

Design and Development of Serial RISC-V Soft Microprocessor Core

ELE
569



Supervisor:

Dr.Aqeel Abbas

Department of Electronics Qau

Co Supervisor:

Dr.Rehan Ahmed

Seecs Nust

By

Abdur Rehman M.Ashraf
September-2023-Mphil-Ele

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of
Philosophy in Electronics (Mphil Ele)

In


Department of Electronics,
Quaid I Azam University Islamaabd,
Pakistan.

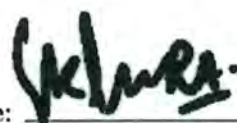
(September 2023)

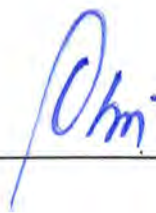


Thesis Acceptance Certificate

Certified that final copy of MPhil thesis entitled “**Design and Development of Serial RISC-V Soft Microprocessor Core**” written by **Abdur Rehman**, (Registration No **02102113013**), of Department of Electronics has been vetted by the undersigned, found complete in all aspects as per QAU Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MPhil degree.

Signature:  ✓
Supervisor: Dr. Aqeel Abbas
Date: _____

Signature: 
Co Supervisor: Dr. Rehan Ahmed
Date: 21st March 2024

Signature (HoD): 
Date: _____

Approval

It is certified that the contents and form of the thesis entitled "**Design and Development of Serial RISC-V Soft Microprocessor Core**" submitted by **Abdur Rehman** have been found satisfactory for the requirement of the degree.

Supervisor: Dr.Aqeel Abbas ✓

Signature: _____

Date: _____

Co Supervisor: Dr.Rehan Ahmed

Signature: _____

Date: 21st March 2024

Dedication

I dedicate this thesis to my beloved parents, Muhammad Ashraf and Shahnaz Begum, my siblings, my teachers, my friends and to all the deserving children who do not have access to quality education, especially young girls.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at Department of Electronics QAU or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at Integrated Circuit Design Lab NUST or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: Abdur Rehman M.Ashraf

Signature:  _____

Acknowledgments

Glory be to Allah (S.W.A), the Creator, the Sustainer of the Universe. There is no power except Allah and HE is the only one who has the power to honor whom HE pleases and to abase whom HE pleases. Verily no one can do anything without HIS will. I bear witness that there is no worthy of worship except ALLAH, the one alone, without a partner and I bear witness that MUHAMMAD (S.A.W.W) is his servant and Messenger. From the day, I came to QAU till the day I came to NUST and now writing this thesis, He was the only one Who blessed me and opened ways for me and showed me the path to success. There is nothing that can payback for His bounties throughout my research period to complete it successfully.

I pay my appreciation to my Advisor Dr.Muhammad Aqeel Abbas who permitted me to conduct my research in the field I choose. And always stood with me in my decisions and guided me throughout my research period.

And now I extend my heartfelt gratitude to the three invaluable gems in my life. [1] My co-advisor Dr.Rehan Ahmed and his two senior research assistants [2] Qazi Shahid Ullah and [3] Shaheer Sajid. They have been a wellspring of inspiration all through this research and above all, during my Mphil thesis. They have consistently been an astounding pioneer and guide. Their reliability, flexibility, and dedication have left a persuasive impact on my character. Without their understanding and difficult work, I would not have the option to finish my research program. And how I overlook the deadlines set by my Co-Advisor Dr.Rehan Ahmed they stand as a wellspring of motivation for me.

Abdur Rehman M.Ashraf

Contents

1	Introduction and Motivation	1
1.1	Objectives	2
1.2	Significance	2
1.3	Background and Motivation	3
1.4	Problem Statement and Contribution	3
2	Literature Review	5
2.1	Microprocessor Architecture	5
2.1.1	Evolution of Microprocessors	5
2.1.2	Instruction Set Architecture (ISA)	6
2.1.3	Data and Control Paths	6
2.1.4	Instruction Execution	6
2.2	Challenges in Microprocessor Design	6
2.3	Bit-Serial Execution	7
2.3.1	Advantages and Limitations	8
2.4	RISC-V Instruction Set Architecture	9
3	Design and Methodology	11
3.1	Types of Instructions	11
3.2	Addressing Modes	12
3.3	Architectural Overview	16

CONTENTS

3.4	Bit-Serial Data Path Design	18
3.5	Design Consideration for Energy Efficiency	20
3.5.1	Serial Execution:	20
3.5.2	Instruction Pipelining	21
4	Implementations Details	23
4.1	Pipelining	23
4.2	Load/Store Unit	24
4.3	Register File to Register File	25
4.3.1	Addition:	25
4.3.2	Subtraction:	26
4.3.3	Logical Operations:	28
4.3.4	Shift Right Arithmetic (SRA):	28
4.3.5	Shift Right Logical (SRL)	29
4.3.6	Shift Left Logical (SLL)	29
4.3.7	Set Less Then(SLT)	30
4.3.8	Set Less Then Unsigned (SLTU)	31
4.3.9	Load Word (LW)	31
4.3.10	Load Half Word (Lh)	32
4.3.11	Load Byte (Lb)	33
4.3.12	Load Upper Immediate	33
4.3.13	Add Upper Immediate to Pc (AUIPC)	33
4.4	Register File to Memory	34
4.4.1	Store Word (Sw)	34
4.4.2	Store Half (Sh)	34
4.4.3	Store Byte (Sb)	34
4.5	Control Flow	35
4.5.1	Jump and Link (JAL)	37

4.5.2	Jump and Link Register (JALR)	37
4.5.3	Branch If Equals to Zero (BEQ)	38
4.5.4	Branch if Not Equal to Zero (BNE)	39
4.5.5	Branch Greater than Equal to (BGE)	39
4.5.6	Branch Greater Then Equal to Unsigned(BGEU)	40
4.5.7	Branch Less Then (BLT)	40
4.5.8	Branch Less Then Unsigned (BLTU)	40
4.6	Data Path	41
4.7	Microarchitecture	42
4.8	Hazard Unit	43
4.8.1	Structural Hazards:	43
4.8.2	Data Hazards:	43
4.8.3	Control Hazard:	43
4.8.4	Handling of Structural Hazards	44
4.8.5	Handling of Data Hazards	44
4.8.6	Handling Control Hazard	44
4.9	Programming and Tools	45
4.9.1	Tools	45
4.9.2	Programming	45
5	Verification	46
5.0.1	GOOGLE DV Instruction Generator	46
5.0.2	Design Under Test (DUT)	47
5.0.3	Spike	47
5.0.4	Python Script	47
6	Analysis And Results	51
6.1	FPGA Implementation	51

CONTENTS

6.2	UART Test	53
6.3	Power Performance Area (PPA)	53
6.3.1	Power	54
6.3.2	Performance	54
6.3.3	Area	54
7	Conclusion and Future Works	56
7.1	Conclusion	56
7.2	Future Works	57
7.2.1	Enhanced Architectural Features	57
7.2.2	Mixed Bit-Serial and Parallel Architectures	57
7.2.3	Multi core Design	57
7.2.4	Advanced Verification and Testing	57
8	References	58

List of Figures

2.1	Serial vs Parallel	7
3.1	Conventional Parallel Branching	14
3.2	Parallel Load Instruction	15
3.3	Parallel Store Instruction	16
4.1	Serial Addition	26
4.2	Serial Subtraction	27
4.3	Serial Subtraction Timing Diagram	27
4.4	Serial Logical Operations	28
4.5	Shift Right Logical	29
4.6	Serial Set Less Than Timing Diagram	31
4.7	Serial Jump and Link Register	38
4.8	Serial Branch Prediction	41
4.9	Serial Branch Prediction Timing Diagram	41
4.10	Serial Branch Prediction	42
5.1	Verification Environment	48
5.2	Instruction Generator	48
5.3	Spike Result (Golden Reference)	49
5.4	Bit-Serial Core Result	49
5.5	Questa Sim Simulation	50

LIST OF FIGURES

6.1	FPGA Demo	52
6.2	Uart Test	53
6.3	Fre vs Power	55
6.4	Fre vs Area	55
8.1	plagrism report	

List of Tables

2.1	RISC-V ISA.	10
3.1	Operation count Parallel vs Serial.	21
4.1	Signed Arithmetic Rules.	30
4.2	Sign Extension.	32
4.3	Tools	45
6.1	LUTS and Register Utilization in FPGA	51
6.2	Frequency Power and Area	54

Abstract

It's a fact! That in modern microprocessor designs, energy efficiency and performance are two critical factors that drive research and innovation. As traditional approach to design the microprocessor has employed parallel processing techniques to enhance performance, but such approach often come with high power and area consumption. In recent few decades, we witnessed the remarkable advancement in microprocessor design leading to increased computational power and energy efficiency in modern computing system. Therefore, the investigation of innovative architectures that utilize bit-serial processing approach is an important field of research in processor design. This thesis work presents the design, implementation, and performance evaluation of a bit-serial approach to design a 32-bit RISC-V microprocessor. Aiming to leverage the benefits of bit-serial processing. The proposed microprocessor design execute the RISC-V instructions using bit-serial execution (one-bit per clock edge). This technique substantially reduce hardware complexity, increased energy efficiency and achieve higher clock frequency. The proposed microprocessor is pipelined and covered almost all the data, control and structural hazards. This design is made by using Hardware descriptive language name Verilog. And duly verified using Google Design Verification Environment. And its performance evaluation is conducted. Through a comprehensive set of benchmarks to compare the bit-serial RISC-V microprocessor against with the conventional RISC-V microprocessor and state of the art microarchitectures. The benchmarks encompass diverse work loads, including integer arithmetic, memory access, and control intensive tasks. Metrics such as power consumption, throughput, energy efficiency and area efficiency are considered to assess the effectiveness of the proposed architecture.

CHAPTER 1

Introduction and Motivation

In today's rapidly advancing technological landscape, microprocessor designs are continuously playing a pivotal role in shaping computing system's performance and energy efficiency. In the relentless pursuit of energy efficiency, and more faster computing systems, microprocessor architectures have evolved significantly over the past few years. With the constant demand for reduced power consumption and higher computing power, we the researchers are continuously exploring innovative architectures for microprocessors to meet these challenges. For this, one such novel and potential alternative approach to traditional parallel processing is the bit-serial technique, Which offers us the potential for energy-efficient executions while adhering to the open-source RISC-V instruction set architecture (ISA).

In 2010 at the University of California, Berkeley the new instructions set architecture (ISA) was developed. The royalty-free and open-source RISC-V instruction set architecture (ISA) gained prominence due to its flexible and adaptable framework for a wide range of computing devices. The purpose of developing the open-source and extensible instruction set is not only for academic use but also for commercial products. Remember that it is sometimes referred to as the "Linux" of processors. As a consequence, RISC-V received significant support from the open-source community, due to the adaptation and development of programming tools, such as GCC compiler with GFB support, GNU MCU Eclipse, LLVM toolchain, C libraries, and an official ISA simulator (spike).

In terms of the operating system, it has already support for FreeBSD, Linux Kernal, and ports of Debian. Due to this, the list of RISC-V members includes big companies such as NVIDIA, GOOGLE, Western Digital, Samsung, and Qualcomm. And currently Western Digital and Nvidia are working on their RISC-V microcontrollers to incorporate them in their commercial

products, as an alternative to their current solutions.

This thesis delves into the design, implementation, and evaluation of a Bit-Serial RISC-V Microprocessor, aiming to leverage the advantages of bit-serial execution while adhering to the widely adopted RISC-V ISA. Bit-serial execution is a unique approach that executes data one bit per clock edge, in contrast to conventional parallel execution, which handles whole vector simultaneously. By executing data serially, bit-serial architectures can potentially achieve reduced hardware complexity and greater energy efficiency.

The introductory chapter of this thesis lays the foundation for understanding the motivations, objectives, and significance of the research conducted. It presents the rationale behind exploring bit-serial processing, discusses the current challenges in microprocessor design, and provides an overview of the key contributions and organization of the thesis.

1.1 Objectives

The primary objective of this thesis is to design and implement a Bit-Serial RISC-V Microprocessor that explores the feasibility of bit-serial execution within the context of the RISC-V instruction set architecture. Specific goals include:

1. Developing bit-serial Datapath, execution unit, and memory subsystem tailored to the RISC-V ISA.
2. Investigating the strengths and limitations of the bit-serial processing technique for different types of workloads and applications.
3. Evaluating the energy efficiency and performance of the Bit-Serial RISC-V Microprocessor against conventional RISC-V processors and state-of-the-art microarchitectures through extensive benchmarking.

1.2 Significance

The significance of this research lies in its potential to pave the way for more energy-efficient microprocessor architectures. If bit-serial processing demonstrates promising advantages over traditional parallel processing for certain workloads, it could be an essential step toward addressing the energy consumption challenges in computing systems. Additionally, the implementation of a Bit-Serial RISC-V Microprocessor could open new avenues for exploration in the design of

future energy-efficient processors.

1.3 Background and Motivation

Traditional parallel execution techniques have long been the backbone of microprocessor design, enabling high-performance gains and throughput through the simultaneous processing of multiple data elements. However, as microprocessors have grown more complex, power consumption and area have become a major concern, particularly in battery-powered devices and data centers. This has prompted us to seek novel approaches that strike a balance between performance area and energy efficiency.

Bit-serial processing, as a potential alternative, offers a simplified paradigm where data is executed one bit at a time. This approach has the potential to reduce power consumption due to its reduced hardware complexity and inherently sequential nature of execution. By exploring the feasibility of implementing bit-serial processing within the RISC-V ISA, this research aims to contribute to the growing body of knowledge in the area of energy and area-efficient microprocessor design.

