Developing Extended ISA on RISC Based Processor

By

Lee Ang

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in Partial Fulfilment of the Requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JAN 2023

Developing Extended ISA on RISC Based Processor

By

Lee Ang

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in Partial Fulfilment of the Requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JAN 2023

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: DEVELOPING EXTENDED ISA ON RISC BASED PROCESSOR

Academic Session: MAY 2023

I LEE ANG

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

[illegible]

(Author's signature)

[illegible]

(Supervisor's signature)

Address:
50, Jalan Harmoni 5/5,
Taman Desa Harmoni,
81100 Johot Bahru, Johor

Ts. Ooi Joo On
Supervisor's name

Date: 20/4/2023

Date: 20/4/2023

Universiti Tunku Abdul Rahman			
Form Title : **Submission Sheet for FYP/Dissertation/Thesis**			
Form Number: **FM-IAD-004**	Rev No.: **0**	Effective Date: **21 JUNE 2011**	Page No.: **1 of 1**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
UNIVERSITI TUNKU ABDUL RAHMAN

Date: 20/4/2034__________

SUBMISSION OF FINAL YEAR PROJECT/DISSERTATION/THESIS

It is hereby certified that ___Lee Ang_____ (ID No: ___20ACB04056__) has completed this final year project entitled "__Developing Extended Isa On RISC Based Processor___" under the supervision of ____Ts. Ooi Joo On_______ from the Department ____Digital Economy Technology____, Faulty of ___Information and Communication Technology_______.

I understand that Univeristy will ipload softcopy of my final year project in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Your truly,

[illegible]

____
(Student Name)

DECLARATION OF ORIGINALITY

I declare that this report entitled "**DEVELOPING EXTENDED ISA ON RISC BASED PROCESSOR**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature	:	LeeAng
Name	:	Lee Ang
Date	:	18/4/23

ACKNOWLEDGEMENTS

First and foremost, I would like to appreciate my lecturer, Ts Dr Liew Soung Yue who is the Dean of Faculty of Information and Communication Technology, and my tutor, Mr Tou Jing Yi, for teaching me in subject which named Introduction to Inventive Problem Solving and Proposal Writing. They showed their profession of proposal writing to provide a guidance for this proposal.

Furthermore, I would like to thank my supervisor of this proposal, Ts Ooi Joo On who is an expert in computer organisation and architecture. He gave me various of topic that about RSIC-V processor for me to choose for this proposal and the following final year project, and also provided a lot of related to this topic documents for me refer to understand in this area. I also would like to appreciate my lecturer, Mr. Mok Kai Ming who teaching me in Digital System Design and Computer Organisation and Architecture, he has provided a lot of related knowledge to help me on processor design.

Lastly, I would like to show sincere appreciation to my family and friends. Their encourage and supporting help me to complete my proposal.

Thanks for all your supporting.

ABSTRACT

In this model era, the digital market is grown rapidly and great quantities of product required processor for their applications. Therefore, the demand of processor is continuously increased, especially RISC-V processors which are durable and adaptable in its instruction set architectures. RISC-V processors are gaining traction in a variety of applications and research fields.

This work aims to develop extended ISA on RISC-V based processor. Those ISA can help the application to accelerate progress during execution and improve the performance. Therefore, it is a useful product since it preserves software compatibility while also allowing for differentiation and innovation. This project will show how it develops a RISC-V RV32I processor and its extension by logic gates and simulates by using Verilog code in ModelSim.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CNN	Convolution Neural Network
CSR	Control and Status Register
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
ISA	Instruction Set Architecture
IT	Information Technology
LTSM	Long Short-Tern Memory
OS	Operating System
PC	Program Counter
RISC	Reduced Instruction Set Computer
RNN	Recurrent Neural Network
RTL	Register-Transfer Level
SIMD	Single-Instruction Multiple-Data

CHAPTER 1
Introduction

1.1 Problem Statement and Motivation

With configurable extensions and different basic elements, RISC-V offers a modular design. Industry, the research community, and academic institutions collaborated to build the ISA basis and its expansions. The base defines logic (i.e., integer) manipulation, control flow, registers (and their sizes), memory, and addressing, and also specifies ancillaries. A general-purpose computer with complete software support, including a general-purpose compiler, can be implemented using only the base. Despite the fact that, the basic extensions already provided a useful to some general applications, there still have a situation that ISA extension may develop where no appropriate ready-made Isa extension exists to meet the design requirements. The Base Integer Instruction Set are required more memory usage, higher power consumption and more instruction code to execute some specific function like multiplication and division, floating data calculation, etc. Furthermore, some of the function cannot perform by using only base instructions. Hence, designing and come out extensions for only perform base instruction is necessarily.

Let have an example for the problem, [1] shows Figure 1.2.1 depicts the placement of a custom ISA extension in a software stack. A RISC-V-compliant processor with a bespoke ISA extension is present at the lowest level. It runs an OS either simple or well design. Any complier that is compatible with a normal RISC-V processor can be used to compile it. There are three apps are installed on top of OS. App1 is a simple application that does not need to be accelerated, especially can be complied with a freely available off-the-self complier or a pre-complied application and able to be run by the RISC-V processor. Apps2 and App3 are the most crucial and must be executed as quickly as possible. Nevertheless, these must be built with a complier that understands the special ISA extension. The complier can make use of the new instructions to speed up App2 and App3 Thus, the RISC-V specification allows for the addition of a bespoke ISA extension in this instance.

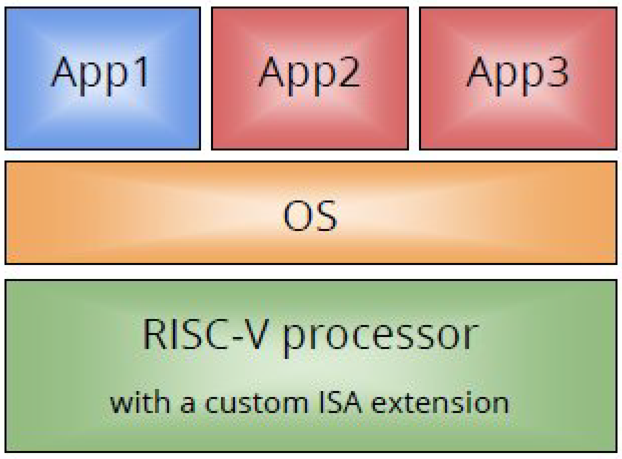


Figure 1.1.1 - Scenario of without the complier for extension

1.2 Project Scope

The scopes of this project are helping the RISC-V based processor to improve performance or add new functionality. This means that RISC-V can be customized for specific use cases, making it more versatile than other ISAs. By adding custom instructions, it can significantly help to improve the performance of their applications, making them run faster and more efficiently. This is particularly important for complex tasks such as machine learning algorithms or encryption and decryption operations, which can benefit greatly from custom instructions.

Another scope is Custom instructions can also help reduce the power consumption of the processor by allowing it to perform certain tasks more efficiently. By tailoring the processor to specific use cases, developers can optimize it for low-power consumption, making it more energy-efficient than other processors.

Encourages innovation of developer also one of the project scopes. Its open-source nature allows developers to freely create and share custom instructions, which can lead to new and exciting advancements in processor design. This fosters an environment of cooperation and creativity, which can ultimately benefit the entire industry.

1.3 Project Objective

The main objective of this project is to develop an extended ISA on RISC-V based processor. It can be divided into several objectives. The first of it is design a five-stage data path RISC-V RV32I processor which its components are instruction fetch unit, registers file, ALU, data memory and control unit. These group of components can perform all 47 of instruction of Base Integer Instruction Set (I).

Besides that, the approaching objectives are identifying instruction of extension and design extended ISA based on instruction. As in this project, it aims to design extended ISA which are Standard Extension for Integer Multiplication and Division (M), These extensions is flexible to integrate with RV32 processor, when the extended instructions are called, then it only be call out to perform the instructions. In other words, it will not be active when the extended instructions are not occurred thereby achieve reducing power consumption of the entire processor.

Furthermore, the M extension must be able to perform calculate multiplication, division and remainder of division with signed or unsigned integer. It aims to increase performance of the calculation, which substituted with using addition or subtraction to perform M extension to reducing instruction line.

1.4 Contributions

This work presents developing extended ISA on RISC-V based processor. First, the processor with extended ISA provides to the potential customer like chipset manufacturers, IoT devices company etc. required powerful, high performance or needed various of different kind of functionality for specific purpose. Extension can help those customers who required a multifunctional or higher performance processor get the processor that achieve fulfilment of processor specification that they needed. They also have optional to customize the specification of processor, which mean choose the designed extension they needed. Hence, the flexibility of customizing ISA helps different types of IT area to be more satisfied to get the processor.

One of the contributions is the growth of RISC-V ISA is potential to disrupt the dominance of proprietary architectures on the market, such as ARM and x86. RISC-V offers a free and open alternative that can be customized for specific use cases, which is particularly attractive for companies that want to reduce their reliance on a single vendor. This has led to increased interest in RISC-V from both large corporations and startups, with companies like Google, NVIDIA, and Western Digital all investing in the technology.

In addition, RISC-V and its extensions are the potential to lower the cost of designing and manufacturing chips, making it more accessible for smaller companies and startups. This could lead to a wave of innovation and new products that would not have been possible before. RISC-V could also enable new applications and use cases, such as low-power IoT devices and specialized machine learning hardware.

1.5 Background Information

First of all, the instruction set architecture defines the functionality of microprocessors (ISA). The instruction set specifies which instructions the CPU is capable of executing. An ISA is a bridge between software and hardware, and it is the specification of microprocessor architecture, according to the layers of abstraction in computers shown in Figure 1.5.1.

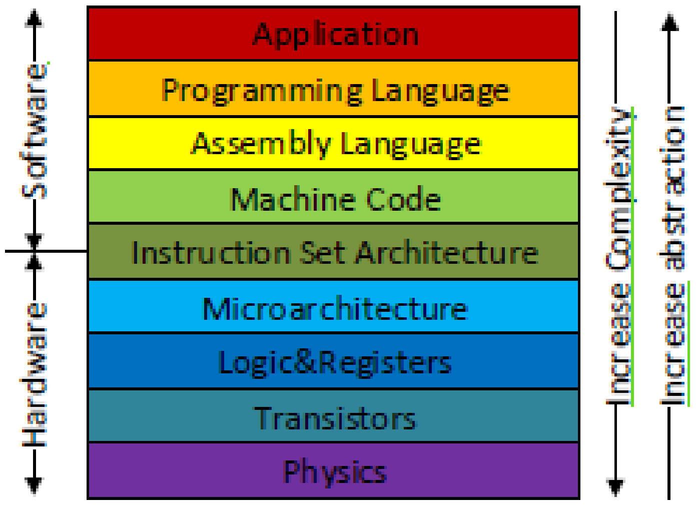
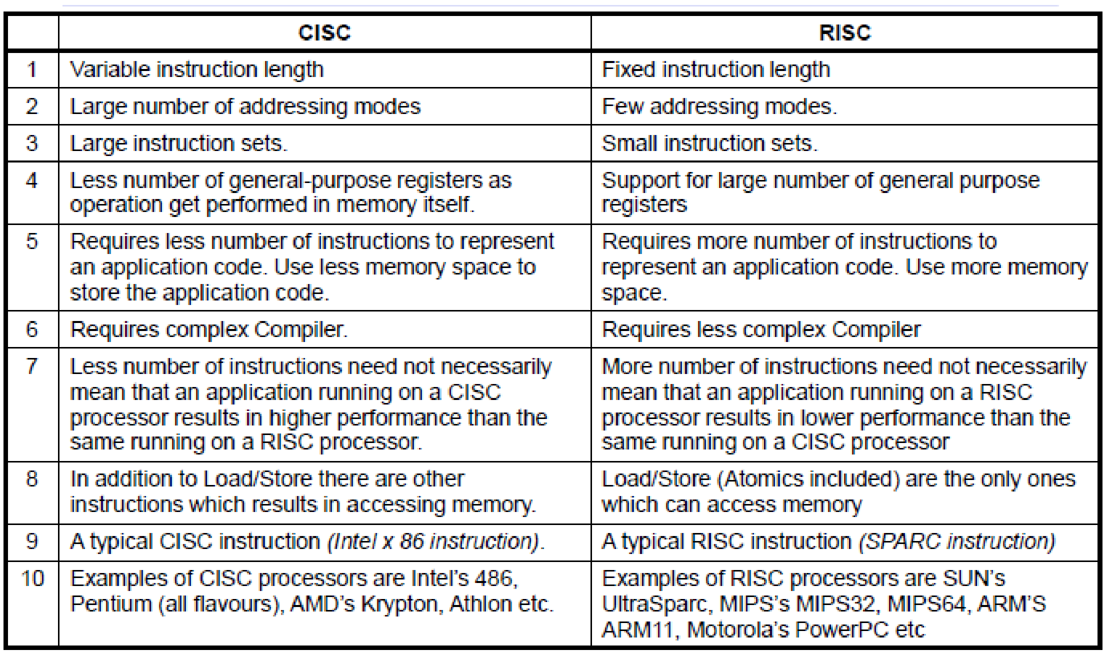


Figure 1.5.1 – Layers of Abstractions

The complex instruction set computer (CISC) and the reduced instruction set computer (RISC) are two types of ISA (RISC). The key differences between CISC and RISC architecture are shown in Table 1.1.1. X86 is a standard CISC ISA, with complicated instructions that may operate directly on memory addresses. RISC instructions, on the other hand, are viewed as an advance over CISC since they simplify the format and operation of each instruction. RISC microprocessors typically execute one instruction per machine cycle so that the design can be pipelined to achieve a higher clock frequency. The simplicity of RISC instructions, on the other hand, adds complexity to software compilers.



	CISC	RISC
1	Variable instruction length	Fixed instruction length
2	Large number of addressing modes	Few addressing modes.
3	Large instruction sets.	Small instruction sets.
4	Less number of general-purpose registers as operation get performed in memory itself.	Support for large number of general purpose registers
5	Requires less number of instructions to represent an application code. Use less memory space to store the application code.	Requires more number of instructions to represent an application code. Use more memory space.
6	Requires complex Compiler.	Requires less complex Compiler
7	Less number of instructions need not necessarily mean that an application running on a CISC processor results in higher performance than the same running on a RISC processor.	More number of instructions need not necessarily mean that an application running on a RISC processor results in lower performance than the same running on a CISC processor
8	In addition to Load/Store there are other instructions which results in accessing memory.	Load/Store (Atomics included) are the only ones which can access memory
9	A typical CISC instruction *(Intel x 86 instruction)*.	A typical RISC instruction *(SPARC instruction)*
10	Examples of CISC processors are Intel's 486, Pentium (all flavours), AMD's Krypton, Athlon etc.	Examples of RISC processors are SUN's UltraSparc, MIPS's MIPS32, MIPS64, ARM'S ARM11, Motorola's PowerPC etc

Table 1.5.1 – Differences between CISC and RISC

The efficiency of CISC architecture is measured in instructions per programme, whereas the efficiency of RISC architecture is measured in cycles per instruction. In terms of performance, there is a trade-off. The growing market for smartphones and embedded projects, on the other hand, has raised concerns about power usage. RISC ISA is currently dominating the mobile device market due to complicated CISC commands requiring more logic and transistors to delay with higher power consumption.

Furthermore, RISC-V ISA is categorised by few sets of instructions that can be combined in any way as design needs. For example, the bare minimum or all ISA extensions can be implemented by a RISC-V processor. Hence, it also can be enabled or disabled as the application needs, without having to consume power when it does not be used. Every group of the instructions is unique and does not be predefined. In the below, table 1.5.2 shows some main ISA extensions that are currently are authorized by RISC-V Foundation and be developing. Besides, it will be extended more ISA extensions in the future as mentioned earlier [1].

SA Extension	Authorized	Notes
I/E	Yes	Instructions for basic Integer operations. This is the only extension that is mandatory. It requires 32 registers, E requires only 16.
M	Yes	Instructions for multiplication and division
C	Yes	Compact instructions that have only 16bit encoding. This extension is very important for applications requiring low memory footprint.
F	Yes	Single-precision floating-point instructions
D	Yes	Double-precision floating-point instructions
A	Yes	Atomic memory instructions
B	No	Bit manipulation instructions. The extension contains instructions used for bit manipulations, such as rotations or bit set/clear instructions.
V	No	Vector instructions that can be used for HPC.
P	No	DSP and packed SIMD instructions needed for embedded DSP processors.

Table 1.5.2 – Main RISC ISA extension

CHAPTER 2
LITERATURE REVIEW

2.1 Design of the RISC ISA

In the beginning, [2] has outlined the support for key aspects we think essential for a modern general-purpose ISA in these instruction sets and show in below Table 2.1.1. At least two key technological features are missing from all of the architectures. The nearest standard, ARMv8 [2, 3] is a proprietary standard. SPARC [2, 4] and OpenRISC [2, 5], the two open ISAs are missing some crucial architectural features. Except for the DEC Alpha [2, 6], all of the ISAs include extra characteristics that significantly increase implementation complexity, especially for high-performance implementations.

	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		✓				✓	
64-bit Address	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

Table 2.1.1 - Summary of several ISAs' support for desirable architectural features.

Given these constraints, the customizable set of instructions are required to be developed. Building a free and open ISA for RISC-V that avoids these technological issues and is simple to implement in a variety of microarchitectural styles, with the benefit of hindsight.

[2] has given some RICS-v\V base ISA examples such as RV32I Base ISA [2, 7], RV32E Base ISA [2, 7], RV64I Base ISA [2, 7] and RV128I [2,7] Base ISA, these RISC-V fundamental ISAs are simple to implement and maintain, yet they are comprehensive enough to handle a modern software stack. Architectural characteristics that bring undue complexity costs to both basic and aggressive microarchitectures are

avoided in the based ISA design. However, there are many application domains for which a simple integer ISA is insufficient, such as workloads requiring floating-point computing.

2.2 Superscalar RISC-V Processor with SIMD Vector Extension

[8] demonstrates the hardware implementations of a dual-issue superscalar RISC-V processor with out-of-order execution and a SIMD vector [8, 9] co-processor with tailored vector instructions are presented in this thesis work. The suggested superscalar processor is designed to achieve high performance in general-purpose activities, whereas the proposed vector co-processor, which includes expanded vector instructions, is designed to improve performance in the machine learning field.

To enable out-of-order execution, the Tomasulo algorithm [8, 10] is incorporated in the hardware design. In the instruction fetch step, the Gshare branch prediction [8, 11] approach is used. The CPU can speculatively execute instructions with 5 backups of the renaming register file to reduce the waste cycles caused by branch operations. In comparison to the traditional one-bit busy status, the busy counters in the renaming register file improve instruction throughput. With the busy counters in the renaming file, the processor can constantly dispatch instructions that alter the same destination register in order to fully utilise each pipeline stage. The hardware complexity is decreased by relocating the latest value column in the typical register renaming file to the result column in the commit buffer, which saves space and power. In addition, the simplified prediction recovery system reduces critical paths, allowing for a higher operational clock frequency, as mentioned by [8]. The suggested RISC-V processor enhances average instruction throughput by 18.9% and average prediction hit rate by 4.92 percent when compared to a similar architecture. With the addition of machine-level exception and integer multiplication/division capability, the suggested CPU achieves a 16.9% higher operating frequency.

[8] states that The SIMD architecture is used in the suggested vector co-processor to improve the performance of computation and data-intensive operations. Based on the Cambricon ISA, a customised SIMD instruction set is presented, which is mapped to the standard 32-bit RISC-V instruction format. The suggested vector extension, in comparison to the Cambricon ISA, unifies the internal address mapping to stress the flexibility of the instruction set. The co-processor is made up of the vector

instruction board, wrapped internal memory banks, and processing units that follow the specification of the proposed vector instruction set. The instruction board combines the functions of the reservation station and commit buffer, allowing the processing units to solve data dependencies and enable instruction-level parallelism. To simplify sequencers and maximise memory use, the wrapped memory bank of the true-dual-port memory block provides one-cycle misaligned memory access in hardware. When compared to the basic C programme, the normal vector programme obtains a small amount of throughput improvement and the delicate vector programme with software optimizations achieves around 10 percent throughput improvement in the case study of the LeNet-5 model. When compared to the RISC-V processor alone, the vector co-processor with the superscalar processor can handle more pictures per second, providing 10.18 percent real-time throughput and gains advantage in energy efficiency.

Nevertheless, [8] state that the processor only allows bare-metal processing with no rmware overlay. To improve usability, a number of features need be added to the current processor. The supervisor-level and user-level privileged CSRs for hardware design are also included in the RISC-V privileged specification. In supervisor mode, the rmware kernel should be running. On the other hand, RISC-V programmes should run in user mode, with supervisor mode controlling entry addresses and heap pointers. For those two privileged modes, memory virtualization with ordered peripherals is required to provide the memory hierarchy. At the same time, other common RISC-V standard extensions, including as the compressed "C" extension, the floating-point "F" extension, and the atomic "A" extension, can be de tuned in the processor to increase compatibility.

