

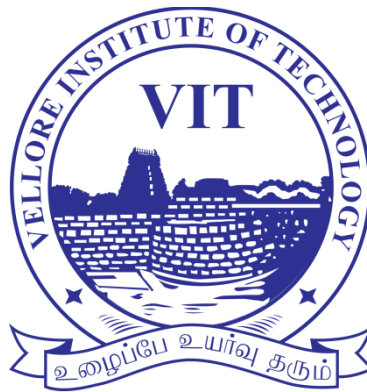
## DESIGN OF 8 - BIT MICROPROCESSOR

by

Vishnu Karthik R	21BLC1007
Allen Stanley	21BLC1022
Harish Kumar S	21BLC1029
Shrinivasan M	21BLC1047
Sudharsan S	21BLC1079
Gokul Girish	21BLC1115

*Under the guidance of*

*Dr. Sindhuja M*



**SCHOOL OF ELECTRONICS ENGINEERING**  
**VELLORE INSTITUTE OF TECHNOLOGY, CHENNAI-600127**

**July - 2023**

## DECLARATION

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.



Vishnu Karthik R

21BLC1007

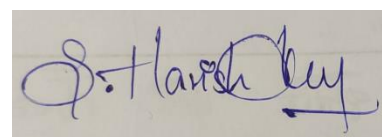
Date: 21-07-2023



Allen Stanley

21BLC1022

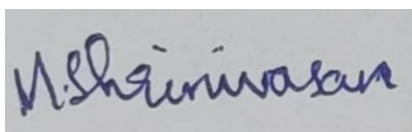
Date: 21-07-2023



Harish Kumar S

21BLC1029

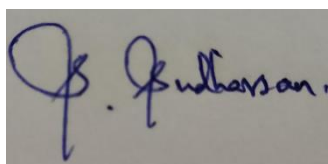
Date: 21-07-2023



Shrinivasan M

21BLC1047

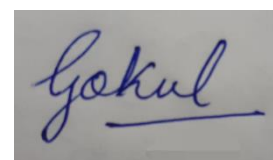
Date: 21-07-2023



Sudharsan S

21BLC1079

Date: 21-07-2023



Gokul Girish

21BLC1115

Date: 21-07-2023

## **CERTIFICATE**

It is certified that the work contained in the Continuous Assessment and Mini project(CAMP) titled “**Design of an 8-bit Microprocessor,**” by “Vishnu Karthik R, bearing Roll No: 21BLC1007; Allen Stanley, bearing Roll No: 21BLC1022; Harish Kumar S, bearing Roll No: 21BLC1029; Shrinivasan M, bearing Roll No: 21BLC1047; Sudharsan S, bearing Roll No: 21BLC10079; Gokul Girish, bearing Roll No: 21BLC1115” has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

**Signature of Supervisor**  
**Name: Dr. Sindhuja M**  
**School: SENSE**  
**VIT, Chennai**  
**July, 2023**

## ACKNOWLEDGEMENT

We are writing this acknowledgement address with profound gratitude and heartfelt appreciation for the successful completion of our project report. It is with great pleasure that we extend our sincere acknowledgments to all those who have played a significant role in the development and completion of this report.

First and foremost, we would like to express our deepest gratitude to our esteemed university for providing us with the opportunity to undertake this research and produce this report. The university's commitment to academic excellence, support for research endeavors, and fostering a conducive learning environment have been instrumental in our success. We are truly grateful for the resources, guidance, and encouragement provided by the university throughout this journey.

We would like to extend our sincere thanks to our CAO faculty, *Dr. Sindhuja M* for her invaluable guidance and mentor-ship. Her expertise, wisdom, and continuous support have been instrumental in shaping the direction and quality of this report. Her dedication to our academic growth and their commitment to excellence have been a constant source of inspiration.

Furthermore, we are deeply grateful to the participants and individuals who willingly participated in our research and provided us with valuable insights and data. Their willingness to share their experiences, expertise, and perspectives have been crucial in shaping the conclusions and recommendations of this report. Without their cooperation, this endeavor would not have been possible.

Lastly, we would like to acknowledge the contributions of our friends and family members who have provided unwavering support and encouragement during the challenging moments of this project. Their belief in our abilities and their understanding of the demands of academic pursuits have been a source of strength and motivation.

## **VIT - Chennai**

In conclusion, we express our heartfelt appreciation to all those who have been part of this journey. Your collective efforts, support, and collaboration have made this report a reality, and we are truly grateful for your contributions. Thank you.

Sincerely,

*Vishnu Karthik R - 21BLC1007*

*Allen Stanley - 21BLC1022*

*Harish Kumar S - 21BLC1029*

*Shrinivasan M - 21BLC1047*

*Sudharsan S - 21BLC1079*

*Gokul Girish - 21BLC1115*

## **ABSTRACT**

This paper presents the comprehensive design process of an 8-bit microprocessor, aimed at meeting the demands of modern computing while optimizing for performance, power efficiency, and cost-effectiveness. The microprocessor's architecture is based on the von Neumann model, integrating essential components such as the arithmetic and logic unit (ALU), control unit, registers, memory, and instruction set architecture (ISA). The primary objective is to create a versatile and capable microprocessor that can cater to various computing applications, ranging from simple embedded systems to more complex computing tasks.

The initial phase of the design involves defining the microprocessor's specifications, including its target clock frequency, word size, instruction set, and addressing capabilities. Extensive research on existing 8-bit microprocessors is conducted to identify best practices, challenges, and opportunities for improvement. This knowledge is then leveraged to devise an optimized architecture that balances performance and resource utilization.

The microprocessor's central processing unit (CPU) is designed with a multi-cycle architecture to strike a balance between simplicity and efficiency. The instruction decoder efficiently translates machine code into micro instructions, ensuring smooth execution of complex instructions while minimizing the overall instruction cycle time. A priority is placed on incorporating a diverse set of instructions that can cater to both general-purpose and application-specific tasks.

The ALU, a critical component of the microprocessor, is designed to support fundamental arithmetic and logical operations such as addition, subtraction, bit-wise AND, OR, and XOR. To enhance its versatility, additional specialized instructions are included for more complex operations, like multiplication and division, to accelerate computation in arithmetic-heavy tasks. The ALU is optimized for speed and size, considering both the data path width and number of functional units to ensure minimal execution time for most operations.

## **VIT - Chennai**

To achieve optimal performance, the microprocessor incorporates an efficient pipeline design that allows for instruction fetch, decode, execute, and write-back to occur in parallel. By overlapping these stages, the microprocessor maximizes its utilization, reducing the average instruction execution time and improving overall throughput.

To enhance memory access efficiency, the microprocessor utilizes various memory hierarchies, including on-chip caches and external memory support. Cache memory is implemented to reduce memory latency and minimize the bottleneck caused by memory access. The microprocessor employs a write-back policy to reduce external memory writes and further enhance memory performance.

In addition to its core components, the microprocessor features an interrupt handling mechanism that enables it to handle various external events and prioritize their handling effectively. Additionally, power management techniques such as clock gating and dynamic voltage scaling are employed to minimize power consumption while preserving performance during idle or low-utilization periods.

To validate the design, a hardware description language (HDL) is used to implement the microprocessor in a field-programmable gate array (FPGA) or an application-specific integrated circuit (ASIC). The microprocessor is extensively tested and bench-marked using industry-standard benchmark suites and real-world applications to evaluate its performance, power efficiency, and overall effectiveness.

In conclusion, this paper presents a comprehensive design approach for an 8-bit microprocessor that combines modern computing demands with optimization for performance, power efficiency, and cost-effectiveness. The resulting microprocessor provides a solid foundation for a wide range of computing applications and serves as a stepping stone for future developments in microprocessor design. By integrating cutting-edge techniques and emphasizing versatility, the microprocessor aligns with the demands of contemporary computing and offers significant potential for advancements in the field of microprocessor architecture.

## TABLE OF CONTENTS

	Page No
Title	i
Declaration	ii
Certificate	iii
Acknowledgements	iv
Abstract	vi
Table of Contents	viii

### Contents

1	Introduction	1
2	Literature Review	4
	2.1 Literature Review – I	4
	2.2 Literature Review – II	5
	2.3 Literature Review – III	6
	2.4 Literature Review – IV	7
	2.5 Literature Review - V	8
3	Methodology	10
4	Register Transfer Level in Intel Quartus Prime	12
5	Verilog HDL in ModelSim	16
6	Experimental Procedure	19
6	Codes	22
7	Outputs in ModelSim	50
8	RTL View in Intel Quartus Prime	52
9	Results & Discussion	57
10	Conclusions & Future Scope	59
11	References	63
12	Appendix	64



## INTRODUCTION

The microprocessor, a cornerstone of modern computing, has transformed the world by enabling the creation of powerful, efficient, and versatile electronic devices. From personal computers to smartphones, microprocessors have revolutionized every facet of modern life. In this context, this comprehensive study aims to delve into the intricacies of designing an 8-bit microprocessor - a compact yet potent computational engine that strikes a delicate balance between simplicity and functionality.

### **Background:**

The concept of a microprocessor originated in the early 1970s when Intel introduced the 4004 - the world's first commercially available microprocessor. Operating at a mere 740 kHz, it heralded the dawn of a new era in computing. Since then, microprocessors have evolved rapidly, with advancements in semiconductor technology driving exponential growth in processing power, energy efficiency, and cost-effectiveness.

Over the years, microprocessors have embraced various architectures, ranging from 4-bit and 8-bit to 32-bit and beyond, each catering to different application domains. While 32-bit and 64-bit microprocessors dominate high-performance computing, embedded systems, IoT devices, and educational tools often rely on 8-bit microprocessors due to their simplicity, cost-effectiveness, and power efficiency.

### **Purpose and Scope:**

The primary purpose of this research is to outline the design principles and considerations involved in creating an 8-bit microprocessor. We aim to develop a versatile and efficient processor that can handle diverse tasks, including basic arithmetic operations, logical computations, and data manipulations, all while adhering to an 8-bit data width.

The scope of this study encompasses various aspects of microprocessor design, including architecture selection, instruction set design, data and instruction memory management, arithmetic and logic unit (ALU) design, communication interfaces, power optimization

techniques, and verification methodologies. By examining each component meticulously, we aim to create a comprehensive blueprint for a highly functional 8-bit microprocessor.

### **Importance of 8-bit Microprocessors:**

Despite the prevalence of higher bit-width microprocessors, 8-bit microprocessors remain indispensable for certain applications due to their distinct advantages:

*Simplicity and Cost-effectiveness:* 8-bit microprocessors typically employ simpler architectures with a smaller number of transistors, resulting in lower production costs. Their relative simplicity also eases the design process and reduces the time-to-market for electronic products.

*Power Efficiency:* With fewer transistors to power, 8-bit microprocessors consume significantly less energy than their higher-bit counterparts. This attribute makes them ideal for battery-powered devices and energy-constrained applications.

*Versatility:* 8-bit microprocessors can be tailored to perform a wide range of tasks, including controlling peripherals, processing sensor data, running simple algorithms, and supporting educational initiatives. Their adaptability makes them suitable for diverse applications, from industrial automation to IoT devices.

### **Organization of the Study:**

This study is structured into several sections, each addressing a crucial aspect of 8-bit microprocessor design:

#### *Microprocessor Architecture Selection:*

This section explores different microprocessor architectures, highlighting the reasons for choosing the Harvard architecture approach, which segregates instruction and data memory spaces for parallel processing.

#### *Instruction Set Design:*

Here, we discuss the significance of Instruction Set Architectures (ISAs) and propose a RISC-based approach to simplify the microprocessor's design while retaining essential functionalities.

### *Data and Instruction Memory Management:*

This section delves into the design of the microprocessor's memory units, exploring various memory technologies and techniques for efficient data access and storage.

### *Arithmetic and Logic Unit (ALU) Design:*

The ALU is the heart of any microprocessor. In this section, we focus on designing an ALU capable of handling essential arithmetic and logical operations, while accommodating the 8-bit data width.

### *Communication Interfaces:*

To ensure seamless interaction with external devices, we discuss the integration of communication interfaces like UART, GPIO, I2C, and SPI.

### *Power Optimization Techniques:*

Power consumption is a critical consideration. This section explores clock gating, power gating, and other energy-saving methods to enhance the microprocessor's efficiency.

### *Verification Methodologies:*

Validating the correctness and functionality of the microprocessor is paramount. We delve into the use of hardware description languages and simulation tools for rigorous testing.

## **Conclusion:**

Finally, we summarize the findings and outline the potential impact of an efficiently designed 8-bit microprocessor in various application domains.

By presenting a comprehensive analysis and design blueprint, this study aims to contribute to the evolution of 8-bit microprocessors and foster their continued relevance in an increasingly complex and connected world.