1.4 Problem Statement and Contribution

The continued evolution of microprocessors now demands new innovative design approaches for energy-efficient and high-performance microprocessors. In the pursuit of making energy efficient computing solution the microprocessor designs tailored for low power applications has become vital. The conventional microprocessor designs which follow the parallel processing of instructions with increased complexity, lead to challenges in power consumption, area utilization, and manufacturing cost. In response to overcome these challenges, we need a design that resolve all these along with enough efficient in computing solutions. Hence the concept of bit-serial microprocessor design emerged as a potential solution. Which meets all these performance parameters.

Now the question that arises here is how it will be efficient in performance while maintaining these parameters. while bit-serial holds the promise of reduced power consumption and area utilization also it introduces new challenges regarding execution time and clock speed limitations. Therefore we have to accept the fact that we have to compromise on something to achieve something. In the bit-serial concept for achieving low power consumption, we have to compro-

CHAPTER 1: INTRODUCTION AND MOTIVATION

mise a little on performance. But we can reduce this fact by making efficient designs and using multiple efficient solutions in a single design like bit-serial with pipe-lining.

Literature Review

2.1 Microprocessor Architecture

The aim of this section is to provide you with an overview of the micro-architecture of conventional microprocessors which refers to the internal structure and design of a microprocessor, which serves as a central processing unit (CPU). It includes the phenomena that how the microprocessor carries out the instruction, memory management, and the communication of the microprocessor with slave devices. Remember that the architecture is a crucial factor in determining the performance, capability, and efficiency of a microprocessor. Micro-architecture is the implementation of Instruction Set Architecture (ISA). From Instruction fetching to the write back data in register file.

2.1.1 Evolution of Microprocessors

A journey of continuous innovation and refinement in microprocessors. In Early eras, microprocessors were simple, single-core with limited computational power and large area. but with advancements in semiconductor technology, transistors became smaller and more efficient, allowing for the integration of multiple cores onto a single chip. This resulted in the transition of single-core microprocessors to multi-core microprocessors, enabling parallel processing and higher performance for tasks that could be divided among cores. Now the evolution has reached a point where we integrate not only a multi-core but multiple multi-cores in the same area using the latest bit-serial technique.

2.1.2 Instruction Set Architecture (ISA)

Instruction set architecture is the language of the microprocessor. It provides the information about what the processor is capable of. It specifies the format of instruction, operations, and addressing modes. There are two commonly known ISAs RISC and CISC. RISC is a reduced instruction set computer and CISC is a complex instruction set computer. The choice of ISA influences the design and performance of a microprocessor, therefore choosing carefully and wisely.

2.1.3 Data and Control Paths

As previously mentioned micro-architecture is the implementation of ISA, it involves two different paths to complete the circuit known as the control path and data path. Data path refers to the path through which data will be moved from instruction memory to the decode stage then towards the execution unit to operate and then the memory and then the register file to write back the result. The control path is responsible for managing the flow of instructions including the branch prediction and jump to the instruction to be executed next. These components would work together to ensure the correct instruction execution.

2.1.4 Instruction Execution

In early eras microprocessors were multi-cycled one instruction takes many cycles to be processed from fetching instructions to storing data. Then after a time, a technique named pipe-lined was introduced. Where we can process multiple instructions at a time. Hence the timing to process the instructions was reduced and throughput was increased. Pipe-line divides the microprocessor into different stages. Commonly it is divided into five stages as Instruction fetch, decode, execute, memory, and write back. But still, it's the designer's choice in how many stages he wants to divide its architecture.

2.2 Challenges in Microprocessor Design

Microprocessor design faces challenges such as clock skew, setup time violation, hold time violation, inferred latches, hazards, and non-hardware blocks. Therefore you have to stay focused when designing a microprocessor we have to keep in mind that each block would make

hardware. the block which will not make any hardware will not run after fabrication or on FPGA (Field Programmable Gate Array). For issues related to the clock, we have to properly divide the microprocessor into equal stages, and place the register or flip-flops clear fully so that not a single extra clock-edge cycle will be produced because a single extra clock cycle will disturb your complete design. We have to keep in mind that our pipeline cycle time will be equal to or greater than the critical path delay so that our instruction in each stage will process correctly without any timing violation.

2.3 Bit-Serial Execution

Bit-serial execution is a computational approach in which data is executed one bit per clock-edge. Computations will be performed on individual bits. By reducing the hardware in contrast to increasing the execution time. This contrasts with parallel processing, where multiple bits or data elements are processed simultaneously. It implies that now we need just one operation block instead of several operation blocks. In conventional microprocessor designs where instructions are processed parallel thirty-two operation blocks are needed for thirty-two-bit instructions and likewise sixty-four blocks for sixty-four bits instructions. but in bit-serial design, just one operation block is used for either thirty-two bits or sixty-four bits instructions.

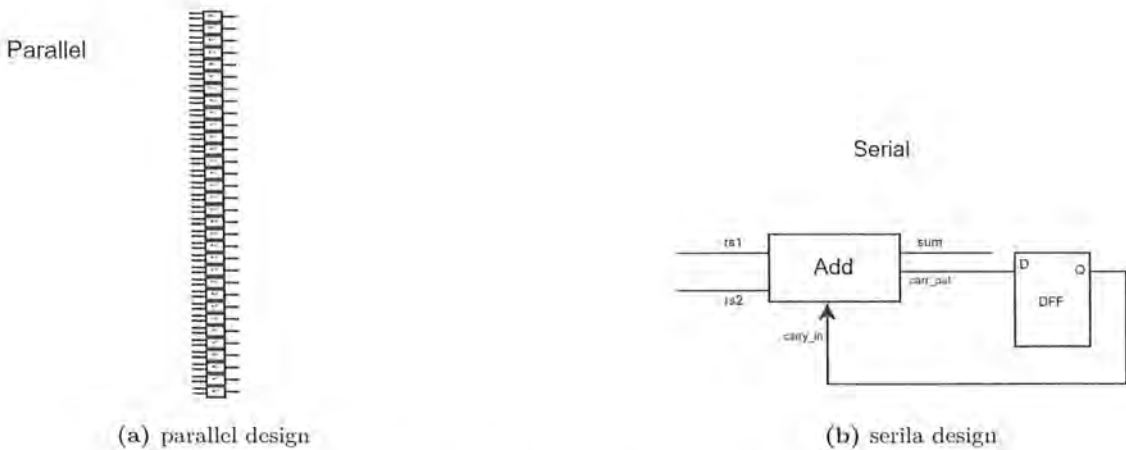


Figure 2.1: Serial vs Parallel

2.3.1 Advantages and Limitations

Advantages:

- **Compact Design:** Bit-serial microprocessors often have compact and simpler designs due to their serial execution nature which can lead to smaller size and low manufacturing cost.
- **Parallelism:** Due to its serial nature it has a lower rate of throughput than conventional parallel microprocessors, but we can increase it by introducing parallelism nature in bit-serial microprocessors. When a microprocessor becomes parallel multiple stages work on the same clock edge and different instructions are processed in different stages at the same time which can lead to higher throughput.
- **Low Complexity:** Bit-serial microprocessor has lower complexity than a conventional parallel microprocessor because the data path is now reduced from thirty-two bit to one bit. Also it simply the arithmetic and logic unit from thirty-two to one bit which can lead to easier chip layout due to lower transistor count.
- **Reduced Interconnects:** Due to the bit-serial nature of microprocessor design it requires less interconnect between processing elements the conventional parallel microprocessors which can improve overall performance due to lower signal delay.
- **High Clock Frequencies:** Bit-serial nature can allow us for higher clock frequency since the complexity of processing is now one bit which is lower than processing a whole word.

Limitations:

- **Limited Data Width:** bit-serial microprocessor can handle only one-bit data which can lead to slower processing for an operation that requires a large data path such as complex arithmetic and memory access.
- **Complex instructions :** Operations that can combine multiple bits can be complex and leads microprocessors to additional pipeline stages which can lead to increased latency.
- **Software Challenges:** Bit-serial nature of microprocessors often requires specialized programming techniques to take full advantage of their architecture. which can limit the portability of software and make it more difficult to optimize the code.

- **Performance Trade-offs:** While bit-serial microprocessor design can achieve higher clock frequency, the performance gains might not always be proportional to conventional parallel microprocessors due to the higher execution latency and additional pipeline stages that will be required for more complex operations.
- **Limited Applicability:** Bit-serial pipelined microprocessors are better suited for specific types of workloads, such as signal processing and certain forms of cryptography. They might not be as effective for general-purpose computing tasks as conventional parallel microprocessors.
- **Bit-Level Operations:** Performing operations at the bit level can introduce additional overhead for certain tasks that could be more efficiently handled by using wider data paths in traditional microprocessors.
- **Complex Hardware:** The design and implementation of bit-serial microprocessors can be complex, especially when dealing with operations that require multi-bit manipulation or inter-bit dependencies.

2.4 RISC-V Instruction Set Architecture

As mentioned earlier ISA is the language of any microprocessor. Therefore always choose wisely. We choose RISC-V ISA due to its open source and extensible nature. And also it is easy in micro-architecture. It has three types of operations. [1] Register file to Register file. [2] Register file to memory. [3] control flow, to jump on different addresses or instructions.

- It has seven seven-bit opcodes to specify the type of instructions.
- It has a small and fixed amount of registers.
- Each register has a bit address which can lead up to thirty-two registers.
- Three bits to address which function to be performed are named funct3.
- Seven bits to differentiate between the operations which have the same funct3 bits.
- And it has a variable amount of immediate based on the type of instructions.

31:25	24:20	19:15	14:12	11:7	6:0	Format
Funct7	Rs2	Rs1	Funct3	Rd	Opcode	R
Imm[11:0]		Rs1	funct3	Rd	Opcode	I
Imm[11:5]	Rs2	Rs1	Funct3	Imm[4:0]	Opcode	S
Imm[12 10:5]	Rs2	Rs1	Funct3	Imm[4:1 11]	Opcode	B
Imm[31:12]				Rd	Opcode	U
Imm[20 10:1 11 19:12]				Rd	Opcode	J

Table 2.1: RISC-V ISA.

Design and Methodology

While designing a microprocessor we were implementing the ISA (Instruction Set Architecture). In our design, we implemented the RISC (Reduced Instruction Set Computer) ISA. And before go onto actual design lets concise the RISC-V ISA first.

As mentioned earlier it has three types of operations.

- Register file to Register file.
- Register file to Memory.
- Control flow.

But there are a lot more in ISA. It has six types of instructions and also it has three different type of addressing modes. let's discuss these one by one.

3.1 Types of Instructions

- **R:** These are Register file to Register file type operations. Contains two source registers and a destinations register.
- **I:** These are also Register file to Register file operations, but in these instructions, one operand is source register one and second operand will be immediate/offset value and the third operand will be destination register. But in these types of operations, one instruction is a little different **JALR** it uses a combination of Register file to Register file and control flow operation. it stores the address of the current instruction in the destination register and also jumps to the next instruction.

- **S:** These are register files to Memory type operations these have one source register and an immediate/offset value and the source register two. Source register two is the data that will be stored in Memory.
- **U:** These are Register file to Register file operations but they have destination register and an immediate/offset value instead of using source register one or two they used the address of current instruction.
- **J:** It is a combination of Register file to Register file and control flow operations. It has one destination register and like U type instructions they used the address of the current instruction instead of the source register one or two. The address of the current instruction will be stored on the destination register and also it calculates the address of the next instruction.
- **B:** These are pure control flow instructions. These include two source registers, which are used for operation and based on their result processor will jump on the next address/instruction.

3.2 Addressing Modes

1. **PC-Relative:** Type of addressing mode used in RISC-V ISA. In this mode, the effective memory address of the next instruction is calculated as an offset using the address of the instruction that is currently being executed.

PC-relative addressing mode is commonly used for control flow or branching instruction, where the target address of instruction is relative to the address of current instruction. This mode is particularly useful for conditional and unconditional branching, as it allows the efficient encoding of small offsets in this type of instruction format.

How it works?

- **Immediate Decoding:** As I mentioned earlier it has two source registers two immediate/offset values and a funct3 bits to specify the function type. The notable point in this format is that we have to carefully entertain the offsets of this instruction and carefully decode them using proper decoding against the offset encoding. $\text{Imm}[4:1|11]$ represents that the seventh bit of instruction is the eleventh bit of Immediate, and the rest of the bits from eight to ten bits of instructions are one to

fourth bits of Immediate.

Imm[12|10:5] same in this twenty-five to thirty bits of instructions are five to ten bits of an immediate, and thirty-first bit of instruction is a twelfth bit of immediate.

- **Calculation:** when instruction is executing based on the comparison result of instruction Program counter will incremented. if the comparison result is true then the address of the current instruction in the execute stage will add with the offset value of the current instruction and then the program counter will jump to that address else the program counter will be incremented as usual by adding four to the current address in the program counter. **Note** that for jump the offset and address that will used is of the instruction presented in the execute stage.

PC-relative addressing mode is only advantageous because it reduces the size of instructions in memory, as only the immediate need to be stored instruction. Especially beneficial for RISC architecture because it has a smaller and typically fixed size of instruction.

A limitation of the Pc-relative addressing mode is that it is only beneficial for short-range branching/jumping. because a large offset needs more bits which typically increases the size of instruction

In summary, PC-relative addressing mode is a key mechanism in RISC ISAs for efficient branching/jumping and control flow instructions by using immediate/offset relative to the Program Counter, contributing to the overall speed and simplicity of RISC architectures.