Only a basic CNN model is used to test the performance of the proposed vector co-processor. The instruction-based accelerator, on the other hand, is adaptable to a wide range of neural network models and layer types. The architecture can also analyse other popular layers including squeeze-and-excite, inception, depth-wise convolution, RNN, and LTSM. In the vector co-processor, the element format is an 8-bit integer. Fixed point values, on the other hand, are insufficient for real-world machine learning applications with a limited data range. The data formats of Minifloat and Posit can be studied on the processing units to create an accurate result with a true set of neural network parameters by keeping the width of each element to 8 bit.

2.3 Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms

In this study, [14] used the RISC-V RV32I ISA to create software-only implementations of eleven important cryptographic algorithms and evaluated their performance against that of a RISC-V processor equipped with additional hardware modules that implement specialised instructions for the single-cycle execution of cryptographic primitives. They have balanced execution speed and code size in our software solutions, with a focus on execution speed. In order to do this, we used loop unrolling where it was most useful while only slightly increasing the programme memory. In addition, unless there is a specific instruction that calculates the SBOX value, all SBOX tables were pre-computed and stored in memory as opposed to being calculated on the fly.

The cryptography instructions were divided into groups based on their organisational structure for the RISC-V processor enhanced with cryptographic hardware, and each group was created as a hardware module. Any subset of modules can be integrated with the CPU thanks to the modular approach. Since only a portion of the 32-bit scalar cryptographic instructions are used by each method, they determined the module utilisation for each algorithm and assessed the implementation costs in accordance in order to ensure a valid performance comparison.

The authors found that for five of the eleven algorithms, implementations with the cryptography set extension offer execution speeds of 1.5 to 8.6 times quicker and programme memory requirements of 1.2 to 5.8 times lower than those utilising only the basic RV32I instruction set. Less than 6% less programme memory is needed and execution speed has increased for the remaining six methods. When compared to software solutions using the RV32I ISA, the hardware crypto implementations have an additional hardware complexity of 0.3% to 7.7%. Figure 31's benefit-cost analysis, which summarises the benefits in execution time as a function of the expenses associated with hardware complexity for each algorithm, depicts the acceleration of execution time as a function of the relative hardware cost graphically. We can see that for the SHA algorithms, we gain an acceleration of roughly 1.7 at a hardware cost increase of less than 7.5% as an example of the benefit vs. cost trade-off. We proposed a new instruction to speed up memory address calculation operations for the 8-bit input SBOX table, which dominates the execution time for four of the eleven algorithms, based on our research of execution durations. With only a 1.1% increase in hardware cost, this new instruction gave the four algorithms execution times that were 1.2 to 1.6 times faster.

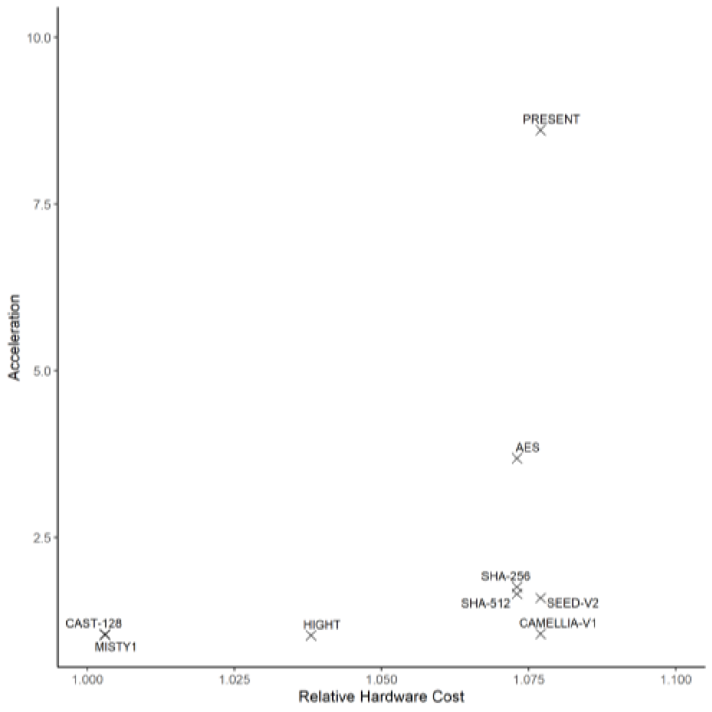


Figure 2.3.1 - Acceleration vs. hardware cost of implementation of crypto implementations.

Besides that, [14] added support for permutation instructions in addition to cryptographic algorithms, as opposed to just cryptographic algorithms as in [15], offering a more comprehensive solution for the implementation of any cryptographic algorithm. This method enables asymmetric algorithms as well as all current and future symmetric cryptographic methods to be supported by software and accelerated by hardware. Only 4K gate equivalent are needed to implement permutation instructions, which is only an 8% increase above the 3.7K gate equivalent needed for cryptographic instruction support. A synthesis of the method suggested in [15], in contrast, would need nearly gate equivalent just for cryptographic instructions.

CHAPTER 3
Proposed Method

3.1 Design Specifications

For specification of RV32I processor, it will be design based on RISC-V base instruction set architecture and few extended ISA (M, F, A, C extension) compatible with pipeline processor. The design and implementation of front-end will be use Verilog to verify the correctness and functionality of the processor built. Below shown the basic feature of processor that implemented based on the functionality of instruction:

- Arithmetic Instructions: ADD, SUB, ADDI.
- Logical Instructions: AND, OR, XOR, SLL, SRL, SRA, etc
- Data Transfer Instructions: LW, SW, etc.
- Branch Instructions: BEQ, BNE, BLT, BGE, etc.
- Jump Instructions: JAL, JALR
- Data Comparison Instructions: SLT, SLTIU.

For specification of RV32I processor, it supports few types of instructions format. Figure 3.1.1 illustrates the six main instruction formats (R, I, S, B, U, J) in the base ISA. Each one must be aligned on a four-byte boundary in memory and have a set length of 32 bits. If the target address is not four-byte aligned, an instruction address misaligned exception is raised on a taken branch or unconditional jump. If a conditional branch is not taken, no instruction fetch misaligned exception is raised.

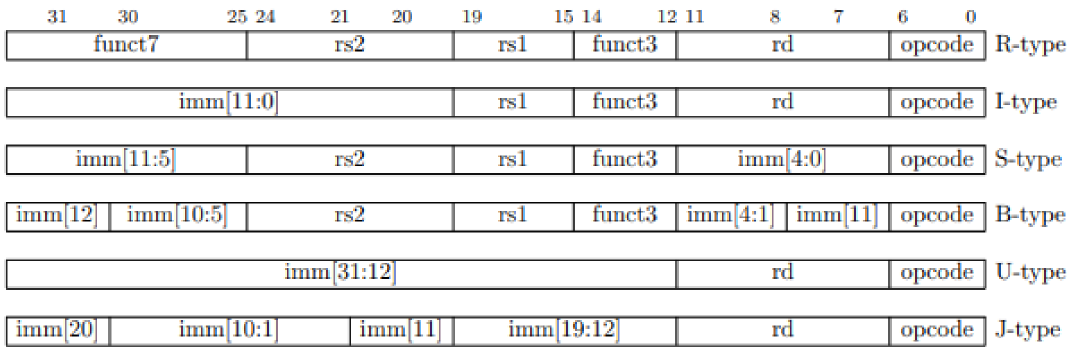


Figure 3.1.1 – RV32 Base Instruction Formats

- R-format

- Normally compute the function with value rs1 and rs2 and store back to destination register.
- The 6-bit of operation code (opcode), 3-bit and 7-bit of function code (funct3 / funct7) is to identify the type of instructions.
- Source register (rs1) [5-bit] is specifically for first source register.
- Target register (rs2) [5-bit] is specifically for second source register.
- Destination register (rd) [5-bit] is specifically for destination register which commonly store the compute result.

- I-format
  - Normally compute the function with value rs1 and the immediate data and store back to destination register.
  - The 6-bit of operation code (opcode) and 3-bit of function code (funct3) is to identify the type of instructions.
  - Source register (rs1) [5-bit] is specifically for first source register.
  - The most 12 significant bits (which the position is R-type format of rs2 and funct7) is specifically for an immediate value for substitute the value of rs2.
  - Destination register (rd) [5-bit] is specifically for destination register which commonly store the compute result.
  - Immediates are always sign-extended, often packed in the instruction's leftmost available bits, and allocated to save hardware complexity. In order to speed up the sign-extension circuitry, the instruction's bit 31 always contains the sign bit for all immediates.

- S-format
  - Normally store the value in rs2 at address of value rs1 with offset of rs1 value.
  - The 6-bit of operation code (opcode) and 3-bit of function code (funct3) is to identify the type of instructions.
  - Source register (rs1) [5-bit] is specifically for first source register.
  - Target register (rs2) [5-bit] is specifically for second source register.

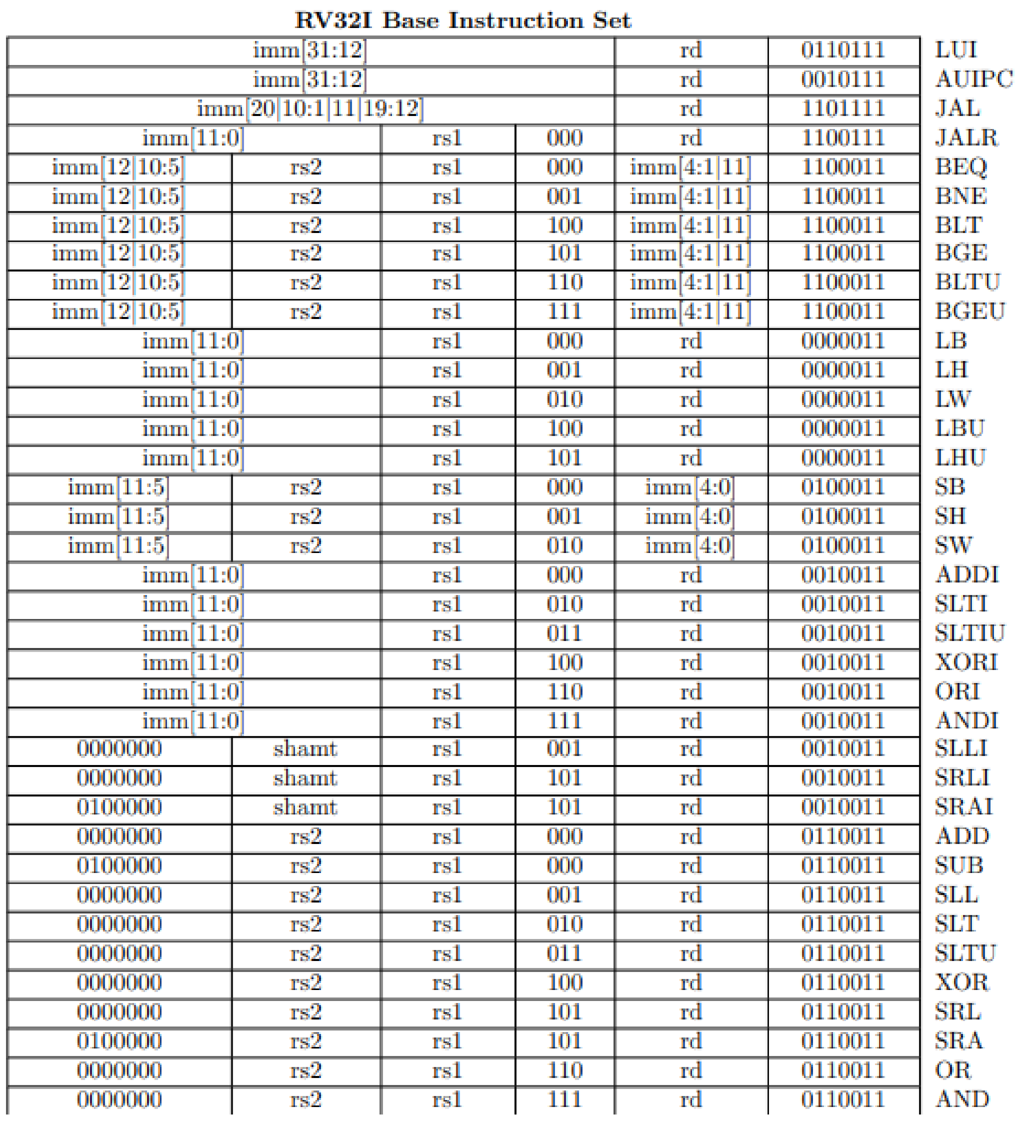
- The separate immediate bits (which the position is R-type format of rd and funct7) is specifically for an immediate value for substitute the value of rd. It needs to combine them together to become a complete immediate value.
- Immediates are always sign-extended, often packed in the instruction's leftmost available bits, and allocated to save hardware complexity. In order to speed up the sign-extension circuitry, the instruction's bit 31 always contains the sign bit for all immediates.

- B-format
  - Normally branch a target address by comparing the values of rs1 and rs2, if the condition is true, the current instruction address will plus with the immediate offset to jump to the target address.
  - The 6-bit of operation code (opcode) and 3-bit of function code (funct3) is to identify the type of instructions.
  - Source register (rs1) [5-bit] is specifically for first source register.
  - Target register (rs2) [5-bit] is specifically for second source register.
  - The separate immediate bits (which same with S-type) is specifically for an immediate value for substitute the value of rd. It needs to combine them together to become a complete immediate value.
  - Immediates are always sign-extended, often packed in the instruction's leftmost available bits, and allocated to save hardware complexity. In order to speed up the sign-extension circuitry, the instruction's bit 31 always contains the sign bit for all immediates.

- U-format
  - Normally compute with an immediate, but zero-extend to the left.
  - The 6-bit of operation code (opcode) is to identify the type of instructions.
  - Immediates are always zero-extended, the immediate values wil be 20 most significant bit and combine 12 zero least significant bit to become a 32-bit immediate value.
  - Only two instructions involved which are load upper immediate (LUI) and add upper immediate with pc address (AUIPC).

- J-format
  - Normally jump to an address to compute a string of instructions and return back to address of value of rd.
  - The 6-bit of operation code (opcode) is to identify the type of instructions which is only jump and link (JAL).
  - Immediates are always sign-extended, often packed in the instruction's leftmost available bits, and allocated to save hardware complexity. In order to speed up the sign-extension circuitry, the instruction's bit 31 always contains the sign bit for all immediates.
  - It stores the next instruction address (pc + 4) into rd which normally is return address (ra, x1) and combine the current instruction address value with immediate offset value, and the result will be the destination of jump target address.

In the figure 3.1.2 shows the 47 instructions of base integer instruction formats that support in this project.

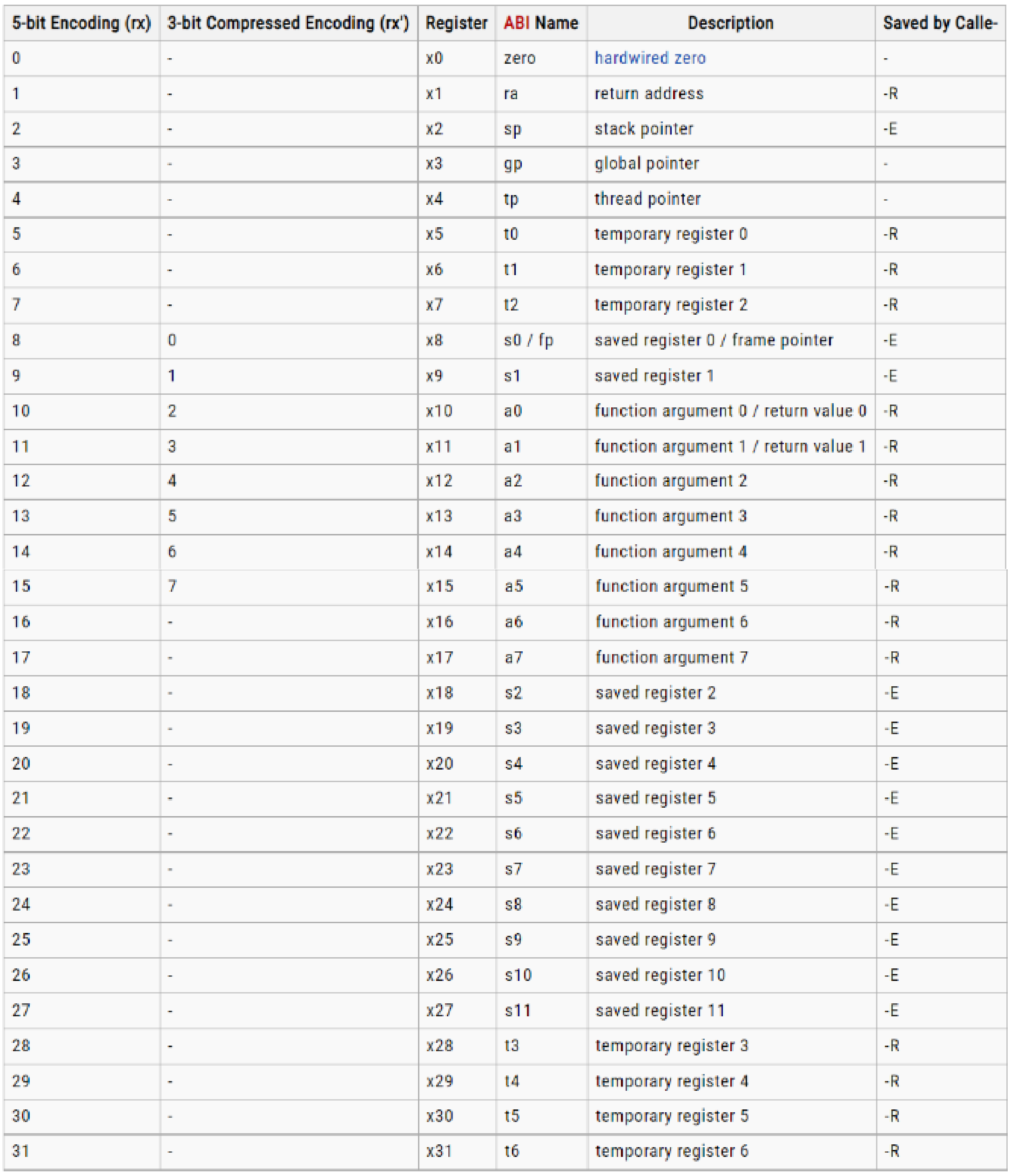


RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 3.1.2 – RV32I Base Instruction Format

Furthermore, the 32 general-purpose registers x1-x31 in the RISC-V base ISA store integer values. The constant 0 is hardwired into the register x0. The address of the current instruction is stored in a separate user-visible programme counter pc register. The normal programme calling convention should use register x1 to hold the return address on a call, even though RISC-V does not define a special subroutine return address link register. Below table 3.1.1 show the description of 32 general-purpose registers file.



5-bit Encoding (rx)	3-bit Compressed Encoding (rx')	Register	ABI Name	Description	Saved by Calle-
0	-	x0	zero	hardwired zero	-
1	-	x1	ra	return address	-R
2	-	x2	sp	stack pointer	-E
3	-	x3	gp	global pointer	-
4	-	x4	tp	thread pointer	-
5	-	x5	t0	temporary register 0	-R
6	-	x6	t1	temporary register 1	-R
7	-	x7	t2	temporary register 2	-R
8	0	x8	s0 / fp	saved register 0 / frame pointer	-E
9	1	x9	s1	saved register 1	-E
10	2	x10	a0	function argument 0 / return value 0	-R
11	3	x11	a1	function argument 1 / return value 1	-R
12	4	x12	a2	function argument 2	-R
13	5	x13	a3	function argument 3	-R
14	6	x14	a4	function argument 4	-R
15	7	x15	a5	function argument 5	-R
16	-	x16	a6	function argument 6	-R
17	-	x17	a7	function argument 7	-R
18	-	x18	s2	saved register 2	-E
19	-	x19	s3	saved register 3	-E
20	-	x20	s4	saved register 4	-E
21	-	x21	s5	saved register 5	-E
22	-	x22	s6	saved register 6	-E
23	-	x23	s7	saved register 7	-E
24	-	x24	s8	saved register 8	-E
25	-	x25	s9	saved register 9	-E
26	-	x26	s10	saved register 10	-E
27	-	x27	s11	saved register 11	-E
28	-	x28	t3	temporary register 3	-R
29	-	x29	t4	temporary register 4	-R
30	-	x30	t5	temporary register 5	-R
31	-	x31	t6	temporary register 6	-R

Table 3.1.1 - 32 General-purpose Registers of RICS-V Processor

In addition, before data are diverted to a data route for operations, memory is a crucial component that stores instructions, stacks, and static data inside. The memory map for the volatile memory used in this project is illustrated in Figure 3.1.3. The memory depicted is a virtual memory and will be built using RTL modelling on a "FPGA," which will use Verilog code to verify that the virtual memory is valid (SD-ram). Because more testing must be done before the real memory is built, physical memory is typically significantly smaller than virtual memory. Memory will employ a lower bit address due to cost and affordability, for example, a smaller flip-flop is used to save the bit value. In this processor is designed in total instructions memory location: $2^{10000000}$ (268 435 456) location each location holds 8-bit value.