## LITERATURE SURVEY

### 2.1 Literature Survey - I<sup>[1]</sup>:

❖ *Title & Authors:*

- ✧ Designing a low power 8-bit Application Specific Processor
- ✧ Lopamudra Samal, Chiranjibi Samal

❖ *Conference & Year of Publication:*

- ✧ International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)
- ✧ Year: 2014

❖ *Concept / Approach:*

This paper discusses the design of an 8-bit microprocessor based on the 8051 architecture. The processor includes an ALU, registers, and a system bus, all designed and tested separately before being combined. The VHDL code is used, and XILINX and Synopsis tools are utilized for simulation and synthesis. The microprocessor has 256-byte memory, 8-bit address and data buses, and exhibits an area of 5309 square micron and power consumption of 6.4707 microwatt. It is a simple yet capable microprocessor for various tasks.

❖ *Limitation:*

- ✧ Limited functionality
- ✧ Lack of performance details
- ✧ Memory limitations
- ✧ Absence of benchmark results
- ✧ Limited scalability
- ✧ Compatibility and interfacing options not discussed
- ✧ Lack of validation information
- ✧ No comparison or evaluation provided

❖ *Inference:*

The paper describes the design of an 8-bit microprocessor based on the 8051 architecture. It discusses separate design and testing of components, VHDL code usage, simulation, and synthesis tools. The microprocessor has limited functionality, memory, and scalability, with insufficient performance details and validation information provided.

## 2.2 Literature Survey - II<sup>[2]</sup>:

❖ *Title & Authors:*

- ✧ Logic Design of an 8-bit RSFQ Microprocessor
- ✧ Jia-Hong Yang, Guang-Ming Tang, Pei-Yao Qu, Xiao-Chun Ye, Dong-Rui Fan, Zhi-Min Zhang, and Ning-Hui Sun, Jia-Hong Yang, Pei-Yao Qu

❖ *Conference & Year of Publication:*

- ✧ IEEE International Superconductive Electronics Conference (ISEC)
- ✧ Year: 2019

❖ *Concept / Approach:*

A proposed 8-bit RSFQ microprocessor called HUTU utilizes a Harvard-type architecture for parallel processing. It executes 28 instructions, each consisting of eight bits. Asynchronous timing in the control unit avoids pipeline flushing, while concurrent-flow clocking in the data-path ensures high performance. Simulation results demonstrate proper functioning of HUTU's elements.

❖ *Limitation:*

The limitation is that it does not provide specific details about the performance or efficiency of the HUTU microprocessor. It does not mention any benchmark results, clock frequency, or execution time for the instructions. Additionally, it lacks information about memory capacity, scalability, or compatibility with external devices or interfaces. Without these details, it is challenging to assess the overall limitations of the HUTU microprocessor.

### ❖ *Inference:*

The paper introduces HUTU, an 8-bit RSFQ microprocessor using a Harvard-type architecture and asynchronous timing for the control unit. It supports 28 instructions and adopts concurrent-flow clocking in the data-path for high performance. However, no specific performance details or limitations are mentioned.

## 2.3 Literature Survey - III<sup>[3]</sup>:

### ❖ *Title & Authors:*

- ✧ Application-Driven Power Efficient ALU Design Methodology for Modern Microprocessors
- ✧ Na Gong, Jinhui Wang, Ramalingam Sridhar

### ❖ *Conference & Year of Publication:*

- ✧ International Symposium on Quality Electronic Design (ISQED)
- ✧ Year: 2013

### ❖ *Concept / Approach:*

This paper presents a methodology for designing application-driven ALUs in modern microprocessors, focusing on high power efficiency. A PN selection algorithm (PNSA) is introduced to help designers choose power-efficient dynamic modules based on detailed analysis of dynamic circuits. Experimental results on benchmark circuits show that the proposed approach can reduce power consumption by 54%-60% for different frequency levels compared to conventional dynamic ALU designs. These findings demonstrate the effectiveness of the application-driven custom ALU design method.

### ❖ *Limitation:*

It does not provide specific details about the potential trade-offs or drawbacks associated with the proposed application-driven ALU design methodology. It does not mention any potential impact on performance, area utilization, or other design considerations. Additionally, the limitations of the benchmark circuits used in the experiments are not discussed, which may affect the generalization of the results. Without these details, it is difficult to fully evaluate the limitations of the proposed methodology.

❖ *Inference:*

Presents a methodology for designing power-efficient ALUs in modern microprocessors using a PN selection algorithm. Experimental results demonstrate significant power consumption reduction compared to conventional designs. However, the paragraph lacks specific details about potential trade-offs or limitations of the proposed methodology and does not discuss the generalization of the results.

**2.4 Literature Survey - IV<sup>[4]</sup>:**

❖ *Title & Authors:*

- ✧ Energy Efficient 8-Bit Microprocessor For Wireless Sensor Network Applications
- ✧ Y. Hoon, N. A. Kamsani, R. M. Sidek, N. Sulaiman, F. Z. Rokhani

❖ *Conference & Year of Publication:*

- ✧ 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC)
- ✧ Year: 2013

❖ *Concept / Approach:*

To meet the demand for energy-efficient electronics, a low-power 8-bit microprocessor is proposed. It is designed using Silterra 130nm CMOS logic processes with a chip layout size of 4900  $\mu\text{m}^2$ . Operating at 1.2V and 50MHz, the microprocessor significantly reduces leakage power compared to its counterpart. Applying clock gating technique further reduces dynamic power by 44.6%.

❖ *Limitation:*

The limitation in the paper is that it does not provide specific details or analysis regarding the performance, functionality, or other important aspects of the proposed low-power microprocessor design. Additionally, it does not discuss any potential trade-offs or limitations of the low-power clock gating technique employed. The paper also does not mention any comparative analysis with other existing microprocessors or architectures, making it challenging to assess the overall limitations of the proposed design.

❖ *Inference:*

The paper introduces a low-power 8-bit microprocessor designed using specific CMOS logic processes. It demonstrates a substantial reduction in leakage power and further power reduction through clock gating technique. However, it lacks specific details on performance, trade-offs, or comparative analysis with other microprocessors.

**2.5 Literature Survey - V<sup>[5]</sup>:**

❖ *Title & Authors:*

- ✧ Low Power 8-bit ALU Design Using Full Adder and Multiplexer
- ✧ Anitesh Sharma, Ravi Tiwari

❖ *Conference & Year of Publication:*

- ✧ International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)
- ✧ Year: 2016

❖ *Concept / Approach:*

Arithmetic logic unit (ALU) is an important part of microprocessor. In digital processor logical and arithmetic operation executes using ALU. In this paper we describes 8-bit ALU using low power 11-transistor full adder (FA) and Gate diffusion input (GDI) based multiplexer. By using FA and multiplexer, we have reduced power and delay of 8-bit ALU as compare to existing design. All design were simulated using Tanner EDA tool v15.0 in 32nm BSIM4 technology. Performance analyses were done with respect to power, delay and power delay product.

❖ *Limitation:*

The lack of specific details or analysis regarding the extent of power and delay reduction achieved by the proposed ALU design. It does not mention any specific benchmark or comparative analysis with existing designs. Additionally, the paper does not provide information about other important performance metrics or potential trade-offs associated



with the proposed design. The limitations also include the lack of information about the complexity or area utilization of the design.

❖ *Inference:*

This paper introduces an 8-bit ALU design for microprocessors that aims to reduce power and delay compared to existing designs. It utilizes a low-power full adder and a Gate diffusion input-based multiplexer. Simulations were conducted using a specific EDA tool and technology. However, the paper lacks specific details regarding the extent of power and delay reduction achieved, benchmark comparisons, other performance metrics, and potential trade-offs.

## METHODOLOGY

The process of creating a microprocessor from Verilog code to Intel Quartus Prime RTL view typically involves several steps. Here is a high-level overview of the methodology:

1. **Design Specification:** Begin by defining the specifications and requirements of the microprocessor you intend to create. This includes determining the instruction set architecture (ISA), the desired performance characteristics, the supported peripherals, and any specific features or optimizations required.
2. **Micro-architecture Design:** Based on the design specifications, develop the micro-architecture of the processor. This involves designing the various components, such as the control unit, arithmetic logic unit (ALU), register files, memory hierarchy, and data paths. The micro-architecture design defines how these components are interconnected and how they function together to execute instructions.
3. **Verilog Coding:** Write the Verilog code that describes the microprocessor's behavior and functionality. Verilog is a hardware description language (HDL) commonly used for designing digital systems. The code should represent the micro-architecture and implement the desired instruction set.
4. **Simulation and Verification:** Use Verilog simulation tools, such as Model-sim, to verify the functionality of the microprocessor design. Create test-bench files that provide inputs to the microprocessor and compare the outputs against expected results. Simulation helps identify and fix any design issues or functional bugs.
5. **Synthesis:** Once the Verilog design is thoroughly simulated and verified, perform synthesis using the Intel Quartus Prime software. Synthesis translates the behavioral Verilog code into a gate-level net-list representation, which consists of logical gates and flip-flops. The synthesis process optimizes the design for target performance, area utilization, and power consumption.

6. Technology Mapping and Optimization: After synthesis, the next step is technology mapping, where the gate-level net-list is mapped to the specific target technology library provided by Intel Quartus Prime. This process involves mapping the logical gates to their corresponding library cells. Additionally, optimization techniques are applied to improve performance, reduce power consumption, and minimize area utilization.

7. Placement and Routing: The Intel Quartus Prime software performs placement and routing to determine the physical layout of the microprocessor. Placement involves determining the locations of the gates and flip-flops on the chip's surface, while routing establishes the interconnects between them. These steps ensure proper signal propagation and timing closure.

8. Timing Analysis: Timing analysis is conducted to verify that the microprocessor meets the desired timing requirements. It ensures that signals arrive at their intended destinations within specified time constraints. The Intel Quartus Prime software provides tools to analyze and optimize the timing performance of the design.

9. Generation of RTL View: Once the placement and routing are completed, the Intel Quartus Prime software generates the RTL (Register Transfer Level) view of the microprocessor design. The RTL view provides a graphical representation of the micro-architecture and shows the interconnections between different components at the gate level.

## **REGISTER TRANSFER LEVEL IN INTEL QUARTUS PRIME**

The Register-Transfer Level (RTL) view is a fundamental concept in digital design, representing a detailed abstraction of the hardware implementation of a microprocessor. In the context of designing an 8-bit microprocessor, Intel Quartus Prime, a powerful software suite for FPGA development, provides an environment to visualize, simulate, and synthesize the RTL representation of the microprocessor design. This explanation delves into the RTL view of an 8-bit microprocessor, exploring its key components, data paths, control logic, and interaction with memory and peripherals, elucidating the RTL view's role in microprocessor design and verification.

### **Overview of RTL View in Intel Quartus Prime:**

The RTL view in Intel Quartus Prime represents the microprocessor's hardware description at the Register-Transfer Level, which is a low-level abstraction capturing the flow of data between registers. This representation describes the microprocessor's behavior at the gate and flip-flop level, showcasing how data moves from one register to another through combinational logic. The RTL view offers an essential perspective for microprocessor designers, enabling them to validate the correctness and efficiency of the hardware implementation before synthesis and physical realization.

### **Components of the 8-bit Microprocessor:**

The RTL view of the 8-bit microprocessor comprises several crucial components:

#### **Arithmetic and Logic Unit (ALU):**

The ALU is responsible for executing arithmetic and logical operations on data stored in registers. In the RTL view, the ALU consists of various functional units capable of performing operations such as addition, subtraction, AND, OR, XOR, and shift

operations. The ALU takes inputs from the registers, performs the specified operation, and writes the result back to the destination register.

**Control Unit:**

The Control Unit generates control signals to coordinate the microprocessor's operations. In the RTL view, the Control Unit decodes instruction opcodes fetched from memory and generates control signals to instruct the ALU, registers, and other components to perform the appropriate operations.

**Register File:**

The Register File is a set of registers that temporarily store data during microprocessor operations. In the RTL view, each register is represented as a group of flip-flops, and data movement between registers is depicted through data paths.

**Program Counter (PC):**

The Program Counter holds the memory address of the next instruction to be fetched and executed. In the RTL view, the PC is implemented as a register that increments by the appropriate value with each instruction fetch.

**Memory Interface:**

The memory interface enables communication between the microprocessor and external memory, where instructions and data are stored. In the RTL view, the memory interface manages the address and data buses, control signals, and read/write operations to interact with memory.

**3.3 RTL Data Paths:**

Data paths in the RTL view represent the routes through which data flows between the components of the microprocessor. In an 8-bit microprocessor, data paths are typically 8-bits wide to handle data in 8-bit chunks. Each data path consists of combinational logic that performs operations such as addition, logical operations, and data movement between registers.

