare the CPU against alternative solutions. Throughput measures the rate at which the CPU executes instructions or processes data, indicating its processing capacity and efficiency in handling control tasks. Latency quantifies the time taken for the CPU to complete a single operation or task, reflecting its responsiveness and real-time performance in control applications. Power consumption is critical in embedded systems, where energy efficiency is paramount. By measuring the power consumption of the CPU under different operating conditions and workloads, designers can optimize power usage and extend the battery life of battery-powered devices. Resource utilization assesses the hardware resources consumed by the CPU, including logic elements, memory blocks, and routing resources on the FPGA chip. Optimizing resource utilization is essential to maximizing the scalability and cost-effectiveness of the CPU design, particularly in resource-constrained embedded systems. Benchmarking involves running standardized test programs or algorithms on the CPU. The subsequent sections of the paper are organized as follows: Methods/Experimental outline the system considerations employed in the design process. A succinct overview of the register array and its simulation is provided. It also thoroughly examines the Arithmetic Logic Unit (ALU) functionality alongside its implementation. Design methodologies for the Comparator and Shifter, accompanied by timing simulations, are elaborated. The control unit design process is elucidated to validate its functionality. The results section discusses the FPGA implementation outcomes in detail, while the conclusions section presents the final remarks. This paper contributes to soft-core processor design by introducing a meticulously crafted eight-bit processor tailored for control applications. Leveraging VHDL and FPGA platforms, along with comprehensive simulation and implementation results, underscores the efficacy and viability of the proposed design.

2. METHODS AND EXPERIMENTAL

2.1 System Architecture

Sections The system architecture is portrayed in Figure 1, illustrating the hierarchical arrangement of the microprocessor being developed. At the topmost level, the system encompasses two core constituents: external memory and the Central Processing Unit (CPU). External memory assumes a critical role as it acts as a pivotal repository for storing both program instructions and data. The chosen external memory chip must demonstrate

operational speeds compatible with the envisaged FPGA-based CPU. Numerous commercially available memory chips meet this criterion satisfactorily. Within the CPU, several internal components play pivotal roles in facilitating computation and control. These components encompass the register array, Arithmetic Logic Unit (ALU), comparator, shifter, and control unit. The register array is a collection of storage elements capable of temporarily holding data during processing operations. The ALU executes arithmetic and logical operations on the data stored in registers. The comparator enables comparison operations, which is essential for decision-making processes within the processor. Conversely, the shifter facilitates the manipulation of data bits, a necessity for various computational tasks. Lastly, the control unit governs the overall operation of the CPU and orchestrates the execution of instructions. Further granularity within the control unit is achieved through its subdivision into three distinct components: the Instruction Register (IR), controller, and Program Counter (PC). The Instruction Register (IR) is a temporary storage unit that executes the current instruction. The controller interprets the instruction fetched from memory and issues appropriate control signals to coordinate the execution flow. Meanwhile, the Program Counter (PC) keeps tabs on the memory address of the subsequent instruction to be fetched, thereby ensuring the sequential execution of program instructions.

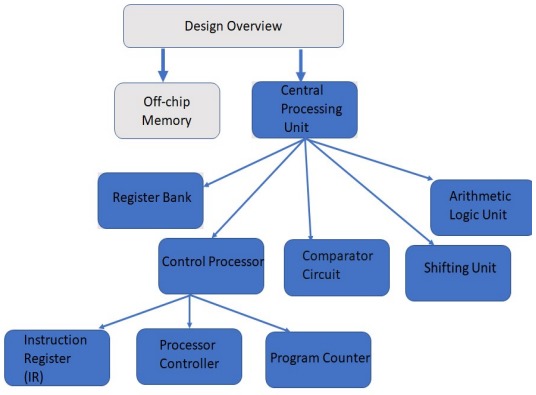

Figure 1: Hierarchical design of the microprocessor

This hierarchical arrangement delineates the essential components and their interconnections within the microprocessor design, laying the foundation for subsequent detailed discussions and implementations.

