DESIGN OF RISC-V ISA (BASE SPECIFICATION) COMPLIANT INSTRUCTION DECODE UNIT

A Project Report Submitted in Partial Fulfillment of the Requirement of the Degree

Of

BACHELOR OF TECHNOLOGY

In

ELECTRONICS AND COMMUNICATION ENGINEERING

by

NAVIN KUMAR SINGH (202000184) & SUBHANJANA GHOSH (202000008)
GROUP:07

Under the guidance of

Dr. BIKASH SHARMA
Associate Professor

SMIT SIKKIM MANIPAL UNIVERSITY

SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
SIKKIM MANIPAL INSTITUTE OF TECHNOLOGY
MAJITAR, EAST SIKKIM - 737136, May 2023

Certificate

This is to certify that the project report entitled “DESIGN OF RISC-V ISA (BASE SPECIFICATION) COMPLIANT INSTRUCTION DECODE UNIT” submitted by NAVIN KUMAR SINGH (202000184) & SUBHANJANA GHOSH (202000008) to Sikkim Manipal Institute Of Technology, Sikkim in partial fulfillment for the award of degree of Bachelor of Technology in Electronics And Communication Engineering, is a bonafide record of the project work carried out by them under my guidance and supervision during the academic session January– May, 2023.

Dr. Bikash Sharma
Associate Professor
Dept. Of Electronics and Communication
Sikkim Manipal Institute of Technology

Prof. (Dr.) Sourav Dhar
Head of the Department
Dept. Of Electronics and Communication
Sikkim Manipal Institute of Technology

Abstract

The aim of this project is to use Verilog to create an instruction decoder that complies with the RISC-V ISA (base specification). Due to its adaptability to different computer systems and versatility, the RISC-V ISA instruction set architecture has grown in favor in recent years. Any processor must have an instruction decode unit that is in charge of decoding the instructions fetched from memory and translating them into signals that the execution unit can understand.

Implementing the logic required to decode the RISC-V instructions, as well as the control signals, essential to carry out arithmetic, logical, and memory operations, is a key component of the instruction decode unit's design. Additionally, the project entails testing the design's accuracy by replicating it with test scenarios and comparing the results with the expected outcome.

For this project, Navin Kumar Singh and Subhanjana Ghosh worked together to make use of their knowledge of Verilog and computer architecture. A working instruction decode unit that may be incorporated into a RISC-V processor will be produced if this project is completed successfully, adding to the expanding body of open-source hardware designs.

Acknowledgement

We would like to extend our sincere gratitude to everyone who helped this project be completed successfully. We want to start by expressing our gratitude to Sikkim Manipal Institute of Technology, our college, for providing us with the tools and facilities needed to complete this project.

We would like to express our sincere gratitude to Dr. Bikash Sharma, Edwin Joy and Dr. Pragnajit Dutta who served as our project's mentor, for his invaluable advice and support. His knowledge of electronics and communication engineering has been extremely helpful to us in developing our project and achieving our objectives.

We also want to express our gratitude to the entire faculty and HOD Dr. Sourav Dhar of the department of electronics and communication engineering for their support and inspiration during our academic careers. Our knowledge and expertise in the area of electronics and communication engineering have greatly benefited from their insightful opinions and constructive criticism.

Finally, we would like to thank our peers and friends for their encouragement and support during the project. We have been kept motivated and focused on the objectives of our project by their ongoing encouragement and feedback.

Navin Kumar Singh (202000184)

Subhanjana Ghosh (202000008)

Department of Electronics and Communication, B.Tech

Abbreviations

Abbreviations	Details
RISC	Reduced Instruction Set Computers
Shamt	Shift Amount
PC	Program Counter
ALU	Arithmetic Logic Unit
Imm	Immediate
func	function
CPU	Central Processing Unit
ISA	Instruction Set Architecture

List of figures

Table of contents

Chapter 1

Introduction

1.1 What is RISC?

RISC is the abbreviation for Reduced Instruction Set Computer. It is a kind of microprocessor architecture that prioritizes a condensed set of instructions with the intention of enhancing performance, lowering power consumption, and simplifying the design and production of the processor.

Instructions in a RISC architecture are created to be executed in a single clock cycle, leading to quicker execution times. In order to do this, both the number of instructions that the processor can carry out and the instructions themselves must be made simpler.

A modest number of general-purpose registers, which are used to store data and address information, are often present in RISC processors. The data is loaded and stored in memory according to requirement, and the instructions are created to direct the type of operation to be performed with the register data.

The RISC architecture has gained popularity in recent years as a result of its ease of use, effectiveness, and ability to be adapted to a variety of computing systems. It is utilized in a variety of gadgets, such as servers, embedded systems, smartphones, and tablets.

RISC-V is an open-source instruction set architecture (ISA) based on the RISC principles. It was developed at the University of California, Berkeley, and is now overseen by the RISC-V Foundation, a non-profit organization that promotes and supports the adoption and development of the RISC-V ISA.

1.2 Why RISC-V?

The creation of RISC-V was motivated by several factors, including:

Flexibility and Openness: The RISC-V ISA was intended to be flexible and open-source, meaning that anybody may use, alter, and distribute it without paying any licensing costs. This makes it a well-liked option for both commercial and academic research since it permits more freedom and creativity in processor design.

Scalability: The RISC-V ISA was intended to be very scalable, supporting various bit widths (such 32-bit, 64-bit, and 128-bit) and extensions that may be introduced as necessary. This enables designers to select exactly the guidelines and features they need for their application, leaving out unrequired functionalities.

Modularity: The RISC-V ISA was created to be modular, with multiple modules that may be coupled in a few different ways to construct processors with diverse capabilities. As a result, processors may be made to order and tailored to work best for tasks.

Innovation: By offering an adaptable and open architecture that can be customized and optimized for application, the RISC-V ISA was created to promote innovation in processor design. As a result, the RISC-V architecture is being improved and expanded by a growing community of academics and developers.

Standardization: The RISC-V ISA was planned to be a standard engineering that can be utilized over a wide run of applications, from low-power inserted gadgets to high-performance computing frameworks. This makes it less demanding for designers to form program and equipment that can be utilized over distinctive stages and gadgets.

By and large, the motivation behind RISC-V was to form an open and adaptable instruction set design that can bolster a wide variety of applications and energize development in processor plan.

1.3 History

The origins of RISC (Reduced Instruction Set Computing) can be traced back to the 1970s, when IBM, Stanford University, and University of California, Berkeley researchers first started experimenting with the idea of streamlining computer architecture by limiting the number of instructions that a processor could carry out. As a result, the first RISC processors were created in the 1980s and had a condensed instruction set, fewer addressing modes, and fewer registers.

Over time, RISC gained popularity as a method for designing processors, and numerous businesses created their own exclusive RISC architectures. But this resulted in fragmentation and incompatibility among various RISC processors.

As a result, in 2010, scientists at the University of California, Berkeley started creating the open-source RISC architecture known as RISC-V. The objective was to develop a free, open ISA that would not require any licensing fees or proprietary technologies and hence could be used and modified by anyone. Professor of computer science Krste Asanovi oversaw the RISC-V project, which also benefited from the work of academics from UC Berkeley and other institutions.

With a limited set of base instructions that may be expanded with unique instructions for certain applications or devices, the RISC-V ISA was created to be modular and flexible. Its open-source nature made it more accessible and economical for developers, and this strategy made it more versatile and adaptable than proprietary RISC ISAs.

Since its launch, RISC-V has become more popular in academia, research, and business and has garnered the support of organizations like Google, NVIDIA, and Western Digital. In order to encourage and assist the use and advancement of the RISC-V ISA, the RISC-V Foundation was created in 2015. Since then, it has grown to include more than 500 member businesses and organizations.