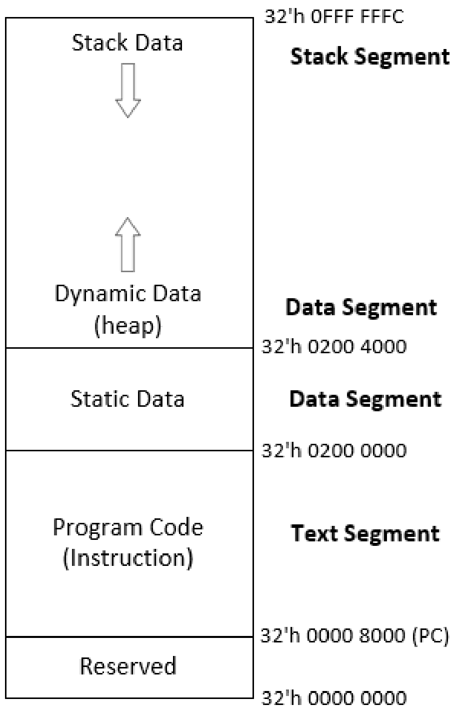
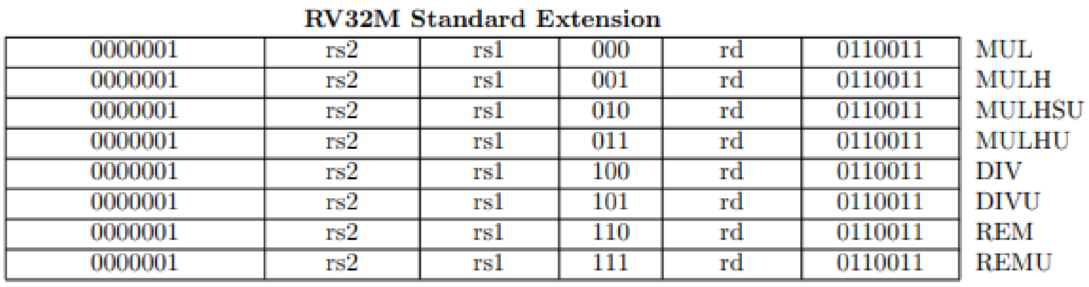


Figure 3.1.3 – Memory map of RV32 Processor

RV32 memory allocation is divided into 3 segments:

1. Stack segment: Hold or store the register values which used by procedure during the execution.
2. Data segment: Hold the object value whose lifetime is the program's execution and cease during the program exit.
3. Text segment: Store the program instructions which flashes assembly code compiled to machine code.

Besides that, the "M" standard integer multiplication and division instruction extension, which comprises instructions to multiply or divide values stored in two integer registers. MUL writes the lower XLEN bits to the destination register after multiplying rs1 by rs2 by XLEN bits. For signed-signed, unsigned-unsigned, and signed rs1-unsigned rs2 multiplication, MULH, MULHU, and MULHSU do the identical multiplication but return the upper XLEN bits of the complete 2XLEN-bit product, respectively. DIV and DIVU divide rs1 by rs2 into XLEN bits by XLEN bits signed and unsigned integers, rounding to zero. The remainder operations of the respective division are handled by REM and REMU. The dividend's sign and the result's sign are identical for REM. In the figure 3.1.4 shows the multiplication and division instruction formats.



RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 3.1.4 – RV32 M Extension Instruction

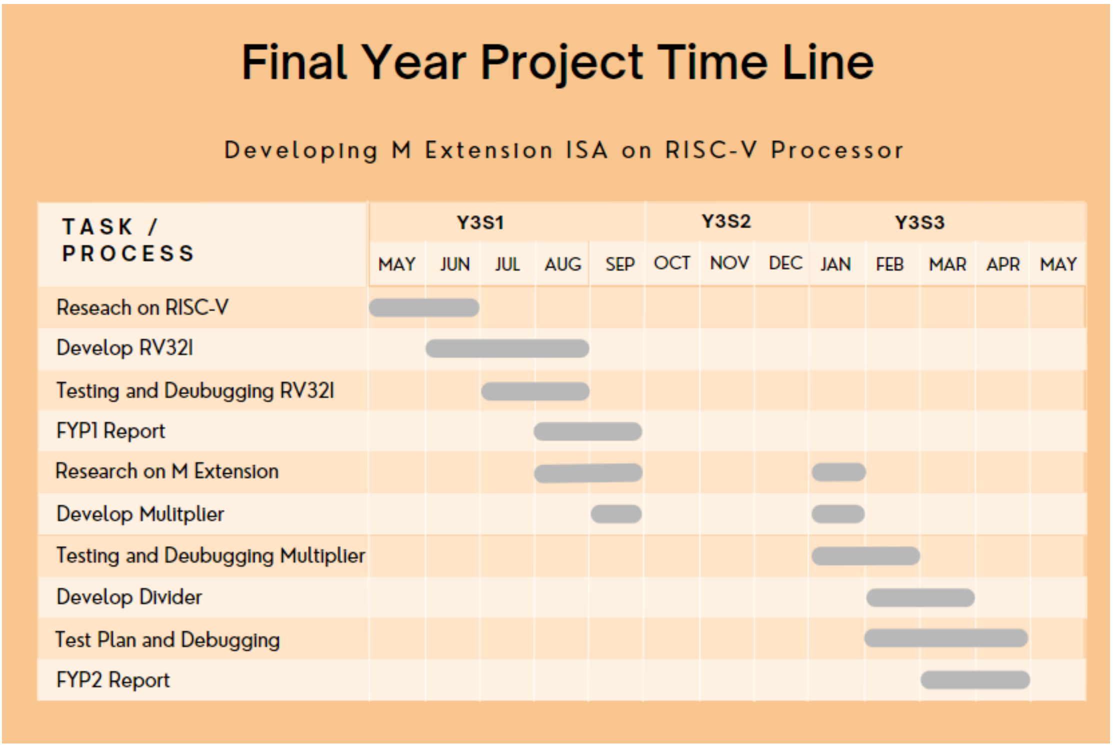
3.2 Software

The entire project will use ModelSim [12] which are a software to design and verification of digital circuits by using HDL such as Verilog [13] code will be selected as coding language for designing the RISC-V RV32I Processor. This software provides a great capability for uncover design flaws and show data for analysis and debug with an intelligently constructed debug platform. It is great for FPGA design because it has a wide range of intuitive feature for Verilog.

The Verilog is an HDL for modelling electronic systems that is specified as IEEE Standard. At the RTL of abstraction, it is widely employed in the design and verification of digital circuits. It is also applied in the design of genetic circuits as well as the verification of analogue and mixed-signal circuits. The syntax of language is similar to C programming language which is famous in software development according to [13]. Hence, it is easy to get started on coding with HDL.

3.3 Timeline

In this project, there are two semester to develop a RV32I and M extension. At the first of the half project time, it starts to study about the RISC-V knowledge. After have a roughly understanding about RISC-V architecture, then start developing and debugging a RV32I processor as a general functional processor and also multiplier. During around the end of semester, the FYP1 report required to generate and hand on. The later semester which are the short semester suspended due to industry training. After the industry training period end, this project plan to finish up the development of multiplier and start study on division. Lastly, complete all the things including FYP2 report at the end of project timeline. Below shown the whole year project timeline.


Final Year Project Time Line

Developing M Extension ISA on RISC-V Processor

TASK / PROCESS	Y3S1					Y3S2			Y3S3				
	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	JAN	FEB	MAR	APR	MAY
Reseach on RISC-V	■	■											
Develop RV32I		■	■	■									
Testing and Deubugging RV32I			■	■									
FYP1 Report				■	■								
Research on M Extension				■	■				■				
Develop Mulitplier					■				■				
Testing and Deubugging Multiplier									■	■			
Develop Divider										■	■		
Test Plan and Debugging										■	■	■	
FYP2 Report											■	■	

Figure 3.3.1 – Gannt Chart of FYP Timeline

CHAPTER 4
System Design

4.1 Macro System Design

First of all, Figure 3.2.1 shows data path of instructions execution in this RISC-V processor that required 5 stages to complete. In the beginning, fetching out the instruction from instruction memory that program counter (PC) point to the address of instruction. At the same time, PC always plus 4 to prepare to output the next instruction address to execute the following instruction. Then, instructions will be decoded into several part such as operation code that determine which function is called and register address that output the value of register which the register address point to. After that, the values execute in ALU like doing arithmetic, comparison, logical or calculating target address operation, and send the result to data memory to store the data, if the result is required store back to registers, it will skip access data memory part and write back to register.

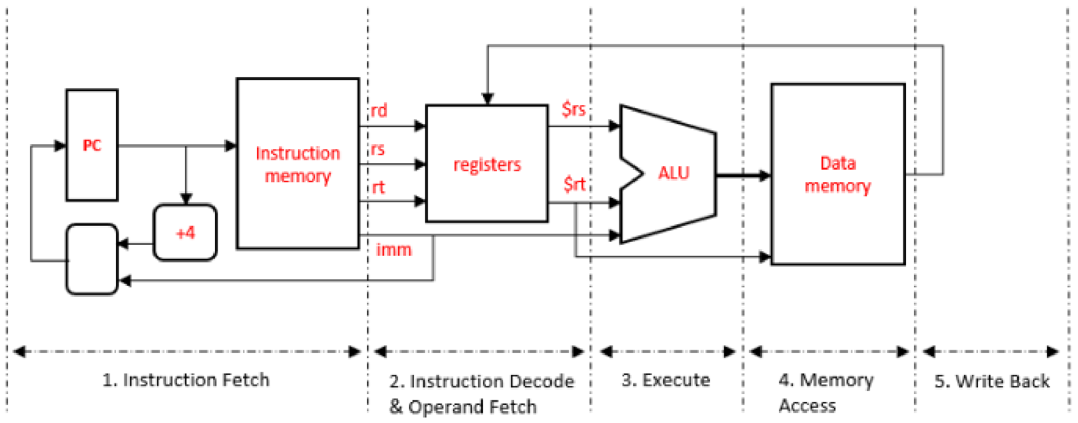


Figure 4.1.1 – Top Level of Structural RISC Instructions Execution Datapath Overview

Addressing modes referring how an operand or instruction are being design and specified in memory by the architecture itself. Before the operand is actually performed, the addressing mode defines a rule for interpreting or changing the address field of the instruction. In this project, there are 6 type of address mode that same as instruction types mentioned before:

- Register Addressing (R-format)
  - The operation basically predefined as compute with two registers value (rs1, rs2) within ALU and mostly used by arithmetic (ADD, SUB), logical (AND, OR, XOR), bit-shifting (SLL, SRL) and program control instructions (SLT, SLTU)
  - The two register data will be fetched according to its address and compute in ALU then store back to the register that address location from rd.

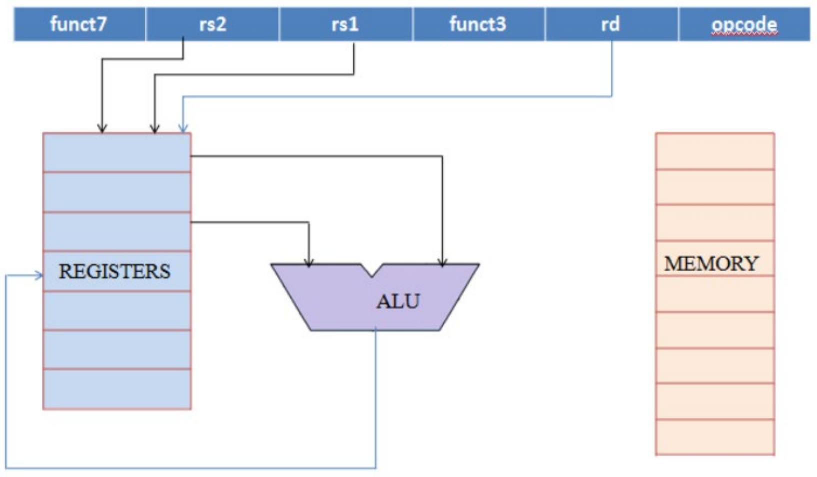


Figure 4.1.2 – Simple Instruction Operation of R-format in Datapath

- Immediate Addressing (I-format)
  - The operation basically predefined as compute with registers value (rs1) and an immediate value from instruction within ALU and mostly used by arithmetic (ADDI), logical (ANDI, ORI, XORI), bit-shifting (SLLI, SRLI) and program control instructions (SLTI, SLTIU) with an immediate.
  - The register data and immediate value with signed or unsigned extend will be fetched according to its address and compute in ALU then store back to the register that address location from rd.

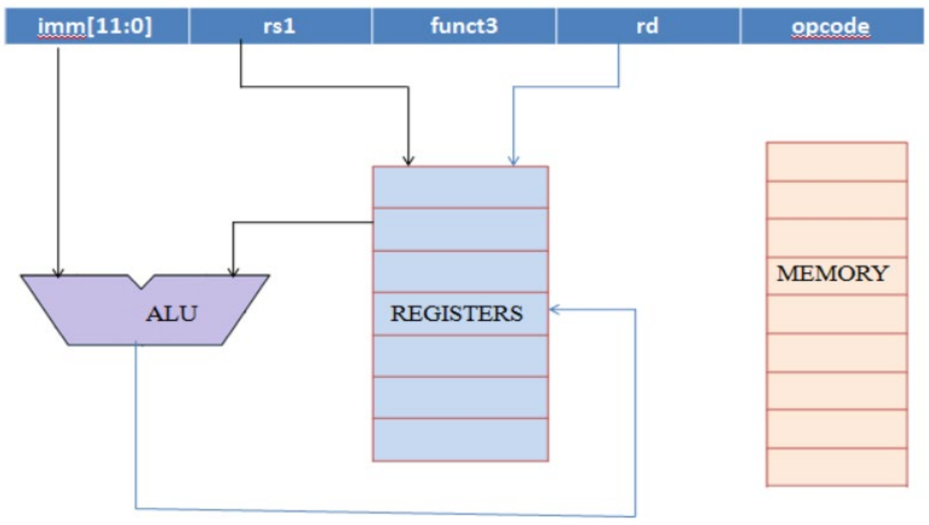


Figure 4.1.3 – Simple Instruction Operation of I-format in Datapath

- Store Addressing (S-format)
  - The operation basically predefined as compute data memory address with registers value (rs1) that contain an address value and an immediate value that are offset of address within ALU and storing the value of rs2 in data memory (SB, SH, SW).
  - The address and immediate value with signed or unsigned extend that is offset of address will be fetched and compute addition in ALU to get the target address then send to data memory to reach the target address and store value of rs2.

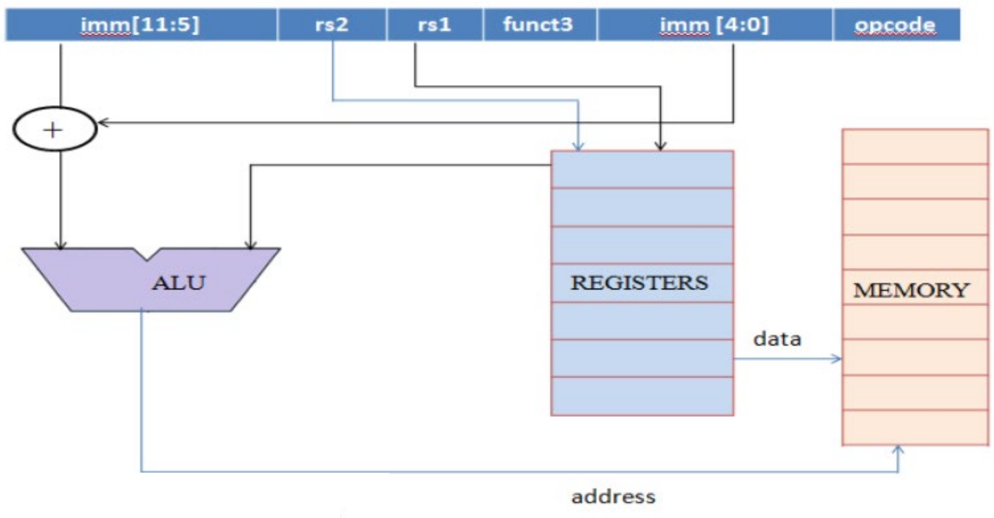


Figure 4.1.4 – Simple Instruction Operation of S-format in Datapath

- Branch Addressing (B-format)
  - The operation basically predefined as compare with two registers value (rs1, rs2) within ALU to checking comparison condition is true or not and it is mostly used by branching to a target address (BEQ, BNE, etc).
  - The two registers data will be fetched compute in ALU by using subtraction to check comparison condition. If the condition is true, it will branch to the target address that is the combination of the current instruction address and the offset which is a separated immediate in instruction with signed or unsigned extend.

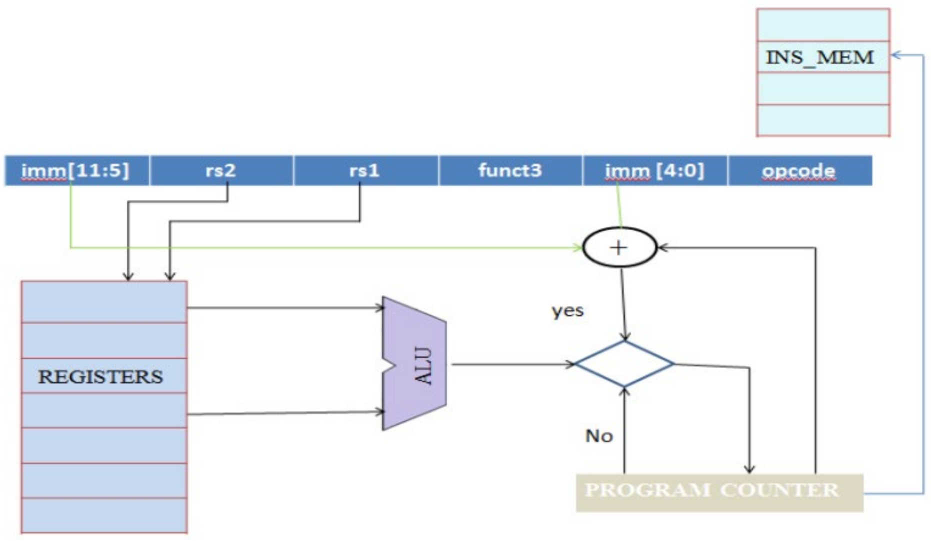


Figure 4.1.5 – Simple Instruction Operation of B-format in Datapath

- Upper Addressing (U-format)
  - The operation basically predefined as compute a data that an immediate is zero extend to right to make the immediate becomes most significant bit within ALU and it has two instructions which are LUI and AUIPC
  - The LUI instruction predefined as storing an immediate that with zero extend into the register and the AUIPC instruction predefined as storing the value of combination of current instruction address and an immediate with zero-extend in register.

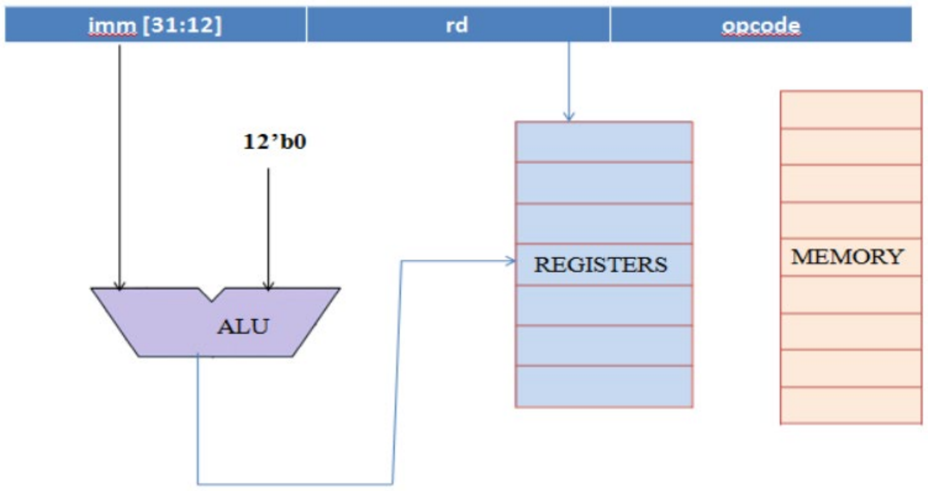


Figure 4.1.6 – Simple Instruction Operation of LUI (B-format) in Datapath

- Jump Addressing (J-format)
  - The operation basically predefined as store the next instruction address into a register for after jump back to next instruction and jump to target address (JAL).
  - The immediate data with signed extend that offset to jumop and current instruction address compute addition then go to instruction memory to branch to the target address.