2.2 Register Array

The register array is a critical CPU component that provides temporary data storage during processing operations. It comprises a collection of registers, each capable of holding a fixed number of bits. These registers serve various purposes, including holding operands for arithmetic and logical operations, storing intermediate results, and facilitating data movement within the processor. The register array facilitates data manipulation and computation in the designed microprocessor. By providing a set of storage elements directly accessible by the CPU, the register array enables efficient instruction execution. Moreover, the availability of multiple registers allows for parallel data processing, thereby enhancing the microprocessor's overall throughput. The register array balances capacity and speed to ensure optimal performance and resource utilization. Careful consideration is given to the number of registers, their width, and the access latency to meet the requirements of the target application. Additionally, techniques such as register renaming and pipelining may be employed to enhance the efficiency of the register array further. The design incorporates an array of eight 8-bit registers denoted as R0 through R7. These registers serve as temporary storage units during program execution. The VHDL model representing the register array is depicted in Figure 2, illustrating a block diagram representation.

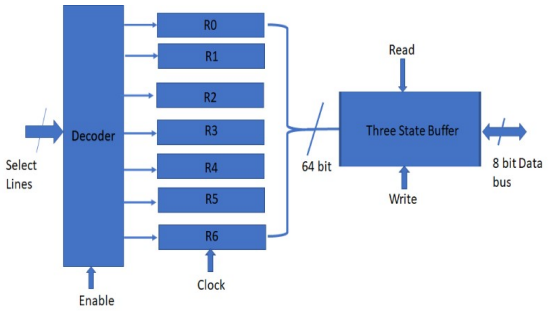

Figure 2: VHDL representation of a Register Array

A 3:8 decoder selects one of the eight registers for read or write operations [10-12]. The logic values at the SEL input determine which register is being operated upon. If the RD signal is asserted (RD = 1), the content of the selected register is loaded onto the data bus. Similarly, if the WR signal is asserted (WR = 1), the content present on the data bus is written into the selected register.
Simulation results of the register array, depicted as a timing diagram, are obtained using the software, (ModelSim XE-III) as shown in Figure 3. During the first clock period, register R0 is selected for

write and read operations (SL = 000, WR = 1, and RD = 1). As illustrated in Figure 3, the input data (data in = 11111111) is written into R0, and subsequently, the same data is read from R0. R1 is exclusively selected for a read operation in the subsequent clock period. Since no data is written into R1, an undefined value is read. Only a write operation is permitted for R0 in the third clock period. However, since RD = 0, the output enters a high impedance state (indicated by dataout = tristate output). The data stored in R0 during the third clock period (equal to 11110000) is read during the ninth, highlighting that modifying the stored data within a specific register without the asserted WR control signal is impossible.

Figure 3: Simulation Outcome of the Register Array

2.3 ALU Functionality and Implementation

The Arithmetic Logic Unit (ALU) serves as the computational heart of the microprocessor and is responsible for performing arithmetic, logical, and comparison operations on data operands. It accepts input data from the register array, processes them according to the operation specified in the instruction, and produces the resulting output. The ALU supports various arithmetic and logical operations, including addition, subtraction, multiplication, division, bitwise AND, OR, XOR, and shift operations. Depending on the application's specific requirements, the ALU may incorporate specialized hardware to accelerate certain operations or support custom instructions.

In implementing the ALU, careful attention is paid to optimizing performance, area utilization, and power efficiency. Various design techniques, such as parallelism, pipelining, and hardware optimization, achieve the desired balance between these factors. Moreover, the ALU is thoroughly verified through simulation and functional testing to ensure correct operation under all scenarios. The ALU (Arithmetic Logic Unit) comprises eight distinct operations, with the first four categorized as arithmetic and the subsequent four as logical. Given the selection of eight operations, three bits are assigned to signify them. Table 1 provides a comprehensive overview of the ALU operations alongside their corresponding bit assignments:

Table 1: Bit Assignments for ALU Operations.

Sl. No	Arithmetic Logic Unit	Operations | Task Allocations (SEL)
1	Add	0 0 0
2	Sub	0 0 1
3	Inc	0 1 0
4	Dec	0 1 1
5	AND	1 0 0
6	OR	1 0 1
7	XOR	1 1 0
8	NOT	1 1 1

The first column enumerates the serial number of each operation within the ALU, while the second column denotes the specific operation. The third column illustrates the corresponding bit assignments, denoted as SEL(2), SEL(1), and SEL(0). For instance, for addition, the bit assignments are 0 0 0, whereas for subtraction, they are 0 0 1.The Arithmetic Logic Unit comprises two 8-bit registers, A and B, at its input. Both registers are utilized for operations necessitating two operands, whereas only register A is employed for single-operand operations. The outcome of an operation is conveyed to another 8-bit register, C. Additionally, the ALU incorporates three flags—Carry (CY), Zero (ZF), and Sign (SF)—to indicate the data state resulting from an arithmetic logic operation. When the ALU is not selected for operation, it remains in a high-impedance state.Simulation results of the ALU, presented in the form of a timing diagram (Figure 4), comprehensively depict all eight operations with typical data input alongside their corresponding output. Table 2 elucidates the simulation results in further detail.