Today, RISC-V is becoming more popular across a range of sectors, including data centers, high-performance computing and embedded systems. Its modularity and flexibility make it suitable for use with customized hardware, and the fact that it is open-source makes it simpler for businesses to work together and share technology.

RISC-V's expanding ecosystem of tools and hardware implementations is one of the technology's main benefits. The creation of hardware and software solutions that facilitate the adoption and integration of RISC-V into various computing systems is supported by the RISC-V Foundation. For instance, RISC-V compilers, simulators, and development boards are now available, making it simpler for programmers to begin using the ISA.

Support for vector processing is another perk of RISC-V. For tasks like multimedia processing, machine learning and scientific computing, RISC-V supports vector processing instructions. As a result, RISC-V-based vector processors, including those in the intelligence family of CPUs, have been developed.

It is anticipated that RISC-V will have an effect on the semiconductor market and computing in general as it continues to gain popularity. Companies may develop and collaborate more easily thanks to its open-source nature and expanding ecosystem of tools and implementations, which may one day result in new RISC-V applications and use cases.

Chapter 2

RISC-V ISA

2.1 RV-32I Instruction Formats

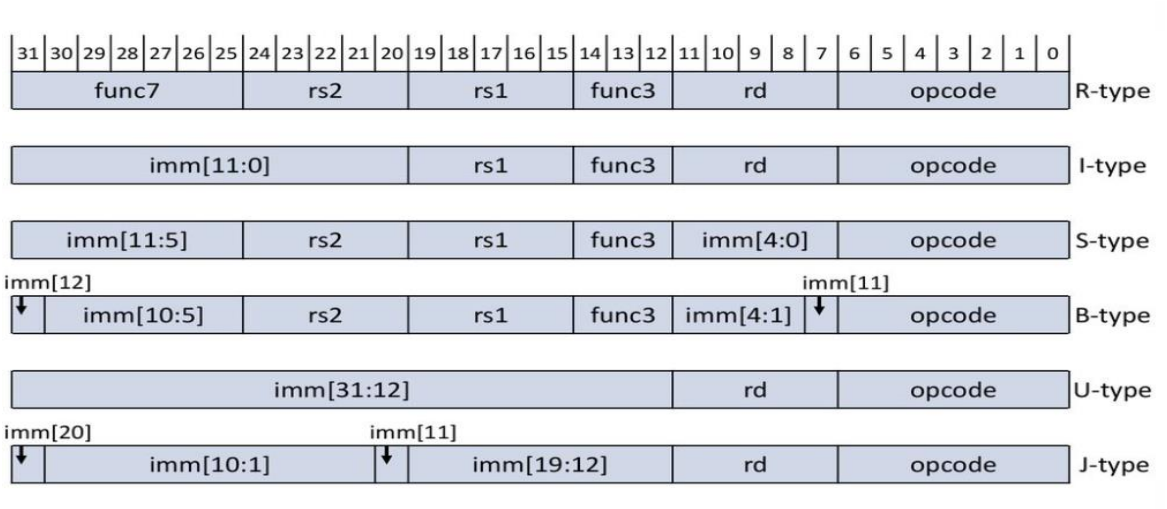


Fig2.1: RV-32I Instruction Formats
(https://slideplayer.com/slide/16185058/)

RV-32I Instructions provide following fields:

- **opcode**
  gives a general classification of the instruction and determines which of the remaining fields are needed, and how they are encoded in the remaining instruction bits.

- **functionfield (funct3/funct7)**
  specifies the exact function performed by the instruction, if not fully specified by the opcode.

- **rs1/rs2**
  identifying the register(s) in the register file containing the source operand values on which the instruction operates.

- **rd**
  identifying the register into which the instruction's result is to be written.

- **immediate**
  value contained within the instruction bits themselves. This value may provide an

offset for indexing into memory or a value upon which to operate (in place of the register value indexed by rs2).

All instructions are 32 bits in length. The R-type encoding provides a general layout of the instruction fields used by all instruction types. R-type instructions have no immediate value. Other instruction types use a subset of the R-type fields and provide an immediate value in the remaining bits. A 32 bits implementation of RISC-V supports 32 registers, each 32 bits wide.

2.2 RISC-V Pipelining

2.2.1 Stages of Execution

A RISC-V pipeline processor is a CPU architecture that processes instructions in a sequential manner, where each instruction passes through five stages of execution. The five stages are **Instruction Fetch (IF)**, **Instruction Decode (ID)**, **Execute (EX)**, **Memory (MEM)**, and **Writeback (WB)**. In this document, the high-level design of a 5 Stage RISC-V Pipeline Processor will be discussed in detail.

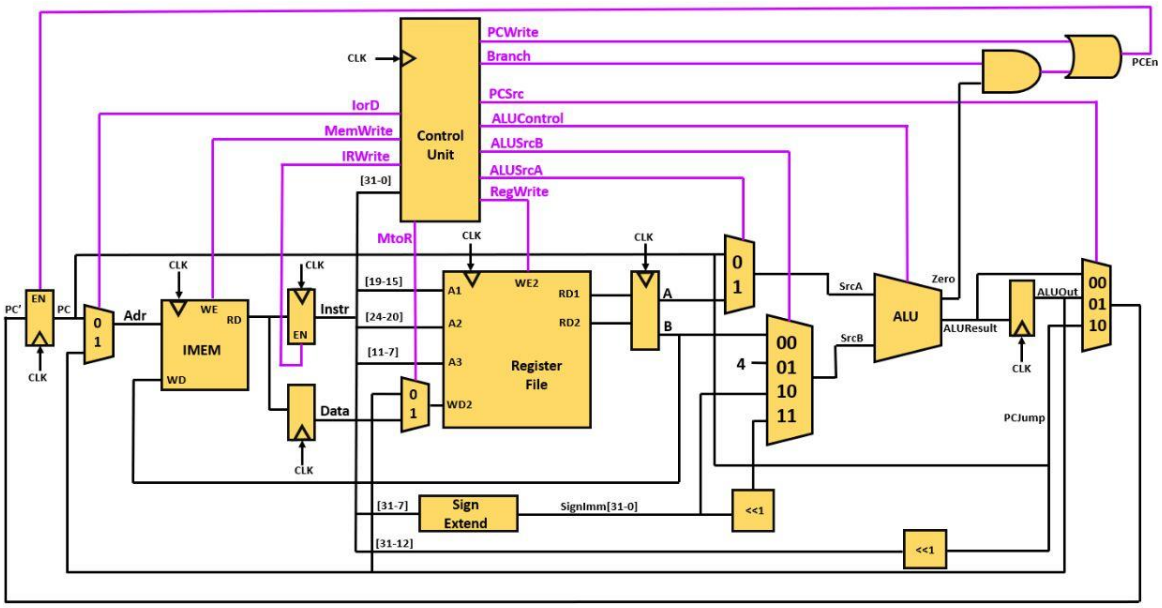


Fig2.2: Block Diagram of CPU Multi-Cycle Pipeline

1. **Instruction Fetch (IF) Stage**: The first stage of the pipeline is the instruction fetch (IF) stage. This stage fetches the instruction from memory and stores it in the instruction register (IR). The program counter (PC) is incremented by four after each instruction fetch.

2. **Instruction Decode (ID) Stage**: The second stage is the instruction decode (ID)

stage. This stage decodes the instruction and extracts the operands from the registers. It then sends the operands to the next stage of execution.

3. **Execute (EX) Stage**: The third stage is the execute (EX) stage. This stage performs the operation specified by the instruction. For example, if the instruction is an add operation, the EX stage will perform the addition operation on the operands.