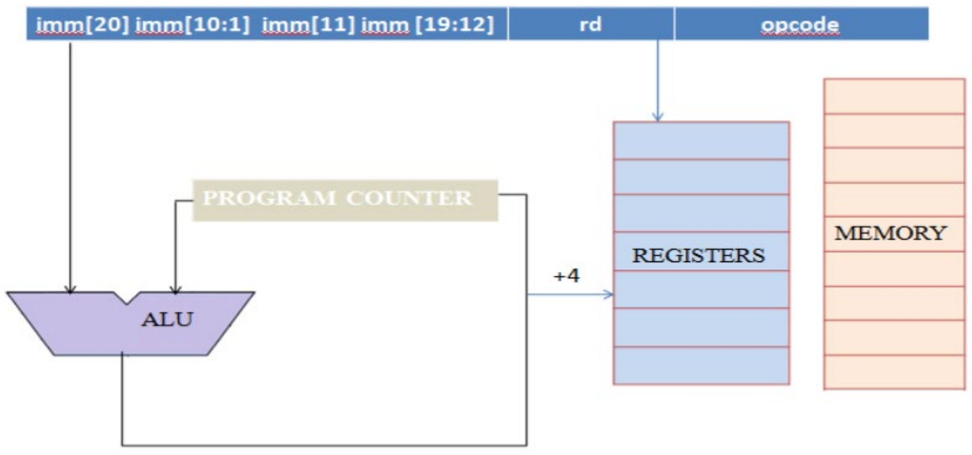


Figure 4.1.7 – Simple Instruction Operation of J-format in Datapath

In RV32 Processor, the chip (Top Level) has been partitioned in several unit level at system level and unit level also will be partitioned in few blocks as shown in table 3.2.1.

Chip Partitioning at (Top Level) at System Level	**Unit and Block Partitioning (Micro-Architecture Level)**	
	Unit-level Partitioning	**Block-level Partitioning**
Processor Chip (c_processor)	Instruction Fetch Unit (u_instr_fecth)	-
	Datapath (u_datapath)	Register File (b_reg_file)
		ALU Block (b_ALU)
		Data Memory Blovk (b_data_mem)
		M Extension Block (m_m_ext)
	Control Unit (u_ctrl)	Main Control Block (b_main_ctrl)
		ALU Control Block (b_ALU_ctrl)

Table 4.1.2 – Block Hierarchy

4.2 Instruction Fetch Unit

4.2.1 Functionality and Feature

Instruction fetch unit is an IF stages where fetch and generate new address value for next instructions. Instructions code to be save in instruction memory where, instruction fetch unit generate the new address value to fetch the instruction and ask instruction memory to send instruction code based on the address generated for processing and operations.

4.2.2 Interface and I/O Pin Description

- **Interface**

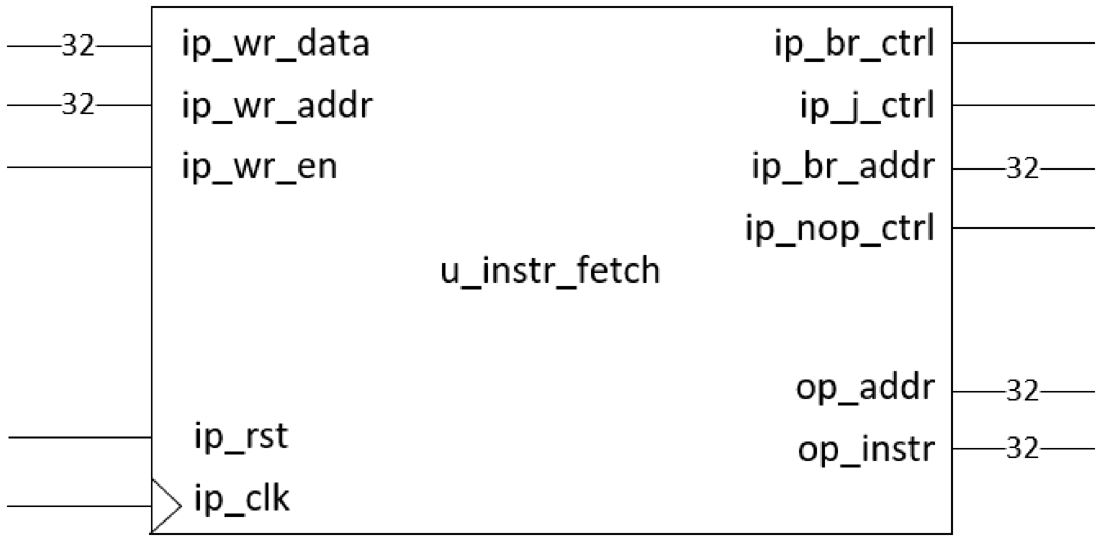


Figure 4.2.2 – Instruction fetch unit interface

- **I/O Pin Description**

Pin name: ip_wr_data[31:0] **Pin class:** data **Pin function:** A 32 bits data for flash into instruction memory.	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: ip_wr_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of instruction memory for flash data.	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: ip_wr_en **Pin class:** control **Pin function:** A pin to control for enable write data into instruction memory.	**Source:** Datapath **Destination:** Instruction Fetch Unit

Pin name: ip_rst **Pin class:** global **Pin function:** A pin to reset program counter and instruction fetch unit	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: ip_br_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of instruction memory for branch or jump the target address.	**Source:** ALU **Destination:** Instruction Fetch Unit
Pin name: ip_br_ctrl **Pin class:** control **Pin function:** A pin to control for enable branch to a target address.	**Source:** ALU **Destination:** Instruction Fetch Unit
Pin name: ip_j_ctrl **Pin class:** control **Pin function:** A pin to control for enable jump to a target address.	**Source:** Control Unit **Destination:** Instruction Fetch Unit
Pin name: ip_nop_ctrl **Pin class:** control **Pin function:** A pin to control for stalling when an instruction required multi clock cycle to operate	**Source:** M Extension **Destination:** Instruction Fetch Unit
Pin name: op_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of instruction memory for output to address calculation.	**Source:** Instruction Fetch Unit **Destination:** ALU
Pin name: op_instr[31:0] **Pin class:** data **Pin function:** A 32 bits data of instruction	**Source:** Instruction Fetch Unit **Destination:** Datapath

Table 4.2.2 – Instruction fetch unit pin description

4.2.3 Internal Operation

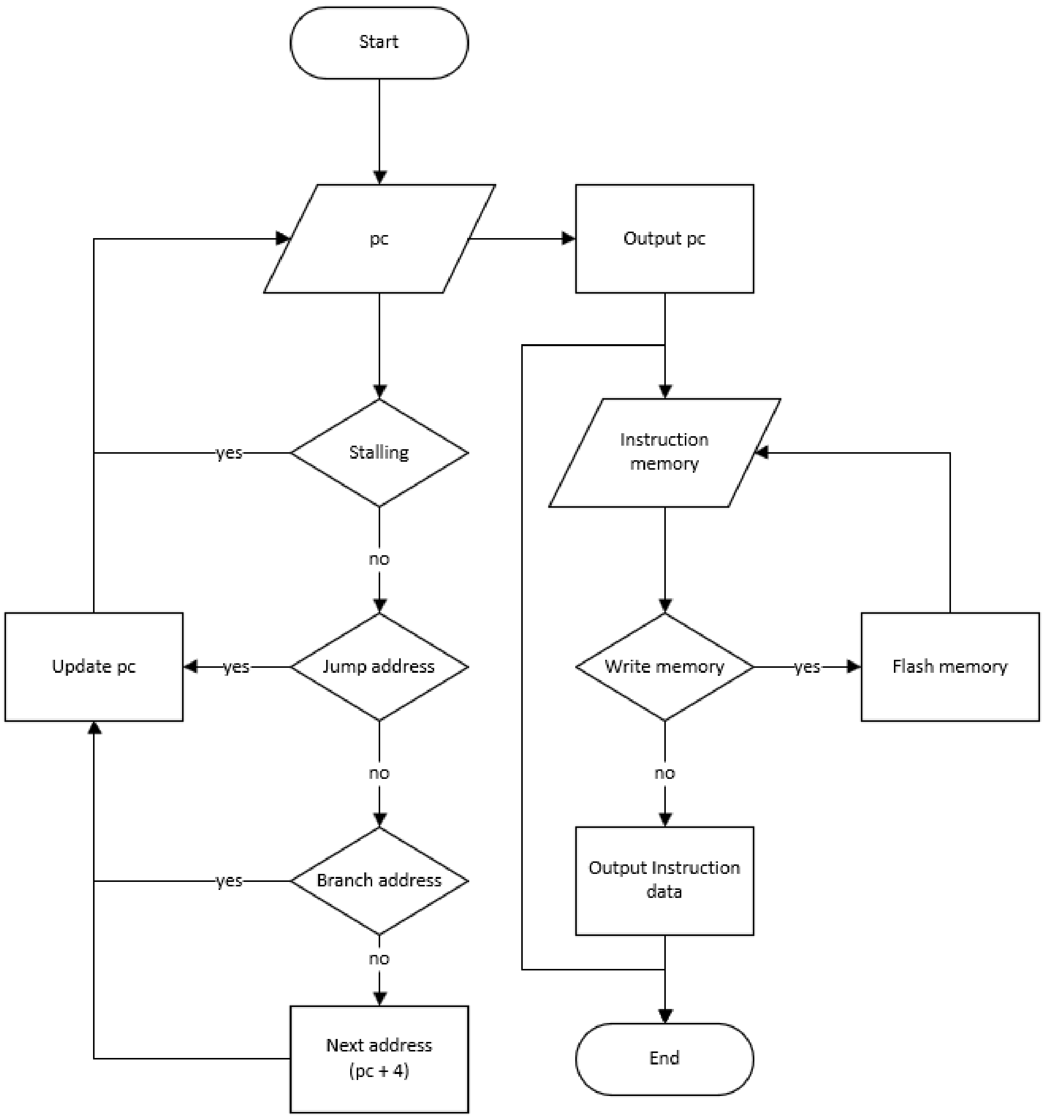


Figure 4.2.3 – Flowchart of instruction fetch unit operation

4.2.4 Schematic Design

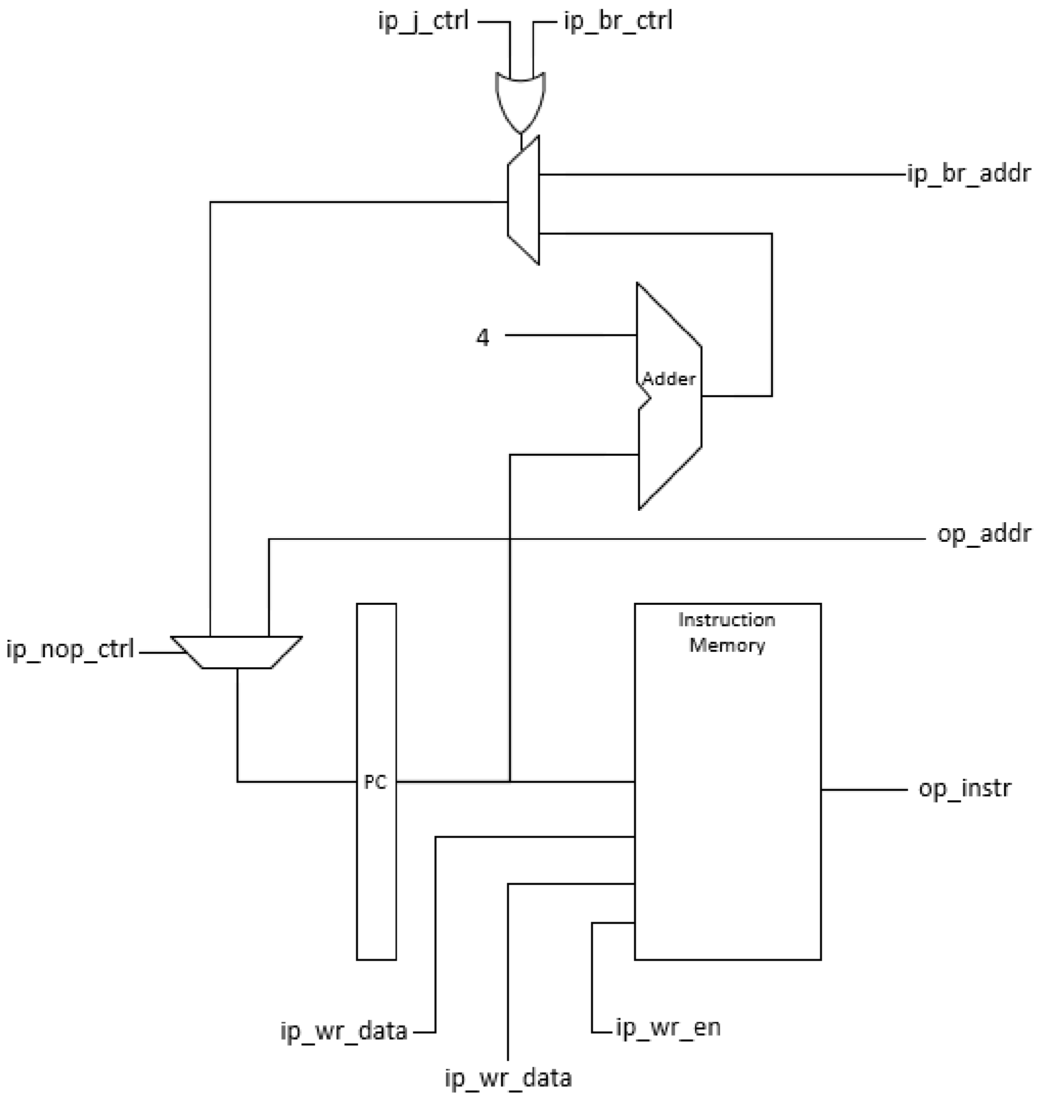


Figure 4.2.4 – Schematic design of instruction fetch unit

4.2.5 Verilog Model

```
/**********************************************************************
Project: Develop Extended ISA on RISC-V Based Processor
Module: b_instr_fetch.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: Instruction Fetch Unit
**********************************************************************/

`default_nettype none // to catch typing errors due to typo of signal names

module u_instr_fetch
#( // declare all the parameter needed
parameter initial_addr = 32'h00008000, // initial pc address
parameter last_addr = 32'h01FFFFFF, // last pc address
parameter SIZE = 6'b100000 // 32
)
( // declare all the input and output pin needed
        input wire [31:0] ip_wr_data, ip_wr_addr, ip_br_addr, ip_j_addr,
        input wire ip_clk, ip_rst, ip_en, ip_wr_en, ip_br_ctrl, ip_j_ctrl, ip_nop_ctrl,
        output reg [31:0] op_addr, op_instr
);

reg [31:0] pc; // program counter
reg [8:0] instr_mem [initial_addr:last_addr]; // instruction memory
reg [31:0] offset_pc; // offset of next address (4 byte)
reg [32:0] carry_pc; // carry bit of pc + 4 adder
integer i; // for loop

assign pc[31:0] = initial_addr; // set pc to first address
assign offset_pc[31:0] = 32'h4; // offset of next address = 4 byte
assign carry_pc[0] = 1'b0; // no negetive  in pc + 4 adder

// program counter
always @(posedge ip_clk) begin
        if(ip_rst == 1'b1) begin // if reset
                // reset to default value
                pc[31:0] <= initial_addr;
        end

        else if (ip_rst == 1'b0) begin // if no reset
                if(ip_en == 1'b1)begin // if enable to run pc
                        op_addr[31:0] <= pc[31:0]; // output current pc

                        if (ip_nop_ctrl == 1'b0) begin // if not stalling
                                if((ip_br_ctrl == 1'b1) || (ip_j_ctrl == 1'b1)) begin // if branch or
jump to target instruction address
                                        pc[31:0] <= ip_br_addr[31:0]; // pc branch to target address
                                end

                                else if((ip_br_ctrl == 1'b0) && (ip_j_ctrl == 1'b0)) begin // if branch
to next instruction address
                                        // instruction address + 4 in 32-bit Adder
                                        for(i = 0; i < SIZE; i = i + 1) begin
                                                // calculation of add result
                                                pc[i] <= pc[i] ^ offset_pc[i] ^ carry_pc[i];
                                        // calculation of carry bit
```

```
                                                  carry_pc[i + 1'b1] = (pc[i] & offset_pc[i]) | (pc[i]
& carry_pc[i]) | (offset_pc[i] & carry_pc[i]);
                                        end
                              end
                    end

                    else if (ip_nop_ctrl == 1'b1) begin // if stalling
                              pc[31:0] <= pc[31:0];
                    end
          end
     end
end

// instruction memory
always @(posedge ip_clk) begin
     if(ip_rst == 1'b1) begin // if reset
          // reset to default value
          for(pc[31:0] = initial_addr; pc[31:0] <= last_addr; pc[31:0] = pc[31:0] + 32'h1) begin
                    instr_mem[pc[31:0]][7:0] <= 8'b0;
          end
     end

     else if(ip_rst == 1'b0) begin // if no reset
          if(ip_en == 1'b1)begin // if enable to run instruction memory
                    // output the instruction from program counter
                    op_instr[7:0] <= instr_mem[pc[31:0]][7:0];
                    op_instr[15:8] <= instr_mem[pc[31:0] + 32'h1][7:0];
                    op_instr[23:16] <= instr_mem[pc[31:0] + 32'h2][7:0];
                    op_instr[31:24] <= instr_mem[pc[31:0] + 32'h3][7:0];
          end

          if(ip_wr_en == 1'b1) begin // // if enable to flash data in instruction memory
                    // flash data into instruction memory
                    instr_mem[ip_wr_addr[31:0]][7:0] <= ip_wr_data[7:0];
                    instr_mem[ip_wr_addr[31:0] + 32'h1][7:0] <= ip_wr_data[15:8];
                    instr_mem[ip_wr_addr[31:0] + 32'h2][7:0] <= ip_wr_data[23:16];
                    instr_mem[ip_wr_addr[31:0] + 32'h3][7:0] <= ip_wr_data[31:24];
          end
     end
end
endmodule
```

4.3 Register File

4.3.1 Functionality and Feature

Register is the fastest and most powerful temporary storage available inside central processing unit (CPU) that receive, hold, and transfer data (instruction). It also used to stage data between memory and the functional units on the chip. The register file contains all the general-purpose registers which are used for data transfer such as read and write data.