Table 2: Simulation Results in Further Detail.

Sn	Arithmetic Logic Unit	EN	SEL	In		Output			
				A	B	C	CY	ZF	SF
1	Add	1	0 0 0	F0	11	01	1	0	0
2	Sub	1	0 0 1	40	28	18	0	0	0
3	Inc	1	0 1 0	10	67	11	0	0	0
4	Dec	1	0 1 1	F0	40	FF	0	0	1
5	ANDing	1	1 0 0	10	0F	00	0	1	0
6	ORing	1	1 0 1	20	0F	FF	0	1	1
7	X-OR	1	1 1 0	F3	28	FB	1	0	1
8	Inverting	1	1 1 1	FF	03	00	0	1	0
9	-	0	z	zz	zz	zz	z	z	z

Journal of Theoretical and Applied Information Technology
15th July 2025. Vol.103. No.13

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

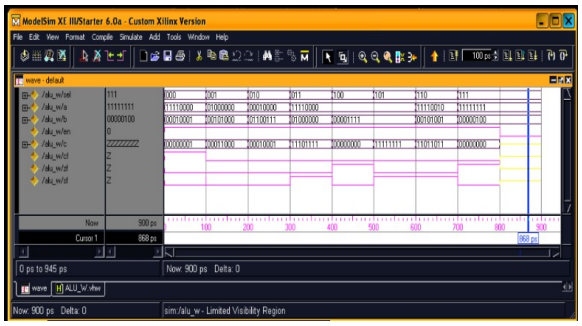

Figure 4: Arithmetic Logic Unit (ALU) Simulation Outcome.

2.4 Comparator and Shifter Design

The comparator and shifter are indispensable components within the microprocessor, playing pivotal roles in comparing and manipulating data bits. These functionalities facilitate conditional branching, decision-making, and various computational tasks within the processor architecture. In the ensuing discourse, we embark on a comprehensive exploration of the design and implementation of these crucial components, with a keen focus on efficiency, speed, accuracy, and robustness.

2.4.1 Comparator Design and Functionality

The comparator circuit forms the cornerstone of the microprocessor's decision-making capabilities, enabling the comparison of two input values and the generation of output signals indicative of the comparison result. Based on the comparison outcome, the comparator evaluates the relative magnitudes of the provided numbers and produces output signals, such as Carry Flag (CF) and Zero Flag (ZF). These flags are pivotal in conditional branching and program execution flow control within the processor.

2.4.2 Comparator Operation

The operation of the comparator revolves around analyzing two input values, denoted as A and B, and determining their relationship. Table 3 elucidates the correspondence between the input values and the resultant states of the CF and ZF flags. It is noteworthy that the compare operation does not alter the contents of the source registers, ensuring data integrity during comparison operations. Moreover, the outputs CF and ZF maintain a high-impedance state when the comparator is not actively selected, mitigating unintended effects on the system.

2.4.3 Simulation Results and Analysis

Figure 5 provides insights into the simulation results of the comparator circuit, offering a glimpse into its operational characteristics under varying input scenarios. During the initial three operations, the enable input (EN) remains asserted (high), while it assumes a de-asserted state (low) during the fourth operation. Consequently, the outputs CF and ZF remain in a high-impedance state, underscoring the passive behavior of the comparator when not actively engaged. Notably, specific input combinations lead to distinct flag configurations, reflecting the outcome of the comparison operation. For instance, when A equals 10H, and B equals 00H, CF and ZF assume a logical low state (0), indicating equality between the input values. Conversely, in scenarios where A and B are both FFH, CF assumes a low state (0). At the same time, ZF is asserted (high), signifying A being more significant than B. Lastly, when A equals 10H and B equals 3FH, CF is asserted (high), and ZF assumes a low state (0), denoting A being less than B.