2. **Register Offset:** addressing mode sometimes refer as "register plus" or "register index" addressing mode. Commonly used in computer architecture including RISC architecture. in this addressing mode effective memory address will be calculated by adding the immediate/offset with the data specified by one of the source operand registers of instruction. This addressing mode is often used in **Load/Store** instruction for accessing the data memory.

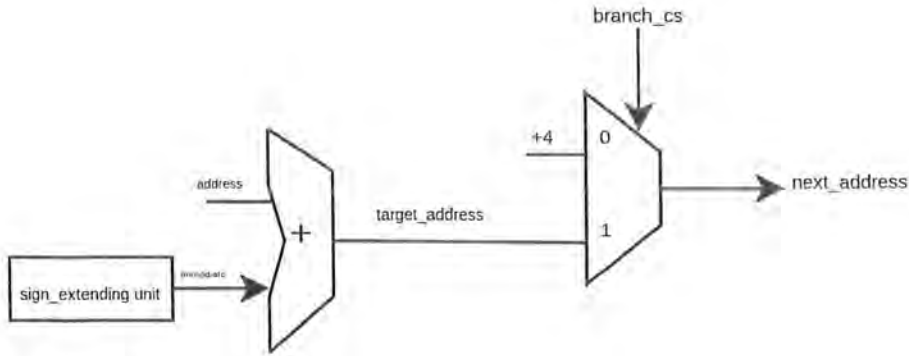


Figure 3.1: Conventional Parallel Branching

How it works?

- **Immediate Decoding:** There are two types of instructions format **I** and **S** that use this type of addressing mode. let's discuss it one by one
 - (a) **Load Immediate Decoding:** It has a simple twelve bits offset presented all bits together from twenty fifth bit to thirty-first bit of instruction. you have to just sign and extend it to thirty-two bits to decode it.
 - (b) **Store Immediate Decoding:** Carefully observes the arrangement of the bit then seven to eleven bits of the instruction are the first four bits of immediate/offset. then the twenty-fifth to thirty-first bits are the rest bits. for decoding care fully arrange these bits then sign extend these bits up to 32 bits.
- **Calculation:**
 - (a) **Load Address Calculation:** for Load instruction we have to add the data located at the address of the source register with the sign-extended immediate/offset presented in the twelve MSB (Most Significant Bits) bits of instruction to make the target address for data memory.
 - (b) **Store Address Calculation:** for Store instruction we have to first arrange the offset/immediate presented in the different bits of instruction the sign extends it up to thirty-two bits then adds it with the data located at the address of the source register one to get the target address of data memory.

Note: The question arises here then what will be the purpose of the source register two here?

The answer for that is the data located at the address of source register two is the data to be stored in memory.

Register offset address mode is beneficial because it gives us the flexibility of accessing memory without requiring separate load and store instructions, for different addressing modes such as a store, load byte or store, load half word or store, load word. Also it has the advantage that using register offset addressing mode we can access a large number of location inside the data memory, without the need of any extra bits. But! how much there for it has limitations based on the bits of source operand data and immediate bits. There for we can access the large number of locations but also a limited number of locations.

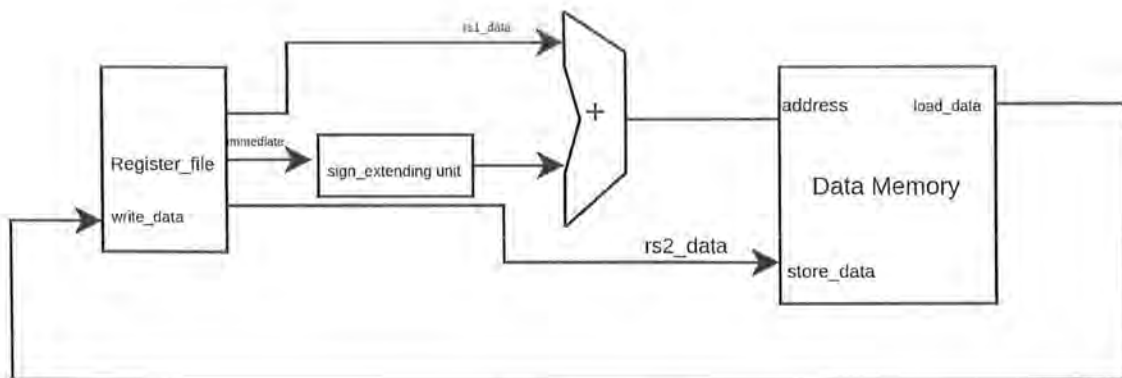


Figure 3.2: Parallel Load Instruction

3. **Absolute Addressing:** In absolute addressing mode, the memory address of a data or operand is directly specified within the instruction itself. The instruction contains the exact memory location where data should be stored or from where data should be fetched. Also, one of the instructions is named **Lui (Load Upper Immediate)** this instruction contains data itself that should be stored in the destination register. In this Instruction no extra logic is needed we just assign the data to the destination register directly. In other instructions we calculate data as its types as discussed earlier the only difference

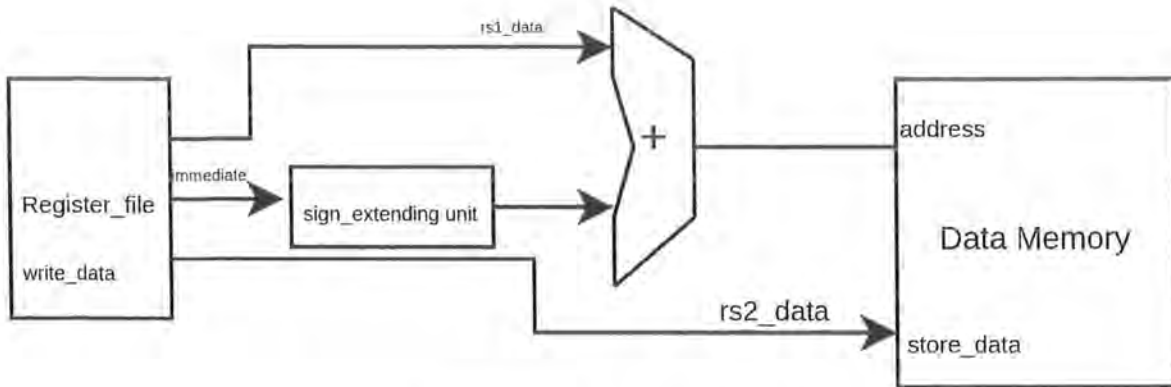


Figure 3.3: Parallel Store Instruction

is that we don't need any extra logic for targeted source register or memory location we just use the immediate/offset of instruction and calculate the address or data directly.

3.3 Architectural Overview

The bit-serial microprocessor has a single-bit microarchitecture, processing/executing instructions one-bit per clock edge rather than executing a whole word or byte like conventional parallel microprocessors. It operates on a single bit sequentially. Bit-serial microprocessor follows the same ISA as conventional microprocessors but still, it has a different microarchitecture, due to their serial nature. its data path is single-bit wide.

The data path for the fetch and decoding stages are the same as conventional parallel microprocessors the difference in the data path started after the decoding stage. before the decoding stage, it has thirty two bit wide data path but after the decode stage data path is converted from thirty-two to one-bit wide. Bit-serial microprocessor required 34 clock-edges at each stage to process the thirty-two-bit instructions. Two cycles overhead is to handle the data dependencies. let's discuss the main architectural components in each stage one by one.

1. **Fetch Stage** Used to fetch the instruction from Imem(Instruction Memory). It contains two main components.

- **Program Counter:** one input and one output, used to assign the memory address to Instruction Memory, based on the mux output connected with the input of the program counter. The program counter either assigns the incremented by four or

plus four (+4) address or it assigns the next targeted address calculated by branch or jump instructions.

- **Instruction Memory:** one input one out, contains pre-stored instructions. Used to assign the instructions to the microprocessor's decoding unit based on the address assigned by the program counter.

2. **Decode Stage** it takes the thirty-two bits instruction from instruction memory and decode it's all operands. and assign to their destinations. it includes

- **Register File:** Five inputs and two outputs, addresses of source registers and destination register are input and are assigned to register file after decoding from instruction. Write enable signal and write data are also inputs to the register file. The register file provides the data located on the source register address to the execution unit.
- **Control Unit:** It takes the seven-bit opcode and three bits of funct3 and seven bits of funct7 from the instruction then makes the data and control signals based on the opcode, funct3 and funct7 of instruction, which helps to control the data path of the microprocessor.
- **Sign Extending Unit:** take the immediate/offset as input and sign extend it to the thirty bits. As you observed in ISA different instructions have different bits of immediate/offset but remember microprocessor will operate on fixed defined bits if the microprocessor is thirty-two bit then it processes all data on thirty-two bits, and the same as for 16 or sixty-four bits. therefore we extend the immediate/offset to thirty-two bits.
- **Load Unit:** use to sign extend the resultant data after loading from data memory. As we have different types of load instructions such as **Lh(Load half)** or **Lb(load byte)** there for before storing it in the destination register we have to extend it up to thirty-two bits.
- **Alu Decoder:** make the operation control bits for execution unit **ALU (Arithmetic Logic Unit)** based on instruction.

3. **Execute Stage** Contain only one component,

- **Alu(Arithmetic Logic Unit):** perform the operation based on the operation control unit assigned by the alu decoder.

4. **Memory Stage** It contain two component,

- **Data Memory:** take alu output as the address of location and data located as a source register two as data to be stored.
- **Store Unit:** used to build the byte enable for data memory. based on which data will be stored on the specific bits at the located address.

5. **Write Back Stage** only contains a mux for selecting the data that will be stored on destination register.

3.4 Bit-Serial Data Path Design

It's clear from the name bit-serial that the whole data path will be one bit wide, but in our design not the whole data path will be one bit wide, As our design in pipelined there for our data path is the same as conventional parallel microprocessor data path till decode stage. The serial data path will start after the decode stage, this is to avoid the multi-cycle phenomena in pipelined microprocessor.

Note that we can not fetch instruction one bit per clock edge from instruction memory and also can not decode the instruction on one bit per clock edge to obtain this format we have to convert an instruction from whole word to bit stream and then convert the bit stream into whole word to decode the instruction which implies the multi-cycle phenomena in decode stage. Likewise, after the write back stage, we can not store the bit stream in the destinations register one bit per clock edge there for we have to convert the bit stream into a complete word to write it back.

Therefore when designing a bit-serial data path, we have to convert the conventional data path that is thirty-two bit wide into one bit wide data path there for now in the bit-serial data path all the components that are designed to process the thirty-two bits word are now design to process the bit stream that will be one-bit per-clock-edge. Two Key components helps us to design the bit-serial data path.

- **Counter:** responsible to to count the bits and to monitor the each stage to active for required cycles to process the instruction. In our Design it will count up to thirty four cycles, to process the thirty two bits instructions, which means each stage is activate for thirty four cycles, to process the instruction.

- **Shift Register:** It is an **Heart** of bit-serial data path. used to convert the thirty two bit word into a bit-stream, to process one bit per-clock-edge. Two types of Shift Registers are used in our Design

1. **Parallel-In-Serial-Out (PISO) Shift Register:** take thirty-two bit word as input and convert it into a bit-stream of thirty-two bits. In PSIO Shift register we have two types of shift rigters,

(a) **MSB to LSB PSIO Shift Register:** it take thirty two bit word as input and convert into bit stream that will starts from MSB (Most Significant Bit) end at last gave us LSB(Least Significant Bit).

(b) **LSB to MSB PSIO Shift Register:** It's output starts from LSB (Least Significant Bit) and ends at MSB(Most Significant Bit).

2. **Serial-In-Parallel-Out (SIPO) Shift Register:** it take the thirty bit stream of thirty two bits and combine it in a thirty two bits word. Like PSIO we used two types of SIPO registers.

(a) **LSB First:** it takes LSB first and MSB at last.

(b) **MSB First:** it takes MSB first and LSB at last.

As we discuss the key components of bit-serial microprocessor, now lets discuss some important components is serial manner. As in our design serial data path will start after decode stage therefor the first components will be

(a) **Arithmetic and Logic Unit (ALU):** in serial manner alu will be one bit wide there for the input should be one bit wide, in bit-serial we have one block for each operation, instead of using thirty two block like in conventional parallel microprocessor. For this purpose we have to convert the thirty two bit word into a bit stream and for this we used sift registers like PISO or SIPO based on instructions. As in conventional microprocessor instructions are executed in one clock cycle but in bit-serial microprocessor instructions will be execute in thirty two clock cycles, because data width is thirty two bits and operation will perform on each individual bit, there for in bit-serial data path ALU will be activated for thirty two clock cycles.

Note: In our design of bit-serial alu most of the instructions are executed using addition block.

Here question rises how ? we will discuss it in implementation chapter.

- (b) **Data Memory (DMem):** as i mentioned earlier we can not access data memory using serial bit. we need complete thirty two bits word to access the memory. there for we convert the alu result which will be the address of data memory and *rs2_data* which will be the data that will stored in dmem, from bit stream to a complete word, for this purpose we will use SIPO shift register.
- (c) **Write Back Mux:** as write aback mux used for two purpose one is to write back the data in register file, second is to handle the hazard there for in bit-serial we used two type of write back muxes
- **Parallel in Parallel Out Mux:** for write back purpose.
 - **Parallel in Serial Out Mux:** for handling the hazards, to forward the data towards alu.

3.5 Design Consideration for Energy Efficiency

This section involves the techniques that we use to make our design suitable for low power applications with high efficiency.