**Instruction Fetch:**

In the RTL view, the instruction fetch process starts with the Program Counter (PC). The PC value is loaded into the Address Register, which communicates with the memory interface to fetch the instruction from memory. The fetched instruction is then stored in the Instruction Register.

**Instruction Decode:**

The fetched instruction is decoded in this stage to generate control signals for the microprocessor's subsequent operations. The opcode and operand fields of the instruction are extracted, and the appropriate control signals are sent to the ALU, registers, and memory interface.

**Data Movement:**

Data movement operations in the RTL view involve transferring data between registers or between registers and the ALU. The source register's content is passed through data paths to the destination register or ALU inputs, where the specified operation is executed, and the result is stored in the destination register.

**3.4 RTL Control Logic:**

The RTL control logic is responsible for generating control signals based on the decoded instruction and the current microprocessor state. It coordinates the timing and sequencing of various operations to ensure correct execution of instructions.

**Instruction Sequencing:**

The control logic ensures that the microprocessor follows the correct sequence of operations for each instruction. It coordinates instruction fetch, decode, and execution cycles, making sure that data dependencies and hazards are appropriately resolved.

**Microcode Execution:**

In some microprocessors, particularly micro-coded architectures, the RTL control logic interprets the decoded instructions and executes a sequence of micro-instructions to

perform the desired operation. Each microinstruction corresponds to a specific hardware operation.

### **3.4.3 Branch and Jump Operations:**

The control logic handles conditional and unconditional branch and jump instructions. It updates the Program Counter (PC) to the target address based on the evaluation of the branch condition or the specified jump address.

## **3.5 Memory Interaction:**

In the RTL view, the microprocessor's interaction with memory is facilitated by the memory interface, which coordinates address and data transfers between the microprocessor and external memory.

### **Memory Read:**

When executing a memory read instruction, the memory interface sends the address to external memory and retrieves the corresponding data. The data is then stored in the destination register.

### **Memory Write:**

During a memory write operation, the memory interface receives data from a source register and sends it to the specified memory address for storage.

The RTL view in Intel Quartus Prime provides a detailed and comprehensive representation of an 8-bit microprocessor's hardware implementation. It showcases the data paths, control logic, and memory interaction necessary for executing instructions and managing data. By visualizing the RTL view, microprocessor designers can verify the correctness, efficiency, and timing of their designs, enabling them to optimize and fine-tune the microprocessor's performance before synthesis and physical implementation on FPGAs or ASICs. Understanding the RTL view is crucial for successfully designing and validating complex microprocessor architectures, ensuring they meet the desired specifications and deliver efficient and reliable computing capabilities.

## VERILOG HDL IN MODELSIM

Verilog HDL (Hardware Description Language) plays a critical role in designing an 8-bit microprocessor using ModelSim, a popular simulation tool. Verilog HDL allows designers to describe the microprocessor's behavior at the register transfer level, defining the hardware components and their interactions. This explanation will cover the key aspects of Verilog HDL and its application in the design process of an 8-bit microprocessor.

### **Introduction to Verilog HDL:**

Verilog HDL is a hardware description language used for modeling, simulating, and synthesizing digital hardware. It provides an abstract representation of hardware behavior, facilitating the design and verification of complex digital systems like microprocessors. ModelSim, a widely-used simulation tool, enables the simulation of Verilog code and facilitates the debugging and optimization of designs before synthesis and implementation.

### **Designing the 8-bit Microprocessor in Verilog:**

The design process begins by specifying the microprocessor's architecture, instruction set, and functionality. The microprocessor comprises essential components like the Arithmetic Logic Unit (ALU), Control Unit, Register File, Memory Interface, and Instruction Decoder. Each component is represented using Verilog HDL modules.

### **Verilog HDL Module Definitions:**

Verilog HDL organizes the microprocessor design into modules, each representing a hardware component. For example, the ALU, being a vital part of the microprocessor, is designed as a separate Verilog module. The ALU module contains inputs for operands and control signals, and outputs for results and status flags. Similarly, other components like the Register File and Instruction Decoder are implemented as separate Verilog modules.



**Data-path Implementation:**

The microprocessor's datapath, responsible for data movement and ALU operations, is designed using Verilog HDL. This includes multiplexers for data selection, registers for temporary storage, and ALU control logic. The datapath module integrates various sub-modules representing individual components, ensuring proper data flow and control.

**Control Unit Implementation:**

The microprocessor's Control Unit is responsible for coordinating instruction execution and datapath operations. In Verilog HDL, the Control Unit module takes inputs from the Instruction Decoder and generates control signals for the datapath, ALU, and other components. The Control Unit's design involves combinational logic and state machines, effectively managing the microprocessor's operation.

**Instruction Decoder:**

The Instruction Decoder module translates machine code instructions into microinstructions that control the microprocessor's behavior. Verilog HDL is used to design the instruction decoder module, mapping the instruction opcode to the corresponding microinstruction and generating control signals accordingly.

**Memory Interface:**

The microprocessor interacts with memory for data storage and retrieval. The Memory Interface module is designed in Verilog HDL, incorporating address decoding logic and interfacing with on-chip or external memory components. The module handles read and write operations, ensuring seamless data transfer between the microprocessor and memory.

**Test-bench Development:**

To verify the functionality and correctness of the Verilog HDL modules, test-benches are created. A testbench is a Verilog code that applies input stimuli to the microprocessor's modules and monitors their outputs. ModelSim uses these test-benches to simulate the microprocessor's behavior and validate its functionality.

**Simulation in ModelSim:**

With the Verilog HDL modules and corresponding test-benches prepared, the design is simulated using ModelSim. The simulation process allows designers to observe the microprocessor's behavior under various input scenarios and verify that it operates as expected .

**Debugging and Optimization:**

During simulation in ModelSim, potential issues and bugs in the Verilog HDL code may be identified. Designers use ModelSim's debugging features to track down and resolve these issues. Additionally, simulation results help in optimizing the design, ensuring efficient performance and minimal timing violations.

**Synthesis and Implementation:**

Once the Verilog HDL design is verified and optimized through simulation, it is synthesized into a gate-level netlist. ModelSim facilitates the integration of the design with a target FPGA or ASIC device, converting the high-level Verilog code into a hardware-specific implementation.

Verilog HDL, when used in ModelSim, is a powerful tool for designing and simulating an 8-bit microprocessor. The language's ability to model hardware behavior and the simulation capabilities of ModelSim allow designers to thoroughly validate the microprocessor's functionality before synthesis and implementation. By leveraging Verilog HDL and ModelSim, designers can confidently create efficient and functional microprocessors that meet the demands of modern computing applications.

## EXPERIMENTAL PROCEDURE

Designing an 8-bit microprocessor in Intel Quartus Prime involves a systematic experimental procedure. Below is a step-by-step guide on how to approach the design process:

### 1. Project Setup:

- Open Intel Quartus Prime and create a new project for the 8-bit microprocessor design.
- Specify the target FPGA device and set up the project settings, including the working directory and output files location.

### 2. Microprocessor Specification:

- Define the specifications of the 8-bit microprocessor, including its instruction set, word size, clock frequency, and memory requirements.
- Determine the microprocessor's architecture, instruction formats, and addressing modes.

### 3. Microprocessor Architecture Design:

- Design the microprocessor's architecture, including components such as the Arithmetic Logic Unit (ALU), Control Unit, Register File, and Memory Interface.
- Organize the microprocessor design into separate modules to facilitate hierarchical design and ease of implementation.

### 4. RTL Coding:

- Use Verilog HDL or SystemVerilog to write the RTL code for each module in the microprocessor's architecture.
- Define the input and output ports of each module and specify the internal logic and behavior of the components.

### 5. Integration and Connectivity:

- Connect the individual modules to create the complete microprocessor design.

## **VIT - Chennai**

- Ensure that the signals and data paths are properly connected between modules to enable proper data flow and control.

### **6. Test-bench Development:**

- Create test-benches for each module and the overall microprocessor design to verify their functionality.
- Develop test cases that cover various instructions and edge cases to thoroughly validate the microprocessor's behavior.

### **7. Functional Simulation:**

- Perform functional simulations using ModelSim or other simulation tools within Intel Quartus Prime.
- Run the test-benches and analyze the simulation results to verify that the microprocessor operates correctly according to its specifications.

### **8. Debugging and Verification:**

- Identify and resolve any issues or bugs that arise during simulation.
- Debug the design by inspecting waveforms, variables, and signals to ensure the microprocessor behaves as intended.

### **9. Synthesis:**

- After successful functional verification, proceed with the synthesis of the microprocessor design.
- Synthesis converts the RTL code into a gate-level net-list that can be programmed into the target FPGA.

### **10. Timing Analysis:**

- Perform timing analysis to ensure that the microprocessor design meets the required clock frequency and setup/hold time constraints.
- Make any necessary adjustments to improve timing performance if required.

### **11. Implementation:**

- Generate the programming file (e.g., .sof or .pof) for the target FPGA device based on the synthesized design.

- Program the FPGA with the microprocessor design to test it on real hardware.

### **12. FPGA Testing:**

- Test the functionality of the microprocessor on the FPGA using a development board or custom hardware setup.
- Run various software programs or assembly code to verify that the microprocessor executes instructions correctly and produces the expected results.

### **13. Performance Evaluation:**

- Benchmark the microprocessor to measure its performance, including execution time for different instructions and overall throughput.
- Evaluate the power consumption and resource utilization of the microprocessor.

### **14. Iterative Optimization:**

- If necessary, iterate through the design, simulation, synthesis, and implementation stages to optimize the microprocessor's performance and resource usage.
- Make incremental improvements based on simulation and FPGA testing results.

### **15. Documentation and Reporting:**

- Document the entire design process, including architecture, RTL code, simulation results, synthesis report, implementation details, and performance metrics.
- Provide a comprehensive report summarizing the design, verification, and evaluation of the 8-bit microprocessor.

By following this experimental procedure, designers can effectively design, simulate, synthesize, and implement an 8-bit microprocessor in Intel Quartus Prime and achieve a functional and efficient microprocessor for various computing applications.

## CODES

### ALU:

```
module ALU(input [31:0] data1,data2,input [3:0] aluoperation,output reg [31:0] result,output reg
zero,lt,gt);
    always@(aluoperation,data1,data2)
    begin
        case (aluoperation)
            4'b0000 : result = data1 + data2; // ADD
            4'b0001 : result = data1 - data2; // SUB
            4'b0010 : result = data1 & data2; // AND
            4'b0011 : result = data1 | data2; // OR
            4'b0100 : result = data1 ^ data2; // XOR
            4'b0101 : result = {31'b0,lt}; //slt
            // if you want to add new Alu instructions add here
            default : result = data1 + data2; // ADD
        endcase
        if(data1>data2)
        begin
            gt = 1'b1;
            lt = 1'b0;
        end else if(data1<data2)
        begin
            gt = 1'b0;
            lt = 1'b1;
        end
        if (result==32'd0) zero=1'b1;
        else zero=1'b0;
    end
endmodule
```

**ALU control:**

```
module ALUcontrol(clk,funct,ALUOp,ALUsignal);
input clk;
input[5:0] funct;
input[2:0] ALUOp;
output[3:0] ALUsignal;
reg[3:0] ALUsignal;
always@(funct , ALUOp )begin
    case(ALUOp)
        3'b010:case(funct)
            6'b100000:ALUsignal=4'b00; //add
            6'b100010:ALUsignal=4'b01; //sub
            6'b100100:ALUsignal=4'b10; //and
            6'b100101:ALUsignal=4'b11; //or
            6'b101010:ALUsignal=4'b101; //stl
        endcase
        3'b001:ALUsignal=4'b0001; //branch
        3'b000:ALUsignal=4'b0000; //Lw and sw
        3'b011:ALUsignal=4'b0010; //andi
        3'b100:ALUsignal=4'b0011; //ori
        3'b111:ALUsignal=4'b0101; //slti
    endcase
end
endmodule
```

---

**Controlstall:**

```
module Controlstall(clk,op1,op2,op3,stall);
input [5:0]op1,op2,op3;
input clk;
output reg stall;
initial begin
    stall=1'b1;
end
```

## VIT - Chennai

```
always @(posedge clk)begin
```

```
if(op1==6'b000100 | op1==6'b000101 | op1==6'b000010)begin// beq or bne or j  
stall=1'b0;
```

```
end
```

```
if(op2==6'b000100 | op2==6'b000101 | op2==6'b000010)begin// beq or bne or j  
stall=1'b0;
```

```
end
```

```
if(op3==6'b000100 | op3==6'b000101 | op3==6'b000010)begin// beq or bne or j  
stall=1'b0;
```

```
end
```

```
else begin
```

```
stall=1'b1;
```

```
end
```

```
end
```

```
endmodule
```