2.4.4 Shifter Design and Functionality

Complementing the comparator, the shifter plays a pivotal role in facilitating the manipulation and reorganization of data bits within the microprocessor. Through the shifter's capabilities, the microprocessor can execute various operations, including arithmetic shifts, logical shifts, and rotations, essential for multiplication, division, and data formatting.

2.4.5 Shifter Operation

The shifter operates by shifting the bits of a given data word left or right within a register or between registers, as dictated by the specific instruction being executed. This operation enables various data manipulation tasks, including shifting bits into or out of position, rotating bits cyclically, and filling vacated bit positions with designated values. The shifter's versatility and flexibility empower the microprocessor to execute complex computational tasks precisely and efficiently.

2.4.6 Hardware Optimizations

Various hardware optimizations, including parallel comparison and bit-level shifting techniques, underpin efforts to enhance the shifter's efficiency and performance. By exploiting parallelism in the shifter's design, multiple bits can be shifted simultaneously, minimizing latency and maximizing throughput. Additionally, bit-level shifting optimizations streamline the data

manipulation process, ensuring rapid and efficient execution of shift operations.

2.4.7 Comprehensive Testing and Validation

The rigorous testing and validation regimen to ensure correct operation across all possible input scenarios are central to the comparator and shifter's design and implementation. Testbenches, comprising diverse test cases and stimuli sets, are meticulously crafted to validate the functionality of individual components and the overall system. Comprehensive testing encompasses functional verification, performance analysis, and corner-case testing to uncover potential anomalies or discrepancies.

2.4.8 Functional Verification

Functional verification encompasses tests for the comparator and shifter's correct operation under various input conditions. Test cases are devised to evaluate edge cases, boundary conditions, and exceptional scenarios, ensuring that the components behave as expected across the entire spectrum of possible inputs. Through exhaustive functional verification, designers gain confidence in the reliability and accuracy of the comparator and shifter implementations.

2.4.9 Performance Analysis

Performance analysis delves into assessing the efficiency, speed, and resource utilization of the comparator and shifter designs under different operating conditions. Key performance metrics, including throughput, latency, and resource utilization, are quantified and analyzed to identify potential bottlenecks and areas for optimization. By scrutinizing performance metrics, designers can iteratively refine the designs to achieve optimal efficiency and throughput while minimizing resource overhead. Essentially, the comparator and shifter are linchpins of the microprocessor's functionality, embodying efficient data comparison and manipulation principles. These components emerge as stalwarts of computational efficiency through meticulous design, optimization, and validation efforts, empowering the microprocessor to execute various tasks with precision and reliability.

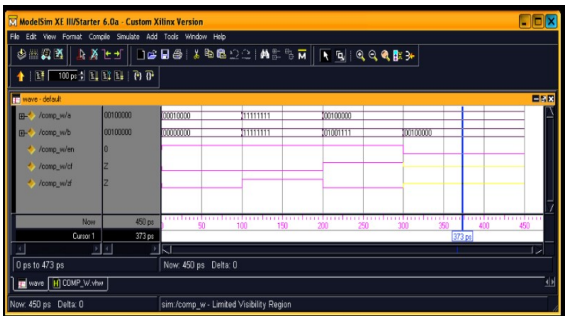

Figure 5a): simulation outcomes of the Comparator

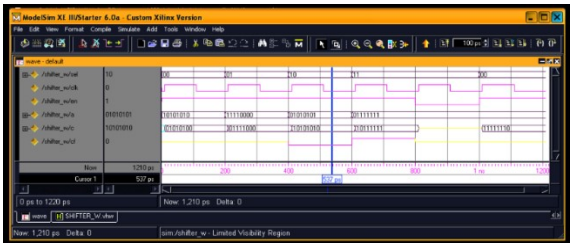

Figure 5b): simulation outcomes of the Shifter

Figure 5 a) and b) illustrate the simulation outcomes of the Comparator and the Shifter, respectively.

Table 3: Comparison Results of the Comparator Circuit.

A	B	CF	ZF
10H	00H	0	0
FFH	FFH	0	1
20H	2FH	1	0

The shifter circuit within the CPU is tasked with altering the contents of register A by shifting or rotating them by one-bit position to the left or right. Subsequently, the resulting output is directed to register C, an 8-bit register. The shifter is designed to accommodate four distinct operations, as outlined in Table 4. Additionally, Table 4 delineates the corresponding actions undertaken by the shifter for each operation.