4. **Memory (MEM) Stage**: The fourth stage is the memory (MEM) stage. This stage accesses memory to read or write data. For example, if the instruction is a load operation, the MEM stage will read data from memory and store it in a register.

5. **Writeback (WB) Stage**: The fifth and final stage is the writeback (WB) stage. This stage writes the results of the operation back to the register file. For example, if the instruction is an add operation, the WB stage will write the result of the addition back to a register.

2.2.2 Step-by-Step Processes of Each Stage

Instruction Fetch (IF) Stage:

- PC register holds the address of the next instruction to be fetched.
- The instruction cache stores the instructions to be fetched.
- A multiplexer selects the next instruction to be fetched based on the current value of the PC register.
- The instruction is fetched from the instruction cache and stored in the instruction register (IR).
- The PC register is incremented by four.

Instruction Decode (ID) Stage:

- The instruction register (IR) holds the fetched instruction.
- The instruction is decoded to determine the operation to be performed.
- The operands are extracted from the register file.
- The operand values are sent to the execute (EX) stage.

Execute (EX) Stage:

- The operation specified by the instruction is performed.
- The result of the operation is sent to the memory (MEM) stage.

Memory (MEM) Stage:

- The memory address is computed based on the result of the execute (EX) stage.
- The data is read from or written to memory.

Writeback (WB) Stage:

- The result of the operation is written back to the register file.

2.3 RISC-V Instruction Details

Usage Template		Type	Description	Detailed Description
add	rd, rs1, rs2	R	Add	rd ← rs1 + rs2, pc ← pc+4
addi	rd, rs1, imm	I	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4
and	rd, rs1, rs2	R	And	rd ← rs1 & rs2, pc ← pc+4
andi	rd, rs1, imm	I	And Immediate	rd ← rs1 & imm_i, pc ← pc+4
auipc	rd, imm	U	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4
beq	rs1, rs2, pcrel_13	B	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)
bge	rs1, rs2, pcrel_13	B	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
bgeu	rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
blt	rs1, rs2, pcrel_13	B	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bltu	rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bne	rs1, rs2, pcrel_13	B	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)
jal	rd, pcrel_21	J	Jump And Link	rd ← pc+4, pc ← pc+imm_j
jalr	rd, imm(rs1)	I	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i)&~1
lb	rd, imm(rs1)	I	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4
lbu	rd, imm(rs1)	I	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4
lh	rd, imm(rs1)	I	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4
lhu	rd, imm(rs1)	I	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4
lui	rd, imm	U	Load Upper Immediate	rd ← imm_u, pc ← pc+4
lw	rd, imm(rs1)	I	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4
or	rd, rs1, rs2	R	Or	rd ← rs1 | rs2, pc ← pc+4
ori	rd, rs1, imm	I	Or Immediate	rd ← rs1 | imm_i, pc ← pc+4
sb	rs2, imm(rs1)	S	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4
sh	rs2, imm(rs1)	S	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4
sll	rd, rs1, rs2	R	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4
slli	rd, rs1, shamt	I	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4
slt	rd, rs1, rs2	R	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
slti	rd, rs1, imm	I	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltiu	rd, rs1, imm	I	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltu	rd, rs1, rs2	R	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
sra	rd, rs1, rs2	R	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srai	rd, rs1, shamt	I	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
srl	rd, rs1, rs2	R	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srli	rd, rs1, shamt	I	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
sub	rd, rs1, rs2	R	Subtract	rd ← rs1 - rs2, pc ← pc+4
sw	rs2, imm(rs1)	S	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4
xor	rd, rs1, rs2	R	Exclusive Or	rd ← rs1 ^ rs2, pc ← pc+4
xori	rd, rs1, imm	I	Exclusive Or Immediate	rd ← rs1 ^ imm_i, pc ← pc+4

Fig2.3: RISC-V Basic Instructions

(Releases · johnwinans/rvalp (github.com))

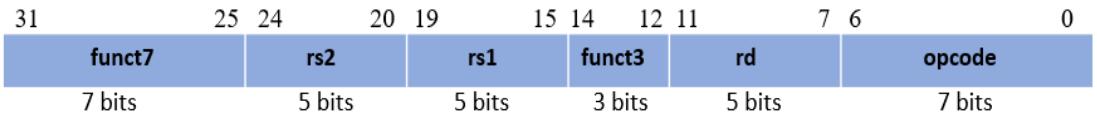
2.4 RISC-V Instructions' Logic

The Decode Unit is a part of the pipelining in the CPU Design. The instruction is fetched from the **Instruction Memory** block, read and the 32-bit instruction code is passed to the decoding block.

The address of the instruction is provided by the **Program Counter** which points to the next instruction that needs to be decoded and executed. The Decode Unit accesses the instructions pointed by the program counter and performs the decoding as directed by the control unit.

The detailed information of one example of each type of instruction is given below:

- **R-Type:**

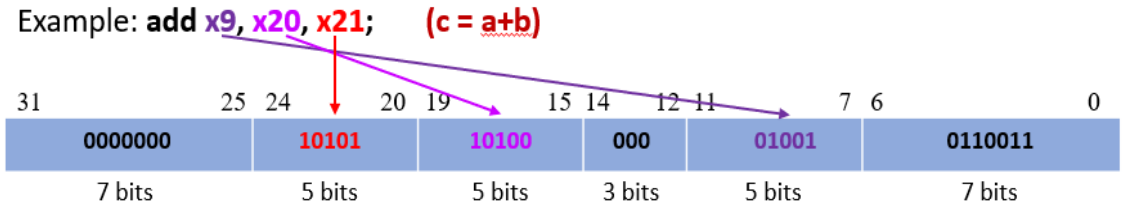


R-type of instructions have their unique **opcode[6-0]** as "0110011". Based on the values of the **funct7[31-25]** & **funct3[14-12],** the exact operation to be done is identified. A few are shown below.

0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type add
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type sub
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type sll
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type slt
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type sltu
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type xor
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type srl
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type sra
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type or
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type and

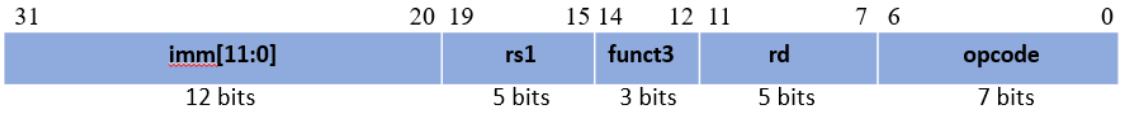
Fig2.4.1: R-Type Instructions
(Releases · johnwinans/rvalp (github.com))

After recognizing the exact operation to be done, the index of the source registers is decoded from the instruction fields i.e. here **rs2[24-20]** & **rs1[19-15]** contains the address of the source registers where the data is stored on which, the operation is to be done. Similarly, the destination register's index i.e. **rd[11-7]** is decoded and the result is to be stored in that address after the operation is completed.



After reading all the input, it is forwarded to **ALU** which does the operation on the data.