Note: that when we improving one factor we have to compromise on other factor to gain some thing we have to lose some thing. As our main goal is to make our design suitable for low power application there for we have to compromise a little bit on through put (Efficiency). we use two techniques.

3.5.1 Serial Execution:

we make our design bit-serial which means in our design instructions will executed serially one-bit per-clock-edge. operation will perform on each individual bits. which will make our design smaller then conventional parallel microprocessor design. and as the area is reduced power will automatically be reduces because the number of components in design is reduced.

in conventional microprocessor design each operation will use thirty two operation block to execute instruction in one clock edge which will used huge area. and each operation will have different blocks. Such as "**shift left, shift right, comparison, subtraction, addition**" each operation will required different operation block and each operations have different voltage level. In contrast with this our design will used single operation block of **Addition** for most of instruction and as we observe from thirty two parallel block to one block area will reduce with

large number.

Now lets do a rough comparison,

Operation Blocks	Parallel	Serial
Addition	32	1
Subtraction	32	1
Right Shift	32	1
Left Shift	32	1
Less then	32	1
Greater Then	32	1
Total	192	1

Table 3.1: Operation count Parallel vs Serial.

As you can see that in just a rough calculation where in conventional parallel design number of operation blocks are 192 and in serial its only one, and these are only six operation from thirty two operations and difference in numbers is huge and if we make a supposition,

Suppose one operation blocks consume 1nm area then for these operations 192nm area will required for parallel design and only 1nm are is required for serial design.

this difference is only for counted operation in just ALU which is only one component of micro-processor, now just imagine area required for muxes registers and all other components that are thirty two bit wide in parallel design. Now imagine how much area will be reduced by making it serial. And power that will be required for 192 components is reduce to only for one component. Suppose if one block use 1mv then 192mv for parallel design and only 1mv for serial design.

3.5.2 Instruction Pipelining

As i said earlier to achieve some thing we have to lose some thing, there for in our design we lose the efficiency because the instruction that will execute in one clock cycles will now execute in thirty two clock cycles hence time for execution is increased in contrast the through put is decreased.

To over come this issue we make our Design pipe-lined. As in no-pipelined design the only one instruction will process on thirty two clock cycles but now in pipelined design the five different instructions will process in same thirty two clock cycles because we divide our design oin five

CHAPTER 3: DESIGN AND METHODOLOGY

different stages using pipelining. hence throughput will increased from one to five.

Implementations Details

This chapter includes the detail explanation regarding how the ISA was implemented using bit-serial and pipelining technique. We will discuss all three types of operations and the instructions under these types, with complete microarchitecture and data path.

4.1 Pipelining

is fundamental concept in microprocessor design used to improve the through put and overall performance of design, by breaking down the instruction execution process into a discrete stages. Each stage is responsible for handling a specific part of the execution process, and allowing the microprocessor to process multiple instruction simultaneously. This parallelism helps the microprocessor to make better use of the hardware resources and increase the overall processing speed. we divide our design in to a five different stages discussed as bwlowed.

1. **Instruction Fetch:** stage is responsible for fetching the instruction from instruction memory based on a given address from program counter.
2. **Decode Stage:** responsible to decode the instruction, to identify the operation, required register and control and data signals.
3. **Execute Stage:** responsible to perform the desired operation on the gievn data.
4. **Memory Access:** stage responsible for accessing the data memory using different Load and Store instruction.
5. **Write Back** responsible for writing the data ta the destination register.

4.2 Load/Store Unit

one of the key components of design used to properly aligned the data before writing it at the destination it either the register file or data memory.

- **Load Unit:** used in decode stage to properly aligned the data before writing at the destination register used for Load instruction. when the instruction is Load half or Load byte, the bits we will write are either sixteen or eighth. **Lh=16bits** and **Lb=8bits**. As we know the data we load from memory is a complete word of thirty two bits, then how we know which of the sixteen or eighth bits will write at the destination register? this decision will made in Load unit on the basis of lowest two bits of the data memory address.

– Load Byte:

1. $dmem_address[1:0]==2'b00$ the lowest eighth bits [7:0] will write at the destination register.
2. $dmem_address[1:0]==2'b01$ from bit eight to bit fifteen [15:8] will write at the destination register.
3. $dmem_address[1:0]==2'b10$ from bit sixteen to bit twenty three [16:23] will write at the destination register.
4. $dmem_address[1:0]==2'b01$ from bit twenty four to bit thirty one [31:24] will write at the destination register.

– Load half

1. $dmem_address[1:0]==2'b00$ the lowest sixteen bits [15:0] will write at the destination register.
2. $dmem_address[1:0]==2'b01$ the highest sixteen bits [31:16] will write at the destination register.

- **Load Word** default (complete word load from memory) data will write at the destination register.

Remember this data will be first sign extend after selecting the desired bits then write at the destination register

- **Store unit:** used in Memory stage to properly aligned the data before storing it in the data memory. Now the point of consideration is that at the targeted location on which

bits we store the data if we have the store byte or store half instruction. Same as Load unit this decision will also made on the basis of address of data memory.

– **Store Byte:**

1. $dmem_address[1:0] == 2'b00$ data will store at the [7:0] bits of targeted address.
2. $dmem_address[1:0] == 2'b01$ data will store at the [15:8] bits of targeted address.
3. $dmem_address[1:0] == 2'b10$ data will store at the [16:23] bits of targeted address.
4. $dmem_address[1:0] == 2'b01$ data will store at the [31:24] bits of targeted address.

– **Store half**

1. $dmem_address[1:0] == 2'b00$ data will store at the [15:0] bits of targeted address.
2. $dmem_address[1:0] == 2'b01$ data will store at the [31:16] bits of targeted address.

– **Store Word** default (given by instruction) data will store at the targeted address.

Remember this data will not be sign extended

4.3 Register File to Register File

this types of operation contains all Integer and Integer-Immediate instructions with some other instructions. It contain Basic integer and Integer Immediate instruction the process for both type of instruction is same only the difference is that we have source register two in Basic Integer instruction , and in Integer immediate instruction we have Immediate value instead of source register two.

4.3.1 Addition:

Addition is base operation of our design, and to implement the addition in serial manner, we have to first make our data that will add from thirty two bits word to a bit stream. there for we use PISO shift register to convert the data. then we implement the addition on each individual

bits.

Remember that the carry out of each individual sum will be add with the next bits with clock-edge delay because each bit will shifted after one clock-edge there for we have to sync the carry-out with the inputs bits.

Serial

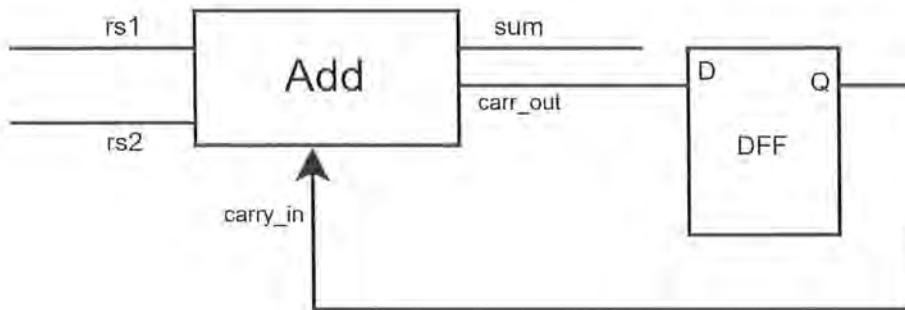


Figure 4.1: Serial Addition

4.3.2 Subtraction:

is also implemented using addition, by using twos compliment method. And remember the twos complement will be implemented parallely with the operation so no extra clock-cycles will required but if we first calculate the two's complement then perform the subtraction the execution cycles will increased from thirty two to sixty for first thirty two cycles for two's compliment and the next thirty two clock cycles for adding the twos complement with the data. how the operation will perform if we have subtraction instruction.

e.g `sub x5,x2,x1`

as in given instruction we have x5 as destination register and x2 and x1 are source register that contains data to e subtracted. this instruction implies that we have to subtract x1 from x2 there for we take x1's two's compliment and add it with x2.

Adding phenomena is same as addition instructions carryout will be add with next bits after one clock edge delay.

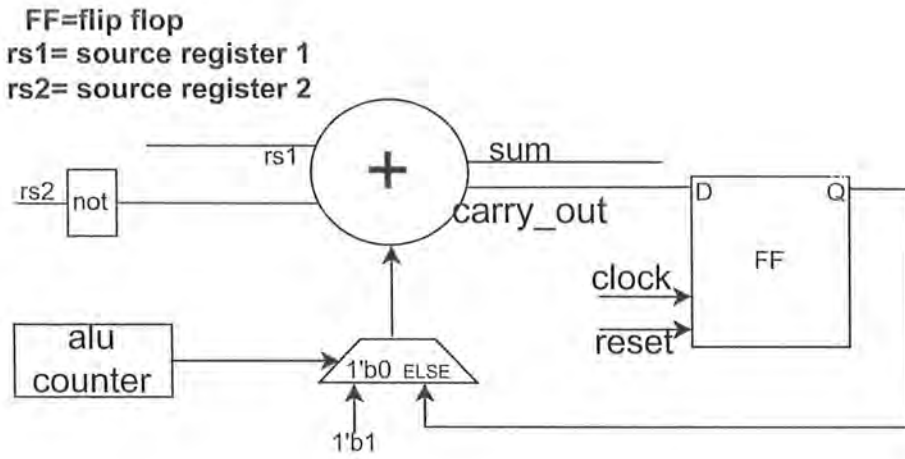


Figure 4.2: Serial Subtraction

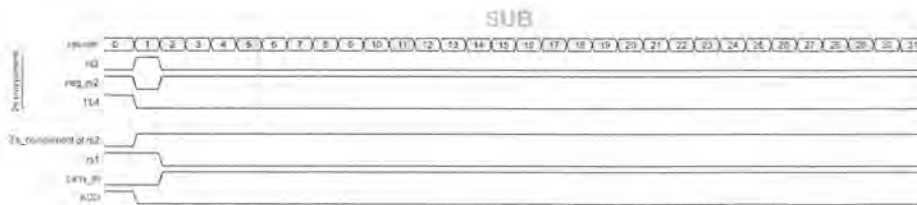


Figure 4.3: Serial Subtraction Timing Diagram

4.3.3 Logical Operations:

And,Or,Xor All these three operation will execute in same manner. we just serialize the data using the PISO shift register and perform the operation on each individual bits no extra logic's are required for these operations.

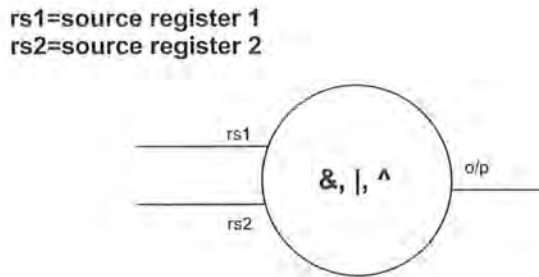


Figure 4.4: Serial Logical Operations

4.3.4 Shift Right Arithmetic (SRA):

This instruction is used to shift the operand based on shift amount given in second operand.

eg. `sra x3,x2,x4`

this instruction contains x3 as destination register and x2 is sources register that should be shifted and the shift amount is given in the source register two which is x4.

Questions rises here how we shift individual bits based on a shift amount?

shift instructions are implemented using the shift registers. For right shift operation PISO shift register will hold data for shift amount, then start shifting from MSB (most significant Bit) towards LSB (Least Significant Bit) by left shifting the data. **Be Care full** data should be hold till shift amount cycles are not passed, other wise you will lose your data.

Now in ALU we place the MSB at the hold bits, as shift right arithmetic operation shift the data and place the MSB at the empty places. there for we place MSB on the hold bits in alu.

After ALU we got the result but its not properly aligned we got data who's MSB is at LSB and

LSB is at MSB there for in SIPO shift register, it takes MSB first and shift it towards the MSB to properly aligned the data.

4.3.5 Shift Right Logical (SRL)

This instruction use to shift the operands logically towards right side. Now logical! means in this instruction we will place the **zero** on the empty places.

PISO shift register use to convert the data into a bit stream and in required alignment based on instruction, and like **SRA** we need MSB first in ALU and ALU will place the zero on the hold bits. and remember like SRA your data should be hold for shift amount then PISO shift register will strts shifting the data from the bit that is required first it may be either MSB or LSB based on instruction.

Operation of SIPO shift register placed after alu will be same as in SRA.



Figure 4.5: Shift Right Logical

4.3.6 Shift Left Logical (SLL)

SLL used to shift the data logically towards left. and as SRL logical means to place the zero at empty places.

eg. `srl x3,x2,x1`

to shift the data towards the left side the PISO shift register will act slightly different then previous one. we shift the data Right to perform the left shift operations. initially data will hold for shift amount then PSIO shift register will out the LSB first by right shifting the data. ALU will place the zero at the hold bits till the shift amount cycles are not passed.

Then SIPO shift register take LSB first from ALU and starts right shifting the data from MSB to LSB and hence we got the desired data in required alignment.

For all the shift operation we have to synch all the control and data signals of PISO shift register, ALU and SIPO shift Register. And the shift direction in shift register is very important

4.3.7 Set Less Then(SLT)

SLT is **Signed Instruction**. Now what is means by Signed instructions?

Signed Instruction: Instruction that will implement under the Signed Arithmetic Rules. Now what are the signed arithmetic rules?

Operands Signs	Result
+,+	+
+,-	-
-,+	-
-,-	+

Table 4.1: Signed Arithmetic Rules.