Table 4: Bit assignments for shifter operations.

Sl. No	Shift Operation	SL	Shifter Action
1	left	0 0	A(0) → 0; A(7:1) → A'(6:0)
2	right	0 1	0 → A(7); A'(7:1)→A(6:0)
3	Left_R	1 0	CF ← A'(7); A(0) ← A'(7); A(7:1) ← A'(6:0)
4	Right_R	1 1	A'(0) → A(7) ; A'(0) → CF; A'(7:1)→ A(6:0)

Note: A' and A stand for register A's contents before and after the operation, respectively.

Table 5 interprets the shifter's simulation results, shown in Figure 6. The four procedures are tested using standard data inputs, and the appropriate outputs are shown. It is crucial to remember that when the shifter is not selected (enable= 0), it enters a high-impedance state.

Table 5: Interpretation of Shifter Simulation Results.

Sl. No	Shift Operation	SL	In A	Op C	CF
1	left	00	10101011	01010101	-
2	right	01	11110001	11111000	-
3	left-rotate	10	11010101	10101011	0
4	right rotate	11	11111110	01111111	1

These results offer insights into the functionality and behavior of the shifter circuit, showcasing its ability to perform shifting and rotating operations on the input data stored in register A while producing the corresponding output in register C along with the state of the Carry Flag (CF).

2.5 Control Unit Design and Functionality

The control unit represents the nucleus of the microprocessor, orchestrating the execution of instructions and regulating the data flow within the system. Its primary function entails deciphering the instructions retrieved from memory, generating control signals to synchronize the operation of other components, and ensuring the accurate sequencing of instruction execution.

Masticating the control unit's design requires meticulous attention to instruction decoding, adherence to timing constraints, and generating requisite control signals. Finite State Machine (FSM) design methodologies emerge as pivotal tools for modeling the intricate control logic, affording precise command over the microprocessor's behavior. Furthermore, the design ethos emphasizes the creation of a control unit characterized by flexibility and extensibility, thereby facilitating the seamless integration of new instructions or modifications to existing ones. The control unit's validation process entails rigorous simulation and verification exercises to ascertain its functionality and correctness comprehensively. Testbenches are meticulously crafted to simulate diverse instruction sequences and edge cases, ensuring the control unit operates reliably under all conceivable conditions. Additionally, exhaustive timing analysis is conducted to verify that the microprocessor aligns with the stipulated performance specifications, assuring optimal operational efficiency. Let us delve deeper into each aspect of the control unit design and validation:

2.5.1 Instruction Decoding:

Instruction decoding constitutes a pivotal aspect of the control unit's functionality. It involves parsing the fetched instructions and discerning their intended operations. Various decoding techniques are employed, ranging from simple lookup tables to more sophisticated FSM-based approaches. The goal is to translate each instruction into a series of control signals that govern the behavior of the microprocessor's functional units.

2.5.2 Timing Constraints:

Adherence to timing constraints is paramount to ensuring the microprocessor's reliable operation. Timing analysis is conducted to ascertain that critical paths within the control unit and associated components meet the specified timing requirements. This involves evaluating signal propagation delays, setup and hold times, and clock skew across the system. By meticulously optimizing timing parameters, the design can achieve the desired performance metrics while mitigating the risk of timing violations.

2.5.3 Control Signal Generation

The generation of control signals is central to the control unit's role in coordinating the operation of other components within the microprocessor. Different control signals are activated depending on the instruction to enable specific functions within the ALU, register array, memory interface, and other units. The control signal generation logic must be robust, responsive, and dynamically adapted to changing operational requirements.

2.5.4 Finite State Machine (FSM) Design

Finite State Machine (FSM) design techniques are widely employed in modeling the microprocessor's control logic. FSMs offer a systematic framework for representing the control unit's sequential behavior, delineating distinct states and transitions based on input conditions and internal state variables. By leveraging FSMs, designers can achieve granular control over the microprocessor's operation while maintaining modularity and scalability in the design.

2.5.5 Flexibility and Extensibility

The control unit's design prioritizes flexibility and extensibility, allowing for seamless integration of new instructions or modifications to existing ones. This involves modularly designing the control unit

architecture with well-defined interfaces facilitating easy expansion and customization. Additionally, provisions are made to accommodate enhancements or updates without necessitating extensive redesign or reconfiguration of the control unit.