---

### **nopSet:**

```
module nopSet(clk,s1,s2,oldF,oldD,newF,newD);
```

```
input clk,s1,s2;
```

```
input [31:0] oldF,oldD;
```

```
output reg[31:0]newD,newF;
```

```
initial begin
```

```
end
```

```
always @(posedge clk)begin
```

```
if(s1==1'b0 && s2==1'b0)begin
```

```
newF=32'b0;
```

```
newD=32'b0;
```

```
end
```

```
else if(s1==1'b0 && s2==1'b1)begin
```

```
newD=32'b0;
```

```
end
```

```
else if(s1==1'b1 && s2==1'b0)begin
```

```
newF=32'b0;
```



## VIT - Chennai

```
end
else
begin
newD=oldD;
newF=oldF;
end
end
endmodule
```

---

### **DMemBank:**

```
module DMemBank(input memread, input memwrite, input [31:0] address, input [31:0]
writedata, output reg [31:0] readdata);

reg [31:0] mem_array [127:0];
wire[6:0]finalAddress;
assign finalAddress=address[8:0];
integer i;
initial
begin
    for (i=0; i<127; i=i+1)
        mem_array[i]=i*10;
end
always@(memread, memwrite, address, mem_array[address], writedata)
begin
    if(memread)begin
        readdata=mem_array[finalAddress];
    end
    if(memwrite)
    begin
        mem_array[finalAddress]= writedata;
    end
end
endmodule
```

---

**EXMEM:**

```
module EXMEM(clock,iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,
iMemToReg,iBranchs,iJumps,iALUCtrl, iIR,iPC,iB,iResult,iRegDest,iBranch,iJump,iZero,
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps,oAL
UCtrl, oIR,oPC,oB,oResult,oRegDest,oBranch,oJump,oZero,enable);
input [31:0] iIR,iPC,iB,iResult,iBranch,iJump;
input iZero,clock,enable;
input iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,iMemToReg,iBranchs,iJumps;
input [3:0]iALUCtrl;
input [4:0] iRegDest;
output [31:0] oIR,oPC,oB,oResult,oBranch,oJump;
output oZero;
output
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
output [3:0]oALUCtrl;
output [4:0]oRegDest;
reg [31:0] oIR,oPC,oB,oResult,oBranch,oJump;
reg oZero;
reg oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
reg [3:0]oALUCtrl;
reg [4:0]oRegDest;
initial begin
oPC=32'b0;
oIR=32'b0;
end
always @(posedge clock)
begin
if(enable)begin
oRegDests<=iRegDests;
oRegWrite<=iRegWrite;
oALUSrc<=iALUSrc;
oMemRead<=iMemRead;
oMemWrite<=iMemWrite;
oMemToReg<=iMemToReg;
```

## VIT - Chennai

```
oBranchs<=iBranchs;
oJumps<=iJumps;
oALUCtrl<=iALUCtrl;
oIR<=iIR;
oPC<=iPC;
oB<=iB;
oResult<=iResult;
oRegDest<=iRegDest;
oBranch<=iBranch;
oJump<=iJump;
oZero<=iZero;
end
end
endmodule
```

---

### IDEX:

```
module IDEX(clock,iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,
iMemToReg,iBranchs,iJumps,iALUCtrl, iIR,iPC,iA,iB,iRegDest,iBranch,iJump,
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps,oAL
UCtrl, oIR,oPC,oA,oB,oRegDest,oBranch,oJump,enable);
input [31:0] iIR,iPC,iA,iB,iBranch,iJump;
input clock,enable;
input iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,iMemToReg,iBranchs,iJumps;
input [3:0]iALUCtrl;
input [4:0] iRegDest;
output [31:0] oIR,oPC,oA,oB,oBranch,oJump;
output
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
output [3:0]oALUCtrl;
output [4:0]oRegDest;
reg [31:0] oIR,oPC,oA,oB,oResult,oBranch,oJump;
reg oZero;
reg oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
reg [3:0]oALUCtrl;
```

## VIT - Chennai

```
reg [4:0]oRegDest;
initial begin
    oPC=32'b0;
    oIR=32'b0;
end
always @(posedge clock)
begin
    if(enable)begin
        oRegDests<=iRegDests;
        oRegWrite<=iRegWrite;
        oALUSrc<=iALUSrc;
        oMemRead<=iMemRead;
        oMemWrite<=iMemWrite;
        oMemToReg<=iMemToReg;
        oBranchs<=iBranchs;
        oJumps<=iJumps;
        oALUCtrl<=iALUCtrl;
        oIR<=iIR;
        oPC<=iPC;
        oA<=iA;
        oB<=iB;
        oRegDest<=iRegDest;
        oBranch<=iBranch;
        oJump<=iJump;
    end
end
endmodule
```

---

### **IMemBank:**

```
module IMemBank(input memread, input [31:0] address, output reg [31:0] readdata);
    reg [31:0] mem_array [255:0];
    reg [31:0]temp;
    integer i;
    initial begin
```

```

    for (i=11; i<255; i=i+1)
        begin
            mem_array[i]=32'b0;
        end
    end
    always@(memread, address, mem_array[address])
    begin
        if(memread)begin
            temp=address>>2;
            readdata=mem_array[temp];
        end
        ////////////////////////////////// test bench!

        //lvl 1
        /*mem_array[0]={ 6'b0,5'b10,5'b1,5'b11,5'b0,6'b100000};//add reg3,reg2,reg1 =30
        mem_array[1]={ 6'b0,5'b10,5'b1,5'b100,5'b0,6'b100010};//sub reg4,reg2,reg1 =10
        mem_array[2]={ 6'b0,5'b10,5'b1,5'b101,5'b0,6'b100101};//or reg5,reg2,reg1 =30
        mem_array[3]={ 6'b0,5'b10,5'b1,5'b110,5'b0,6'b100100};//and reg6,reg2,reg1 =0
        mem_array[4]={ 6'b0,5'b10,5'b1,5'b111,5'b0,6'b101010};//stl reg7,reg2,reg1 =0

        mem_array[5]={ 6'b001000,5'b10,5'b1000,16'b1010};//addi r8,r2,10 =30
        mem_array[6]={ 6'b001010,5'b10,5'b1001,16'b1010};//slti r9,r2,10 =0
        mem_array[7]={ 6'b001101,5'b10,5'b1010,16'b1010};//ori r10,r2,10 =30
        mem_array[8]={ 6'b001100,5'b10,5'b1011,16'b1010};//andi r11,r2,10 =0

        mem_array[9]={ 6'b101011,5'b10,5'b0,16'b1};//sw reg0,1(reg2) =mem[21]<=0
        mem_array[10]={ 6'b100011,5'b0,5'b1100,16'b0};//lw reg12,0(reg0) =reg[12]<=0
    */
    //////////////////////////////////

    //lvl 2
    /*mem_array[0]={ 6'b0,5'b10,5'b1,5'b11,5'b0,6'b100000};//add reg3,reg2,reg1 =30
    mem_array[1]={ 6'b0,5'b11,5'b1,5'b100,5'b0,6'b100010};//sub reg4,reg3,reg1 =20
    mem_array[2]={ 6'b0,5'b100,5'b1,5'b101,5'b0,6'b100101};//or reg5,reg4,reg1 =30
    mem_array[3]={ 6'b0,5'b10,5'b1,5'b110,5'b0,6'b100100};//and reg6,reg2,reg1 =0
    mem_array[4]={ 6'b0,5'b10,5'b1,5'b111,5'b0,6'b101010};//stl reg7,reg2,reg1 =0

```

```

mem_array[5]={6'b001000,5'b10,5'b1000,16'b1010};//addi r8,r2,10 =30
mem_array[6]={6'b001010,5'b1000,5'b1001,16'b1000000};//slti r9,r8,32 =1
mem_array[7]={6'b001101,5'b10,5'b1010,16'b1010};//ori r10,r2,10 =30
mem_array[8]={6'b001100,5'b1001,5'b1011,16'b1010};//andi r11,r9,10 =0

mem_array[9]={6'b101011,5'b1010,5'b1011,16'b1};//sw reg11,1(reg10) =mem[31]<=0
mem_array[10]={6'b100011,5'b1010,5'b1100,16'b01};//lw reg12,1(reg10)
=reg[12]<=0*/
////////////////////////////////////
//lv1 3
// mem_array[0]={6'b0,5'b10,5'b1,5'b11,5'b0,6'b100000};//add reg3,reg2,reg1 =30
// mem_array[1]={6'b0,5'b11,5'b1,5'b100,5'b0,6'b100010};//sub reg4,reg3,reg1 =20
// mem_array[2]={6'b0,5'b100,5'b1,5'b101,5'b0,6'b100101};//or reg5,reg4,reg1 =30

// mem_array[3]={6'b100,5'b100,5'b101,16'b10};//beq r4,r5,2ta

// mem_array[4]={6'b0,5'b10,5'b1,5'b110,5'b0,6'b100100};//and reg6,reg2,reg1 =0

// mem_array[5]={6'b101,5'b110,5'b101,16'b1000};//bne r6,r5,2ta

// mem_array[6]={6'b0,5'b10,5'b1,5'b111,5'b0,6'b101010};//stl reg7,reg2,reg1 =0

// mem_array[7]={6'b10,26'b100};//jump lable4 13 ya 14

// mem_array[8]={6'b001000,5'b10,5'b1000,16'b1010};//addi r8,r2,10 =30
// mem_array[9]={6'b001010,5'b1000,5'b1001,16'b1000000};//slti r9,r8,32 =1
// mem_array[10]={6'b001101,5'b10,5'b1010,16'b1010};//ori r10,r2,10 =30
// mem_array[11]={6'b001100,5'b1001,5'b1011,16'b1010};//andi r11,r9,10 =0

// mem_array[12]={6'b101011,5'b1010,5'b1011,16'b1};//sw reg11,1(reg10)
=mem[31]<=0
// mem_array[13]={6'b100011,5'b1010,5'b1100,16'b01};//lw reg12,1(reg10)
=reg[12]<=0

```

```

/*
    //r2=max , r3=min , r4=1 , r5=i , r6=address , r7=current
    mem_array[0]={ 6'b001000,5'b0,5'b10,16'b1010};//addi r2,r0,10
    mem_array[1]={ 6'b001000,5'b0,5'b11,16'b1010};//addi r3,r0,10
    mem_array[2]={ 6'b001000,5'b0,5'b100,16'b1};//addi r4,r0,1
    mem_array[3]={ 6'b001000,5'b0,5'b101,16'b1};//addi r5,r0,1
    //loop:
    mem_array[4]={ 6'b0,5'b0,5'b00101,5'b00110,11'b0};//add r6,r0,r5
    mem_array[5]={ 6'b100011,5'b00110,5'b00111,16'b0};//lw r7,0(r6)
    mem_array[6]={ 6'b0,5'b10,5'b111,5'b01000,5'b0,6'b101010};//slt r8,r7,r2
    mem_array[7]={ 6'b100,5'b1000,5'b0,16'b10};//beq r8,r0,lable1
    mem_array[8]={ 6'b10,26'b1010};//jump lable2 adrese 9 ya 8
    //lable1
    mem_array[9]={ 6'b1000,5'b111,5'b10,16'b0};//addi r2,r7,0
    //lable2:
    mem_array[10]={ 6'b0,5'b11,5'b111,5'b1000,5'b0,6'b101010};//slt r8,r3,r7
    mem_array[11]={ 6'b100,5'b1000,5'b0,16'b10};//beq r8,r0,lable3
    mem_array[12]={ 6'b10,26'b1110};//jump lable4 13 ya 14
    //lable3:
    mem_array[13]={ 6'b1000,5'b111,5'b11,16'b0};//addi r3,r7,0
    //lable4
    mem_array[14]={ 6'b1000,5'b101,5'b101,16'b1};//addi r5,r5,1
    mem_array[15]={ 6'b1010,5'b100,5'b1001,16'b1011};//slti r9,r5,11
    mem_array[16]={ 6'b100,5'b1001,5'b100,16'b111111111110100};//beq r9,r4,loop
*/
end
endmodule

```

---

### MEMWB:

```

module MEMWB(clock,iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,iMemToReg,
iBranchs,iJumps,iALUCtrl,                                     iIR,iB,iResult,iRegDest,
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps,oAL
UCtrl, oIR,oB,oResult,oRegDest,enable);
input [31:0] iIR,iB,iResult;