There for we have to be very care full when implementing comparison instructions either they are signed or unsigned.

e.g `slt x3,x2,x1`

this instruction compare the data located at the source register one and source register two, if the data of source register one is less then the data of source register two then it will write the one (1) at the destination register. other wise zero (0).

for implementing the comparison instructions like this we perform the addition operation on data of source register one with the twos compliment of data of source register two, long story short we implement the subtraction operation for implementing the set less then instruction. And the result will be accumulate on the basis of MSBs of source register one and source register two data, and the subtraction results MSB.

Note: result will accumulate paralley and required combinational logic which means it does

not required any extra cycles. and hence instruction will execute in thirty two clock cycles.

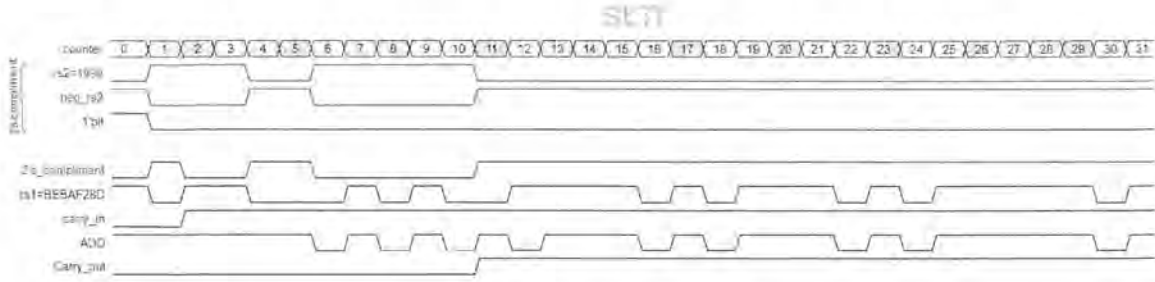


Figure 4.6: Serial Set Less Than Timing Diagram

4.3.8 Set Less Than Unsigned (SLTU)

SLTU instruction is same as SLT the only difference is that SLT is computed under signed arithmetic rules, but SLTU will not compute under signed arithmetic rules. SLTU will also implement using Subtraction but now the logic for computing the result is changed.

As in SLT instruction for computing the result under signed arithmetic rules we used the MSB of subtraction result, but in unsigned comparison we compute the result on the basis of source operands data and the carry-out bit that will obtain after MSB subtraction. And same as SLT and output will calculate parallelly and logic will be pure combinational because we have to calculate result in thirty two bits, no extra cycles will be needed using combinational logic.

4.3.9 Load Word (LW)

Load word instruction used to load the whole thirty two bit data from the data memory placed at targeted address in the destination register.

e.g `lw x3,x1,108`

- **Address Calculation:** Targeted address of data memory was calculated by adding the data in source register one with the sign extended immediate/offset value. as immediate has twelve bits only but to execute the instruction we need each operand of thirty two bits there for we have to extend it up to thirty two bits. Now **Sign extend** refer to extend the MSB bit up to remaining thirty two bits. now both operands are of thirty two bits now we can easily add them to calculate the targeted address. After perform the addition we have to convert the data from bit stream to a vector form(thirty two bit word). As

Immediate	Sign Extended
11'b101000000001	31'b1111111111111111111111111111111101000000001

Table 4.2: Sign Extension.

As mentioned earlier we can not access the data memory with one bit per clock edge data or address. Now as we have data in vector form there for now we can load the data from memory and write back it at the destination register.

- **Write Back:** But before write it to destination register we have to pass it through the load unit. Load unit is used to extend the data up to thirty two bits if its not, and as LW instruction load the whole thirty two bit word from data memory there for the Load unit will give us the same data at the output as it was at input.

4.3.10 Load Half Word (Lh)

Load half word instruction is used to load the half word from data memory and write it back at the destination register. But! the **Question** rises here that how can we load the half word from data memory?. Lets discuss it one by one. First the address calculation.

e.g `lh x2,x3,108`

- **Address Calculation:** The target address will calculate same as Lw Instruction. the source register one is added with the sign extended immediate/offset, and addition was performed one bit per-clock-edge. then we convert the data from bit stream to a vector. Pass it to the data memory and data memory gave us the whole thirty two bit word as output.
- **Write Back:** before writing it to the destination register we pass it to the load unit now load unit will decide it which of the sixteen bits are use to write as the destination register. Sixteen bits because the instruction is **Load Half** there for it load only half word and half word is of sixteen bits. But we can not write the sixteen bits at destination register there for load unit select the sixteen bits and also sign extend it up to thirty two bits, for writing at destination register.

4.3.11 Load Byte (Lb)

Load byte instruction is used to load the one byte data from data memory and write it back to the destination register.

e.g `lb x2,x3,108`

- **Address Calculation:** Address calculation is same as other load instruction. first convert the data in to a bit stream. Sign extend the immediate/offset and then add it with the data of source register one.
- **Write Back:** Same as other instruction before writing it at the destination register we have to pass it through the Load unit for selecting the desired byte that should be write at destination register. And then sign extend this byte up to thirty two bits, then write it at the destination register.

4.3.12 Load Upper Immediate

instruction used to to load the twenty bits of immediate/offset value in the destination register after sign extending it to the destination register. in load upper immediate instruction we will not sign extend the immediate value because in this instruction we have to write the twenty bits of immediate value along with twelve zero bits at the LSB.

`IMM[19:0],12'b0` like this. to perform this type of operation we have two ways

- Either you perform the logical left shift operation as discussed earlier for the amount of ten bits and then write it back to the destination register.
- Or we can just assign the immediate after write back stage to the load unit and then make the data by making the lower twelve bit zero and write it back to the register.

4.3.13 Add Upper Immediate to Pc (AUIPC)

instruction use to implement the PC-relative addressing mode. in this instruction we will make the immediate first like `Lui` instruction `IMM[19:0],12'b0`. and now add this immediate with the address of current instruction. and the write the result at the destination register.

4.4 Register File to Memory

4.4.1 Store Word (Sw)

As it's clear from the name the instruction is used to store the whole word of thirty two bytes in the data memory at the desired location. The format of instruction is

e.g `Sw x3,x2,108`

in this instruction we don't have any destination register as we are writing the data in the memory not in the register file. There for no need of destination register.

- **Address Calculation:** Address calculation for store word is same as load word we serialise the data sign extend the immediate/offset and add source register one with the immediate/offset.
- **Store Data:** for writing the data we now pass the data from the store unit, Store unit will select the bits and sign extend it and then we store it at the targeted address. But! in store word case we don't need of this because we store the complete thirty two bits there for store unit will out the default data as it was a input.

4.4.2 Store Half (Sh)

store half instruction used to store the half word of data at the targeted address.

e.g `sh x3,x4,109`

- **Address Calculation:** same as Store word.
- **Store Data:** before storing the data we pass it through the store unit now store unit will select the sixteen bits that will store in the data memory, and sign extend it up to thirty two bits. then we store the data at the targeted address.

4.4.3 Store Byte (Sb)

store byte instruction used to store the one byte of data at the targeted address.

e.g `sb x3,x4,109`

- **Address Calculation:** same as Store word.
- **Store Data:** same as store half and store word before storing the data we pass it through the store unit now store unit will select the eight bits that will store in the data memory, and sign extend it up to thirty two bits. then we store the data at the targeted address.

Note: for storing the data in data memory we need parallel data instead of serial data because we cant store data one-bit per-clock-edge or i say serially.

4.5 Control Flow

these instruction are used to control the flow of instruction in instruction memory.

Question: How? these instructions decides whta will be the next instruction that will execute. lets discus it using example.

We want to display **This core is running on RV-32i ISA** using UART (Universal asynchronous receiver-transmitter). Now each alphabet has different ASCII and different Hexadecimal number now we run a bunch of instruction. the flow of instruction will used control flow instruction for making the data because data should be in some bits of instruction and we have to made it properly by using continuous loop of instruction and, and loop of instruction will be implemented using the control flow instructions. the instruction pattern given below will print the desire data **this core is running on RV-32i ISA** and the data will accumulate in the store word instruction and if you notice that store word instruction is in loop of made by using different control flow instructions, and hence like this we control the flow of instruction in instruction memory using control flow instructions.

1. 00011197 auipc
2. 82818193 addi

CHAPTER 4: IMPLEMENTATIONS DETAILS

3. 00018117 auipc
4. FF810113 addi
5. 00010433 add
6. 00010637 lui
7. 00160613 addi
8. 008006B7 lui
9. 00060713 addi
10. 05400793 addi
11. 00F6A023 sw
12. 00170713 addi
13. FFF74783 lbu
14. FE079AE3 bne
15. FE9FF06F jal
16. 008007B7 lui
17. 00A7A023 sw
18. 00008067 jalr
19. 00054783 lbu
20. 00078E63 beq
21. 00150513 addi
22. 00800737 lui
23. 00F72023 sw
24. 00150513 add
25. FFF54783 lbu
26. FE079AE3 bne

27. 00008067 jalr

Now lets discuss each control flow instruction one by one.// we have two types of control flow instructions, conditional and unconditional jumps.

4.5.1 Jump and Link (JAL)

e.g jal x2,-108

as mentioned earlier JAL is a combination of register file to register file and control flow type. Because it writes the data at the register file also it use to jump to the targeted address.

- **Write Back:** it writes the address of the current instruction plus four (Pc+4) at the destination register. For writing this type of data we have two possible ways.
 1. Either you hard encode the four in thirty-two bits and then add these bits with the address of the current instruction.
 2. Or you can directly assign the updated address to the register file.

Note: we used the second method because hard enoded in design is not a good option.

- **Jump address calculation:** In Jal's instruction the immediate/offset is encoded in multiply of twos form, and also its a signed offset. There for before adding the offset with the address we first make it in multiplies of two's form by making its LSB zero and then sign extend it up to thirty two bits. Then we add the address now the obtained result is the next instruction address. where microprocessor will jump.

4.5.2 Jump and Link Register (JALR)

e.g jalr x2,x1,0

it's also a combination of register file to register file and control flow type.

- **Write Back:** same as JAL instruction.
- **Jump address calculation:** the method of calculating the target address is slightly different then Jal instruction, here in JALR we extend the offset up to thirty-two bits and add it with the address of the current instruction. when we got the result we set the LSB

of the result to zero for aligned it with the word boundary.

Note: in our design we don't wait for getting the whole result and then set the LSB to zero because we have an SIPO shift register after ALU where we get the complete result and then making LSB zero will take an additional cycle because SIPO is sequential clock-based, there for when adding the operands we send the zero at the place of LSB addition, but remember if you are sending the zero you still have to add the LSBs because we need carry-out for next bit addition else the result will be wrong.

```
carry_out,sum=rs1+rs2+carry_in;
sum = count_alu=0 ? 0: sum;
```

Note: JAL and JALR are unconditional jumps, means they dont need any condition on which bases they will jump, they just jump

PISO=Parallel in Serial out
 SIPO=Serial in Parallel out
 temp=temporary variable to store 32 bits of add

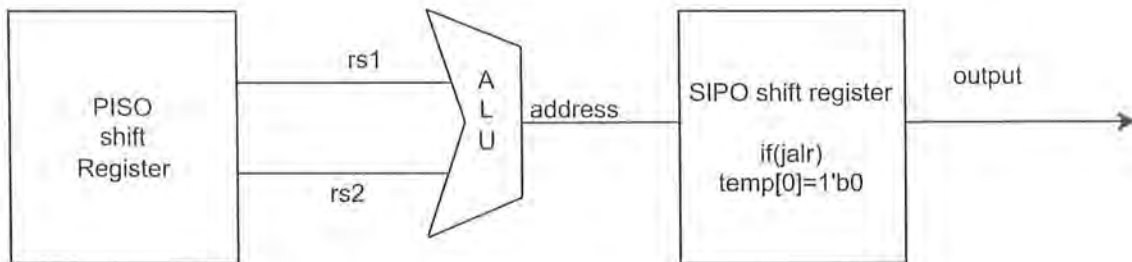


Figure 4.7: Serial Jump and Link Register

4.5.3 Branch If Equals to Zero (BEQ)

implement to jump at the targeted address based on the condition that if the source operand one is equal to the source operand two only then jump to the targeted address. In this instruction three adder blocks will work together. One for checking the equality and rest of the two for calculating the targeted address. **Two address?**

one is PC plus signed extended offset ($Pc+sext(offset)$) and other one is Pc plus four($Pc+4$).

- **Equality:** equality was checked by subtracting the source operand two from source operand one. We take two compliment of source operand two and add it with the source

operand one. Now as we have serial data there for we need counter which will count on the basis of the result of subtraction. If the result is zero the counter will count plus one and at the end if counter will reached to the thirty two it means result is zero and it implies that $rs1$ is equal to the $rs2$. because only then we get zer result of subtraction if both are equal else not.

- **Address Calculation:** We can calculate both address at the same time otherwise, we need extra clock cycles for calculating both address on each clock edge we will calculate $Pc+4$ and $Pc+offset$ and remember offset will be sign extended up to thirty two bits.

Now at which address Pc will jump, this will decide on the basis of counter result if its thirty two then Pc will jump to the $Pc+offset$ address else it will jump to $Pc+4$ address.

4.5.4 Branch if Not Equal to Zero (BNE)

Branch not equal to zero is the opposite of BEQ now the Pc will Jump if and only if source register one is not equal to source register two.