2.5.6 Simulation and Verification

Simulation and verification serve as indispensable phases in validating the functionality and correctness of the control unit design. A comprehensive suite of testbenches is developed to simulate various instruction sequences, corner cases, and error conditions. Simulation results are meticulously analyzed to ensure the control unit behaves as expected under diverse scenarios. Moreover, formal verification techniques may be employed to rigorously verify the correctness of the control unit's operation, providing mathematical certainty of its functionality.

2.5.7 Timing Analysis

Timing analysis is conducted to verify that the microprocessor meets the required performance specifications regarding clock frequency, throughput, and latency. This involves analyzing critical paths within the control unit and associated components to identify potential timing bottlenecks and optimize timing parameters accordingly. By meticulously scrutinizing timing margins and addressing any timing violations, designers can ensure that the microprocessor operates reliably within its specified performance envelope. Through a meticulous synthesis of these design principles and validation methodologies, the control unit emerges as a robust and reliable component within the microprocessor, effectively steering its operation and enabling seamless execution of instructions across a diverse range of applications.

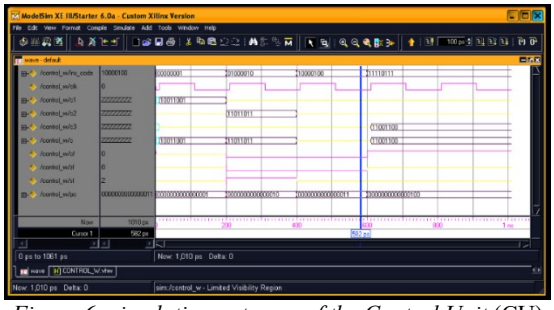

Figure 6: simulation outcome of the Control Unit (CU)

3. TRESILTS AND DISCUSSION

The design journey culminates as the microprocessor transitions into practical realization through its implementation on an FPGA platform. The FPGA, or Field Programmable Gate Array, is an eminently flexible hardware substrate that facilitates rapid prototyping and validation of the designed microprocessor in a real-world context. The process of FPGA implementation entails a series of intricate steps aimed at translating the synthesized Register Transfer Level (RTL) design into a tangible, executable form on the target FPGA architecture. This endeavor encompasses optimizing performance, area utilization, and power efficiency, ensuring that the resultant implementation aligns seamlessly with the intended design objectives.

3.1 Mapping to FPGA Architecture

At the outset of FPGA implementation, the synthesized RTL design undergoes a transformative process wherein it is mapped to the intricate architecture of the target FPGA. This entails allocating logic elements, routing resources, and memory blocks within the FPGA to accommodate the functional units and interconnections specified by the design. The mapping process is guided by resource availability, placement constraints, and routing feasibility, aiming to optimize the utilization of FPGA resources while adhering to design constraints.

3.2 Optimization for Performance and Area Utilization

Optimization lies at the heart of FPGA implementation, encompassing endeavors to enhance performance and area utilization. Techniques such as logic restructuring, pipelining, and parallelization are employed to maximize the throughput and minimize the latency of the microprocessor design. Concurrently, efforts are directed towards efficiently utilizing the available FPGA resources, striking a delicate balance between functionality and resource constraints. Through meticulous optimization, designers strive to extract the optimal performance from the FPGA-based implementation while economizing on hardware resources.

3.3 Power Efficiency Considerations

Power efficiency is critical in FPGA implementation, particularly in embedded systems where energy consumption is paramount. Techniques such as clock gating, power gating, and voltage scaling are employed to mitigate power dissipation and enhance energy efficiency. By dynamically controlling the power consumption of different functional units within the microprocessor, designers seek to strike a judicious balance between performance and power

consumption, extending the battery life of battery-powered devices and minimizing environmental impact.

3.4 Place-and-Route Algorithms

Central to FPGA implementation is the utilization of place-and-route algorithms, which orchestrate the allocation of physical resources and establishment of interconnections within the FPGA fabric. Place-and-route algorithms aim to optimize the placement of logic elements and routing paths to minimize signal propagation delays, routing congestion, and timing violations. These algorithms consider user-defined constraints, including timing constraints, placement directives, and routing preferences, to generate an optimized layout that meets the specified design objectives.