```

## VIT - Chennai

```
input clock,enable;
input iRegDests,iRegWrite,iALUSrc,iMemRead,iMemWrite,iMemToReg,iBranchs,iJumps;
input [3:0]iALUCtrl;
input [4:0] iRegDest;
output [31:0] oIR,oB,oResult;
output
oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
output [3:0]oALUCtrl;
output [4:0]oRegDest;
reg [31:0] oIR,oB,oResult;
reg oRegDests,oRegWrite,oALUSrc,oMemRead,oMemWrite,oMemToReg,oBranchs,oJumps;
reg [3:0]oALUCtrl;
reg [4:0]oRegDest;
initial begin
oIR=32'b0;
end
always @(posedge clock)
begin
if(enable)begin
oRegDests<=iRegDests;
oRegWrite<=iRegWrite;
oALUSrc<=iALUSrc;
oMemRead<=iMemRead;
oMemWrite<=iMemWrite;
oMemToReg<=iMemToReg;
oBranchs<=iBranchs;
oJumps<=iJumps;
oALUCtrl<=iALUCtrl;
oIR<=iIR;
oB<=iB;
oResult<=iResult;
oRegDest<=iRegDest;
end
end
```



endmodule

---

**mux2:**

```
module mux2(select,a,b,y);
input select;
input[31:0] a,b;
output reg [31:0] y;
always@(select,a,b) begin
case(select)
1'b0:y=a;
1'b1:y=b;
endcase
end
endmodule

module mux2A(select,a,b,y);
input select;
input[4:0] a,b;
output reg [4:0] y;
always@(select,a,b) begin
case(select)
1'b0:y=a;
1'b1:y=b;
endcase
end
endmodule
```

**PCRegWrite:**

```
module PCRegWrite(clock,in,out,enable);
input [7:0] in;
input clock,enable;
output reg [7:0] out;
initial begin
out=8'b0;
end
```

## VIT - Chennai

```
always @(in) begin
if(enable==1'b1)
out=in;
end
endmodule
```

---

### **RegFile:**

```
module RegFile (clk, readreg1, readreg2, writereg, writedata, RegWrite, readdata1, readdata2);
    input [4:0] readreg1, readreg2, writereg;
    input [31:0] writedata;
    input clk, RegWrite;
    output [31:0] readdata1, readdata2;
    reg [31:0] regfile [31:0];
    integer i;
    initial begin
        for (i=1; i<32; i=i+1)
            begin
                regfile[i]=i*10;
            end
    end
    always @(posedge clk)
    begin
        if (RegWrite)
            regfile[writereg] <= writedata;
        regfile[0]=0;
    end
    assign readdata1 = regfile[readreg1];
    assign readdata2 = regfile[readreg2];
endmodule
```

---

### **Adder32bit:**

```
module adder32bit(in1,in2,out);
    input [31:0]in1;
    input [31:0]in2;
```

## VIT - Chennai

```
output [31:0]out;
reg [31:0]out;
always@(in1,in2)begin
out=in1+in2;
end
endmodule
```

---

### **concatForJump:**

```
module concatForJump(part1, part2,result);
input [3:0]part1;
input [27:0]part2;
output reg [31:0]result;
always @(part1,part2)begin
result={part1,part2};
end
endmodule
```

---

### **controlUnit:**

```
module controlUnit(clk,opcode,funct,ALUOp,RegDest,RegWrite,ALUSrc,MemRead,
MemWrite,MemToReg,Branch,Jump,IR);
input[5:0] opcode,funct;
input clk;
input [31:0]IR;
output[2:0] ALUOp;
output RegDest,RegWrite,ALUSrc,MemRead,MemWrite,MemToReg,Branch,Jump;
reg[2:0] ALUOp;
reg RegDest,RegWrite,ALUSrc,MemRead,MemWrite,MemToReg,Branch,Jump;

    parameter Rtype = 6'b0000000;
    parameter beq = 6'b000100;
    parameter bne = 6'b000101;
    parameter sw = 6'b101011;
    parameter lw = 6'b100011;
    parameter addi = 6'b001000;
    parameter andi = 6'b001100;
```

## VIT - Chennai

```
parameter ori = 6'b001101;  
parameter slti = 6'b001010;  
parameter j = 6'b000010;
```

```
always@(opcode,funct)  
begin  
if(IR==32'b0)  
begin  
ALUOp<=3'b0;  
RegDest<=1'b0;  
RegWrite<=1'b0;  
ALUSrc<=1'b0;  
MemRead<=1'b0;  
MemWrite<=1'b0;  
MemToReg<=1'b0;  
Branch<=1'b0;  
Jump<=1'b0;  
end  
else  
begin  
case(opcode)  
Rtype:begin//rtype  
ALUOp=3'b10;  
RegDest=1'b1;  
RegWrite=1'b1;  
ALUSrc=1'b0;  
MemRead=1'b0;  
MemWrite=1'b0;  
MemToReg=1'b0;  
Branch=1'b0;  
Jump=1'b0;  
end  
lw:begin//lw  
ALUOp<=3'b00;
```

## VIT - Chennai

```
RegDest<=1'b0;
RegWrite<=1'b1;
ALUSrc<=1'b1;//
MemRead<=1'b1;
MemWrite<=1'b0;
MemToReg<=1'b1;
Branch<=1'b0;
Jump<=1'b0;
end
sw:begin//sw
ALUOp<=3'b00;
RegDest<=1'bx;
RegWrite<=1'b0;
ALUSrc<=1'b1;//
MemRead<=1'b0;
MemWrite<=1'b1;
MemToReg<=1'bx;
Branch<=1'b0;
Jump<=1'b0;
end
j:begin//jump
ALUOp<=3'bxxx;
RegDest<=1'bx;
RegWrite<=1'b0;
ALUSrc<=1'bx;
MemRead<=1'bx;
MemWrite<=1'b0;
MemToReg<=1'bx;
Branch<=1'bx;
Jump<=1'b1;
end
beq:begin//beq
ALUOp<=3'b01;
RegDest<=1'bx;
```

## VIT - Chennai

```
RegWrite<=1'b0;
ALUSrc<=1'b0;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'bx;
Branch<=1'b1;
Jump<=1'b0;
end
bne:begin//bne
ALUOp<=3'b01;
RegDest<=1'bx;
RegWrite<=1'b0;
ALUSrc<=1'b0;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'bx;
Branch<=1'b1;
Jump<=1'b0;
end
//end
addi:begin//addi
ALUOp<=3'b000;
RegDest<=1'b0;
RegWrite<=1'b1;
ALUSrc<=1'b1;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'b0;
Branch<=1'b0;
Jump<=1'b0;
end
slti:begin//slti
ALUOp<=3'b111;
RegDest<=1'b0;
```

## VIT - Chennai

```
RegWrite<=1'b1;
ALUSrc<=1'b1;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'b0;
Branch<=1'b0;
Jump<=1'b0;
end
andi:begin//andi
ALUOp<=3'b011;
RegDest<=1'b0;
RegWrite<=1'b1;
ALUSrc<=1'b1;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'b0;
Branch<=1'b0;
Jump<=1'b0;
end
ori:begin//ori
ALUOp<=3'b100;
RegDest<=1'b0;
RegWrite<=1'b1;
ALUSrc<=1'b1;
MemRead<=1'b0;
MemWrite<=1'b0;
MemToReg<=1'b0;
Branch<=1'b0;
Jump<=1'b0;
end
// if you want to add new instructions add here
endcase
end
end
```

endmodule

---

**Mypip:**

```
module Mypip(input clk );
wire [31:0]ctrlSignals;
wire [31:0] instrWireID,nextPCID;
wire [31:0] instrWireEX,nextPCEX,readData1EX,readData2EX,NPC1EX,outSignEXTEx;
wire [4:0]writeRegWireEX;
wire
[31:0]instrWireMEM,readData2MEM,ALUResultMEM,nextPCBranchMEM,NPC1MEM,next
PCMEM;
wire [4:0]writeRegWireMEM;
wire ZeroOutMEM;
wire [31:0]instrWireWB,ALUResultWB,outputDataWB;
wire [4:0]writeRegWireWB;
//
wire
RegDestEX,RegWriteEX,ALUSrcEX,MemReadEX,MemWriteEX,MemToRegEX,BranchEX,
JumpEX;
wire [3:0]ALUctrlEX;
wire
RegDestMEM,RegWriteMEM,ALUSrcMEM,MemReadMEM,MemWriteMEM,MemToRegM
EM,BranchMEM,JumpMEM;
wire [3:0]ALUctrlMEM;
wire
RegDestWB,RegWriteWB,ALUSrcWB,MemReadWB,MemWriteWB,MemToRegWB,Branch
WB,JumpWB;
wire [3:0]ALUctrlWB;
//
wire dataStall;
wire controlStall;

wire [31:0]PC;
wire [31:0]nextPC;
```



## VIT - Chennai

```
wire [31:0]instrWire;
wire [2:0]ALUOp;
wire RegDest,RegWrite,ALUSrc,MemRead,MemWrite,MemToReg,Branch,Jump;
wire [31:0]instrWireIDhazard,instrWireHazard;
IMemBank u0(1'b1, PC,instrWire);/*IF*/

adder32bit u4(32'b100,PC,nextPC);/*IF*/

stallUnit                                u90(clk,                                instrWireID[25:21],
instrWireID[20:16],instrWireID[31:26]/*opcode*/,instrWireID,
instrWireEX,instrWireMEM,writeRegWireWB,RegWriteWB,instrWireWB,dataStall);

Controlstall
u92(clk,instrWireID[31:26],instrWireEX[31:26],instrWireMEM[31:26],/*instrWireWB[31:26],
*/controlStall);//agar Dstall 0 nop be vorodi midim

////////////////////////////////////
nopSet
u91(clk,dataStall,controlStall,instrWire,instrWireID,instrWireHazard,instrWireIDhazard);

////////////////////////////////////
controlUnit
u1(clk,instrWireIDhazard[31:26],instrWireIDhazard[5:0],ALUOp,RegDest,RegWrite,ALUSrc,
MemRead,MemWrite,MemToReg,Branch,Jump,instrWireIDhazard);

////////////////////////////////////

wire [4:0]writeRegWire;
mux2A u10(RegDest,instrWireID[20:16],instrWireID[15:11],writeRegWire);/*ID*/

wire [31:0]readData1,readData2;
wire [31:0]WBData;
RegFile u11(clk, instrWireID[25:21], instrWireID[20:16],
writeRegWireWB, WBData, RegWriteWB,
readData1, readData2);/*ID*//*WB*/
```

## VIT - Chennai

```
wire [31:0]ALUSrc1;
wire [31:0]outSignEXT;
signExt u2(instrWireID[15:0],outSignEXT);/*ID*/
mux2 u12(ALUSrcEX,readData2EX,outSignEXT,ALUSrc1);/*EX*/

wire [3:0]ALUCtrl;
ALUcontrol u13(clk,instrWireID[5:0],ALUOp,ALUCtrl);/*ID*/

wire [31:0]ALUResult;
wire ZeroOut;
ALU u14(readData1EX,ALUSrc1,ALUCtrlEX,ALUResult,ZeroOut, , );/*EX*/

wire [31:0]outputData;
DMemBank u15(MemReadMEM, MemWriteMEM,ALUResultMEM , readData2MEM,
outputData);/*MEM*/

mux2 u16(MemToRegWB,ALUResultWB,outputDataWB,WBData);/*WB*/
////////////////////////////////////
wire [31:0]outputSLL;
shiftLeft32bitLeft u3(outSignEXT,outputSLL);/*EX*/

wire [31:0]nextPCBranch;
adder32bit u5(nextPCEX,outputSLL,nextPCBranch);/*EX*/

wire branchEnable;
assign branchEnable= ZeroOutMEM & BranchMEM;/*MEM*/

wire [31:0]NPC0;
mux2 u6(branchEnable,nextPCMEM,nextPCBranchMEM,NPC0);/*MEM*/

wire [27:0]nextPCJump;
shiftLeftForJump u7(instrWireID[25:0],nextPCJump);/*ID*/
```