4.3.2 Interface and I/O Pin Description

- **Interface**

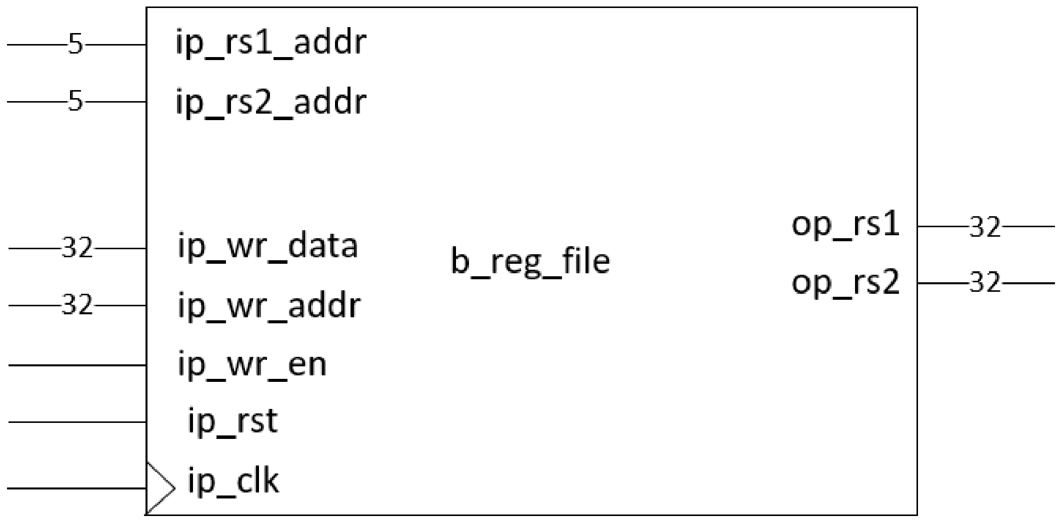


Figure 4.3.2 – Register file interface

- **I/O Pin Description**

Pin name: ip_rs1_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of register 1 for output the register data.	**Source:** Instruction Fetch Unit **Destination:** Register File
Pin name: ip_rs2_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of register 2 for output the register data.	**Source:** Instruction Fetch Unit **Destination:** Register File
Pin name: ip_wr_data[31:0] **Pin class:** data **Pin function:** A 32 bits data for write back into register file.	**Source:** Datapath **Destination:** Register File
Pin name: ip_wr_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of register file for write back data.	**Source:** Datapath **Destination:** Register File
Pin name: ip_wr_en **Pin class:** control **Pin function:** A pin to control for enable write back data into register file.	**Source:** Datapath **Destination:** Register File
Pin name: ip_rst **Pin class:** global **Pin function:** A pin to reset register file	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** Instruction Fetch Unit
Pin name: op_rs1[31:0] **Pin class:** data **Pin function:** A 32 bits data of register 1 for output to calculation in ALU.	**Source:** Instruction Fetch Unit **Destination:** ALU
Pin name: op_rs2[31:0] **Pin class:** data **Pin function:** A 32 bits data of register 2 for output to calculation in ALU.	**Source:** Instruction Fetch Unit **Destination:** ALU

Table 4.3.2 – Register file pin description

4.3.3 Internal Operation

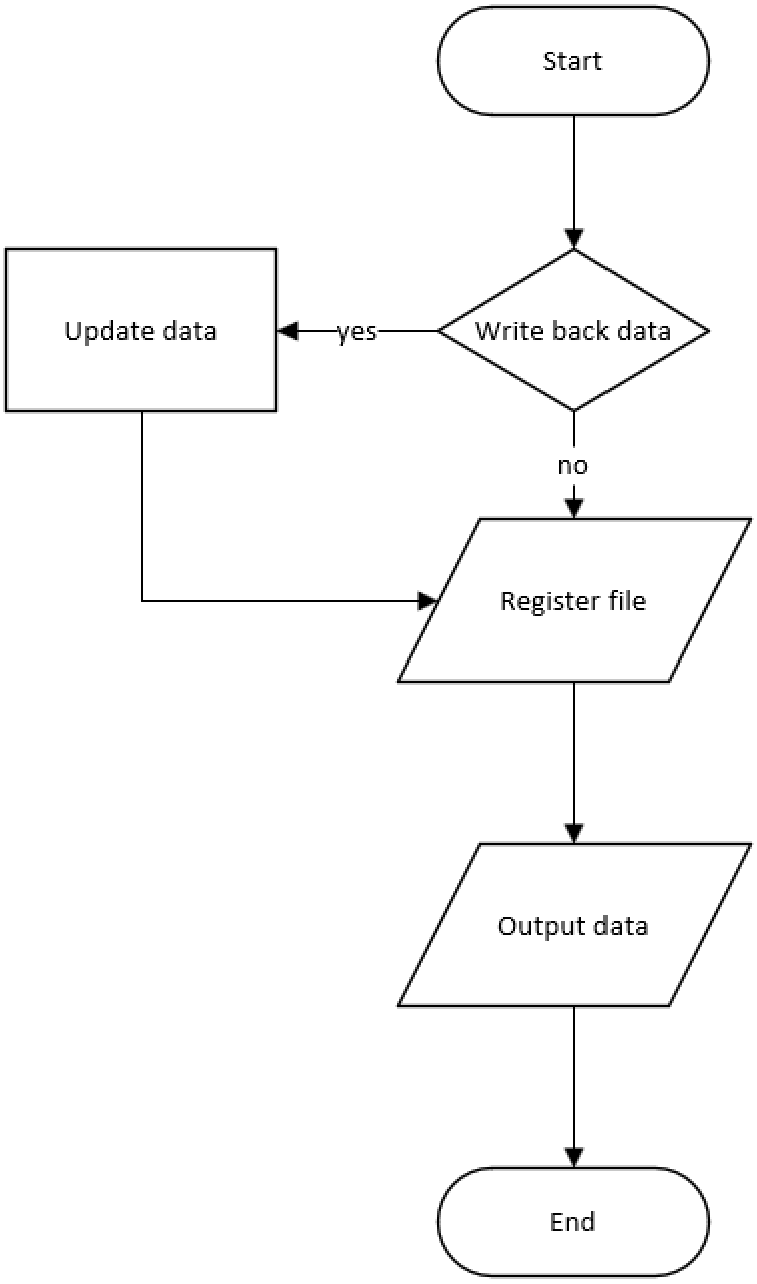


Figure 4.3.3 – Flowchart of register file operation

4.3.4 Schematic Design

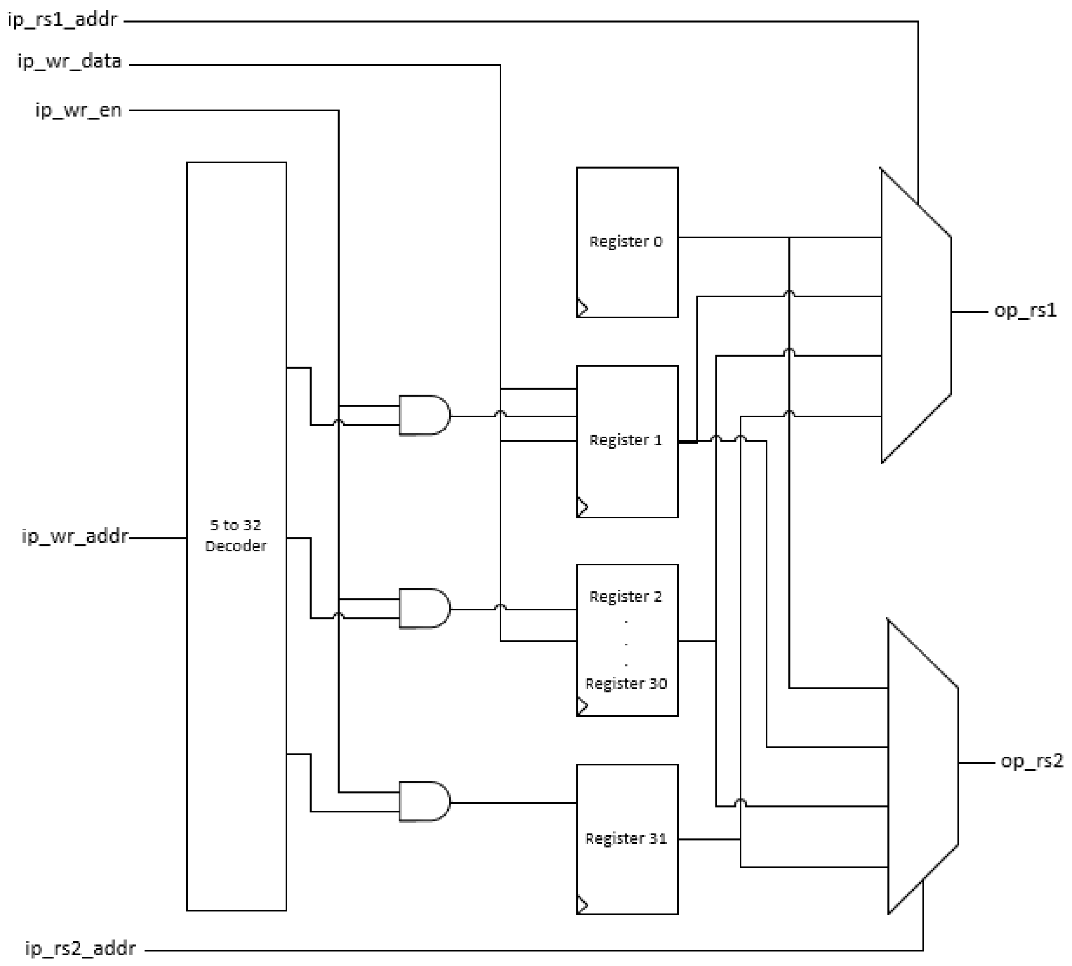


Figure 4.3.4 – Schematic design of register file

4.2.5 Verilog Model

```
/*******************************************************************
Project: Develop Extended ISA on RISC-V Based Processor
Module: b_reg_file.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: 32 of 32-bits General Regsiters
*******************************************************************/

`default_nettype none // to catch typing errors due to typo of signal names

module b_reg_file
#(parameter SIZE = 6'b100000) // declare all the parameter needed
( // declare all the input and output pin needed
        input wire [31:0] ip_wr_data,
        input wire [4:0] ip_rs1_addr, ip_rs2_addr, ip_wr_addr,
        input wire ip_clk, ip_rst, ip_wr_en,
        output reg [31:0] op_rs1, op_rs2
);

reg [31:0] reg_file [0:4]; // 32 of 32-bit general registers

integer i; // for loop to reset

//register zero (x0) always be 0
always @(*) begin
        reg_file[5'b0][31:0] <= 32'b0;
end

always @(posedge ip_clk) begin
        if(ip_rst == 1'b1) begin // if reset
                // reset to default value
                for(i = 1; i < SIZE; i = i + 1) begin
                        reg_file[i[4:0]][31:0] <= 32'b0;
                end
        end

        else if(ip_rst == 1'b0) begin // if no reset
                // output rs1 and rs2
                op_rs1[31:0] <= reg_file[ip_rs1_addr[4:0]][31:0];
                op_rs2[31:0] <= reg_file[ip_rs2_addr[4:0]][31:0];

                if (ip_wr_en == 1'b1) begin // if enable write data
                        // write data into register according to address
                        reg_file[ip_wr_addr][31:0] <= ip_wr_data[31:0];
                end
        end
end
endmodule
```

4.4 ALU

4.4.1 Functionality and Feature

Receives decoded instructions and data from register or an immediate with the help from control signal from control unit to ALU block which then perform arithmetic, logical and bit shifting operations. ALU block do the signed extend if needed and calculate the output from the input from the instructions set. The result is later than send back to the selected register address for others computation or send to data memory to store or load data. It also has ability to calculate target address for jump and branch an address.

4.4.2 Interface and I/O Pin Description

- **Interface**

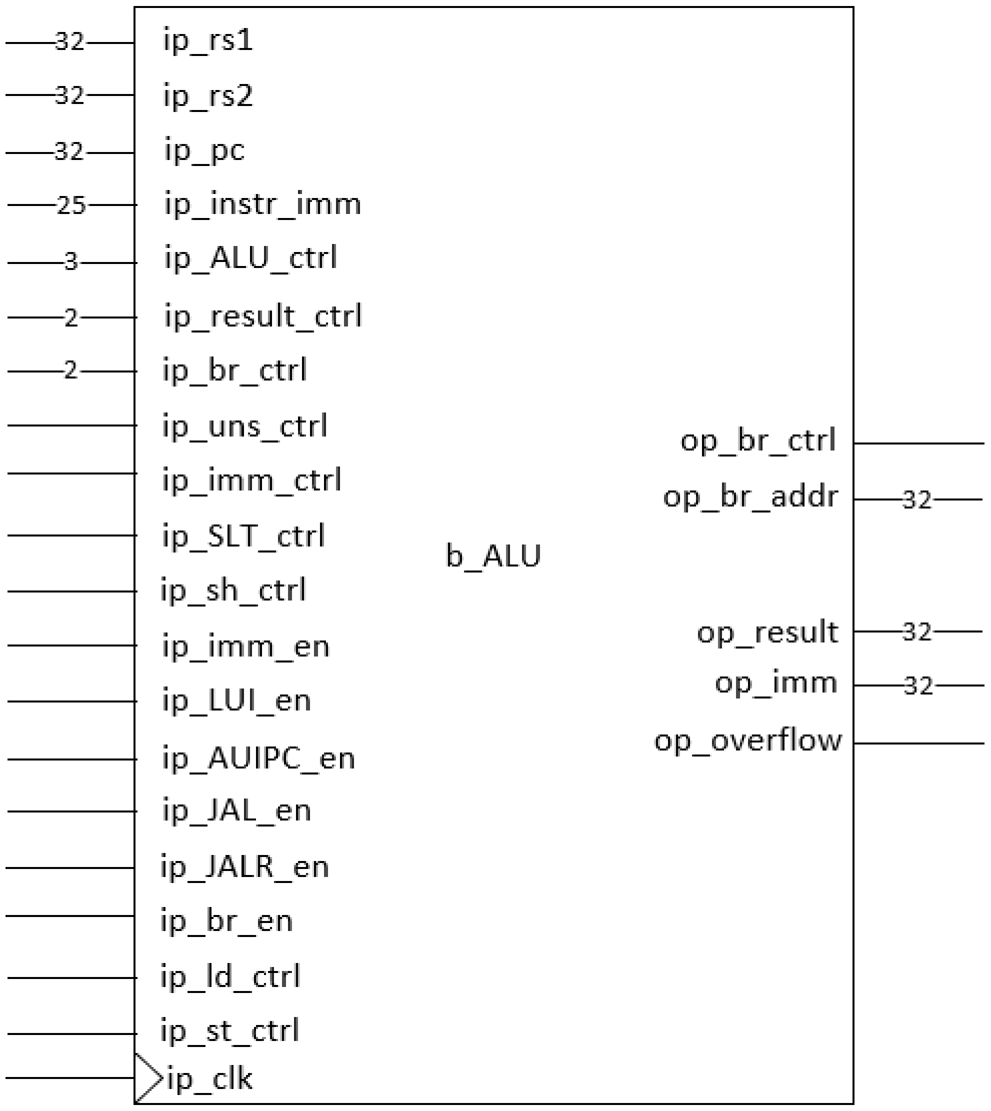


Figure 4.4.2 – ALU interface

- **I/O Pin Description**

Pin name: ip_rs1[31:0] **Pin class:** data **Pin function:** A 32 bits data of register 1 for computing.	**Source:** Register File **Destination:** ALU
Pin name: ip_rs2[31:0] **Pin class:** data **Pin function:** A 32 bits data of register 2 for computing.	**Source:** Register File **Destination:** ALU
Pin name: ip_instr_imm[24:0] **Pin class:** data **Pin function:** A 25 bits immediate data for computing.	**Source:** Instruction Fetch Unit **Destination:** Register File
Pin name: ip_ALU_ctrl[2:0] **Pin class:** control **Pin function:** A 3 bits control pin to determine using AND, OR, XOR, ADD or SUB for arithmetic and logical computing.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_result_ctrl[1:0] **Pin class:** control **Pin function:** A 2 bits control pin to determine output from ALU, barrel shifter or result of set instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_br_ctrl[1:0] **Pin class:** control **Pin function:** A 2 bits control pin to determine which branch instruction is used for comparing two data	**Source:** ALU Control **Destination:** ALU
Pin name: ip_uns_ctrl **Pin class:** control **Pin function:** A control pin to determine the instruction is involved unsigned data.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_imm_ctrl **Pin class:** control **Pin function:** A control pin to determine the instruction is involved immediate data.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_SLT_ctrl **Pin class:** control **Pin function:** A control pin to determine the instruction is involved set instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_sh_ctrl **Pin class:** control **Pin function:** A control pin to determine the barrel shifter shifting direction.	**Source:** ALU Control **Destination:** ALU

Pin name: ip_imm_en **Pin class:** control **Pin function:** A control pin to determine the instruction is involved I-type instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_LUI_en **Pin class:** control **Pin function:** A control pin to determine the instruction is LUI Instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_AUIPC_en **Pin class:** control **Pin function:** A control pin to determine the instruction is AUIPC Instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_JAL_en **Pin class:** control **Pin function:** A control pin to determine the instruction is JAL Instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_JALR_en **Pin class:** control **Pin function:** A control pin to determine the instruction is JALR Instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_br_en **Pin class:** control **Pin function:** A control pin to determine the instruction is involved branch instruction.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_ld_en **Pin class:** control **Pin function:** A control pin to determine the instruction is involved load data instruction from data memory.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_st_en **Pin class:** control **Pin function:** A control pin to determine the instruction is involved store data instruction from data memory.	**Source:** ALU Control **Destination:** ALU
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** ALU
Pin name: op_result[31:0] **Pin class:** data **Pin function:** A 32 bits data of result after computing in ALU.	**Source:** ALU **Destination:** Data Memory / Register File
Pin name: op_imm[31:0] **Pin class:** data	**Source:** ALU **Destination:** Data Memory

Pin function: A 32 bits immediate data for storing into data memory	
Pin name: op_overflow **Pin class:** control **Pin function:** A pin for determine the computing result is overflow or error, it only occurred when over adding or subtract or shifting	**Source:** ALU **Destination:** Data Memory
Pin name: op_br_addr[31:0] **Pin class:** address **Pin function:** A 32 bits of address for branch target address after calculate offset with an address	**Source:** ALU **Destination:** Instruction Fetch Unit
Pin name: op_br_ctrl **Pin class:** ctrl **Pin function:** A control pin to determine the control output branch pin is 0 or 1 by comparing result of trueness.	**Source:** ALU **Destination:** Instruction Fetch Unit