3.5 Testing and Validation

Following FPGA implementation, the microprocessor undergoes rigorous testing and validation to ascertain its correct operation and adherence to design specifications. Testbenches, comprising comprehensive sets of test cases and stimuli, are executed to verify the functionality of individual components and the overall system. Diagnostic programs may be employed to identify and rectify any discrepancies or anomalies encountered during testing. Moreover, performance analysis tools measure key performance metrics such as throughput, latency, and resource utilization under varying operating conditions.

3.6 Performance Analysis:

Performance analysis is a critical aspect of FPGA-based implementation, enabling designers to evaluate the efficacy and efficiency of the microprocessor design in real-world scenarios. Through performance profiling and benchmarking, designers gain insights into the microprocessor's computational capabilities, memory bandwidth, and responsiveness across different workloads. By analyzing performance metrics such as throughput, latency, and resource utilization, designers can identify potential bottlenecks, optimize system parameters, and refine the microprocessor design to effectively meet the demands of target applications. Having traversed the intricate terrain of FPGA implementation, the 8-bit soft processor emerges as a tangible entity, poised to unleash its computational prowess within the laboratory setting. The subsequent sections delineate the logic resource requirements and utilization of FPGA resources, providing a comprehensive overview of the microprocessor's implementation on the Virtex-5 FPGA platform.

Overview of HDL Synthesis	Device Usage In brief
Total Subtractors and Adders: 2	
Number of Slices: 76 out of 3584	
16-bit adder: 1	
Total Registers: 10	
8-bit addsub: 1	
Number of Slice Flip Flops: 56	
9-bit addsub: 1	
Total Latches: 20	
Number of 4 input LUTs: 126	
Total Comparators: 2	
Number of bonded IOBs: 54 out of 141 (38.29%)	
Total Multiplexers: 24	
Number of GCLKs: 1 out of 8 (12%)	
Total Decoders: 1	
Total Tristates: 14	
Total XORs: 1	

These statistics provide a comprehensive overview of the resource utilization of the FPGA, showcasing the efficient allocation and utilization of various logic elements to accommodate the functionality of the soft processor design. A major distinction of this work is its complete design and implementation pipeline, which spans from VHDL-based hardware description and functional simulation using ModelSim to FPGA synthesis and hands-on validation on a VIRTEX-5 platform. In contrast to studies like Amiri et al. [10], which emphasize dataflow architectures tailored for computationally intensive applications, this research focuses on developing a streamlined and predictable processor architecture optimized for embedded control systems, emphasizing minimal latency, efficient resource usage, and architectural simplicity rather than maximizing processing performance.

4. CONCLUSIONS

This paper presents a comprehensive overview of designing and implemention of an eight-bit CPU tailored for control applications. The design process encompasses various stages, including system considerations, register array design, design of the control unit, comparator and shifter, ALU functionality and implementation, FPGA implementation results, and concluding remarks. This research is based on the premise that current soft-core processors, which are primarily designed for general-purpose or high-performance tasks, fall

short in meeting the unique performance and efficiency requirements of control systems with low to moderate complexity. Through meticulous design and optimization, the proposed soft-core processor demonstrates efficient performance and versatility in handling control applications with simple to moderate complexity. The results from ModelSim simulations, along with the successful implementation on FPGA hardware, confirm the processor's capability to fulfill the requirements of control systems that demand predictable behavior, low response time, and efficient use of hardware resources. In contrast to earlier studies centered on high-performance or data-intensive processing, this work demonstrates that a streamlined, application-specific design approach can deliver effective and adaptable solutions for real-time control scenarios. VHDL and FPGA platforms enable rapid prototyping and testing, facilitating the development of custom microprocessor solutions for various applications. Overall, the proposed design does not incorporate advanced functionalities like pipelining, interrupt management, or built-in peripheral controllers, which may be essential for handling more complex or time-sensitive applications. This work advances soft-core processor design methodologies and their practical implementation in real-world applications.

ACKNOWLEDGMENTS

This study was supported by Baekseok University Research Fund.