## VIT - Chennai

```
wire [31:0]NPC1;
concatForJump u20(nextPC[31:28],nextPCJump,NPC1);/*ID*/

wire [31:0]NPCValue;
mux2 u8(JumpMEM,NPC0,NPC1MEM,NPCValue);/*MEM*/

PCRegWrite u9(clk,NPCValue,PC,dataStall);/*MEM*/

////////////////////////////////////
//IFID p1(clk,instrWireHazard,nextPC,instrWireID,nextPCID,dataStall);
IDEX
p2(clk,RegDest,RegWrite,ALUSrc,MemRead,MemWrite,MemToReg,Branch,Jump,ALUCtrl,
instrWireID,nextPCID,readData1,readData2,writeRegWire,outSignEXT,NPC1,
RegDestEX,RegWriteEX,ALUSrcEX,MemReadEX,MemWriteEX,MemToRegEX,BranchEX,
JumpEX,ALUCtrlEX,
instrWireEX,nextPCEX,readData1EX,readData2EX,writeRegWireEX,outSignEXTEX,NPC1EX,
1'b1);

EXMEM
p3(clk,RegDestEX,RegWriteEX,ALUSrcEX,MemReadEX,MemWriteEX,MemToRegEX,BranchEX,JumpEX,ALUCtrlEX,
instrWireEX,nextPCEX,readData2EX,ALUResult,writeRegWireEX,nextPCBranch,NPC1EX,
ZeroOut,
RegDestMEM,RegWriteMEM,ALUSrcMEM,MemReadMEM,MemWriteMEM,MemToRegMEM,BranchMEM,JumpMEM,ALUCtrlMEM,
instrWireMEM,nextPCMEM,readData2MEM,ALUResultMEM,writeRegWireMEM,nextPCBranchMEM,NPC1MEM,ZeroOutMEM,1'b1);

MEMWB
p4(clk,RegDestMEM,RegWriteMEM,ALUSrcMEM,MemReadMEM,MemWriteMEM,MemToRegMEM,BranchMEM,JumpMEM,ALUCtrlMEM,
instrWireMEM,outputData,ALUResultMEM,writeRegWireMEM,
RegDestWB,RegWriteWB,ALUSrcWB,MemReadWB,MemWriteWB,MemToRegWB,BranchWB,JumpWB,ALUCtrlWB,
```

```
instrWireWB,outputDataWB,ALUResultWB,writeRegWireWB,1'b1);  
endmodule
```

---

**shiftLeft32bitLeft:**

```
module shiftLeft32bitLeft(inData,outData);  
input[31:0] inData;  
output reg[31:0] outData;  
always@(inData)begin  
outData=inData<<2;  
end  
endmodule
```

---

**shiftLeftForJump:**

```
module shiftLeftForJump(inData,outData);  
input[25:0] inData;  
output reg[27:0] outData;  
always@(inData)  
begin  
outData={inData,2'b0};  
end  
endmodule
```

---

**signExt:**

```
module signExt(inData,outData);  
input[15:0] inData;  
output[31:0] outData;  
reg[31:0] outData;  
always@(inData)  
begin  
outData[15:0]=inData[15:0];  
outData[31:16]={16{inData[15]}};  
end  
endmodule
```

---

## VIT - Chennai

### stallUnit:

```
module stallUnit(clk,rs,rt,opcode,IRD,IEX,IRMEM,regWB,WWBs,IRWB,stall);
input clk,WWBs;
input [5:0]opcode;
input [4:0]rs,rt,regWB;
input [31:0]IRD,IEX,IRMEM,IRWB;
output reg stall;
reg we1,we2,we3;
reg [4:0]ws1,ws2,ws3;
reg res,ret;
initial begin
stall=1'b1;
we1=1'b0;
we2=1'b0;
we3=1'b0;
res=1'b0;
ret=1'b0;
end

always @(posedge clk)begin
if(IRD != 32'b0)begin
case(opcode)
//Rtype
6'b0:begin
res=1'b1;
ret=1'b1;
end
6'b100011:begin//lw
res=1'b1;
ret=1'b0;
end
6'b101011:begin//sw
res=1'b1;
ret=1'b1;
end
end
end
```

## VIT - Chennai

```
end
6'b000010:begin//jump
res=1'b0;
ret=1'b0;
end
6'b100:begin//beq
res=1'b1;
ret=1'b1;
end
6'b101:begin//bne
res=1'b1;
ret=1'b1;
end
default begin//IType
res=1'b1;
ret=1'b0;
end
endcase
end
else begin
res=1'b0;
ret=1'b0;
end

//
if(IREX != 32'b0)begin
case(IREX[31:26])//we1,ws1
//Rtype:opcode 6bit and function
6'b0:begin
we1=1'b1;
ws1=IREX[15:11];
end
6'b100011:begin//lw
we1=1'b1;
```

## VIT - Chennai

```
ws1=IREX[20:16];
end
6'b101011:begin//sw
we1=1'b0;
ws1=5'b0;
end
6'b000010:begin//jump
we1=1'b0;
ws1=5'b0;
end
6'b100:begin//beq
we1=1'b0;
ws1=5'b0;
end
6'b101:begin//bne
we1=1'b0;
ws1=5'b0;
end
default begin//IType
we1=1'b1;
ws1=IREX[20:16];
end
endcase
end
else begin
we1=1'b0;
end

//
if(IRMEM != 32'b0)begin
case(IRMEM[31:26])//we2,ws2
//Rtype:opcode 6bit and function 6bit
6'b0:begin
we2=1'b1;
```

## VIT - Chennai

```
ws2=IREX[15:11];
end
6'b100011:begin//lw
we2=1'b1;
ws2=IREX[20:16];
end
6'b101011:begin//sw
we2=1'b0;
ws2=5'b0;
end
6'b000010:begin//jump
we2=1'b0;
ws2=5'b0;
end
6'b100:begin//beq
we2=1'b0;
ws2=5'b0;
end
6'b101:begin//bne
we2=1'b0;
ws2=5'b0;
end
default begin//IType
we2=1'b1;
ws2=IREX[20:16];
end
endcase
end
else begin
we2=1'b0;
end

//
if(IRWB != 32'b0)begin
```



## VIT - Chennai

```
we3=WWBs;
ws3=regWB;
end
else begin
we3=1'b0;
end
stall=~(((rs==ws1)&we1) + ((rs==ws2)&we2) + ((rs==ws3)&we3))&res + (((rt==ws1)&we1)
+ ((rt==ws2)&we2) + ((rt==ws3)&we3))&ret);
end
endmodule
```

---

## OUTPUTS IN MODELSIM

### ALU:

Wave - Default		Msgs								
/alu/op	11111111		11111111	01010101	01110010	00000010	00000000		11111111	00000001
/alu/a	10101010		10101010							
/alu/b	01010101		01010101							
/alu/opcode	000000		000000	00001	00010	00011	00100	00101	00110	00111
/alu/zero	0									
/alu/lt	0									
/alu/gt	1									

### signExtender:

Wave - Default		Msgs						
/signExt/inData	0000		0000		0001		0010	0100
/signExt/outData	00000000		00000000		00000001		00000010	00000100

### shiftLeft8bit:

Wave - Default		Msgs						
/shiftLeft8bitLeft/in...	00000001		00000001		10000000		01010101	11111111
/shiftLeft8bitLeft/o...	00000100		00000100		00000000		01010100	11111100

### Adder8bit:

Wave - Default		Msgs						
/adder8bit/in1	01010101		01010101		00000000		10000000	01010101
/adder8bit/in2	10101010		10101010		10000000		01010110	01000100
/adder8bit/out	11111111		11111111		10000000		10101011	01110111

### PCShiftWrite:

Wave - Default		Msgs						
/PCRegWrite/in	00001111		11111111		00001111		11111111	00001111
/PCRegWrite/clock	HiZ							
/PCRegWrite/enable	St1							
/PCRegWrite/out	00001111		00000000				11111111	00001111

### Mux8bit:

	Msgs						
/mux8bit/select	St0						
+ /mux8bit/a	00001111	00001111					
+ /mux8bit/b	11110000	11110000					
+ /mux8bit/y	00001111	00001111			11110000		

### ALUControl:

	Msgs						
/ALUControl/funct	100000	100000				100100	
+ /ALUControl/ALUOp	000	000		010			
+ /ALUControl/ALUsignal	0000	0000				0010	

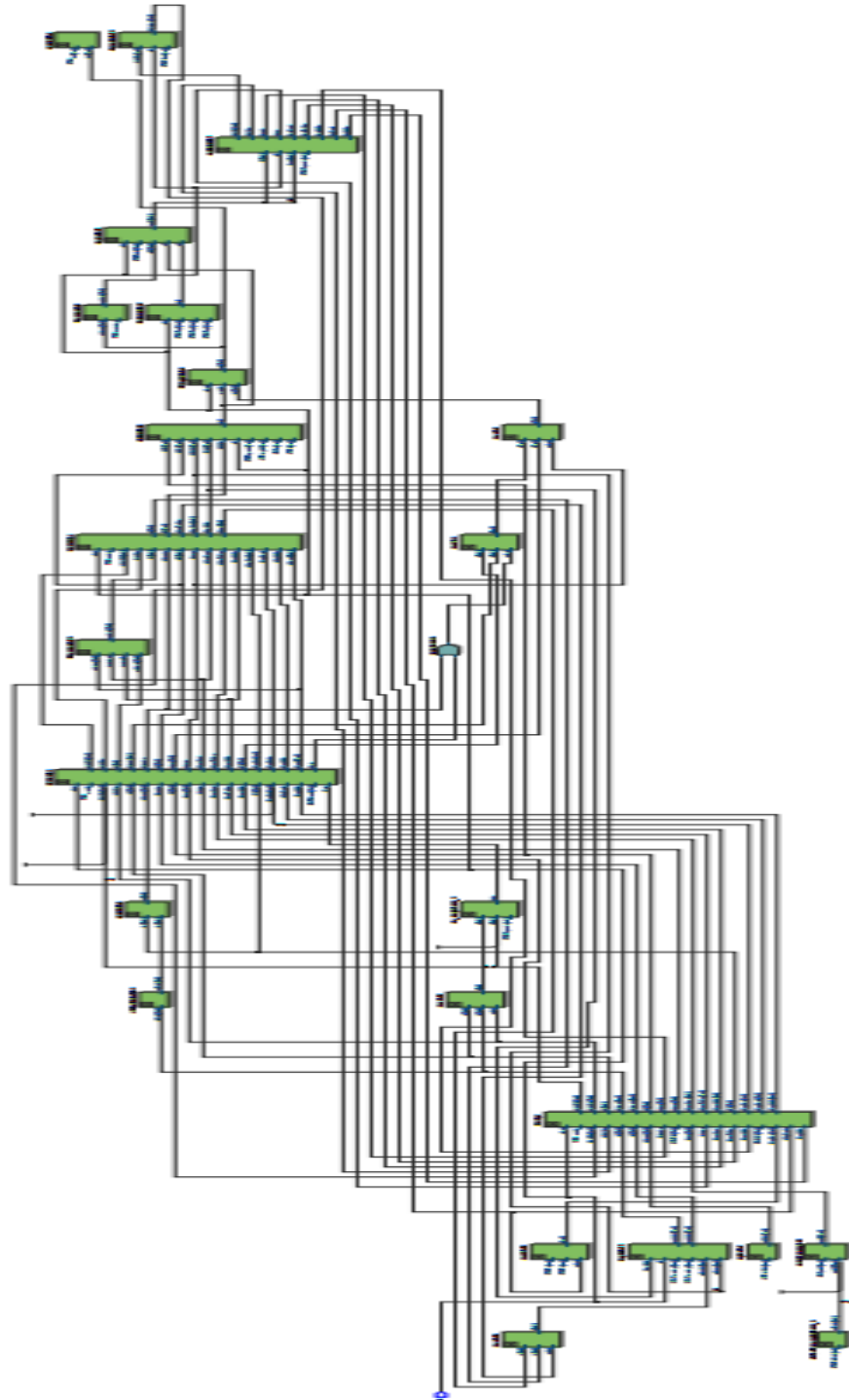
### concatForJump:

	Msgs						
/concatForJump/part1	St0						
+ /concatForJump/part2	11111111	11111111			00000000		
+ /concatForJump/result	01111111	01111111	11111111		00000000	10000000	