- **Equality:** Now as we check the equality for BEQ the same process will follow for BNE, only te difference here in BNE is PC will jump if the counter will not reach to thirty two, because only then we can say that the both operands are not equal.
- **Address Calculation:** same as BEQ we calculate both addresses at the same time else we need extra thirty-two cycles to execute this instruction. There we calculate both addresses at the same time.

now the decision of Target address will made on not reaching the counter to thirty-two, if the counter reaches to thirty-two then the next address will be $Pc+4$ else if Pc not reaches to thirty two then the next address will be $Pc+immediate$.

4.5.5 Branch Greater than Equal to (BGE)

BGE is a signed instruction and jumps only if source register one is equal to or greater the source register two. As it is a signed instruction there for it will execute under signed arithmetic rules that we discussed earlier.

- **Condition Check:**

1. **Greater Than:** As now we have greater than condition, followed by signed arithmetic rules, we used the same phenomena as BLT but in opposite manner, the condition on the basis of which we decides the less then we can use it in opposite manner and use them as greater then.

And same as BLT decision was made on source register one MSB, source register two MSB, and the MSB bit of result.

2. **Equal To:** Equal to the condition will check using the same phenomena as used in BEQ or BNE.

- **Address Calculation:** Address calculation is the same as BEQ or BNE.

4.5.6 Branch Greater Than Equal to Unsigned(BGEU)

All the procedure is the same as BGE the only **difference** is that now we don't check the greater than condition under signed arithmetic rules, now its simple and decision will made on the MSBs of source register one, source register two and carry out bit.

4.5.7 Branch Less Then (BLT)

Branch less then instruction is signed instruction and used to jump the Pc if and only if source register one is less then source register two.

- **Condition Check:** less then the condition will check same as **SLT** instruction same process will used here.
- **Address Calculation:** Same as BEQ or BNE.

4.5.8 Branch Less Then Unsigned (BLTU)

Same as Blt the Pc will jump only if rs1 is less the rs2 but now we don't care of signed arithmetic rules.

- **Condition Check:** less then the condition will check same as **SLTU** instruction same process will used here.

- **Address Calculation:** Same as BEQ or BNE.

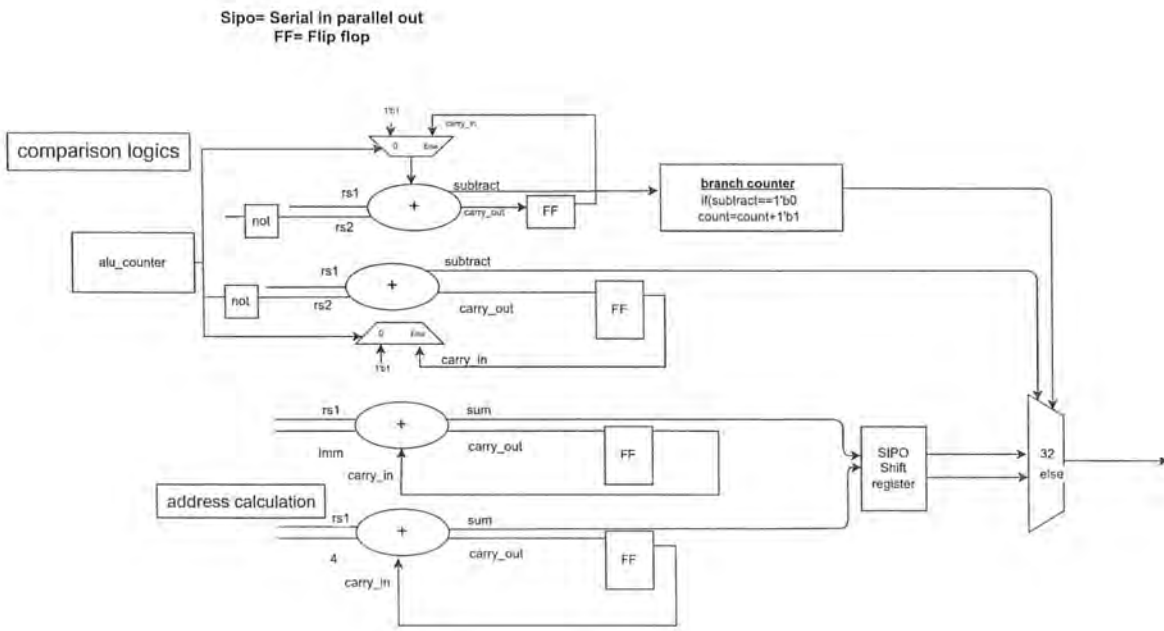


Figure 4.8: Serial Branch Prediction

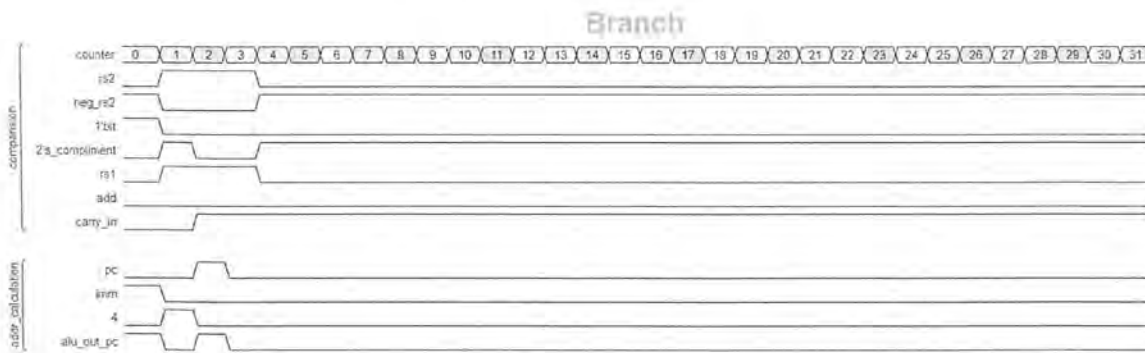


Figure 4.9: Serial Branch Prediction Timing Diagram

4.6 Data Path

- **Register file to Register File Instructions:** data path will start after the instruction memory but before that, we have a Program counter that will make the address for the

instruction memory. Then instruction memory will fetch the instruction and pass it to the PIPO Shift register, after that instruction will decode the control unit will make the control signals register file gave us the data located at the addresses of source registers sign extend extend the immediate/offset then these all data and signals will pass to the PISO shift register the out will goes to the hazard control muxes and then ALU for execution after then the output of ALU will pass to the SIPO shift register to make the data vector, then pass to the memory if load instructions is in process after then write back mux and then load unit for sign extension then write back again to the register file.

- **Register file to Memory:** same procedure as register file to register file but it will stop at data memory that's it.
- **Control Flow:** same as the Register file to Register file till the execution stage after then when the instruction is executed in ALU then it will forward back to the Program Counter for jump and then Pc will jump to the given instruction.

4.7 Microarchitecture

Below given diagram shows the complete microarchitecture of serial RISC-V (RV-32i) design. Which involves all the necessary components required for the bit-serial microprocessor deign.

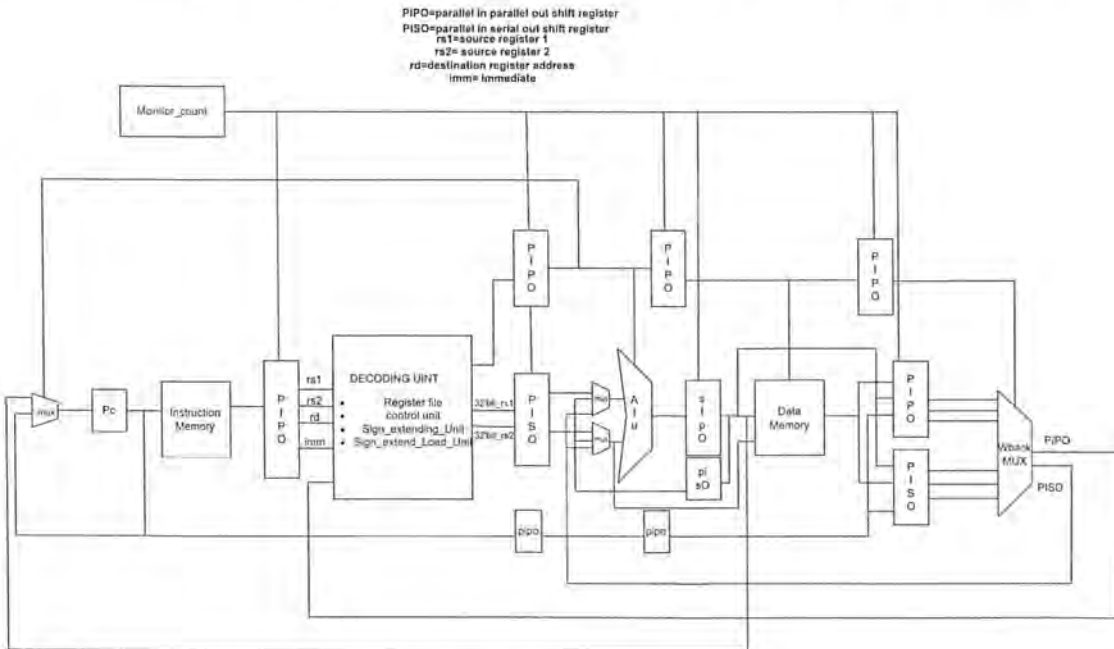


Figure 4.10: Serial Branch Prediction

4.8 Hazard Unit

The most important unit of pipelined microprocessor design. Responsible for the proper flow of instructions in pipelines and rest of the core, by detecting and handling the different types of hazards in the design.

Note: stay focused when designing this unit a single mistake can fail your whole design

there are three different types of hazards in pipelined microprocessor design such as

4.8.1 Structural Hazards:

This occurs when multiple instruction need in use of the same resource or component in the design then this hazard will occur. e.g when two instructions require access of same execution unit at the same time.

4.8.2 Data Hazards:

Occurs when instruction needs the result of an other instruction that is still in the pipeline. we face three types of data Hazards in pipeline design.

```
ADD R1, R2, R8 ; R1 = R2 + R8
SUB R4, R1, R9 ; R4 = R1 - R9
MUL R1, R4, R10 ; R1 = R4 * R10
```

1. **Read-After-Write(RAW):** when instruction needs to read the data that's no ready till yet. As MUL depends on the ADD instruction.
2. **Write-After-Read (WAR):** When the instruction is read before the other instruction write the data. As we can see SUB reads from rs1 and MUL writes at rs1.
3. **Write-After-Write (WAW)** When both instruction writes the data at the same location hazard occur if the not executed in the correct order. As ADD and MUL both write at rs1.

4.8.3 Control Hazard:

These types of hazards occurs when there is a change in flow in instructions due to control flow instructions, such as **JUMP** and **BRANCH**.

These hazards or say dependencies that we will face in pipeline design. Now the **Question** rises here what do we do when face these hazards?

4.8.4 Handling of Structural Hazards

- **Resource Duplication:** duplicate the resources that commonly caused the structural hazard. Such as if ALU causes the the structural hazards to double the ALU. This method caused the increase in area.
- **Bypassing:** Bypass the data from any stage that causes the structural hazard, before waiting for writing at the destination.

4.8.5 Handling of Data Hazards

- **Stalling:** stalling means holding the instruction in the same stage for a number of cycles required to process the instruction properly.
- **Forwarding:** or we say bypassing the data from one stage to another stage.
 1. **Forwarding from Execute Stage:** if the source register at the Decode stage is the same the destination register in the Execute Stage. Then forward data from the execute stage.
 2. **Forward from Memory Stage:** if the source register at the Execute stage is the same as the destination register in Memory Stage the forward data to the Execute stage, also if source register is the same at decode stage is same as destination register in memory stage then forward the data from Memory stage to Decode Stage.

4.8.6 Handling Control Hazard

Control hazard detects at the execute stage only.

- **Stalling:** Stall the decode stage and fetch stage for one clock cycle.
- **Flushing** when a control hazard occurs it means the new instruction will be at the targeted address and the rest of the two instructions in the decode stage and the fetch stages are useless there for we have to flush these stages to maintain the flow of instruction.

4.9 Programming and Tools

4.9.1 Tools

Front-end	Back-end
Intel's Quartus 21.1	Cadence's Genus
Mentors Graphic's Questa Sim	Cadence's Irun Sim

Table 4.3: Tools

4.9.2 Programming

1. **Hardware:** Used Verilog Hardware descriptive language as Front-end to design the bit-serial core.
2. **Software:** Used C and Python as Back-end to run the simulation and various multiple script such as for dumping the core results.

CHAPTER 5

Verification

The verification process involves confirming that the microprocessor core design meets its specification and functions correctly. All the instructions are working properly, and no hazards occurs. The design meets the timing requirements. And also write data and store the data is at the Right time. Verification is a critical step in microprocessor design to ensure that the final product behaves as intended and is free from errors or bugs.

To verify our Design we use the Design Verification Environment ever made bu Google named **GOOGLE DV**. We integrate our design with the verification environment and compare the core result with the predicted result of the verification environment.

5.0.1 GOOGLE DV Instruction Generator

Instruction generator is used to make the different combinations of thousands of instructions. That will pass to the core and the **Golden Reference**. In our case, we use six different types of test instructions.

1. **Basic Arithmetic Test:** contains basic arithmetic tests in the manner that hazards will occur because only then we can test our design on different dependencies.
2. **Jump Stress:** as shown by a name its literally a stress, containing the different types in unconditional and conditional jumps along with the other instructions.
3. **Loop Stress:** it contains the instructions that form the loop.
4. **Memory Stress:** contains the load and store instructions along with the jump and other instructions.

5. **Random Instructions:** contains random instruction by combining all the instructions with a little touch of jump instructions.
6. **Random Jump Instructions:** contain random instructions with a high number of jump instructions.