REFERENCES

[1] Sharif S, Simbule M, Cabbil T, Kondamadugula S, Xiao Z, and Scott A., "Optimization and Implementation of a 32-Bit MIPS-Based Softcore Processor to Run Statistical Algorithms on FPGA", *In: Proceedings of ICIPCN 2024. Springer;* 2024. pp. 371–80. doi:10.1007/978-981-97-5810-4_32

[2] Ribeiro V. M., Santos N. S. M., Kapisch E. B., Silva L. R. M., and Duque C. A., "Real-Time Implementation of Stockwell Transform in FPGA Platform Using Soft-core Processor Applied to Novelty Detection in Power Quality Signals", *J Control Autom Electr Syst.,* 2024, Vol. 35, No. 3, pp. 509–521. doi:10.1007/s40313-024-01083-

[3] Denisov, L., Galimberti, A., Cattaneo, D., Agosta, G., and Zoni, D., "Design-time methodology for optimizing mixed-precision CPU architectures on FPGA", *J Syst Archit*. 2024, Vol. 155, p. 103257. doi:10.1016/j.sysarc.2024.103257.

[4] Sriraman, H., & Ravikumar, A., "Customized FPGA Design and Analysis of Soft-Core Processor for DNN", *Procedia Computer Science*, 2023, Vol. *218*, pp. 469-478. doi:10.1016/j.procs.2023.01.029.

[5] Wegrzyn M., Jamro E., Dąbrowska-Boruch A., and Wiatr K., "Optimal Reduction in the Number of Test Vectors for Soft Processor Cores Implemented in FPGA", *Electronics*. 2021, Vol. 10, No. 20, 2505. doi:10.3390/electronics10202505

[6] Maheswari R., Pattabiraman V., and Sharmila P., "Reconfigurable FPGA-Based Soft-Core Processor for SIMD Applications", *Asian J Pharm Clin Res*., 2017, Vol. 10, No. 1, pp. 217–222. doi:10.22159/ajpcr.2017.v10s1.19632

[7] Salim, M., et al., "Assembler design techniques for a reconfigurable soft-core processor," *Journal of Theoretical & Applied Information Technology*, 2014, Vol. 64, No. 2., pp. 461-469.

[8] Brown, S., & Rose, J., "FPGA and CPLD Architectures: A Tutorial", *IEEE Design & Test of Computers*, 1996, pp. 42-57. doi:10.1109/54.500200

[9] Jayakrishnan, V., & Parikh, C., "Embedded Processors on FPGA: Soft vs Hard", *In ASEE North Central Section Conference,* 2019, pp. 1-9.

[10] Amiri, M., Siddiqui, F. M., Kelly, C., Woods, R., Rafferty, K., & Bardak, B., "FPGA-Based Soft-Core Processors for Image Processing Applications", *Journal of Signal Processing System*, 87, 2017, pp. 139-156. doi:10.1007/s11265-016-1185-7

[11] Tong, J. G., Anderson, I. D. L., & Khalid, M. A. S., "Soft-Core Processors for Embedded Systems", *In 18th International Conference on Microelectronics (ICM),* 2006, pp. 170-173. doi:10.1109/ICM.2006.373294

[12] Minev, P. B., & Kukenska, V. S., "Implementation of soft-core processors in FPGAs", *In UNITECH'07 International Sceintific Conference.* 2007, November.

[13] Ball, J. "Designing soft-core processors for FPGAs", *In Processor Design: System-on-Chip Computing for ASICs and FPGAs, Dordrecht: Springer Netherlands*, 2007, pp. 229-256.

[14] Yiannacouras, P., Rose, J., & Steffan, J. G., "The microarchitecture of FPGA-based soft

processors", *In Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems,* 2005, September, pp. 202-212. doi:10.1145/1086297.1086325

[15] Makni, M., Baklouti, M., Niar, S., Jmal, M. W., & Abid, M., "A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems", *11th International Design & Test Symposium (IDT)*, 2016, December, pp. 154-159.
doi:10.1109/IDT.2016.7843032

[16] Elkateeb, A., "A Processor Design Course Project: Creating Soft-Core MIPS Processor Using Step-by-Step Components' Integration Approach", *International Journal of Information and Education Technology*, Vol. 1, No. 5, 2011, pp. 432-440.

[17] Mano, M. M., "Digital Logic and Computer Design", *Prentice Hall of India*, 2002.

[18] Leach, D. P., & Malvino, A. P., "Digital Principles and Applications (5th ed.)", *Tata McGraw-Hill*, 2002.

[19] Hamacher, V. C., Vranesic, Z. G., Zaky, S. G., Vransic, Z., & Zakay, S., "Computer organization", New York: McGraw-Hill, 1984, pp. 367-390.