- **I-Type:**

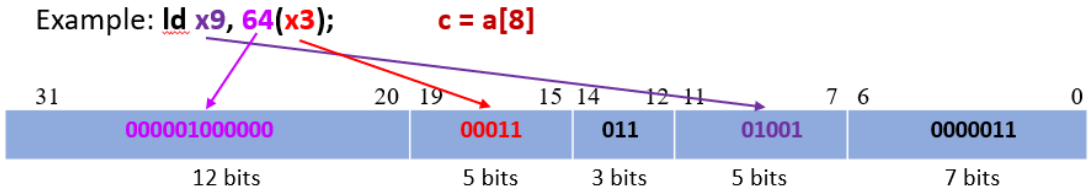


Here opcode values are different according to their respective purpose. Like for **loading instruction** it is "**0000011**", for **register jump** it is "**1100111**", for **immediate arithmetic**, **logical operation** and for **shift operation** it is "**0010011**". So similar to R-type we extract values in the **funct3[14-12]** and find the specific operation to be done.

imm[11:0]	rs1	0 0 0	rd	0 0 0 0 0 1 1	I-type	lb
imm[11:0]	rs1	0 0 1	rd	0 0 0 0 0 1 1	I-type	lh
imm[11:0]	rs1	0 1 0	rd	0 0 0 0 0 1 1	I-type	lw
imm[11:0]	rs1	1 0 0	rd	0 0 0 0 0 1 1	I-type	lbu
imm[11:0]	rs1	1 0 1	rd	0 0 0 0 0 1 1	I-type	lhu
imm[11:0]	rs1	0 0 0	rd	1 1 0 0 1 1 1	I-type	jalr
imm[11:0]	rs1	0 0 0	rd	0 0 1 0 0 1 1	I-type	addi
imm[11:0]	rs1	0 1 0	rd	0 0 1 0 0 1 1	I-type	slti
imm[11:0]	rs1	0 1 1	rd	0 0 1 0 0 1 1	I-type	sltiu
imm[11:0]	rs1	1 0 0	rd	0 0 1 0 0 1 1	I-type	xori
imm[11:0]	rs1	1 1 0	rd	0 0 1 0 0 1 1	I-type	ori
imm[11:0]	rs1	1 1 1	rd	0 0 1 0 0 1 1	I-type	andi
0 0 0 0 0 0 0 shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	I-type	slli
0 0 0 0 0 0 0 shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli
0 1 0 0 0 0 0 shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai

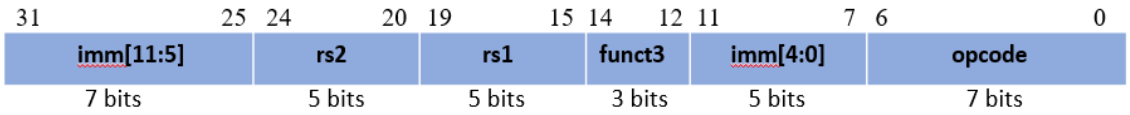
Fig2.4.2: I-Type Instructions
(Releases · johnwinans/rvalp (github.com))

Here, for each instruction the immediate input (data provided in the instruction itself), lies in the position [31-20]. In case of shift operation, it is of 5 bits size depicting the **shamt[24-20]** by which the data is to be shifted right or left.



Similar to R-type instructions, the opcode is decoded and type of immediate instruction is found out to analyze the immediate data accordingly. For the example instruction above, the data present in the address location specified by the sum of "address stored in register **x3** and the **offset**(64)" has to be copied to the register **x9**. So this specified address is calculated by the **ALU** block and the data present in that location is copied to the register **x9**.

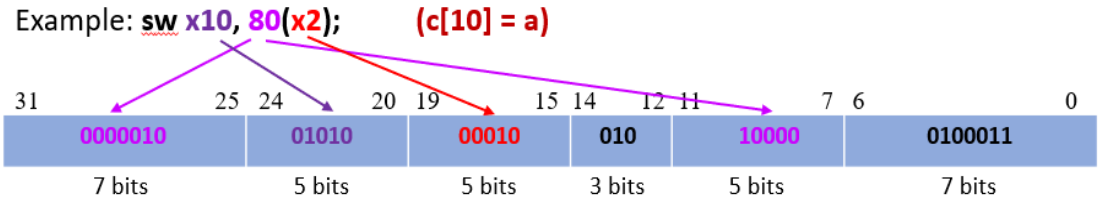
- **S-Type:**



S-type are used for storing the data in the immediate destination which is provided in the instruction itself. Its opcode is given by "**0100011**". Here again the opcode is decoded and discriminated according to its type. The immediate data part is divided into two blocks i.e. **{imm[31-25], imm[11-7]}** which can be accessed by concatenating the values stored in them.

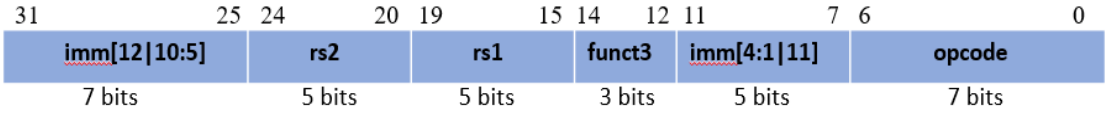
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sb
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	S-type	sh
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sw

Fig2.4.3: S-Type Instructions
(Releases · johnwinans/rvalp (github.com))



In the example given, the data stored in register **x10** is being copied to the address location pointed by the sum of the address stored in register **x2** and **offset**(80). So based on the **funct3**, the exact operation of store to be done is found and the data is stored to the pointed address accordingly. The **ALU** block calculates this destination address.

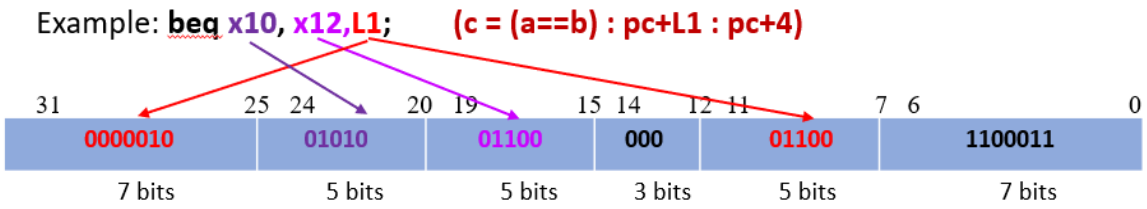
- **B-Type:**



These type of instructions are used for conditional jump. The order of the immediate data is present in the way represented above. These instructions have the unique opcode "**1100011**".
The blocks, like **rs1**, **rs2**, and **funct3** are decoded in the same manner as mentioned in the above types and are used for specific comparison to be made between operands. The immediate data is the address where the **PC** should point for the next instruction, so this offset value is provided back to the **program counter** for branching.

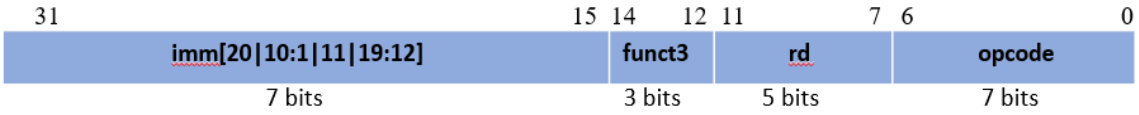
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	beq
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bne
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	blt
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bge
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bltu
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bgeu

Fig2.4.4: B-Type Instructions
(Releases · johnwinans/rvalp (github.com))



Here, in this example, if data stored in **x10** and **x12** registers are same then the program counter is required to branch to the instruction pointed by the immediate address (**L1**) given in the instruction itself. Based on the **funct3** the exact type of branch instruction is to be decoded and implemented.

- **J-Type:**



This type of instruction is used for unconditional jump. Here opcode value is "**1101111**". No **funct3** is present as this instruction has no different types. The order of the immediate data is present in the way represented above. This instruction will increment **program counter** by the immediate value present in the instruction. The result is stored in the address pointed by **initial PC value + 4**.

Fig2.4.5: J-Type Instructions

(Releases · johnwinans/rvalp (github.com))

So, we record the immediate data part and pass it on to the next stage for incrementing the PC to desired address i.e. **(PC+imm_data)**.