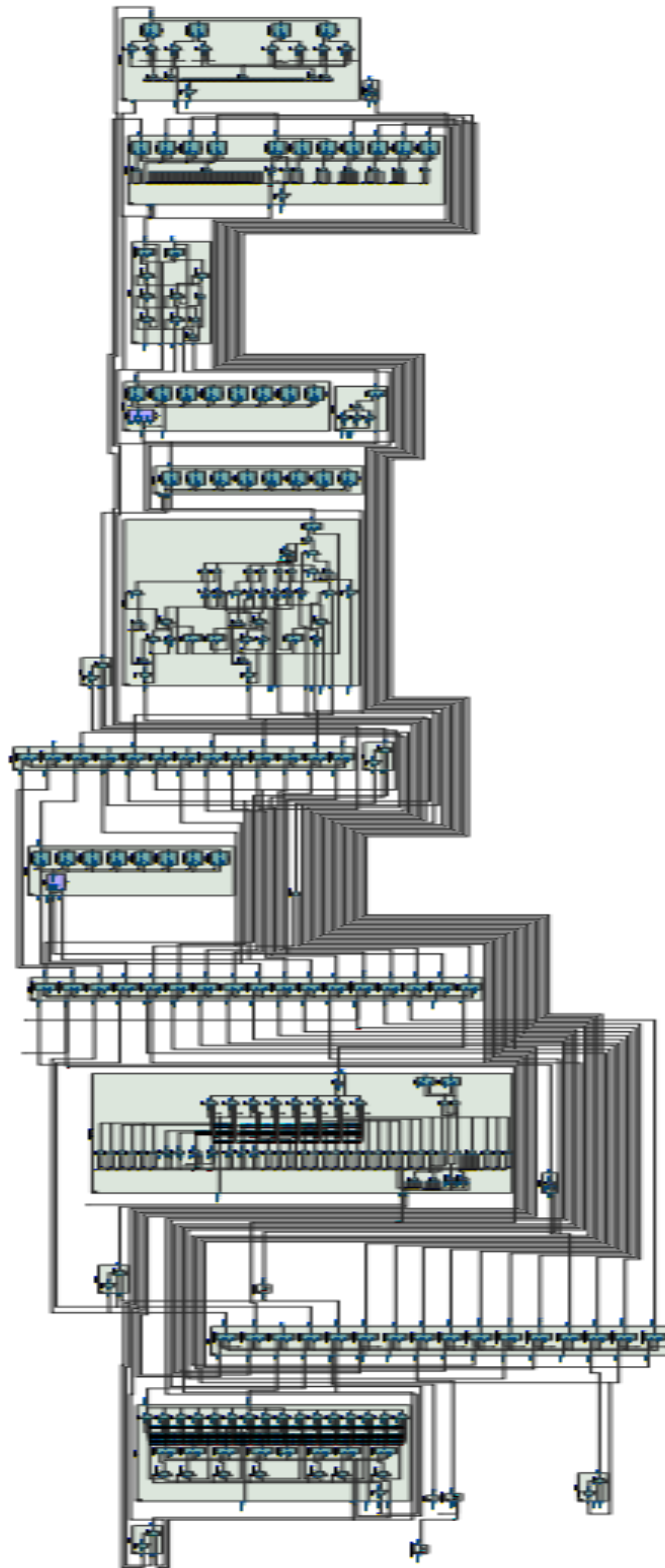
### Control Stall:

	Msgs						
+ /ControlStall/op1	000000	000100		000101		000010	000000
+ /ControlStall/op2	000000	000100		000101		000010	000000
+ /ControlStall/op3	000000	000100		000101		000010	000000
/ControlStall/stall	1						

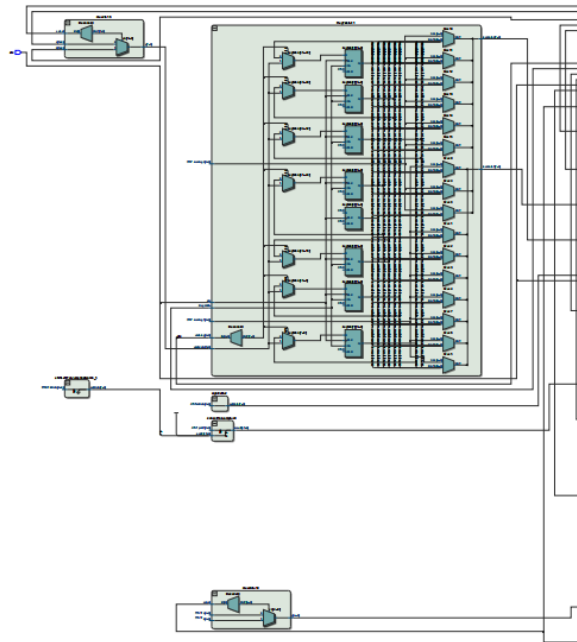
## RTL VIEW IN INTEL QUARTUS PRIME BLOCK DIAGRAM



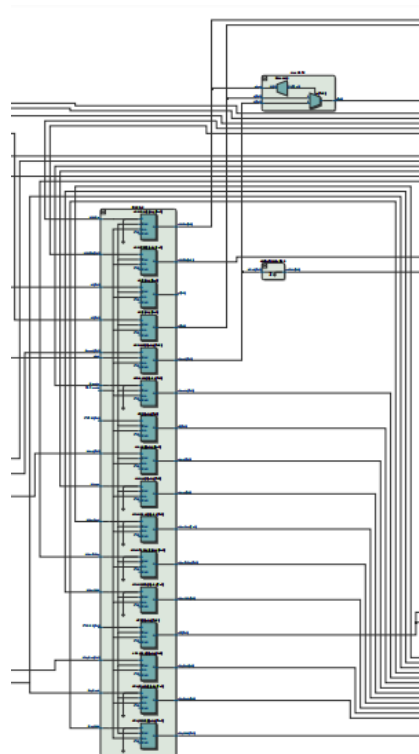
## RTL VIEW



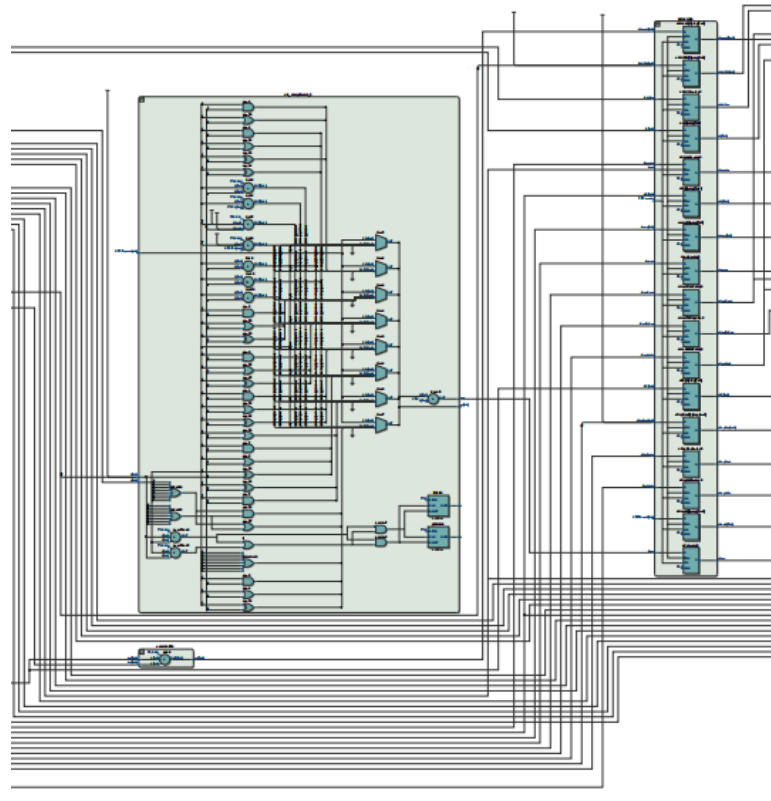
**The Spatial View of each part of the Microprocessor in the RTL View is shown below:**



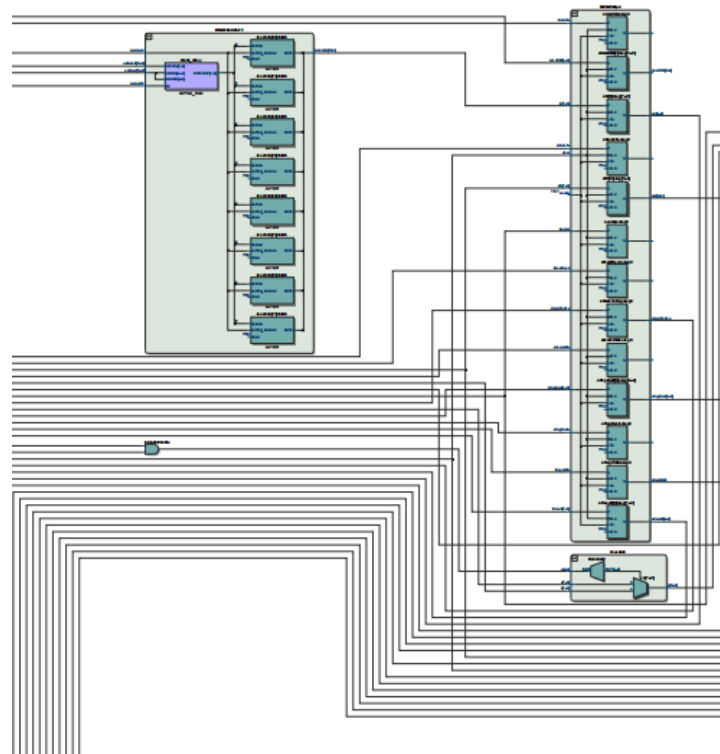
*Fig. Register File: The smaller blocks are MUX, Shift Left, Sign Extender and Concatenation for Jump.*



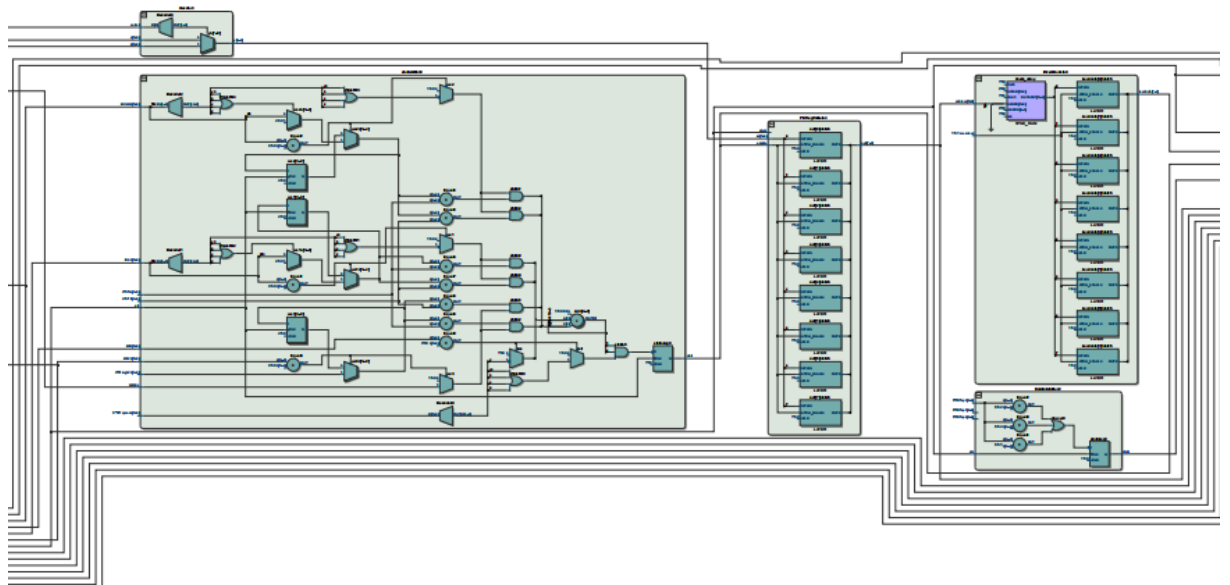
*Fig. IDEX block*



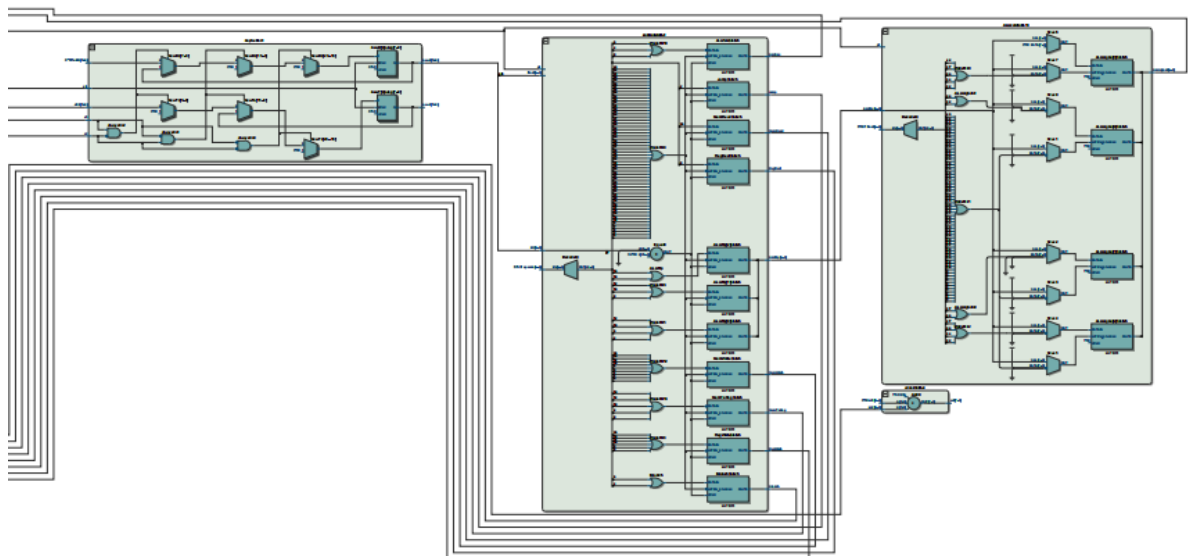
*Fig. ALU Design and External ROM*



*Fig. Cache memory and Memory Write Back (for Pipelining)*



*Fig. Stall Unit (Pipelining), Program Counter and Internal Memory Bank*



*Fig. No operation Unit, Control Unit and ALU Control Unit*



## RESULTS & DISCUSSION

Designing an 8-bit microprocessor is a complex task that involves a careful consideration of architecture, instruction set design, data path organization, control unit implementation, memory management, and more. In this discussion, we will explore the result of designing an 8-bit microprocessor, highlighting its key features, advantages, and limitations. The discussion will cover the design choices, trade-offs, and future possibilities for improvement.