All tests include ten (10) seeds. Each seeds contain a minimum amount of eighth thousand instructions. And in result, we test approximately six lac instructions which contains a high number of dependencies.

5.0.2 Design Under Test (DUT)

includes the core that will be verified and a Tracer IP.

- **Core:** core is our case is a bit-serial microprocessor.
- **Tracer IP** use to dump write-back data, address of destination address, instruction address and instructions for comparison with the expected result.

5.0.3 Spike

it is a RISC-V ISA Simulator and in our case, it's our Golden Reference. All the instructions that pass to the core also pass to the spike and it gave us the correct expected result that will be used for comparison with the core result.

5.0.4 Python Script

used to compare the results given by the core and the golden reference. As we can see the core results are for 34 time its just for testing the functionality in actual it only write for once and python scripts takes it only for once to compares the instructions, address, write back data, and the destination register address of core with the spike. If all things are matched pythons script gave us the message **PASS** else **FAIL**.

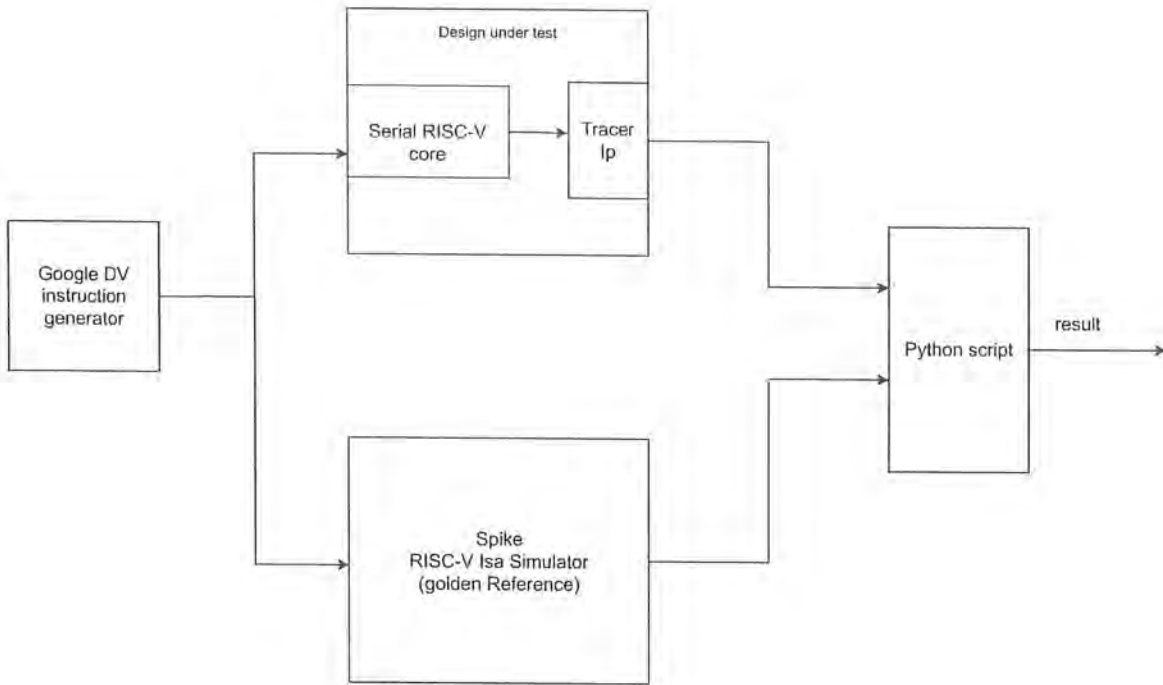


Figure 5.1: Verification Environment

```

14 core 0: 0x80000008 (0x00628263) beq    t0, t1, pc + 4
15 core 0: 0x80000008 (0x00628263)
16 core 0: 3 0x80000008 (0x00628263)
17 core 0: 0x8000000c (0x00000117) auipc  sp, 0x0
18 core 0: 3 0x8000000c (0x00000117) x 2 0x8000000c
19 core 0: 0x80000010 (0x00c10113) addi   sp, sp, 12
20 core 0: 3 0x80000010 (0x00c10113) x 2 0x80000018
21 core 0: 0x80000014 (0x00010067) jr     sp
22 core 0: 3 0x80000014 (0x00010067)
23 core 0: 0x80000018 (0x40000a37) lui    s4, 0x40000
24 core 0: 3 0x80000018 (0x40000a37) x20 0x40000000
25 core 0: 0x8000001c (0x100a0a13) addi   s4, s4, 256
26 core 0: 3 0x8000001c (0x100a0a13) x20 0x40000100
27 core 0: 0x80000020 (0x301a1073) csrw   misa, s4
28 core 0: 3 0x80000020 (0x301a1073) c769_misa 0x40140100
29 core 0: 0x80000024 (0x0002a817) auipc  a6, 0x2a
30 core 0: 3 0x80000024 (0x0002a817) x16 0x8002a024
31 core 0: 0x80000028 (0xcc080813) addi   a6, a6, -832
32 core 0: 3 0x80000028 (0xcc080813) x16 0x80029ce4
33 core 0: 0x8000002c (0x00007a17) auipc  s4, 0x7
34 core 0: 3 0x8000002c (0x00007a17) x20 0x8000702c
35 core 0: 0x80000030 (0x2d0a0a13) addi   s4, s4, 720
36 core 0: 3 0x80000030 (0x2d0a0a13) x20 0x800072fc
    
```

Figure 5.2: Instruction Generator

```

1 pc,instr,gpr,csr,binary,mode,instr_str,operand,pad
2 80000000,,t0:00000000,,f14022f3,3,"csrr    t0, mhartid",,
3 80000004,,t1:00000000,,00000313,3,"li      t1, 0",,,
4 8000000c,,sp:8000000c,,00000117,3,"auipc  sp, 0x0",,,
5 80000010,,sp:80000018,,00c10113,3,"addi   sp, sp, 12",,,
6 80000018,,s4:40000000,,40000a37,3,"lui    s4, 0x40000",,,
7 8000001c,,s4:40000100,,100a0a13,3,"addi   s4, s4, 256",,,
8 80000024,,a6:8002a024,,0002a817,3,"auipc  a6, 0x2a",,,
9 80000028,,a6:80029ce4,,cc080813,3,"addi   a6, a6, -832",,,
10 8000002c,,s4:8000702c,,00007a17,3,"auipc  s4, 0x7",,,
11 80000030,,s4:800072fc,,2d0a0a13,3,"addi   s4, s4, 720",,,
12 80000034,,s4:800072fc,,000a6a13,3,"ori    s4, s4, 0",,,
13 8000003c,,s4:8000003c,,00000a17,3,"auipc  s4, 0x0",,,
14 80000040,,s4:80000064,,028a0a13,3,"addi   s4, s4, 40",,,
15 8000004c,,s4:00002000,,00002a37,3,"lui    s4, 0x2",,,
16 80000050,,s4:00001800,,800a0a13,3,"addi   s4, s4, -2048",,,
17 80000058..s4:00000000..00000a13.3."li    s4. 0"..

```

Figure 5.3: Spike Result (Golden Reference)

```

76 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
77 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
78 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
79 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
80 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
81 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
82 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
83 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
84 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
85 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
86 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
87 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
88 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
89 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
90 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
91 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
92 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
93 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
94 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
95 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
96 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
97 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
98 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
99 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
100 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
101 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
102 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
103 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
104 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
105 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
106 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
107 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
108 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",
109 80000018,lui,s4:40000000,,40000a37,, "lui    x20,0x40000", "s4,0x40000",

```

Figure 5.4: Bit-Serial Core Result

CHAPTER 6

Analysis And Results

In this chapter, we will discuss the findings of the implementation discussed in chapter four. And will discuss the PPA(Power Performance Area) graph.

6.1 FPGA Implementation

Test our core on FPGA to validate it working on hardware. Implementation was performed on Intel DE1 SoC FPGA board on 50Mhz frequency and displayed the results on its HEX display. The FPGA consumed 2301 LUTS **Lookup table** used to implement all the combinational logic of the core. And Total Number of registers was 2284 used to implement the sequential logic. The number of luts and registers were high due to pipeline fashion. The maximum achievable frequency was 600MHz but FPGA is only capable of 50MHz therefor the entioned frequency in the table is 50MHz.

Decimal numbers displayed on the HEX display of FPGA is the half part of write back stage instruction due to only six HEX display we first display the six msbs of instruction and next cycle we display the remaining bits. This verified that each stage from fetch to write back all are completely synthesizable because the instruction only reaches at write abck stage if it process through all stages in the hardware.

Operating Frequency	Number of LUTS	Number of Registers
50MHz	2301	2284

Table 6.1: LUTS and Register Utilization in FPGA

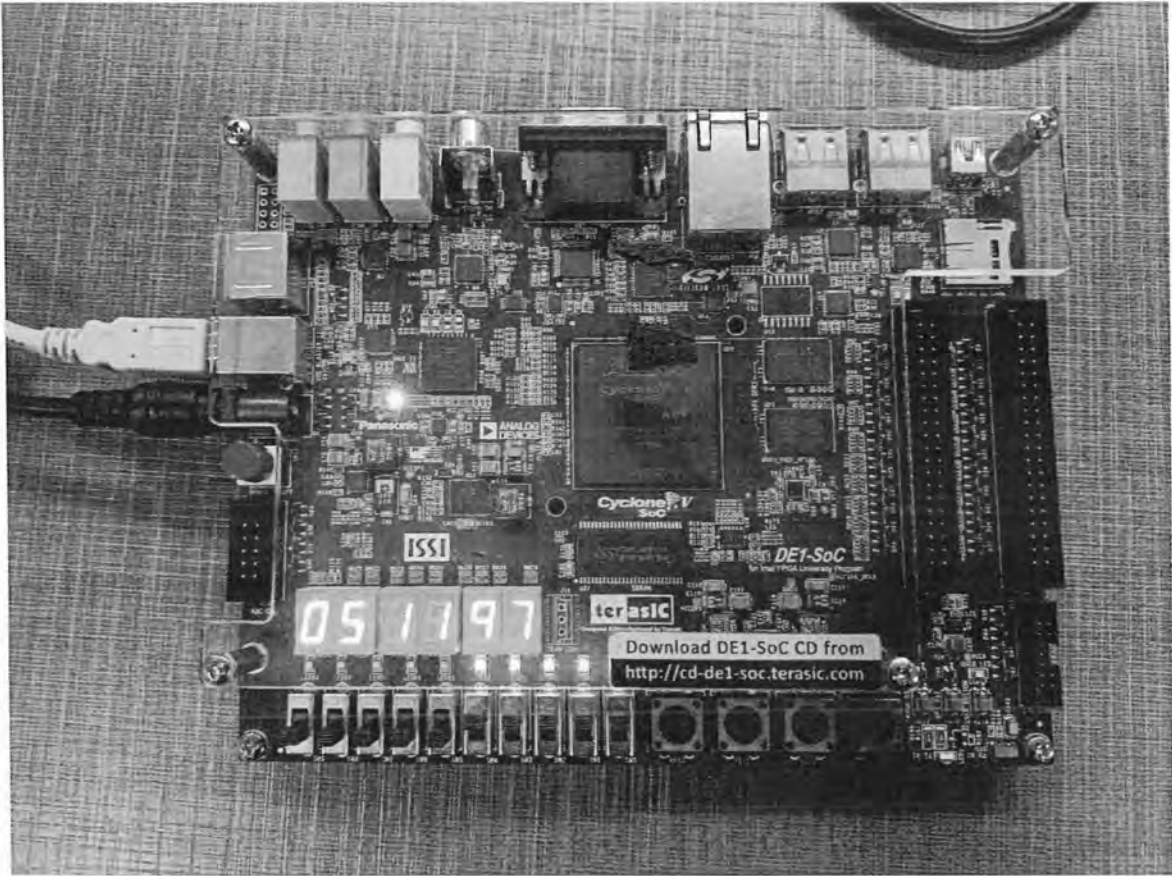


Figure 6.1: FPGA Demo

6.2 UART Test

UART (Universal Asynchronous Receiver-Transmitter) hardware interface communication protocol used for serial communication between microprocessor, micro controller, and various other peripheral devices. WE perform this test to validate our core speed and time, and to validate the communication of microprocessors with peripherals and slave devices such micro-controller. For this, we write a C program which prints a message displayed on GUI and use the infinite loop to see wether its working or got some bug after some time. Then we use RISC-V GCC to compile that C program and convert it from C to a .hex file which contains the instructions in hexadecimal form. Then pass those instructions to core and from the core to UART with a specific display address. Also, we use basic interconnect between core and uart in which we specify some address based on which uart will get input and display the output.