2.5 Problem Definition

The need for a component that can efficiently translate instructions from the RISC-V ISA into the control signals necessary to execute those instructions within a RISC-V processor is the issue that this project involving the construction of a RISC-V ISA based Decode Unit tries to answer.

The issue can be described in more detail as follows:

- Although RISC-V is an open and adaptable ISA that supports customization and

extension, it needs a specialized Decode Unit to convert its instruction set into the processor's internal functions.

- Decoding instructions from the RISC-V ISA and producing the relevant control signals that will be used to execute those instructions within the processor are tasks performed by the Decode Unit.
- The Decode Unit needs to be scalable and flexible in order to accommodate RISC-V ISA's many instruction formats, opcodes, and operands in addition to being built to enable the ISA's potential future additions and modifications.
- By creating a RISC-V ISA based Decode Unit that is tailored for a particular purpose, such as high-performance computing or low-power embedded systems, the project aims to solve the issue.

2.6 Proposed Solution

The following phases make up a suggested approach for the project that entails designing a RISC-V ISA-based Decode Unit:

Study and Analysis: The initial step is to thoroughly investigate and examine the RISC-V ISA and the unique application needs. Understanding the trade-offs between space, power, and performance as well as the instruction set's main requirements and restrictions were necessary for this.

Design Requirements: The next stage is to produce a thorough design specification for the Decode Unit based on the investigation and analysis. The Decode Unit must be able to support a variety of instruction formats, opcodes, and operands. In addition, the Decode Unit must be able to support the control signals and logic necessary to carry out those instructions.

Architecture Design: The next phase is to design the Decode Unit's general architecture, which includes all of its many parts, connections, and interfaces. Specifically, this entails disassembling the instruction set into its component pieces and developing logic circuits that can comprehend and carry out each instruction. The Decode Unit's hardware design will be created in the following stage using a hardware description language, such as Verilog. To do this, a simulation of the processor must be built and tested to guarantee that it performs as planned.

Verification and Validation: Testing the Decode Unit in various scenarios to make sure it operates correctly and dependably constitutes the final stage in the design verification and validation process. This may entail executing functional simulations, performing functional testing, and measuring performance metrics such as latency and throughput.

2.7 Expected Outcome

An improvement in the instruction decoding performance of the CPU is expected. The goal is to provide optimum use of all present resources to make the RISC V decoding unit more practically feasible.

Chapter 3

Project Methodologies

3.1 Explanation of Source Code

The "decode" Verilog module, which simulates a decoder for a computer processor, is the source code utilised in this instance. It requires a number of inputs, including the decoded **instruction (instruction**), **reset signal (reset),** and **clock signal (clk).** Additionally, it includes numerous outputs that correspond to different control signals used by the processor.

The code inside the module will be performed on the positive edge of the clock signal because it has a sequential behaviour set using the "**always @(posedge clk)**" block. The module initialises all the registers and signals to their default settings when the **reset** signal is active (**reset = 1**).

The stage of the instruction decoding process is identified using a case statement inside the else block based on the "**stage**" variable. The stages stand in for several phases of the decoding process involved in any instruction.

Stage 0 (Fetching):

- The instruction's **opcode, rs1, rs2, rd, funct3, and funct7** fields are extracted.
- The **ALU (Arithmetic Logic Unit)** and programme counter (**PC**)'s various control signals and registers are initialised.
- For R-type, I-type (load), and S-type instructions, the module switches to stage 2 (**Execution/MemAdr**) based on the opcode.

Stage 2 (Execution/MemAdr):

- The ALU control signal is set to specify the ALU operation to be carried out in accordance with the values of the opcode and funct7 fields.
- **Sources A and B** of the ALU are set up.
- The module moves on to stage 5 (**Writeback**), where the ALU operation is completed and the outcome is stored.

Stage 3 (MemRead):

- This stage is specifically for I-type (load) instructions.
- The **IorD (Instruction or Data memory)** control signal is set to indicate a memory read operation.

Stage 5 (Writeback):
Various activities are carried out by the module depending on the opcode:

- The **MtoR (Memory to Register)** signal is cleared and the **RegWrite (Register Write)** signal is set for R-type instructions.
- The **MtoR** signal and the **RegWrite** signal are both set for I-type (load) instructions.
- The **IorD (Instruction or Data memory)** and **MemWrite (Memory Write)** signals are set for S-type instructions.
- For the purpose of beginning the following instruction's decoding, the module returns to **stage 0 (Fetching).**

There are in total 6-7 type of instructions in **rv32i** version of RISC-V ISA from which 3 are implemented in this project:

3.1.1 R-Type Instruction Decoding:

The R-type instructions in RISC-V have a specific format. Each instruction consists of different bit fields that hold specific information.

These bit fields are **opcode**, **rs1 (source register 1), rs2 (source register 2), funct7 (additional function bits), rd (destination register),** and **funct3 (function bits).**

- After receiving an instruction as input, the module uses bitwise operations to extract the appropriate bit fields. For subsequent operations, a corresponding register is assigned to each bit field.

- . Based on the **funct3/funct7** parameters and the opcode, the module sets control signals. These control signals dictate how various processing components will behave.

- The **AluSrcA_reg** control signal is set to **1'b1**, indicating that a register should be used as the ALU's first input.

- The **AluSrcB_reg** control signal is set to **2'b00**, indicating that a register should serve as the second input to the ALU.

- The module chooses the appropriate ALU control value (**AluControl_reg**) based on the opcode and funct7 fields. To determine the appropriate control value, a nested case statement is used.

- The module updates the current stage to **stage = 5** after establishing the required control signals. This suggests the pipeline should relocate to the next stage of execution.

- In the last **stage (Writeback),** the module executes the following tasks:

- The ALU result should be read when the **MtoR_reg** control signal is set to **0**.

- When the **RegWrite_reg** control signal is set to **1**, the ALU result can be written to the target register.

3.1.2 Load-Type Instruction Decoding:

- An instruction passes through each stage of the pipeline when it enters the decode module. We'll focus on the steps connected to load instructions in this discussion.

- The module takes the relevant bit fields from the instruction during the fetching stage **(stage 0).** The **opcode field (bits [6:0])** will match the value **7'b0000011**, indicating a load instruction, for I-type instructions.

- The module conducts further decoding depending on the opcode field at the **Execution/MemAdr** stage (**stage 2**) of the process. The module decodes the particular **funct3 field (bits [14:12])** to identify the load operation type since it matches the value for load instructions.

- The module tells the ALU that the initial input should come from a register by setting the **AluSrcA_reg** control signal to **1'b1**. Additionally, it sets the **AluSrcB_reg** control signal to **2'b10**, designating that a register shall serve as the second input to the ALU.

- **Bits [31:20]** of the instruction are used to extract the immediate value (**sign-extended 12-bit immediate),** which is then put in the **Imm_reg** register.

- The module sets the **AluControl_reg** control signal to the appropriate value for the ALU operation based on the **funct3** field. This establishes the precise arithmetic or logical action that will be carried out while the load instruction is being executed.

- The module switches to the **MemRead** stage by updating the current stage to **stage = 3.**

- The module continues processing the load instruction in the **MemRead stage (stage 3).** To confirm that it corresponds to a load instruction, it once more verifies the opcode field.

- The **IorD_reg** control signal is set to **1'b1** in the case of load instructions, indicating that the memory location for the load operation should be determined from the ALU result.

- According to the prior general decoding description, the module then moves on to the **Writeback step (stage 5)** to finish executing the load instruction. The proper control signals are configured at this point i.e. **MtoR_reg = 1** and **Reg_Write =1** to carry out the tasks required by load instructions, such as reading data from memory and putting it to the destination register.