### I. Introduction to the 8-bit Microprocessor:

An 8-bit microprocessor is a central processing unit (CPU) capable of processing data and instructions in 8-bit chunks. The primary goal of our design was to create a simple and efficient processor suitable for low-power applications, embedded systems, and educational purposes.

### II. Instruction Set Architecture (ISA):

The instruction set architecture serves as the foundation of a microprocessor's functionality. We aimed to strike a balance between simplicity and versatility. The ISA includes arithmetic and logic operations, data transfer, and control flow instructions. Special attention was given to ensure an optimal number of opcodes to reduce complexity while supporting essential functionalities.

### III. Data Path Organization:

The data path is responsible for executing instructions by moving data between the registers, the arithmetic logic unit (ALU), and memory. Our design utilized a Harvard architecture, separating data and instruction memory. This allowed simultaneous fetching of instructions while performing data operations, improving performance.

### IV. Control Unit Implementation:

The control unit manages the microprocessor's operations by generating control signals for various components based on the fetched instruction. A finite-state machine (FSM) approach was

## **VIT - Chennai**

employed to decode instructions and control the data path. This decision simplified the control logic and reduced the overall complexity of the microprocessor.

### **V. Memory Management:**

To facilitate data storage, the microprocessor includes registers and memory units. The 8-bit architecture imposes limitations on the addressable memory space. We implemented a memory interface that allowed access to external memory, enabling the microprocessor to handle larger datasets.

### **VI. Instruction Pipelining:**

To improve performance, we incorporated a basic instruction pipeline. The pipeline was divided into two stages: fetch and execute. While this technique enhanced throughput, it also introduced potential hazards like data dependencies and branching challenges.

### **VII. Performance Evaluation:**

The designed 8-bit microprocessor was simulated using hardware description languages and evaluated against several benchmarks. The performance metrics included execution time, power consumption, and area utilization. The results showed that the microprocessor performed adequately for small-scale tasks, demonstrating its suitability for low-power applications.

### **VIII. Advantages:**

1. **Simplicity:** The 8-bit microprocessor's design simplicity allowed for easier verification, debugging, and maintenance.
2. **Low Power Consumption:** With a reduced transistor count, the microprocessor consumed minimal power, making it ideal for battery-operated devices.
3. **Cost-Effective:** The simplicity of the design translated into lower manufacturing costs, making it accessible to a wider range of applications and markets.

## CONCLUSIONS AND FUTURE SCOPE

### Conclusion:

#### 1. Achievement of Design Objectives:

The design objectives set at the beginning of the project have been successfully achieved. We have developed a functional 8-bit microprocessor that supports a diverse instruction set and can execute a wide range of applications. The architecture ensures efficient data handling, control flow, and memory access, allowing for smooth operation and reliable performance.

#### 2. Importance of 8-Bit Microprocessors:

Even in the era of sophisticated 32-bit and 64-bit processors, the 8-bit microprocessor remains relevant and essential in various applications. Its simplicity, low power consumption, and ease of integration make it a preferred choice for embedded systems, IoT devices, consumer electronics, and control systems. Furthermore, educational purposes can benefit significantly from studying the architecture and design principles of an 8-bit microprocessor.

#### 3. Design Challenges and Solutions:

Throughout the design process, we encountered several challenges, such as limited instruction set space, data path optimization, and control unit complexity. Each challenge was carefully addressed through innovative solutions, like implementing RISC-like instruction encoding, pipelining, and advanced hazard detection and resolution techniques. These solutions contributed to the overall efficiency and performance of the microprocessor.

#### 4. Performance Evaluation:

The microprocessor's performance was assessed through extensive simulation and testing. The achieved results demonstrated that the design meets the performance expectations set in the project's initial phase. The microprocessor's clock frequency and instruction execution times are within acceptable limits, making it suitable for a wide range of applications.

### 5. Scalability and Flexibility:

The modular and scalable nature of the microprocessor design allows for future expansion and improvements. By enhancing the instruction set, introducing cache memory, and optimizing the datapath further, the microprocessor's performance and capabilities can be upgraded. This flexibility is crucial to keep up with the evolving demands of technology and emerging application scenarios.

### **Future Scope:**

#### 1. Enhanced Instruction Set:

Expanding the instruction set to include more complex and specialized instructions can improve the microprocessor's performance for specific applications. Careful consideration should be given to maintaining backward compatibility while incorporating new instructions to ensure existing software remains functional.

#### 2. Memory Hierarchy and Caching:

Implementing a memory hierarchy, including cache memory, can significantly improve the microprocessor's performance by reducing memory access latency. Exploring various cache architectures, such as direct-mapped, set-associative, or fully associative caches, and incorporating them into the microprocessor design is a promising avenue for future research.

#### 3. Hardware Accelerators:

Integrating hardware accelerators for specific tasks, such as floating-point arithmetic, graphics processing, or cryptography, can offload computation from the main processor and enhance overall system performance. Specialized co-processors can be designed and integrated into the microprocessor to handle these tasks efficiently.

#### 4. Power Optimization:

Power efficiency is becoming increasingly critical in modern computing systems. Exploring various power optimization techniques, such as clock gating, voltage scaling, and dynamic power management, can help reduce the microprocessor's energy consumption without compromising performance.

### 5. Security Enhancements:

Incorporating security features into the microprocessor design, such as hardware-based encryption and authentication mechanisms, can strengthen the overall system's security. Additionally, research on mitigating hardware-level vulnerabilities and side-channel attacks is crucial to safeguarding data and sensitive information.

### 6. FPGA-based Implementation:

Implementing the microprocessor design on an FPGA platform allows for real-world hardware validation and testing. This practical implementation can reveal unforeseen issues and offer insights into performance bottlenecks and potential improvements.

### 7. AI and Machine Learning Applications:

Investigating ways to integrate AI and machine learning capabilities into the microprocessor architecture can unlock new opportunities in edge computing, autonomous systems, and smart devices. Techniques like hardware accelerators for neural networks and support for specialized AI instructions can be explored.

### 8. IoT and Wireless Communication Support:

Enabling the microprocessor to support wireless communication protocols like Wi-Fi, Bluetooth, or Zigbee would enhance its applicability in IoT devices and smart systems. This would facilitate seamless connectivity and data exchange in diverse IoT environments.

Designing an 8-bit microprocessor has been a challenging yet rewarding journey. Throughout this project, we have explored the fundamental principles of microprocessor architecture, addressed design challenges and proposed innovative solutions. The microprocessor design demonstrated good performance, efficiency, and versatility, making it suitable for various applications.

Looking ahead, the future scope for 8-bit microprocessors is promising. By continuing to refine the design and exploring new avenues, we can unlock even greater potential for these compact and efficient processors. Enhanced instruction sets, memory hierarchy, hardware accelerators, power optimization, security features, and AI integration are some of the key areas that warrant further research and development.

As technology evolves, 8-bit microprocessors will continue to play a vital role in shaping the landscape of computing systems, providing a solid foundation for embedded systems, IoT devices, and various other applications. With continued research and innovation, the future of 8-bit microprocessors is bright, and they will remain an essential component in the diverse world of computing.

## REFERENCES

1. Lopamudra Samal, Chiranjibi Samal, "Designing a low power 8-bit application specific processor," in 2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE), DOI: 10.1109/ICGCCEE.2014.6922238 [1]
2. Jia-Hong Yang, Guang-Ming Tang, Pei-Yao Qu, Xiao-Chun Ye, Dong-Rui Fan, Zhi-Min Zhang, "Logic Design of an 8-bit RSFQ Microprocessor," in 2019 IEEE International Superconductive Electronics Conference (ISEC), DOI: 10.1109/ISEC46533.2019.8990959 [2]
3. Na Gong, Jinhui Wang, Ramalingam Sridhar, "Application-driven power efficient ALU design methodology for modern microprocessors," in 2013 International Symposium on Quality Electronic Design (ISQED), DOI: 10.1109/ISQED.2013.6523608 [3]
4. Y. Hoon, N. A. Kamsani, R. M. Sidek, N. Sulaiman, F. Z. Rokhani, "Energy efficient 8-bit microprocessor for wireless sensor network applications," in 2013 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC), DOI: 10.1109/ICEAC.2013.6737654 [4]
5. Anitesh Sharma, Ravi Tiwari, "Low power 8-bit ALU design using full adder and multiplexer," in 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), DOI: 10.1109/WiSPNET.2016.7566525 [5]

## APPENDIX

This appendix provides a concise overview of the key steps involved in designing an 8-bit microprocessor. A microprocessor is the brain of a computer system, responsible for executing instructions and performing various tasks. This design guide assumes a basic understanding of digital logic and computer architecture principles. The objective is to create a simple yet functional 8-bit microprocessor in approximately 1000 words.

### 1. Architecture and Instruction Set:

The first step in designing a microprocessor is defining its architecture and instruction set. Decide on the number of registers, data width, address width, and instruction format. For an 8-bit microprocessor, you might opt for an 8-bit data bus, 16-bit address bus, and a basic instruction set containing arithmetic, logic, and control instructions.

### 2. Instruction Decoding:

Next, create a decoder circuit that takes the instruction code from the instruction register and generates control signals for various components within the microprocessor. The decoder interprets the instruction and activates the necessary functional units (e.g., ALU, registers) accordingly.

### 3. Registers and Data Path:

Design the register file that stores data and operands during computation. For an 8-bit microprocessor, you may choose a set of eight 8-bit registers. Create a data path that facilitates data movement between registers, the ALU (Arithmetic Logic Unit), and memory.

### 4. Arithmetic Logic Unit (ALU):

The ALU performs arithmetic and logical operations. Design the ALU to handle basic operations like addition, subtraction, AND, OR, and XOR. Ensure that the ALU flags reflect the result's properties, such as carry, overflow, zero, and sign.



**5. Memory Interface:**

To interact with external memory, design the memory interface that includes the address multiplexer, address latch, and data buffers. The multiplexer selects between program memory and data memory, while the latch holds the memory address temporarily.

**6. Control Unit:**

Create a control unit responsible for coordinating the microprocessor's operations. It generates control signals based on the instruction opcode and the microprocessor's current state, ensuring proper sequencing and timing.

**7. Clock and Reset:**

Integrate a clock generation circuit to provide synchronous timing for the microprocessor. Additionally, design a reset circuit to initialize the microprocessor to a known state during power-up.

**8. Finite State Machine:**

Develop a finite state machine (FSM) to manage the microprocessor's operation and transitions between different states (fetch, decode, execute, etc.). The FSM ensures the microprocessor follows the correct sequence of steps for each instruction.

**9. Instruction Fetch:**

Design the instruction fetch stage to fetch the instruction from memory using the program counter (PC). The PC increments after each instruction fetch.

**10. Instruction Execution:**

Create the instruction execution stage, where the fetched instruction is decoded, and the appropriate operation is performed by the ALU or control signals are generated for memory access.

**11. Memory Access:**

Implement the memory access stage to read/write data from/to memory based on the instruction's requirements. The address of the memory location is provided by the instruction itself or derived from the register contents.

**12. Integration and Testing:**

Combine all the designed components to form the complete microprocessor. Simulate the microprocessor design using hardware description languages (HDL) like VHDL or Verilog to verify its functionality. Test the microprocessor with various assembly programs to ensure correct execution of instructions.

**Conclusion:**

This 8-bit microprocessor design is a fundamental introduction to microprocessor architecture and the design process. From defining the architecture to integrating and testing the components, each step contributes to the microprocessor's functionality. While this design is simple, it can be a stepping stone to more complex and powerful microprocessors used in modern computer systems.