Table 4.4.2 – ALU pin description

4.4.3 Internal Operation

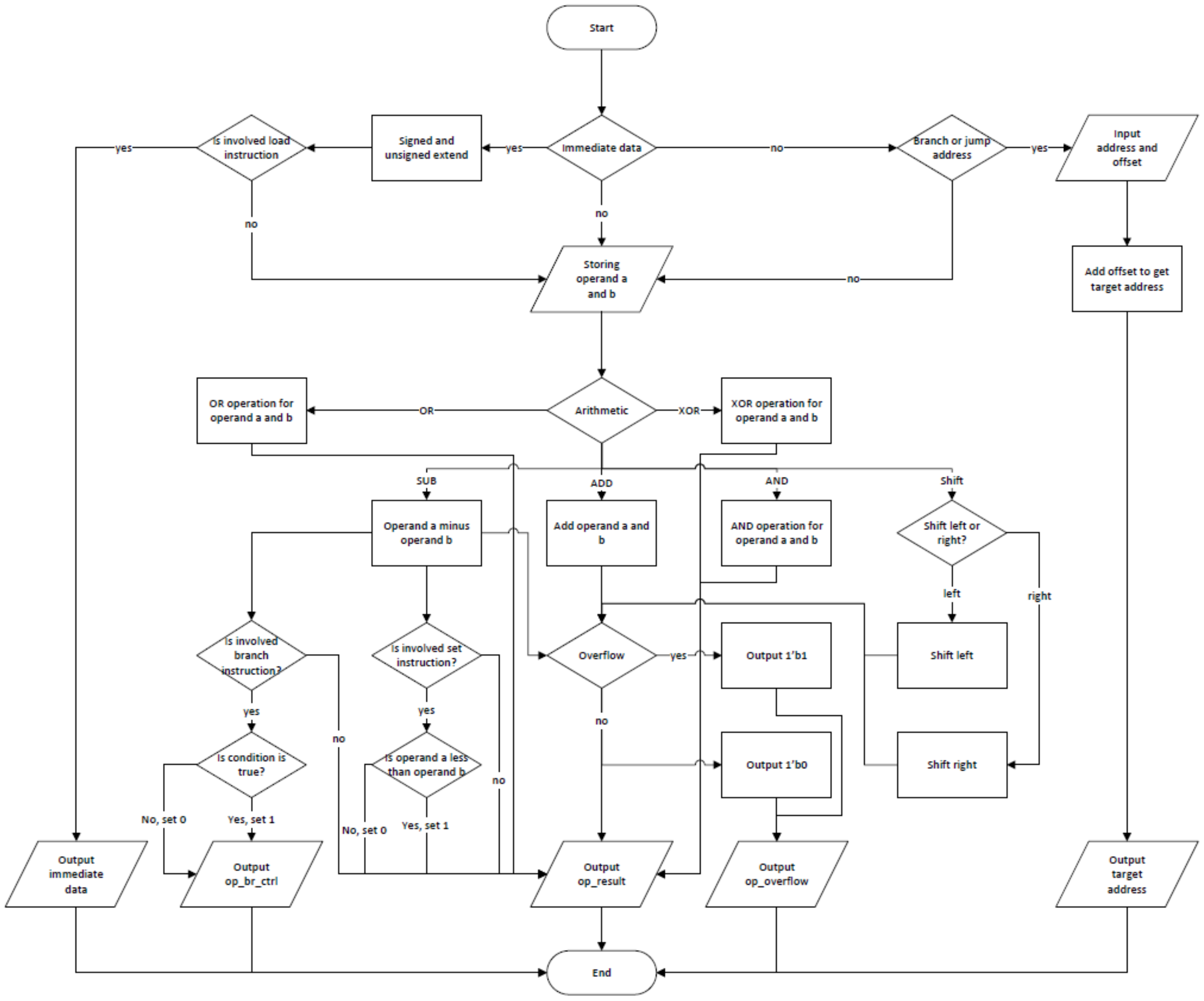


Figure 4.4.3 – Flowchart of ALU operation

4.4.4 Schematic Design

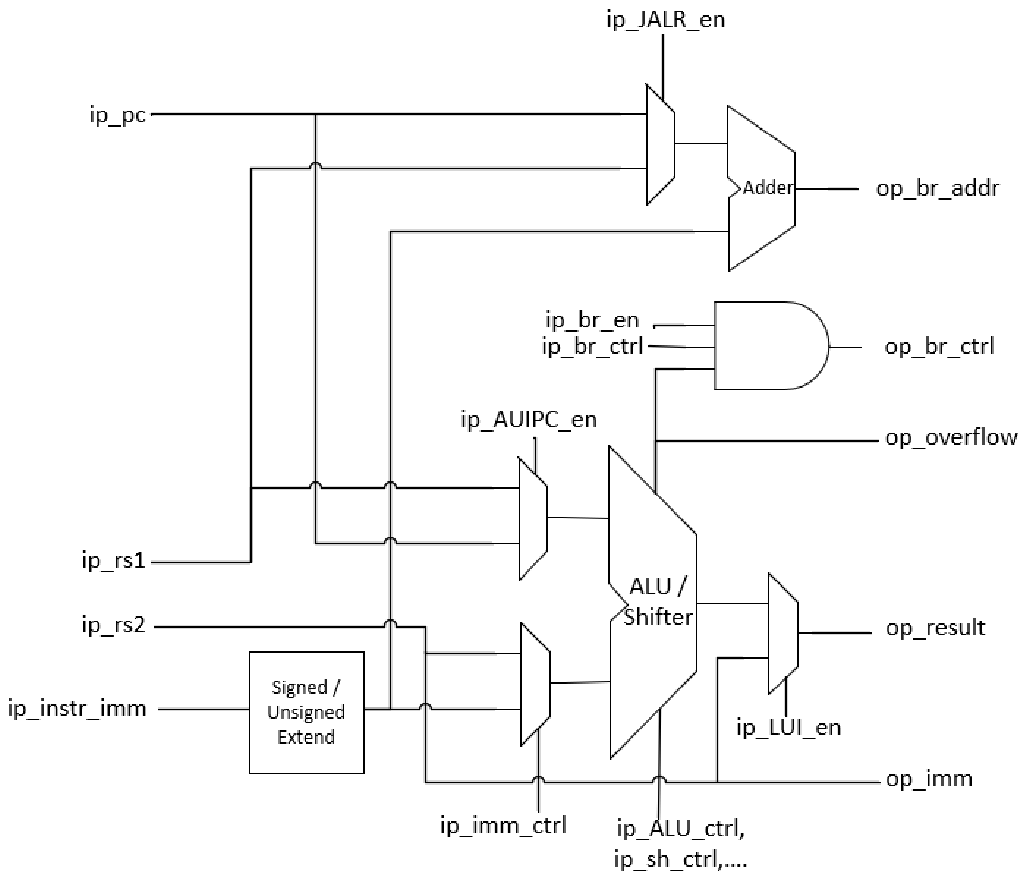


Figure 4.4.4 – Schematic design of ALU

4.4.5 Verilog Model

```
/*********************************************************************
Project: Develop Extended ISA on RISC-V Based Processor
Module: b_ALU.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: ALU
*********************************************************************/

`default_nettype none//to catch typing errors due to typo of signal names

module b_ALU
#()
( // declare all the input and output pin needed
        input wire [31:0] ip_rs1, ip_rs2, ip_pc,
        input wire [24:0] ip_instr_imm,
        input wire [2:0] ip_ALU_ctrl,
        input wire [1:0] ip_result_ctrl, ip_br_ctrl,
        input wire ip_clk, ip_uns_ctrl, ip_imm_ctrl, ip_SLT_ctrl, ip_sh_ctrl, ip_imm_en, ip_LUI_en,
ip_AUIPC_en, ip_JAL_en, ip_JALR_en, ip_br_en, ip_ld_ctrl, ip_st_ctrl,
        output reg [31:0] op_result, op_imm, op_br_addr,
        output reg op_overflow, op_br_ctrl
);

reg [31:0] operand_a; // operand a of ALU
reg [31:0] operand_b; // operand b of ALU
reg [31:0] imm_ext; // immediate data extand
reg [31:0] result_ALU; // result of ALU
reg [31:0] result_sh; // result of barrel shifter
reg result_SLT; // result of SLT
reg [32:0] carry_bit_ALU; // adder carry bit
reg [31:0] pc; // address for adding offset
reg [31:0] br_addr; // branch address shift left 2 bit
reg [32:0] carry_bit_br_adder; // branch adder carry bit


integer i; // for loop

// zero/sign extand if immediate data
always @(posedge ip_clk) begin
        if(ip_JAL_en == 1'b1) begin // if is J-type instruction (jump instruction, JAL)
                if(ip_instr_imm[24] == 1'b0) begin
                        br_addr[31:0] = {12'b0, ip_instr_imm[24:5]};
                end

                else if(ip_instr_imm[24] == 1'b1) begin
                        br_addr[31:0] = {12'b111111111111, ip_instr_imm[24:5]};
                end
        end

        else if (ip_JALR_en == 1'b1) begin
                if(ip_instr_imm[24] == 1'b0) begin
                        br_addr[31:0] = {20'b0, ip_instr_imm[24:13]};
                end

                else if(ip_instr_imm[24] == 1'b1) begin
                        br_addr[31:0] = {20'b11111111111111111111, ip_instr_imm[24:13]};
        end
    end

    else if(ip_br_en == 1'b1) begin // if is B-type instruction (branch instruction)
        if(ip_uns_ctrl == 1'b0) begin
            if(ip_instr_imm[24] == 1'b0) begin
                br_addr[31:0] = {20'b0, ip_instr_imm[24:18], ip_instr_imm[4:0]};
            end

            else if(ip_instr_imm[24] == 1'b1) begin
                br_addr[31:0]        =        {20'b11111111111111111111,
ip_instr_imm[24:18], ip_instr_imm[4:0]};
            end
        end

        else if(ip_uns_ctrl == 1'b1) begin
            br_addr[31:0] = {20'b0, ip_instr_imm[24:18], ip_instr_imm[4:0]};
        end
    end

    else if((ip_imm_en == 1'b1) || (ip_ld_ctrl == 1'b1)) begin // if is I-type instruction (instruction
with immediate)
        if(ip_uns_ctrl == 1'b0) begin
            if(ip_instr_imm[24] == 1'b0) begin
                imm_ext[31:0] = {20'b0, ip_instr_imm[24:13]};
            end

            else if(ip_instr_imm[24] == 1'b1) begin
                imm_ext[31:0]        =        {20'b11111111111111111111,
ip_instr_imm[24:13]};
            end
        end

        else if(ip_uns_ctrl == 1'b1) begin
            imm_ext[31:0] = {20'b0, ip_instr_imm[24:13]};
        end
    end

    else if(ip_st_ctrl == 1'b1) begin
        if(ip_instr_imm[24] == 1'b0) begin
            imm_ext[31:0] = {20'b0, ip_instr_imm[24:18], ip_instr_imm[4:0]};
        end

        else if(ip_instr_imm[24] == 1'b1) begin
            imm_ext[31:0]    =    {20'b11111111111111111111,    ip_instr_imm[24:18],
ip_instr_imm[4:0]};
        end
    end

    else if((ip_AUIPC_en == 1'b1) || (ip_LUI_en == 1'b1)) begin // if is U-type instruction (upper
instrcution AUIPC, LUI)
        imm_ext[31:0] = {ip_instr_imm[24:5], 12'b0};
    end


    // a operand for calculate in ALU
    if((ip_AUIPC_en == 1'b0) && (ip_LUI_en == 1'b0) && (ip_JAL_en == 1'b0) || (ip_JALR_en
== 1'b1)) begin // normal case or JALR
        operand_a[31:0] = ip_rs1[31:0];
    end
```

```
        else if((ip_AUIPC_en == 1'b1) || (ip_JAL_en == 1'b1)) begin // if AUIPC or JAL
                operand_a[31:0] = ip_pc[31:0];
        end

        else if(ip_LUI_en == 1'b1) begin // if LUI
                operand_a[31:0] = 32'b0;
        end

        // b operand for calculate in ALU
        if((ip_JAL_en == 1'b1) || (ip_JALR_en == 1'b1)) begin // if jump, then +4 to store the next
address
                operand_b[31:0] = 32'h00000004;
        end

        else if((ip_JAL_en == 1'b0) && (ip_JALR_en == 1'b0)) begin // if not jump, store the signed
extended or rs2 data
                if(ip_imm_ctrl == 1'b0) begin
                        operand_b[31:0] = ip_rs2[31:0];
                end

                else if(ip_imm_ctrl == 1'b1) begin
                        operand_b[31:0] = imm_ext[31:0];
                end
        end
end

// ALU
always @(posedge ip_clk) begin
        if(ip_ALU_ctrl[2] == 1'b1) begin // if subtration selected
                // convert operand b
                for(i = 0; i < 32; i = i + 1) begin
                        operand_b[i] = operand_b[i] ^ ip_ALU_ctrl[2];
                end
        end

        if(ip_ALU_ctrl[1:0] == 2'b00) begin
                result_ALU[31:0] = operand_a[31:0] & operand_b[31:0];
        end

        else if(ip_ALU_ctrl[1:0] == 2'b01) begin
                result_ALU[31:0] = operand_a[31:0] | operand_b[31:0];
        end

        else if(ip_ALU_ctrl[1:0] == 2'b10) begin
                result_ALU[31:0] = operand_a[31:0] ^ operand_b[31:0];
        end

        else if(ip_ALU_ctrl[1:0] == 2'b11) begin
                // 32-bit Adder
                carry_bit_ALU[0] = ip_ALU_ctrl[2]; // subtraction gate
                for(i = 0; i < 32; i = i + 1) begin
                        // calculation of add result
                        result_ALU[i] = operand_a[i] ^ operand_b[i] ^ carry_bit_ALU[i];
                        // calculation of carry bit
                        carry_bit_ALU[i + 1'b1] = (operand_a[i] & operand_b[i]) | (operand_a[i] &
carry_bit_ALU[i]) | (operand_b[i] & carry_bit_ALU[i]);
                end

        // signed addition overflow
        op_overflow = (carry_bit_ALU[32] ^ carry_bit_ALU[31]) & (~ip_uns_ctrl);
        // set if SLT/SLTU is true
        result_SLT = ((op_overflow ^ result_ALU[31]) & ip_SLT_ctrl & (~ip_uns_ctrl)) |
(~(carry_bit_ALU[32]) & ip_SLT_ctrl & ip_uns_ctrl);
        end

        // jump condition
        if(ip_br_en == 1'b1) begin
                if(ip_br_ctrl[1:0] == 2'b00) begin // if BEQ
                        if(result_ALU[31:0] == 32'b0) begin
                                op_br_ctrl = 1'b1;
                        end

                        else if(result_ALU[31:0] != 32'b0) begin
                                op_br_ctrl = 1'b0;
                        end
                end

                else if(ip_br_ctrl[1:0] == 2'b01) begin // if BNE
                        if(result_ALU[31:0] != 32'b0) begin
                                op_br_ctrl = 1'b1;
                        end

                        else if(result_ALU[31:0] == 32'b0) begin
                                op_br_ctrl = 1'b0;
                        end
                end

                else if(ip_br_ctrl[1:0] == 2'b10) begin // BLT
                        if(result_ALU[31] == 1'b1) begin
                                op_br_ctrl = 1'b1;
                        end

                        else if(result_ALU[31] == 1'b0) begin
                                op_br_ctrl = 1'b0;
                        end
                end

                else if(ip_br_ctrl[1:0] == 2'b11) begin // BGE
                        if(result_ALU[31] == 1'b0) begin
                                op_br_ctrl = 1'b1;
                        end

                        else if(result_ALU[31] == 1'b1) begin
                                op_br_ctrl = 1'b0;
                        end
                end
        end

        else if (ip_br_en == 1'b0) begin
                op_br_ctrl = 1'b0;
        end

        // control the output result
        if(ip_result_ctrl[1:0] == 2'b00) begin
                op_result[31:0] = result_ALU[31:0];
        end

        else if(ip_result_ctrl[1:0] == 2'b01) begin
                op_result[31:0] = {31'b0000000000000000000000000000000, result_SLT};
        end

        if(ip_st_ctrl == 1'b1) begin
                op_imm[31:0] = ip_rs2[31:0];
        end

        else if(ip_st_ctrl == 1'b0) begin
                op_imm[31:0] = 32'b0;
        end
end

// barrel shifter
always @(posedge ip_clk) begin
        if(ip_sh_ctrl == 1'b0) begin // shift left logical and arithmetic
                if(operand_b[0] == 1'b1) begin
                        result_sh[31:0] = {ip_rs1[31:1], 1'b0};
                end

                if(operand_b[1] == 1'b1) begin
                        result_sh[31:0] = {ip_rs1[31:2], 2'b00};
                end

                if(operand_b[2] == 1'b1) begin
                        result_sh[31:0] = {ip_rs1[31:4], 4'b0000};
                end

                if(operand_b[3] == 1'b1) begin
                        result_sh[31:0] = {ip_rs1[31:8], 8'b00000000};
                end

                if(operand_b[4] == 1'b1) begin
                        result_sh[31:0] = {ip_rs1[31:16], 16'b0000000000000000};
                end
        end

        else if(ip_sh_ctrl == 1'b1) begin // shift right
                if(ip_uns_ctrl == 1'b1) begin // shift right logical
                        if(operand_b[0] == 1'b1) begin
                                result_sh[31:0] = {1'b0, ip_rs1[31:1]};
                        end

                        else if(operand_b[1] == 1'b1) begin
                                result_sh[31:0] = {2'b00, ip_rs1[31:2]};
                        end

                        else if(operand_b[2] == 1'b1) begin
                                result_sh[31:0] = {4'b0000, ip_rs1[31:4]};
                        end

                        else if(operand_b[3] == 1'b1) begin
                                result_sh[31:0] = {8'b00000000, ip_rs1[31:8]};
                        end

                        else if(operand_b[4] == 1'b1) begin
                                result_sh[31:0] = {16'b0000000000000000, ip_rs1[31:16]};
                        end
                end

                else if(ip_uns_ctrl == 1'b0) begin // shift right arithmetic
                        if(operand_b[0] == 1'b1) begin
```

```
            if(ip_rs1[31] == 1'b0) begin
                result_sh[31:0] = {1'b0, ip_rs1[31:1]};
            end

            else if(ip_rs1[31] == 1'b1) begin
                result_sh[31:0] = {1'b1, ip_rs1[31:1]};
            end
        end

        else if(operand_b[1] == 1'b1) begin
            if(ip_rs1[31] == 1'b0) begin
                result_sh[31:0] = {2'b00, ip_rs1[31:2]};
            end

            else if(ip_rs1[31] == 1'b1) begin
                result_sh[31:0] = {2'b11, ip_rs1[31:2]};
            end
        end

        else if(operand_b[2] == 1'b1) begin
            if(ip_rs1[31] == 1'b0) begin
                result_sh[31:0] = {4'b0000, ip_rs1[31:4]};
            end

            else if(ip_rs1[31] == 1'b1) begin
                result_sh[31:0] = {4'b1111, ip_rs1[31:4]};
            end
        end

        else if(operand_b[3] == 1'b1) begin
            if(ip_rs1[31] == 1'b0) begin
                result_sh[31:0] = {8'b00000000, ip_rs1[31:8]};
            end

            else if(ip_rs1[31] == 1'b1) begin
                result_sh[31:0] = {8'b11111111, ip_rs1[31:8]};
            end
        end

        else if(operand_b[4] == 1'b1) begin
            if(ip_rs1[31] == 1'b0) begin
               result_sh[31:0] = {16'b0000000000000000, ip_rs1[31:16]};
            end

            else if(ip_rs1[31] == 1'b1) begin
               result_sh[31:0] = {16'b1111111111111111, ip_rs1[31:16]};
            end
        end
    end
end

if((ip_result_ctrl[1:0] == 2'b10) && (operand_b[31:6] != 27'b0)) begin
    op_overflow = 1'b1;
end

else begin
    op_overflow = 1'b0;
end

if(ip_result_ctrl == 2'b10) begin
```

```
			op_result[31:0] = result_sh[31:0];
		end
end

// branch or jump address adder
always @(posedge ip_clk) begin
		br_addr[31:0] = {br_addr[30:0], 1'b0};
		if(ip_JALR_en == 1'b1) begin
			pc[31:0] = ip_rs1[31:0];
		end

		else if((ip_JAL_en == 1'b1) || (ip_br_en == 1'b1)) begin
			pc[31:0] = ip_pc[31:0];
		end

		// 32-bit Adder
		carry_bit_br_adder[0] = 1'b0;
		for(i = 0; i < 32; i = i + 1) begin
			// calculation of add result
			op_br_addr[i] = pc[i] ^ br_addr[i] ^ carry_bit_br_adder[i];
			// calculation of carry bit
			carry_bit_br_adder[i + 1'b1] = (pc[i] & br_addr[i]) | (pc[i] & carry_bit_br_adder[i]) |
(br_addr[i] & carry_bit_br_adder[i]);
		end
end
endmodule
```

4.5 Data Memory

4.5.1 Functionality and Feature

Data Memory used to store or load data, and stack for supply additional places despite register are smallest memory unit. Static data, dynamic data and stack data all are defined in the data memory. It can load or store different length of data such as byte, half and word. The data will be according to the address given by instruction which is calculated by ALU to store or load relatively in data memory.

4.5.2 Interface and I/O Pin Description

- **Interface**

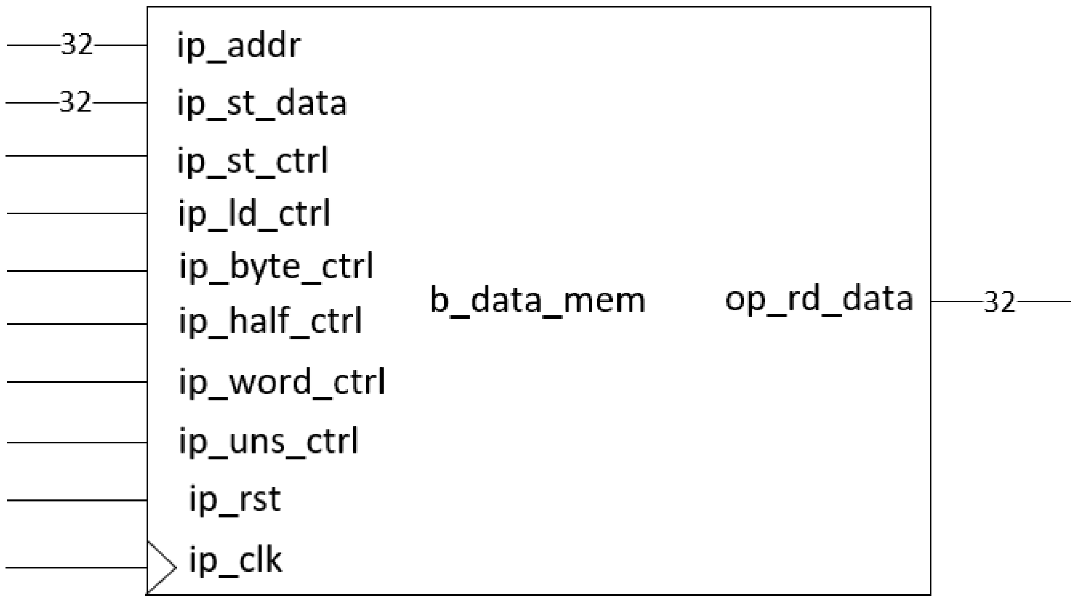


Figure 4.5.2 – Data memory interface

- **I/O Pin Description**

Pin name: ip_addr[31:0] **Pin class:** address **Pin function:** A 32 bits address of data memory for load or store a data.	**Source:** ALU **Destination:** Data Memory
Pin name: ip_st_data[31:0] **Pin class:** data **Pin function:** A 32 bits data for storing data.	**Source:** ALU **Destination:** Data Memory
Pin name: ip_st_ctrl **Pin class:** control **Pin function:** A control pin to enable for storing data.	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_ld_ctrl **Pin class:** control **Pin function:** A control pin to enable for load a data.	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_byte_ctrl **Pin class:** control **Pin function:** A control pin to allow for load or store a byte of data.	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_half_ctrl **Pin class:** control **Pin function:** A control pin to allow for load a half of data.	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_word_ctrl **Pin class:** control **Pin function:** A control pin to allow for load a word of data.	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_uns_ctrl **Pin class:** control **Pin function:** A control pin to control extension of signed or unsigned when load out a data	**Source:** Control Unit **Destination:** Data Memory
Pin name: ip_rst **Pin class:** global **Pin function:** A pin to data memory	**Source:** Datapath **Destination:** Data Memory
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** Data Memory
Pin name: op_rd_data[31:0] **Pin class:** data **Pin function:** A 32 bits data that load out from data memory according to the address	**Source:** Data Memory **Destination:** Register