3.1.3 Store-Type Instruction Decoding:

- An instruction passes through each stage of the pipeline when it enters the decode module. We'll focus on the processes involved with store instructions in this discussion.

- The module takes the relevant bit fields from the instruction during the **fetching stage (stage 0**). The **opcode field (bits [6:0])** for **S-type instructions (store)** will match the value **7'b0100011**, designating a store instruction.

- The module conducts further decoding depending on the opcode field at the **Execution/MemAdr stage (stage 2)** of the process. The module continues to decode the specific **funct3 field (bits [14:12])** to ascertain the type of store operation because it matches the value for store instructions.

- The module tells the ALU that the initial input should come from a register by setting the **AluSrcA_reg** control signal to **1'b1**. Additionally, it sets the **AluSrcB_reg** control signal to **2'b10**, designating that a register shall serve as the second input to the ALU.

- **Bits [31:25]** and **[11:7]** of the instruction are used to derive the instant value **(sign-extended 12-bit immediate).** To determine the memory address for the store operation, the offset or displacement value will be added to the base address in these bits.

- The module sets the **AluControl_reg** control signal to the appropriate value for the ALU operation based on the **funct3** field. This establishes the precise arithmetic or logical operation that will be carried out while the store instruction is being executed.

- The module switches to the Writeback stage after updating the current stage to **stage = 5**.

- The module continues to handle the store command during the **Writeback** step (**stage 5).** To confirm that it corresponds to a store instruction, it once more verifies the opcode field.

- The **IorD_reg** control signal is set to **1'b1** in the case of store instructions, indicating that the memory location for the store operation should be determined from the ALU result.

- Additionally, the data from the source register should be written to memory at the created address, according to the setting of the **MemWrite_reg** control signal, which is **1'b1**.

- The module then returns to **stage = 0** to begin the pipeline for the following instruction.

3.2 FSM of Decode Unit

In order to verify the working of the design model, checking of the control signals for each stage is done. Thus, for a particular instruction, the control signals are verified using simulation of the hardware design logic using Verilog coding. For this project, simulation is done in Vivado Software. Two types of instructions have been considered for the verification- R-type and S-type.

FSM for Load & Store-type:

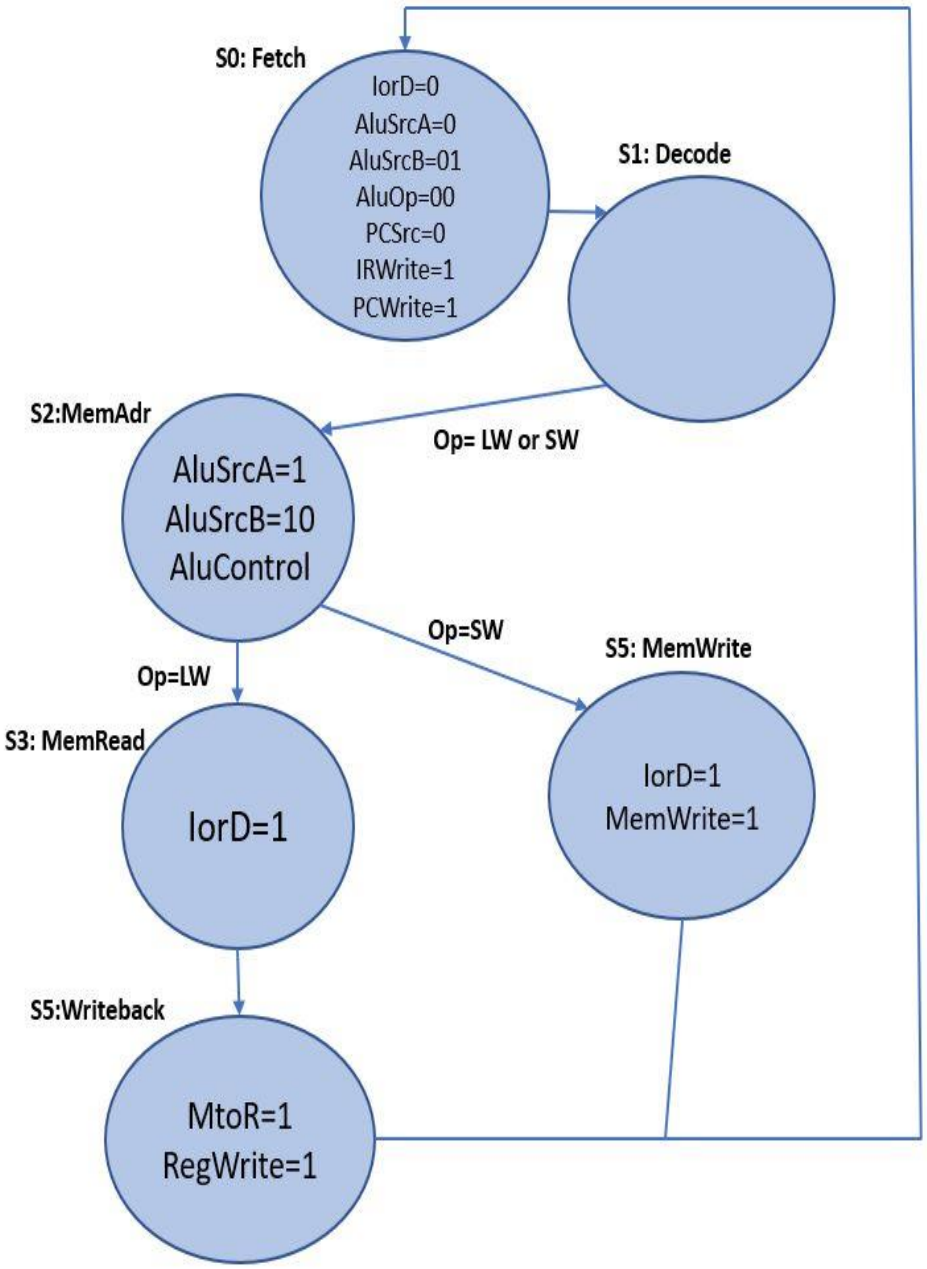


Fig3.2.1: FSM for Load & Store Type Instructions

FSM for R-type:

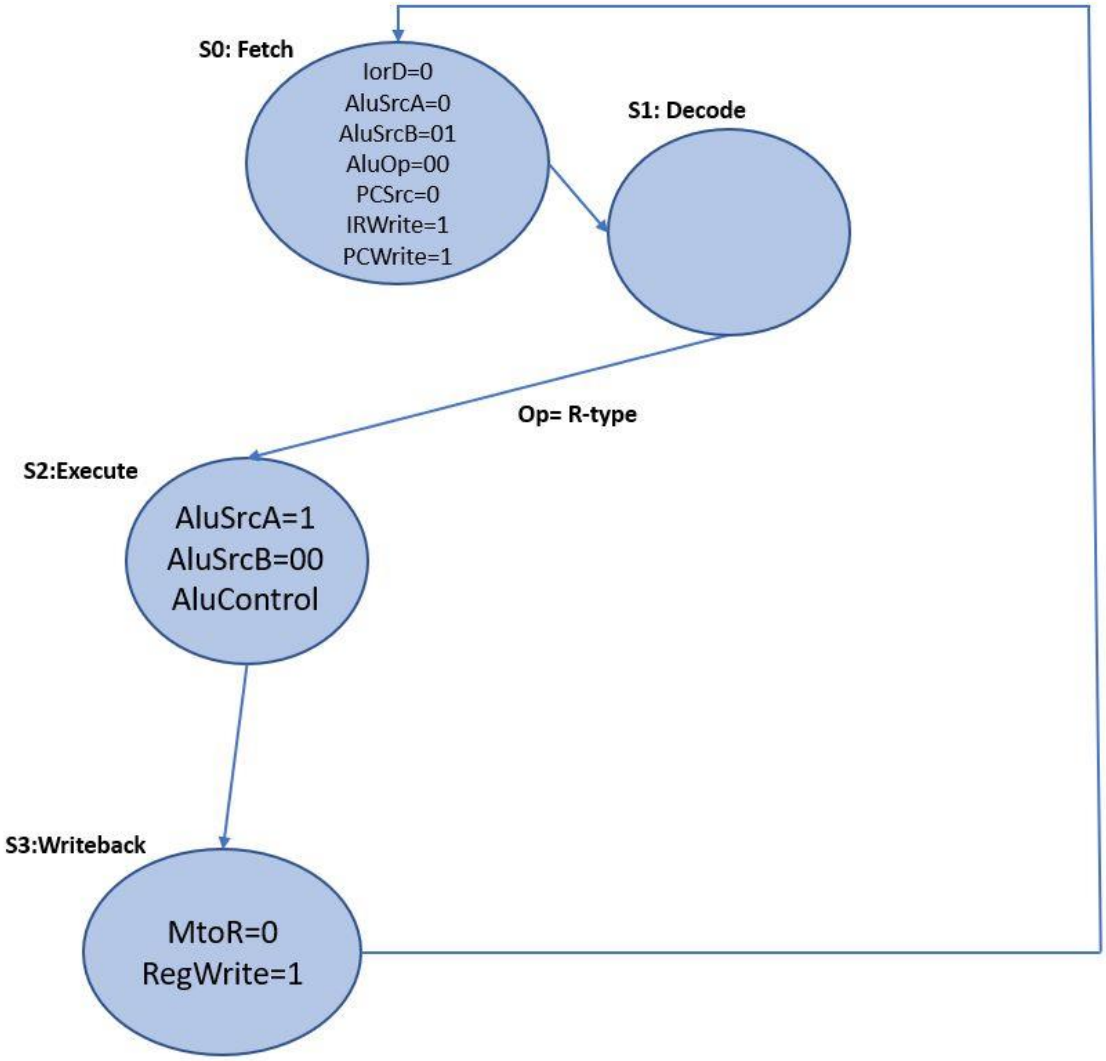


Fig3.2.2: FSM for R-Type Instructions

3.3 Simulation Output

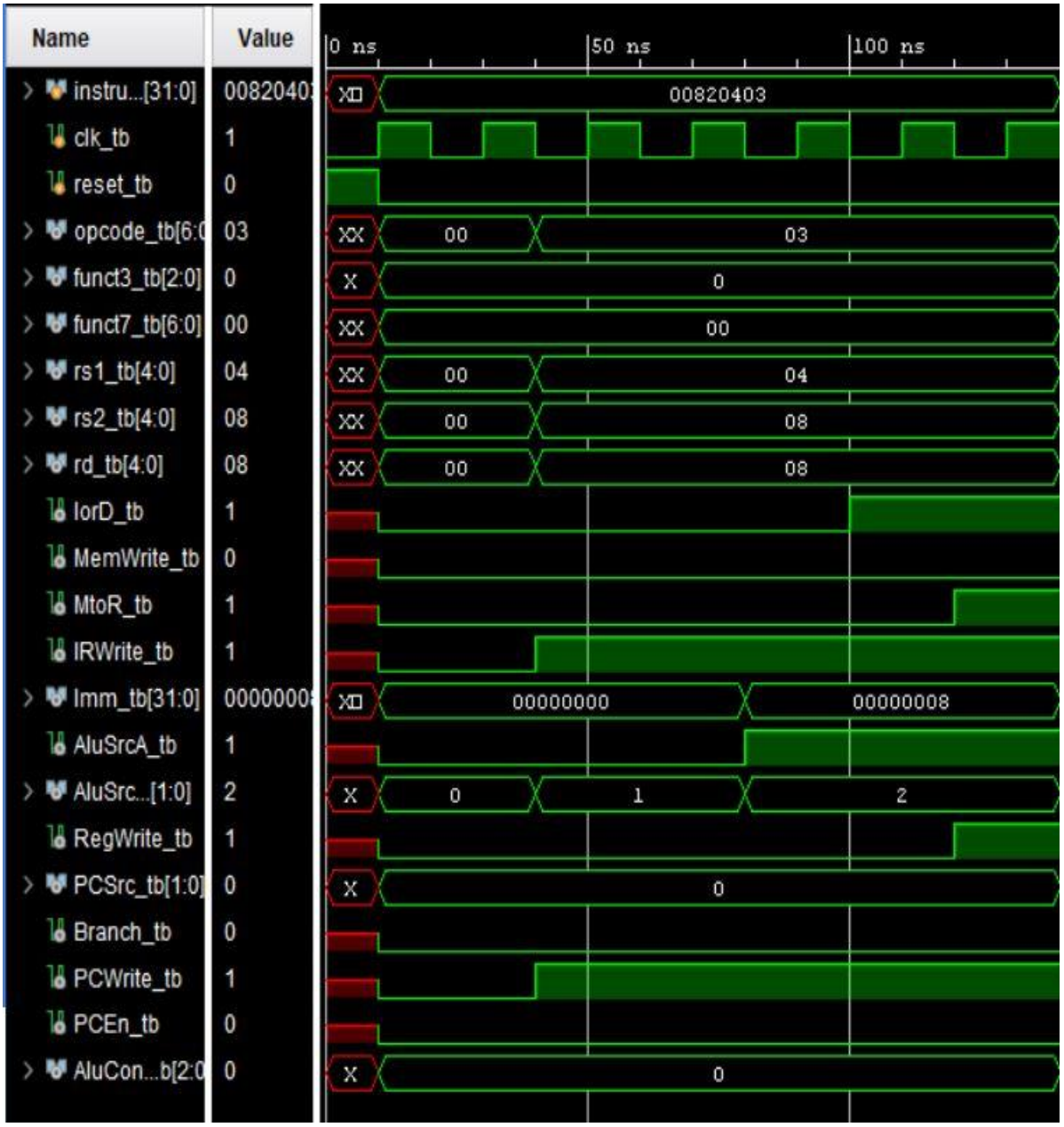


Fig3.3.1: Simulation output for lw x8, 8(x4)

Fig3.3.2: Simulation output for sw x0, 11(x5)