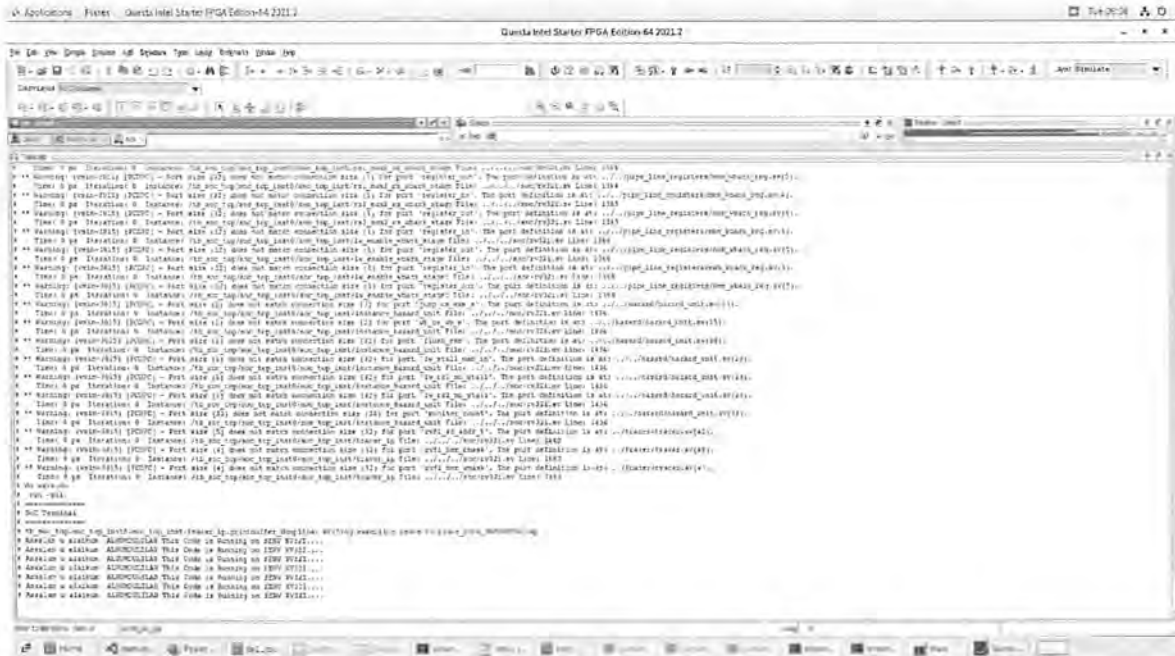


Figure 6.2: Uart Test

6.3 Power Performance Area (PPA)

Three critical factors of any digital hardware design, need to be balanced carefully. When a designer start to design any hardware the will set their goal in this term what they want and in contrast what they sacrifice.

6.3.1 Power

specified the application for what you are designing your hardware, either the application in low power or high power, we need to set this in starting otherwise at the end it is out of the hands to change the power. In our case, the goal was low power, and there we choose the bit-serial approach to reduce the overall power of design. and through this technique, we exceedingly reduce the power against the conventional parallel microprocessor.

6.3.2 Performance

Performance specifies the speed of the core. In contrast to conventional parallel microprocessors bit-serial achieve higher frequency, due to its reduced data width which means generally shorter propagation delay, and smaller area, but perform means data process in time. And it is obvious the performance of conventional parallel microprocessors are greater the the bit-serial microprocessor, because they can process the whole 32-bit vector in just one cycle, rather than bit-serial microprocessors which process 32-bit vectore in at least 32 cycle. but we can reduce its impact by using pipelining technique along with bit-serial.

6.3.3 Area

Bit-serial microprocessor cores have smaller areas then conventional parallel microprocessor designs due to their single bit data path and single-bit operating resources.

Frequency (MHz)	Power (W)	Area (μm^2)
50	$1.33843e^{-3}$	40665
100	$2.49871e^{-3}$	40672
200	$4.82859e^{-3}$	40753

Table 6.2: Frequency Power and Area

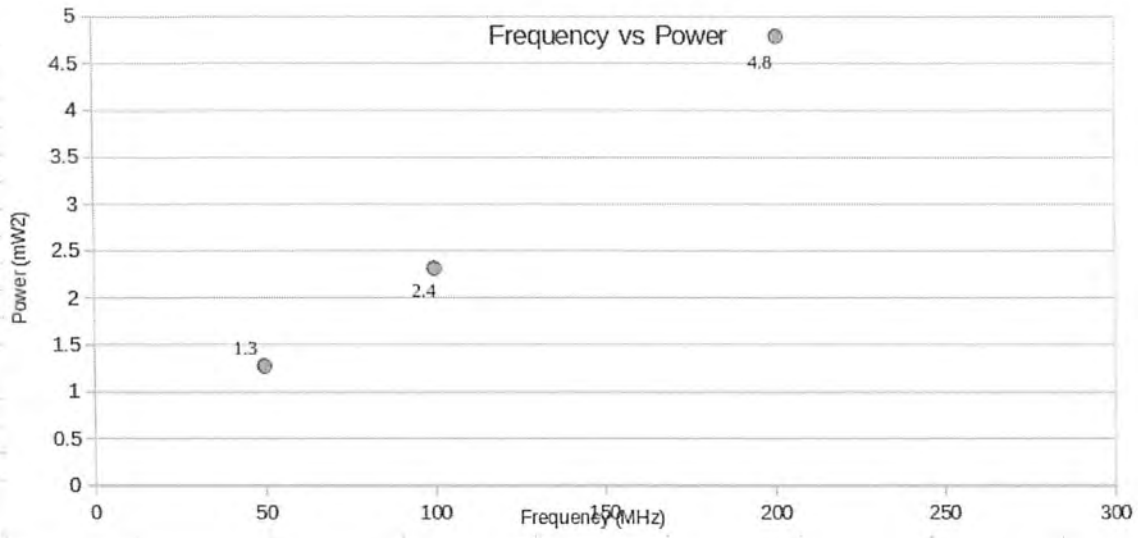


Figure 6.3: Fre vs Power

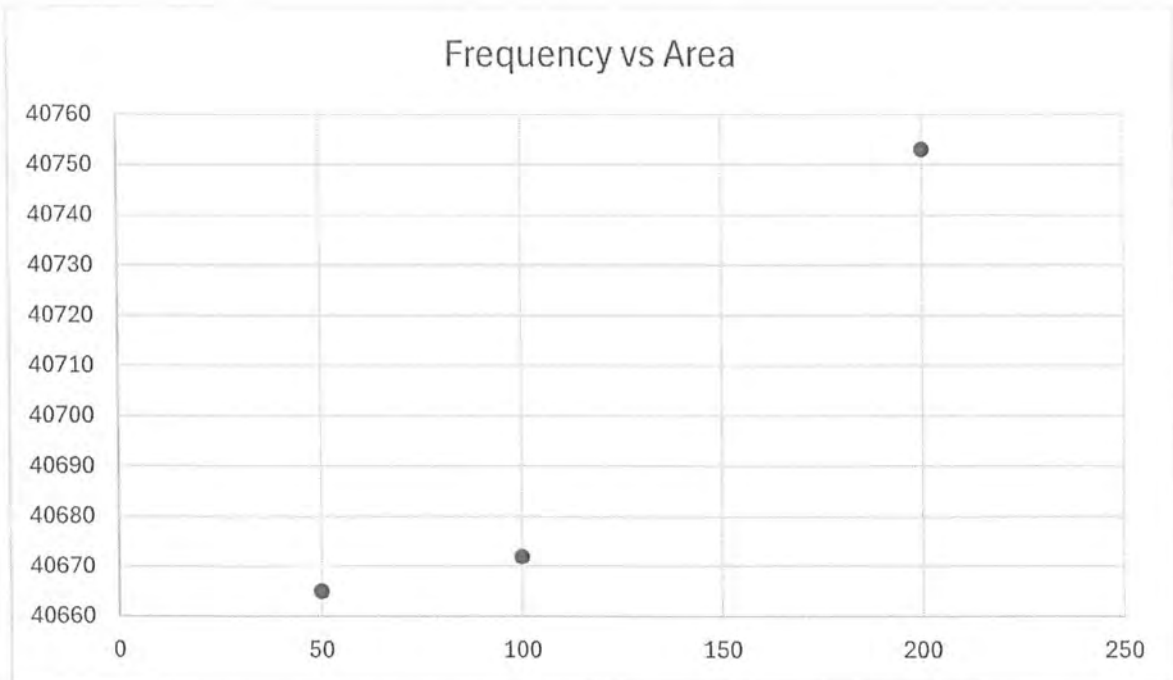


Figure 6.4: Fre vs Area

Conclusion and Future Works

7.1 Conclusion

In this thesis, we embarked on a journey to design, analyze, and evaluate the capabilities of a low-power bit-serial RISC-V microprocessor. The objective was to explore the potential benefits of bit-serial architectures in achieving efficient processing while minimizing power consumption. Through a rigorous research process, we have arrived at several significant findings that shed light on the strengths and limitations of such an approach. We implemented the RISC-V ISA (instruction Set Architecture) serially by making its data path from thirty-two-bit to one-bit wide. Using different types of shift register such as PIPO or PISO. Implemented most of the instructions using one addition block.

Our findings illuminate the potential advantages in terms of power efficiency, as the narrow data paths and reduced capacitance offer promising avenues for optimizing power consumption. We observed that the reduced complexity of the bit-serial data path design could indeed translate into improved energy efficiency, particularly in scenarios where parallelism is not a primary requirement.

However, our exploration also brought to light the inherent limitations of bit-serial architectures. The constrained parallelism, complex synchronization, and specialized design considerations necessitate a careful assessment of application suitability. We recognized that the applicability of bit-serial RISC-V microprocessors thrives in domains where the advantages of minimized area and energy efficiency align with the specific requirements of the workload.

In conclusion, this thesis underscores the value of investigating unconventional architectural paradigms such as bit-serial designs in the pursuit of energy-efficient microprocessors. Our work

offers a stepping stone towards a more nuanced comprehension of low-power microprocessor design and encourages the exploration of innovative pathways for achieving efficient processing capabilities.

7.2 Future Works

While this thesis has provided valuable insights into the design and analysis of a low-power bit-serial RISC-V microprocessor, there remain several avenues for further research and development in this domain. The following directions offer potential opportunities for expanding upon the current study and advancing the field:

7.2.1 Enhanced Architectural Features

Extend the architecture to incorporate the additional instructions or custom functional units optimized for bit-serial processing. Investigate how these enhancements impact both energy efficiency and performance for specific workloads.

7.2.2 Mixed Bit-Serial and Parallel Architectures

Explore hybrid architecture that combines both parallel and bit-serial design. Analyze how such mixed architecture can provide a balanced trade-off between power consumption and performance across a diverse range of work loads.

7.2.3 Multi core Design

Extend this single core up to multi-core design and analyze the performance effect for the specific workloads in-contrast with power consumption.

7.2.4 Advanced Verification and Testing

As we verified it using the Google Dv environment, now in the future try to verify it with the latest technologies such as UVM (Universal Verification Method) and find out the corner cases and their impact on design.

CHAPTER 8

References

1. Stangherlin, K. and Sachdev, M., 2022. Design and implementation of a secure RISC-V microprocessor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11), pp.1705-1715.
2. Serrano, R., Sarmiento, M., Duran, C., Nguyen, K.D., Hoang, T.T., Ishibashi, K. and Pham, C.K., 2021, October. A low-power low-area SoC based in RISC-V processor for IoT applications. In *2021 18th International SoC Design Conference (ISOC)* (pp. 375-376). IEEE.
3. Wu, B.C. and Wey, I.C., 2017. Parallel balanced-bit-serial design technique for ultra-low-voltage circuits with energy saving and area efficiency enhancement. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(1), pp.141-153.
4. Risikesh, R.K., Sinha, S. and Rao, N., 2021, December. Variable Bit-Precision Vector Extension for RISC-V Based Processors. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)* (pp. 114-121). IEEE.
5. Verma, G., 2020, October. Design and analysis of ALU for low power IOT centric processor architectures. In *2020 Global Conference on Wireless and Optical Technologies (GCWOT)* (pp. 1-5). IEEE.
6. Traber, A., Zaruba, F., Stucki, S., Pullini, A., Haugou, G., Flamand, E., Gurkaynak, F.K. and Benini, L., 2016, January. PULPino: A small single-core RISC-V SoC. In *3rd RISC-V Workshop*.
7. Li, Z., Huang, Y., Tian, L., Zhu, R., Xiao, S. and Yu, Z., 2021. A low-power asynchronous

CHAPTER 8: REFERENCES

- RISC-V processor with propagated timing constraints method. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(9), pp.3153-3157.
8. Patsidis, K., Konstantinou, D., Nicopoulos, C. and Dimitrakopoulos, G., 2018. A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed Instruction Set Extension. *Microprocessors and Microsystems*, 61, pp.1-10.
 9. Santos, D.A., Luza, L.M., Zeferino, C.A., Dilillo, L. and Melo, D.R., 2020, April. A low-cost fault-tolerant RISC-V processor for space systems. In *2020 15th Design and Technology of Integrated Systems in Nanoscale Era (DTIS)* (pp. 1-5). IEEE.
 10. Santos, D.A., Luza, L.M., Zeferino, C.A., Dilillo, L. and Melo, D.R., 2020, April. A low-cost fault-tolerant RISC-V processor for space systems. In *2020 15th Design and Technology of Integrated Systems in Nanoscale Era (DTIS)* (pp. 1-5). IEEE.
 11. Cilaro, A. and Mercogliano, S., 2022. Flexible privilege management for microcontroller-class RISC-V cores. *Microelectronics Reliability*, 137, p.114771.

CHAPTER 8: REFERENCES

Mphil_Thesis

ORIGINALITY REPORT

6% <i>W-2</i>	5%	.1%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Higher Education Commission Pakistan Student Paper	2%
2	www.cifr.edu.au Internet Source	1%
3	www.coursehero.com Internet Source	<1%
4	trace.tennessee.edu Internet Source	<1%
5	docplayer.net Internet Source	<1%
6	ir.mu.ac.ke:8080 Internet Source	<1%
7	Maria Markstedter. "Memory Access Instructions", Wiley, 2023 Publication	<1%
8	Submitted to University of The Gambia Student Paper	<1%



Figure 8.1: plagiarism report