Table 4.5.2 – Data memory pin description

4.5.3 Internal Operation

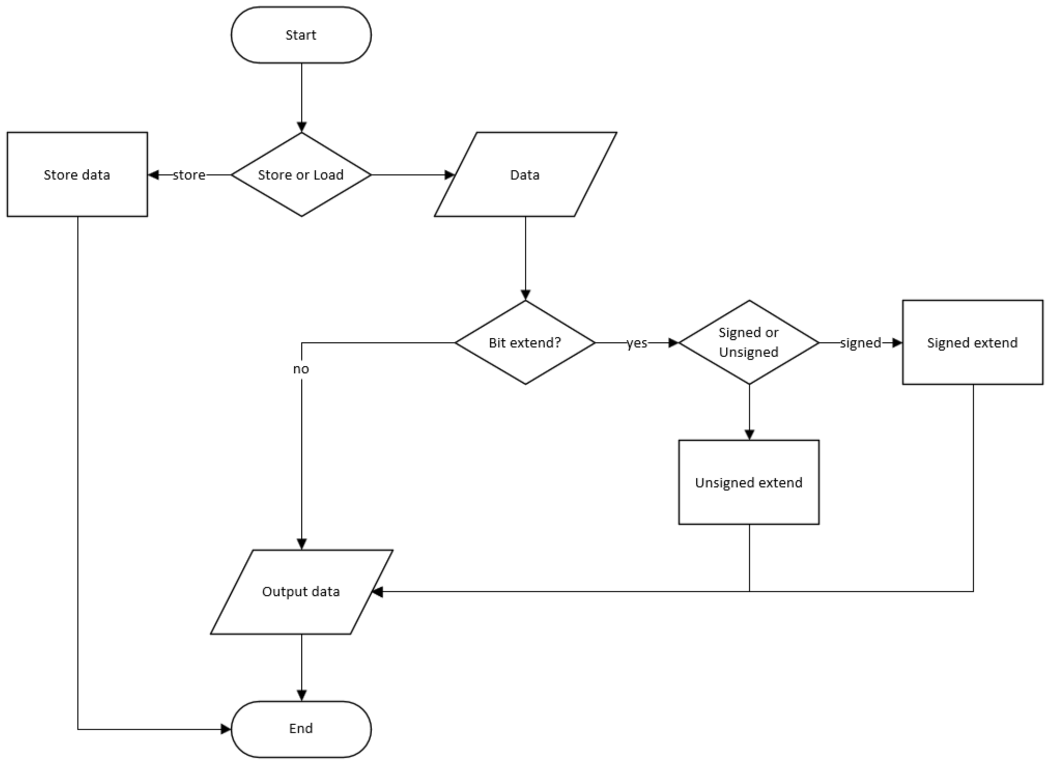


Figure 4.5.3 – Flowchart of data memory operation

4.5.4 Schematic Design

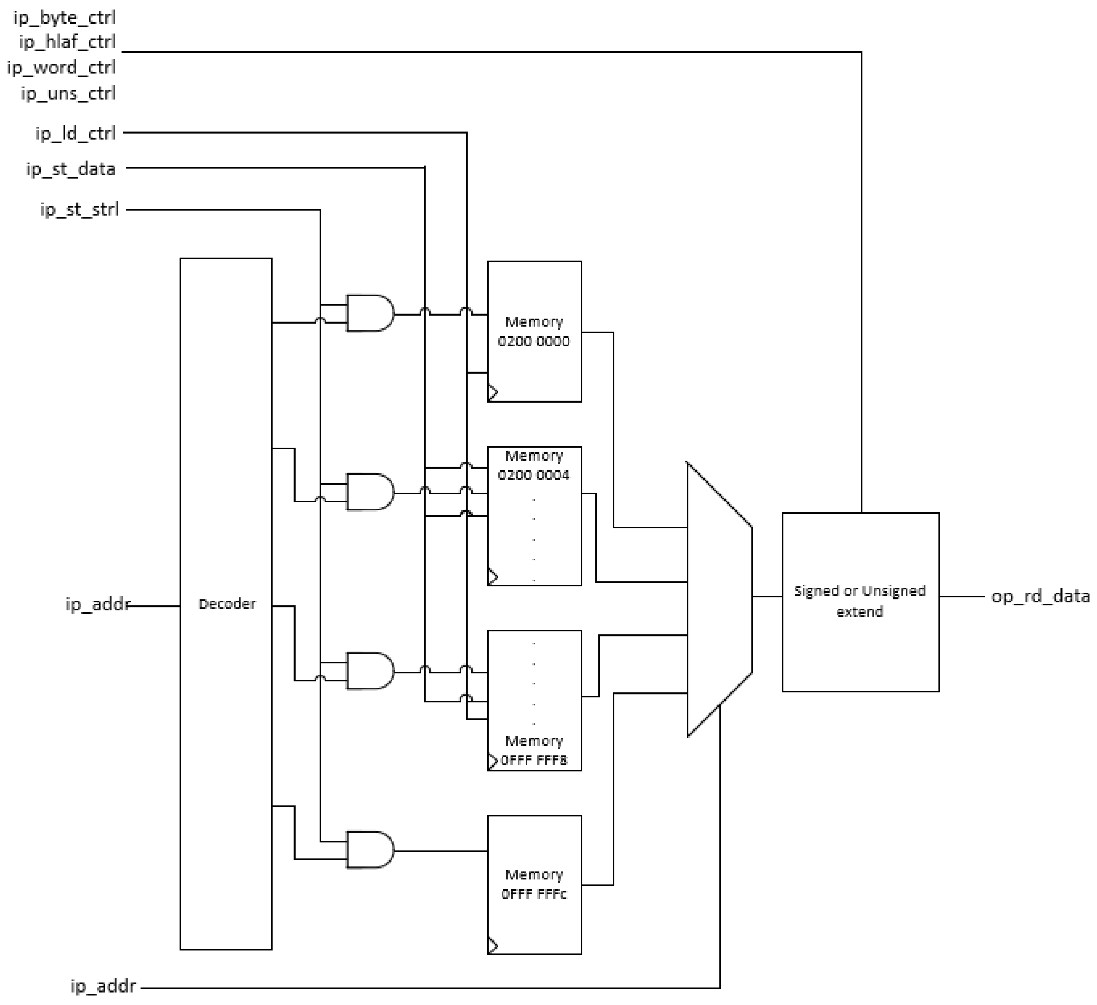


Figure 4.5.4 – Schematic design of data memory

4.5.5 Verilog Model

```
/*********************************************************************
Project: Develop Extended ISA on RISC-V Based Processor
Module: b_data_mem.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: Data Memory Block
*********************************************************************/

`default_nettype none  // to catch typing errors due to typo of signal names

module b_data_mem
#( // declare all the parameter needed
parameter initial_static_data_addr = 32'h02000000, // initial static data address
parameter last_static_data_addr = 32'h02003FFF, // last static data address
parameter initial_dynamic_data_addr = 32'h02004000, // initial dynamic data address
parameter initial_stack_data_addr = 32'h0FFFFFFC, // initial stack_data address
parameter last_data_mem_addr = 32'h0FFFFFFF // last data memory address
)
( // declare all the input and output pin needed
        input wire [31:0] ip_addr, ip_st_data,
        input wire ip_clk, ip_rst, ip_ld_en, ip_st_en, ip_byte_ctrl, ip_half_ctrl, ip_word_ctrl,
ip_uns_ctrl,
        output reg [31:0] op_rd_data
);

reg [8:0] data_mem [initial_static_data_addr:last_data_mem_addr]; // data memory (static + dynamic +
stack)
reg [31:0] data_mem_addr;

// data memory
always @(posedge ip_clk) begin
        if(ip_rst == 1'b1) begin // if reset
                // reset to default value
                for(data_mem_addr[31:0] = initial_static_data_addr; data_mem_addr[31:0] <=
last_data_mem_addr; data_mem_addr[31:0] = data_mem_addr[31:0] + 32'h1) begin
                        data_mem[data_mem_addr[31:0]][7:0] <= 8'b0;
                end
                data_mem_addr[31:0] <= initial_static_data_addr;
        end

        else if(ip_rst == 1'b0) begin // if no reset
                if(ip_ld_en == 1'b1) begin
                        if(ip_byte_ctrl == 1'b1) begin // if LB/LBU
                                if(ip_uns_ctrl == 1'b0) begin
                                        if(data_mem[ip_addr[31:0]][7] == 1'b0) begin
                                                op_rd_data[7:0] <=
data_mem[ip_addr[31:0]][7:0];
                                                op_rd_data[31:8] <= 24'b0;
                                        end

                                        else if(data_mem[ip_addr[31:0]][7] == 1'b1) begin
                                                op_rd_data[7:0] <=
data_mem[ip_addr[31:0]][7:0];
                                                op_rd_data[31:8] <=
24'b111111111111111111111111;
                                        end
                                        end

                                        else if(ip_uns_ctrl == 1'b1) begin
                                                  op_rd_data[7:0] <= data_mem[ip_addr[31:0]][7:0];
                                                  op_rd_data[31:8] <= 24'b0;
                                        end
                              end

                              if(ip_half_ctrl == 1'b1) begin // if LH/LHU
                                        if(ip_uns_ctrl == 1'b0) begin
                                                  if(data_mem[ip_addr[31:0] + 32'h1][7] == 1'b0) begin
                                                            op_rd_data[7:0] <=
data_mem[ip_addr[31:0]][7:0];
                                                            op_rd_data[15:8] <= data_mem[ip_addr[31:0 +
32'h1]][7:0];
                                                            op_rd_data[31:16] <= 16'b0;
                                                  end

                                                  else if(data_mem[ip_addr[31:0] + 32'h1][7] == 1'b1)
begin
                                                            op_rd_data[7:0] <=
data_mem[ip_addr[31:0]][7:0];
                                                            op_rd_data[15:8] <= data_mem[ip_addr[31:0 +
32'h1]][7:0];
                                                            op_rd_data[31:16] <= 16'b1111111111111111;
                                                  end
                                        end

                                        else if(ip_uns_ctrl == 1'b1) begin
                                                  op_rd_data[7:0] <= data_mem[ip_addr[31:0]][7:0];
                                                  op_rd_data[15:8] <= data_mem[ip_addr[31:0 +
32'h1]][7:0];
                                                  op_rd_data[31:16] <= 16'b0;
                                        end
                              end

                              if(ip_word_ctrl == 1'b1) begin // if LW
                                        op_rd_data[7:0] <= data_mem[ip_addr[31:0]][7:0];
                                        op_rd_data[15:8] <= data_mem[ip_addr[31:0] + 32'h1][7:0];
                                        op_rd_data[23:16] <= data_mem[ip_addr[31:0] + 32'h2][7:0];
                                        op_rd_data[31:24] <= data_mem[ip_addr[31:0] + 32'h3][7:0];
                              end
                    end

                    else if (ip_ld_ctrl == 1'b0) begin
                              op_rd_data[31:0] <= 32'b0;
                    end

                    if(ip_st_en == 1'b1)begin // if store data
                              if(ip_byte_ctrl == 1'b1) begin // if SB
                                        data_mem[ip_addr[31:0]][7:0] <= ip_st_data[7:0];
                              end

                              if(ip_half_ctrl == 1'b1) begin // if SH
                                        data_mem[ip_addr[31:0]][7:0] <= ip_st_data[7:0];
                                        data_mem[ip_addr[31:0] + 32'h1][7:0] <= ip_st_data[15:8];
                              end

                              if(ip_word_ctrl == 1'b1) begin // if SW
                                        // write data in to data memory
                                        data_mem[ip_addr[31:0]][7:0] <= ip_st_data[7:0];
                                        data_mem[ip_addr[31:0] + 32'h1][7:0] <= ip_st_data[15:8];
                                        data_mem[ip_addr[31:0] + 32'h2][7:0] <= ip_st_data[23:16];
                                        data_mem[ip_addr[31:0] + 32'h3][7:0] <= ip_st_data[31:24];
                              end
                    end
          end
end
endmodule
```

4.6 Main Control

4.6.1 Functionality and Feature

Main control unit is a component which decoded instructions set and get the certain bits value and generate control signal. Main control block decodes the instructions and get certain bits value and indicate the value for certain control signal which needed to control the execution, operation of the CPU and enables read or write to or from memory nor register.

4.6.2 Interface and I/O Pin Description

- **Interface**

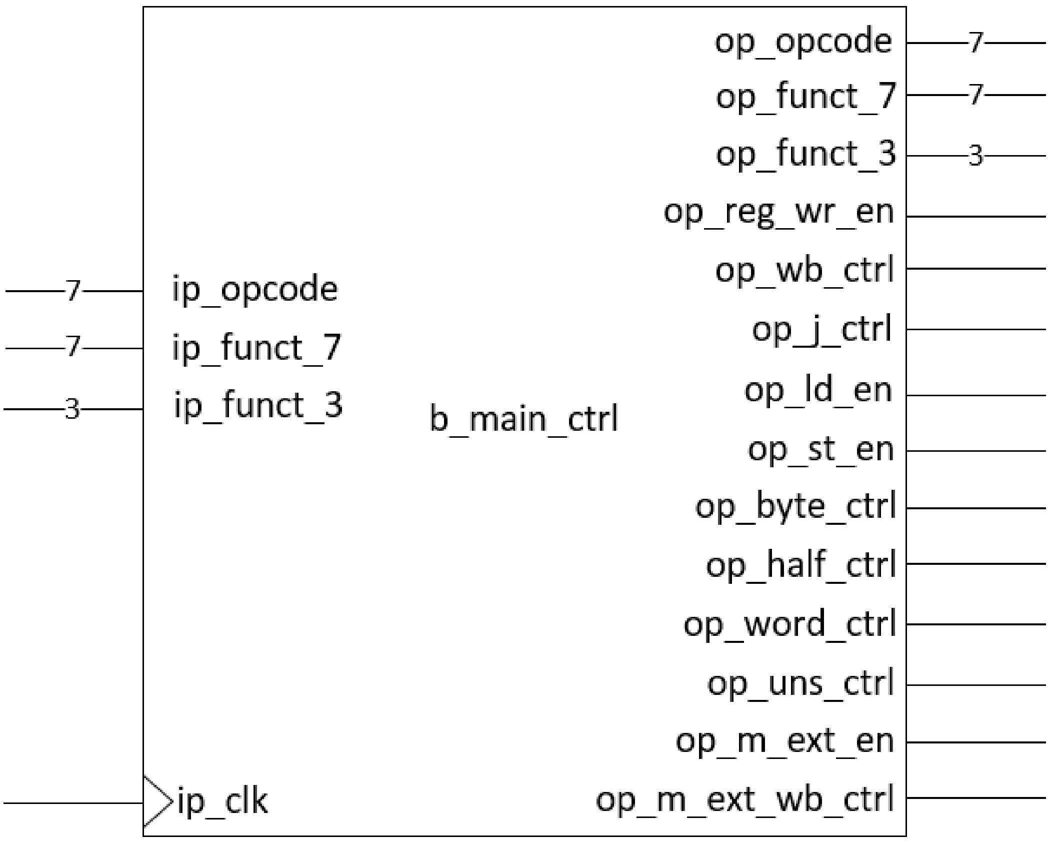


Figure 4.6.2 – Main Control interface

- **I/O Pin Description**

Pin name: ip_opcode[6:0] **Pin class:** control **Pin function:** A 7 bits of instruction ID.	**Source:** Instruction Fetch Unit **Destination:** Main Control
Pin name: ip_funct_7[6:0] **Pin class:** control **Pin function:** A 7 bits function signal for define other types of control.	**Source:** Instruction Fetch Unit **Destination:** Main Control
Pin name: ip_funct_3[3:0] **Pin class:** control **Pin function:** A 3 bits function signal for defined the arithmetic and logical used by instruction.	**Source:** Instruction Fetch Unit **Destination:** Main Control
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** Main Control
Pin name: op_opcode[6:0] **Pin class:** control **Pin function:** Output the same 7 bits of instruction ID from input for other control unit.	**Source:** Main Control **Destination:** ALU Control
Pin name: op_funct_7[6:0] **Pin class:** control **Pin function:** Output the same 7 bits of function signal from input for other control unit.	**Source:** Main Control **Destination:** ALU Control
Pin name: op_funct_3[2:0] **Pin class:** control **Pin function:** Output the same 3 bits of function signal from input for other control unit.	**Source:** Main Control **Destination:** ALU Control
Pin name: op_reg_wr_en **Pin class:** control **Pin function:** A control pin to enable write back the data in register file	**Source:** Main Control **Destination:** Datapath
Pin name: op_wb_ctrl **Pin class:** control **Pin function:** A control pin to determine data from ALU or data memory for write back into register file	**Source:** Main Control **Destination:** Datapath
Pin name: op_j_ctrl **Pin class:** control **Pin function:** A control pin to determine jump instruction is occurred	**Source:** Main Control **Destination:** Instruction Fetch Unit
Pin name: op_ld_ctrl **Pin class:** control	**Source:** Main Control **Destination:** Data Memory

Pin function: A control pin to determine load instruction is occurred	
Pin name: op_st_ctrl **Pin class:** control **Pin function:** A control pin to determine store instruction is occurred	**Source:** Main Control **Destination:** Data Memory
Pin name: op_byte_ctrl **Pin class:** control **Pin function:** A control pin to determine load 8 bit wide data	**Source:** Main Control **Destination:** Data Memory
Pin name: op_half_ctrl **Pin class:** control **Pin function:** A control pin to determine load 16 bit wide data	**Source:** Main Control **Destination:** Data Memory
Pin name: op_word_ctrl **Pin class:** control **Pin function:** A control pin to determine load 32 bit wide data	**Source:** Main Control **Destination:** Data Memory
Pin name: op_uns_ctrl **Pin class:** control **Pin function:** A control pin to determine loaded data is an unsigned extend or signed extend	**Source:** Main Control **Destination:** Data Memory
Pin name: op_m_ext_en **Pin class:** control **Pin function:** A control pin to determine is M extension is required to use	**Source:** Main Control **Destination:** M Extension
Pin name: op_m_ext_wb_ctrl **Pin class:** control **Pin function:** A control pin to determine is data from M extension for write back into register file	**Source:** Main Control **Destination:** Datapath

Table 4.6.2 – Main control pin description

4.6.3 Internal Operation

Input			Output										
ip_funct_7[0]	**ip_funct_3**	**ip_opcode**	**op_reg_wr_en**	**op_wb_ctrl**	**op_j_ctrl**	**op_ld_en**	**op_st_en**	**op_byte_ctrl**	**op_half_ctrl**	**op_word_ctrl**	**op_uns_ctrl**	**op_m_ext_en**	**op_m_ext_wb_ctrl**
x	x	0110111	1	0	0	0	0	x	x	x	x	0	0
x	x	0010111	1	0	0	0	0	x	x	x	x	0	0
x	x	1101111	1	0	1	0	0	x	x	x	x	0	0
x	x	1100111	1	0	1	0	0	x	x	x	x	0	0
x	x	1100011	1	0	0	0	0	x	x	x	x	0	0
x	000		1	1	0	1	0	1	0	0	0	0	0
x	001		1	1	0	1	0	0	1	0	0	0	0
x	010	0000011	1	1	0	1	0	0	0	1	0	0	0
x	100		1	1	0	1	0	1	0	0	1	0	0
x	101		1	1	0	1	0	0	1	0	1	0	0
x	000		1	1	0	0	1	1	0	0	x	0	0
x	001	0100011	1	1	0	0	1	0	1	0	x	0	0
x	010		1	1	0	0	1	0	0	1	x	0	0
x	x	0010011	1	0	0	0	0	x	x	x	x	0	0
0	x	0110011	1	0	0	0	0	x	x	x	x	0	0
1	x	0110011	1	0	0	0	0	x	x	x	x	1	1