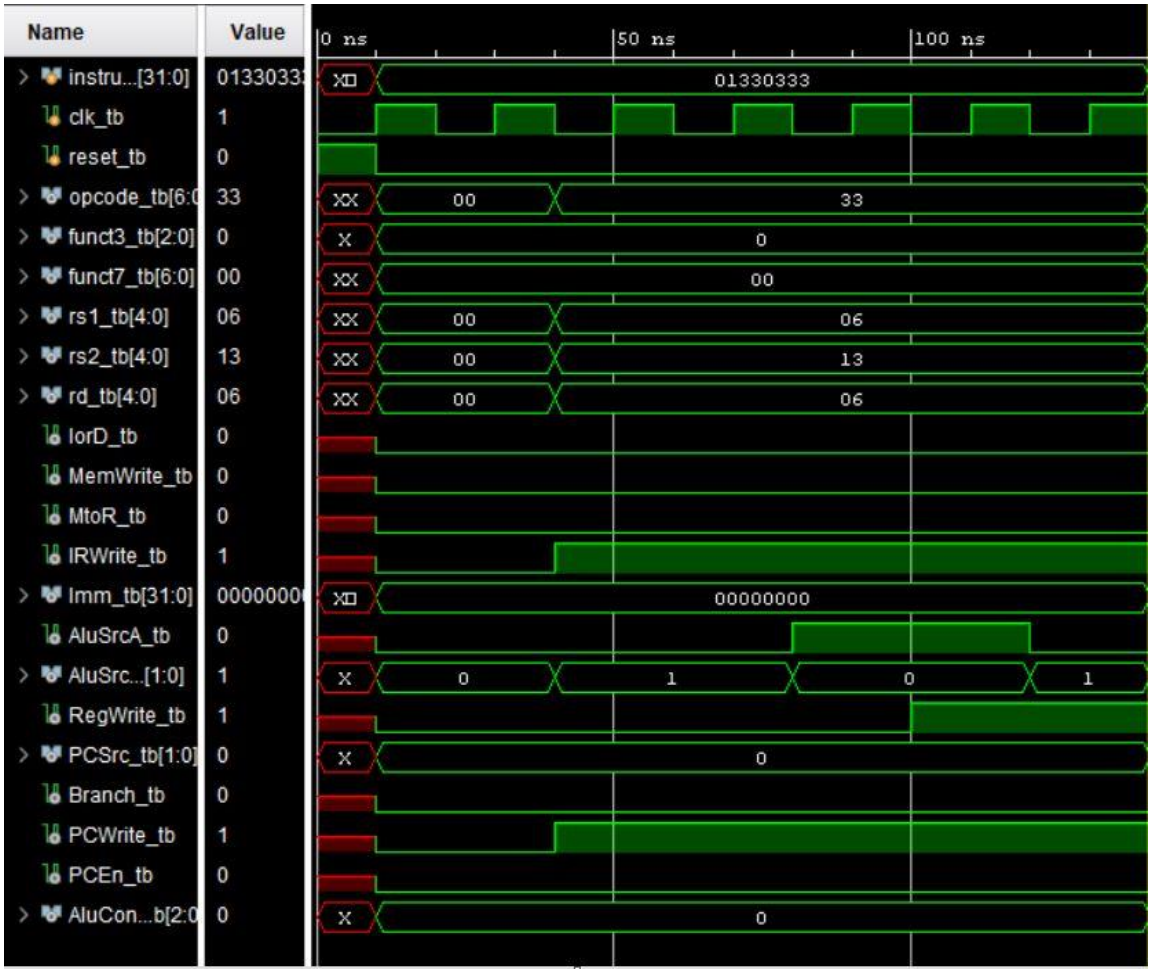


Fig3.3.3: Simulation output for add x6, x13, x6

Future Scope

The RISC-V decode unit's anticipated future uses include a number of potential upgrades and improvements. These include machine learning-based decoding, advanced debugging and tracing capabilities, support for expanded instruction sets, improved pipeline performance, enhanced microarchitecture support, power optimization approaches, and scalability and customization possibilities.

These developments coincide with the extensibility and openness of the RISC-V architecture and aim to enhance the decode unit's performance, power efficiency, adaptability, and versatility.

Plagiarism

Decode_Unit_Final

ORIGINALITY REPORT

12%	8%	4%	8%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

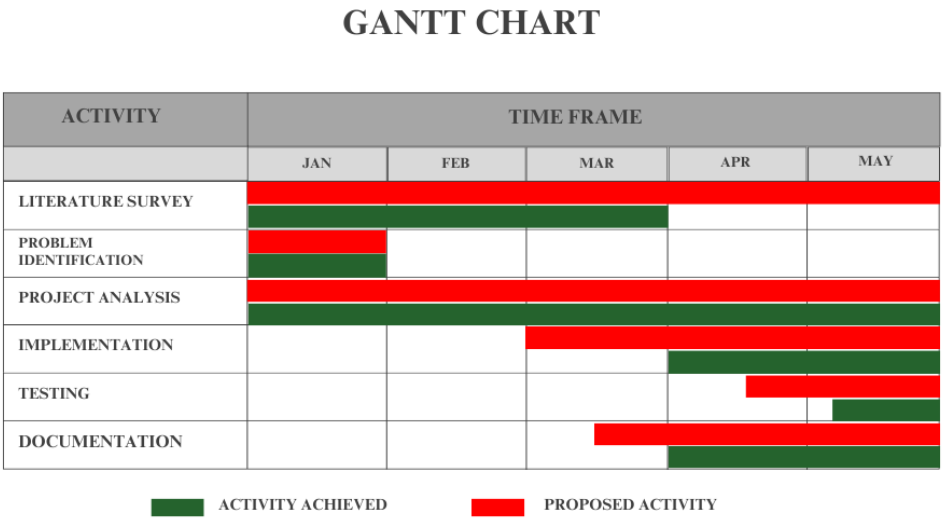
PRIMARY SOURCES

1	Submitted to Sikkim Manipal University Student Paper	1%
2	Submitted to University of Southampton Student Paper	1%
3	dokumen.pub Internet Source	1%
4	www.dixdecoeur.com Internet Source	1%
5	Submitted to Sogang University Student Paper	1%
6	jcst.ict.ac.cn Internet Source	1%
7	Submitted to Pathfinder Enterprises Student Paper	1%
8	dblp.uni-trier.de Internet Source	<1%
9	solutionsadda.in Internet Source	<1%

Literature Survey

Sl. No	Author	Title of the paper	Journal/Conference /Book name (in Details)	Relevance to the project
1.	Yuan-Hu Cheng, Li-Bo Huang, Yi-Jun Cui Sheng Ma, Yong-Wen Wang and Bing-Cai Sui	RV16: An ultra-low-cost embedded RISC-V processor core	Journal of Computer Science and Technology 37(6)	Understanding the different segments required in a CPU core
2.	Scott, Ian	Analysis on the Possibility of RISC-V Adoption	UC Merced Undergraduate Research Journal 12, no. 1	Features that must be present in RISC-V based hardware
3.	Jim Ledin	An introduction to the open source 32-bit and 64-bit RISC-V architecture, instruction set, and extensions	Article on Dzone website 2020	Deeper understanding of RV-32I architecture
4.	Hyeonguk Jang, Kyuseung Han, Sukho Lee, Jae-Jin Lee, Seung-Yeong Lee, Jae-Hyoung Lee, Woojoo Lee	Developing a Multicore Platform Utilizing Open RISC-V Cores	IEEE Access, vol. 9, pp. 120010-120023, 2021, doi: 10.1109/ACCESS.2021.3108475.	Multi-cycle processing implementation

Gantt Chart



References

[1] Yuan-Hu Cheng, Li-Bo Huang, Yi-Jun Cui, Sheng Ma, Yong-Wen Wang, and Bing-Cai Sui. RV16: *An Ultra-Low-Cost Embedded RISC-V Processor Core*[J]. Journal of Computer Science and Technology, 2022, 37(6): 1307-1319.

[2] Scott, Ian. "*Analysis on the Possibility of RISC-V Adoption*." *UC Merced Undergraduate Research Journal* 12, no. 1 (2020). https://doi.org/10.5070/m4121046641.

[3] H. Jang et al., "*Developing a Multicore Platform Utilizing Open RISC-V Cores*," in IEEE Access, vol. 9, pp. 120010-120023, 2021, doi: 10.1109/ACCESS.2021.3108475.

[4] GitHub.com/olofk/serv (last accessed on 13/05/2023)

[5] Github.com/sean-brandenburg/Verilog-Processor-RISC-V (last accessed on 13/05/2023)

[6] Computer Organization and Design RISC-V Edition: The Hardware Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)

[7] Digital Design and Computer Architecture, RISC-V Edition by David Harris

[8] https://dzone.com/articles/introduction-to-the-risc-v-architecture (last accessed on 20/03/2023)

[9] https://riscv.org/technical/specifications/ (last accessed on 15/05/2023)