Table 4.6.3 – Function table of main control

4.6.4 Schematic Design

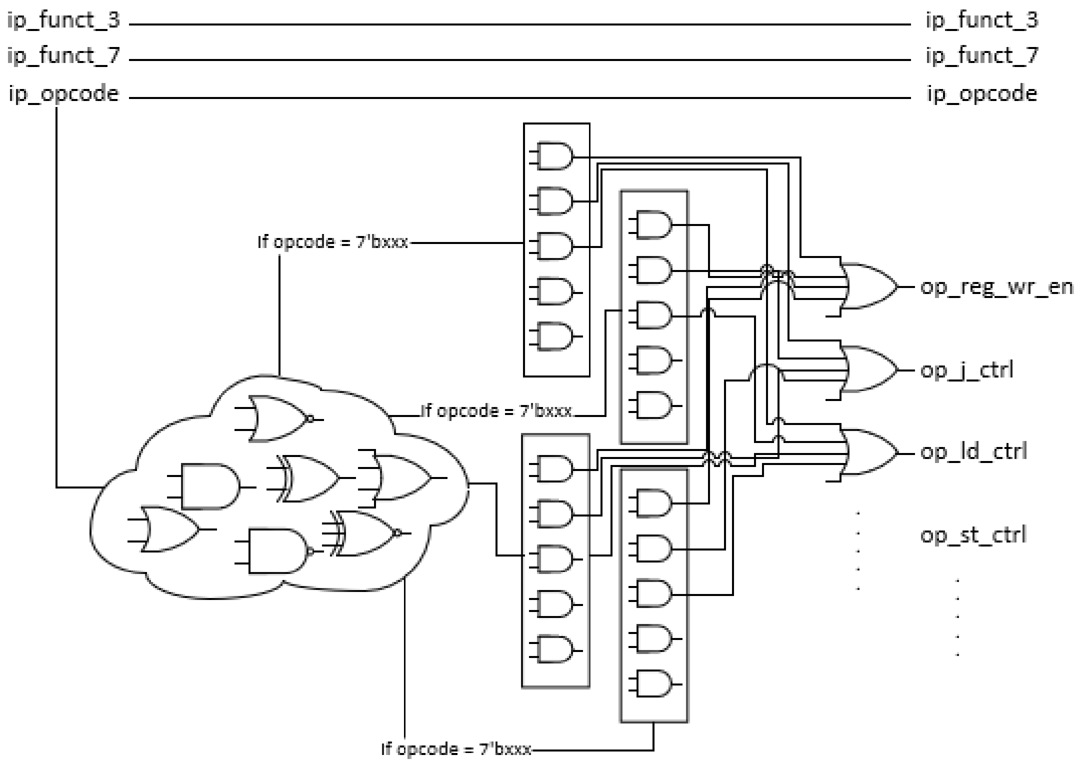


Figure 4.6.4 – Schematic design of main control

4.6.5 Verilog Model

```
/*********************************************************************
Project: Project: Develop Extended ISA on RISC-V Based Processor
Module: b_main_ctrl.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: Main Control Block
*********************************************************************/

`default_nettype none // to catch typing errors due to typo of signal names

module b_main_ctrl
#( // declare all the parameter needed
parameter initial_addr = 32'h00008000, // initial pc address
parameter last_addr = 32'h01FFFFFF // last pc address
)
( // declare all the input and output pin needed
	input wire [6:0] ip_opcode, ip_funct_7,
	input wire [2:0] ip_funct_3,
	input wire ip_clk,
	output reg [6:0] op_opcode, op_funct_7,
	output reg [2:0] op_funct_3,
	output reg op_reg_wr_en, op_wb_ctrl, op_j_ctrl, op_ld_en, op_st_en, op_byte_ctrl,
op_half_ctrl, op_word_ctrl, op_uns_ctrl, op_mul_div_en, op_m_ext_wb_ctrl
);

// main control block
always @(posedge ip_clk) begin
	op_opcode[6:0] = ip_opcode[6:0];
	op_funct_7[6:0] = ip_funct_7[6:0];
	op_funct_3[2:0] = ip_funct_3[2:0];

	// upper instruction, LUI / AUIPC or branch instuction or ALU operation instruction
	if((ip_opcode[6:0] == 7'b0110111) || (ip_opcode[6:0] == 7'b0010111) || (ip_opcode[6:0] ==
7'b1100011) || (ip_opcode[6:0] == 7'b0010011) || ((ip_opcode[6:0] == 7'b0110011) && (ip_funct_7[0]
== 1'b0))) begin
		op_reg_wr_en = 1'b1;
		op_wb_ctrl = 1'b0;
		op_j_ctrl = 1'b0;
		op_ld_en = 1'b0;
		op_st_en = 1'b0;
		op_uns_ctrl = 1'b0;
		op_mul_div_en = 1'b0;
		op_m_ext_wb_ctrl = 1'b0;
	end

	// jump instruction, JAL / JALR
	else if((ip_opcode[6:0] == 7'b1101111) || (ip_opcode[6:0] == 7'b1100111)) begin
		op_reg_wr_en = 1'b1;
		op_wb_ctrl = 1'b0;
		op_j_ctrl = 1'b1;
		op_ld_en = 1'b0;
		op_st_en = 1'b0;
		op_uns_ctrl = 1'b0;
		op_mul_div_en = 1'b0;
		op_m_ext_wb_ctrl = 1'b0;
	end
```

```
// load instruction
else if(ip_opcode[6:0] == 7'b0000011) begin
        op_reg_wr_en = 1'b1;
        op_wb_ctrl = 1'b1;
        op_j_ctrl = 1'b0;
        op_ld_en = 1'b1;
        op_st_en = 1'b0;
        op_uns_ctrl = ip_funct_3[2];
        op_mul_div_en = 1'b0;
        op_m_ext_wb_ctrl = 1'b0;

        if(ip_funct_3[1:0] == 2'b00) begin
                op_byte_ctrl = 1'b1;
                op_half_ctrl = 1'b0;
                op_word_ctrl = 1'b0;
        end

        else if(ip_funct_3[1:0] == 2'b01) begin
                op_byte_ctrl = 1'b0;
                op_half_ctrl = 1'b1;
                op_word_ctrl = 1'b0;
        end

        else if(ip_funct_3[1:0] == 2'b10) begin
                op_byte_ctrl = 1'b0;
                op_half_ctrl = 1'b0;
                op_word_ctrl = 1'b1;
        end
end

// store instruction
else if(ip_opcode[6:0] == 7'b0100011) begin
        op_reg_wr_en = 1'b0;
        op_wb_ctrl = 1'b0;
        op_j_ctrl = 1'b0;
        op_ld_en = 1'b0;
        op_st_en = 1'b1;
        op_uns_ctrl = 1'b0;
        op_mul_div_en = 1'b0;
        op_m_ext_wb_ctrl = 1'b0;

        if(ip_funct_3[1:0] == 2'b00) begin
                op_byte_ctrl = 1'b1;
                op_half_ctrl = 1'b0;
                op_word_ctrl = 1'b0;
        end

        else if(ip_funct_3[1:0] == 2'b01) begin
                op_byte_ctrl = 1'b0;
                op_half_ctrl = 1'b1;
                op_word_ctrl = 1'b0;
        end

        else if(ip_funct_3[1:0] == 2'b10) begin
                op_byte_ctrl = 1'b0;
                op_half_ctrl = 1'b0;
                op_word_ctrl = 1'b1;
        end
end
```

```
        // M Extension
        else if((ip_opcode[6:0] == 7'b0110011) && (ip_funct_7[0] == 1'b1)) begin
                op_reg_wr_en = 1'b1;
                op_wb_ctrl = 1'b0;
                op_j_ctrl = 1'b0;
                op_ld_en = 1'b0;
                op_st_en = 1'b0;
                op_uns_ctrl = 1'b0;
                op_mul_div_en = 1'b1;
                op_m_ext_wb_ctrl = 1'b1;
        end
end
endmodule
```

4.7 ALU Control

4.7.1 Functionality and Feature

ALU control block which decide which operations are being process in ALU and generate control signal which feed into the ALU for computation process and executions of the R-type instructions such as arithmetic and logical operations included addition for load and store or subtraction for branches. ALU control unit received decoded 7-bit opcode and 7-bit and 3-bit funct from main control block. After decoded the control signals are being fetch from ALU control block and ask ALU to perform certain operation and executions based on the instructions.

4.7.2 Interface and I/O Pin Description

- **Interface**

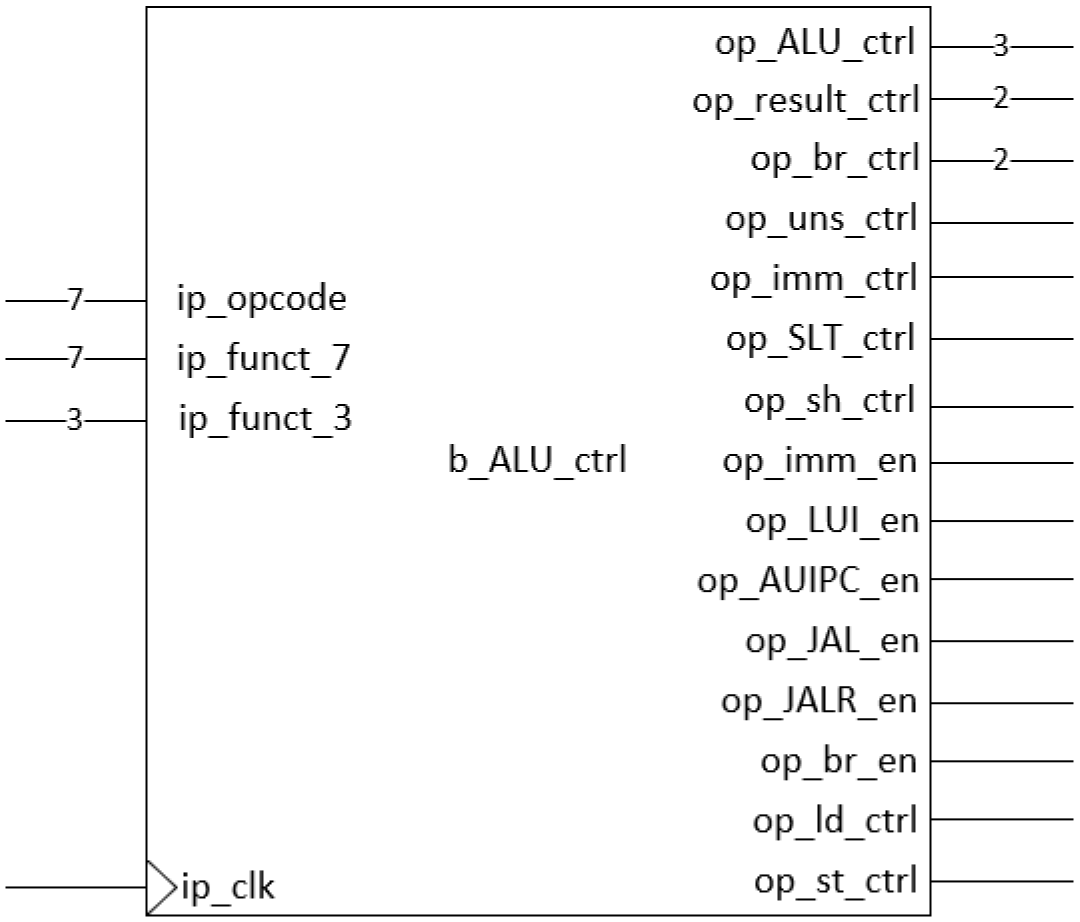


Figure 4.7.2 – ALU Control interface

- **I/O Pin Description**

Pin name: ip_opcode[6:0] **Pin class:** control **Pin function:** A 7 bits of instruction ID.	**Source:** Main Control **Destination:** ALU Control
Pin name: ip_funct_7[6:0] **Pin class:** control **Pin function:** A 7 bits function signal for define other types of control.	**Source:** Main Control **Destination:** ALU Control
Pin name: ip_funct_3[2:0] **Pin class:** control **Pin function:** A 3 bits function signal for defined the arithmetic and logical used by instruction.	**Source:** Main Control **Destination:** ALU Control
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** ALU Control
Pin name: op_ALU_ctrl[2:0] **Pin class:** control **Pin function:** A 3-bit of control pin to determine which arithmetic or logical are required to use.	**Source:** ALU Control **Destination:** ALU
Pin name: op_result_ctrl[1:0] **Pin class:** control **Pin function:** A 2-bit of control pin to determine computing result from ALU or barrel shifter.	**Source:** ALU Control **Destination:** ALU
Pin name: op_br_ctrl[1:0] **Pin class:** control **Pin function:** A 2-bit of control pin to determine which branch instruction is used.	**Source:** ALU Control **Destination:** ALU
Pin name: op_uns_ctrl **Pin class:** control **Pin function:** A control pin to determine computing is involved unsigned or signed data.	**Source:** ALU Control **Destination:** ALU
Pin name: op_imm_ctrl **Pin class:** control **Pin function:** A control pin to determine is immediate data involved.	**Source:** ALU Control **Destination:** ALU
Pin name: op_SLT_ctrl **Pin class:** control **Pin function:** A control pin to determine is set instruction involved.	**Source:** ALU Control **Destination:** ALU
Pin name: op_sh_ctrl **Pin class:** control	**Source:** ALU Control **Destination:** ALU

Pin function: A control pin to determine the shifting direction	
Pin name: op_imm_en **Pin class:** control **Pin function:** A control pin to determine is I-type instruction involved	**Source:** ALU Control **Destination:** ALU
Pin name: op_LUI_en **Pin class:** control **Pin function:** A control pin to determine is LUI instruction	**Source:** ALU Control **Destination:** ALU
Pin name: op_AUIPC_en **Pin class:** control **Pin function:** A control pin to determine is AUIPC instruction	**Source:** ALU Control **Destination:** ALU
Pin name: op_JAL_en **Pin class:** control **Pin function:** A control pin to determine is JAL instruction	**Source:** ALU Control **Destination:** ALU
Pin name: op_JALR_en **Pin class:** control **Pin function:** A control pin to determine is JALR instruction	**Source:** ALU Control **Destination:** ALU
Pin name: op_br_en **Pin class:** control **Pin function:** A control pin to determine is branch instruction involved	**Source:** ALU Control **Destination:** ALU
Pin name: op_ld_ctrl **Pin class:** control **Pin function:** A control pin to determine load instruction is occurred	**Source:** ALU Control **Destination:** ALU
Pin name: op_st_ctrl **Pin class:** control **Pin function:** A control pin to determine store instruction is occurred	**Source:** ALU Control **Destination:** ALU

Table 4.7.2 – ALU control pin description

4.6.3 Internal Operation

Input			Output													
ip_funct_7[5]	ip_funct_3	ip_opcode	op_ALU_ctrl	op_result_ctrl	op_jmm_ctrl	op_uns_ctrl	op_imm_en	op_LUI_en	op_AUIPC_en	op_JAL_en	op_JALR_en	op_br_en	Op_br_ctrl	op_ld_ctrl	op_st_ctrl	op_SLT_ctrl
x	x	0110111	OR	00	1	0	0	1	0	0	0	0	x	0	0	0
x	x	0010111	ADD	00	1	0	0	0	1	0	0	0	x	0	0	0
x	x	1101111	ADD	00	1	0	0	0	0	1	0	0	x	0	0	0
x	x	1100111	ADD	00	1	0	0	0	0	0	1	0	x	0	0	0
x	000	1100011	SUB	00	1	0	0	0	0	0	0	1	00	0	0	0
x	001		SUB	00	1	0	0	0	0	0	0	1	01	0	0	0
x	100		SUB	00	1	0	0	0	0	0	0	1	10	0	0	0
x	101		SUB	00	1	0	0	0	0	0	0	1	11	0	0	0
x	110		SUB	00	1	1	0	0	0	0	0	1	10	0	0	0
x	111		SUB	00	1	1	0	0	0	0	0	1	11	0	0	0
x	000	0000011	ADD	00	1	0	0	0	0	0	0	0	x	1	0	0
	001															
	010															
x	101		ADD	00	1	1	0	0	0	0	0	0	x	1	0	0
	110															
x	x	0100011	ADD	00	1	x	0	0	0	0	0	0	x	0	1	0
x	000	0010011	ADD	00	1	0	1	x	x	x	x	0	x	0	0	0
x	001		x	10	1	0	1	x	x	x	x	0	x	0	0	0
x	010		SUB	01	1	0	1	x	x	x	x	0	x	0	0	1
x	011		SUB	01	1	1	1	x	x	x	x	0	x	0	0	1
x	100		XOR	00	1	0	1	x	x	x	x	0	x	0	0	0
0	101		x	10	1	0	1	x	x	x	x	0	x	0	0	0
1			x	10	1	1	1	x	x	x	x	0	x	0	0	0
x	110		OR	00	1	0	1	x	x	x	x	0	x	0	0	0
x	111		AND	00	1	0	1	x	x	x	x	0	x	0	0	0
0	000	0110011	ADD	00	0	0	0	x	x	x	x	0	x	0	0	0
1			SUB	00	0	0	0	x	x	x	x	0	x	0	0	0
x	001		x	10	0	0	0	x	x	x	x	0	x	0	0	0
x	010		SUB	01	0	0	0	x	x	x	x	0	x	0	0	1
x	011		SUB	01	0	1	0	x	x	x	x	0	x	0	0	1
x	100		XOR	00	0	0	0	x	x	x	x	0	x	0	0	0
0	101		x	10	0	1	0	x	x	x	x	0	x	0	0	0
1	101		x	10	0	0	0	x	x	x	x	0	x	0	0	0
x	110		OR	00	0	0	0	x	x	x	x	0	x	0	0	0
x	111		AND	00	0	0	0	x	x	x	x	0	x	0	0	0

Table 4.7.3 – Function table of ALU control

4.6.4 Schematic Design

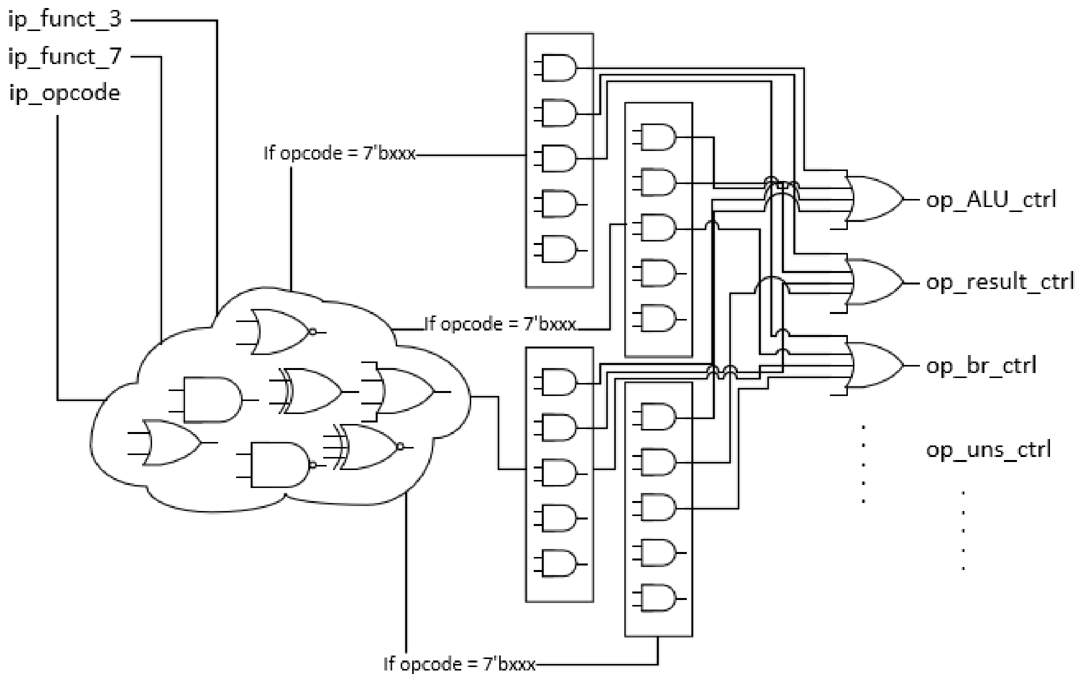


Figure 4.7.4 – Schematic design of ALU control

4.7.5 Verilog Model

```
/**********************************************************************
Project: Project: Develop Extended ISA on RISC-V Based Processor
Module: b_ALU_ctrl.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
Code Type: Verilog
Description: ALU Control Block
**********************************************************************/

`default_nettype none // to catch typing errors due to typo of signal names

module b_main_ctrl
#( // declare all the parameter needed
parameter AND = 3'b000,
parameter OR = 3'b001,
parameter XOR = 3'b010,
parameter ADD = 3'b011,
parameter SUB = 3'b111
)
( // declare all the input and output pin needed
        input wire [6:0] ip_opcode, ip_funct_7,
        input wire [2:0] ip_funct_3,
        input wire ip_clk,
        output reg [2:0] op_ALU_ctrl,
        output reg [1:0] op_result_ctrl, op_br_ctrl,
        output reg op_uns_ctrl, op_imm_ctrl, op_SLT_ctrl, op_sh_ctrl, op_imm_en, op_LUI_en,
op_AUIPC_en, op_JAL_en, op_JALR_en, op_br_en, op_ld_ctrl, op_st_ctrl
);

// ALU control block
always @(posedge ip_clk) begin
        // load upper immediate, LUI
        if(ip_opcode[6:0] == 7'b0110111) begin
                op_ALU_ctrl[2:0] = OR;
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_imm_en = 1'b0; // immediate instruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_LUI_en = 1'b1; // LUI involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        // add upper immediate with pc, AUIPC
        if(ip_opcode[6:0] == 7'b0010111) begin
                op_ALU_ctrl[2:0] = ADD; // addition involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_imm_en = 1'b0; // immediate instruction not involved
                op_AUIPC_en = 1'b1; // AUIPC involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        // jump address, JAL
        if(ip_opcode[6:0] == 7'b1101111) begin
                op_ALU_ctrl[2:0] = ADD; // addition involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_imm_en = 1'b0; // immediate instruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b1; // JAL involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution not involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        // jump address, JALR
        else if(ip_opcode[6:0] == 7'b1100111) begin
                op_ALU_ctrl[2:0] = ADD; // addition involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_imm_en = 1'b0; // immediate idtruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b1; // JALR involved
                op_br_en = 1'b0; // branch instrcution not involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        // branch address
        else if(ip_opcode[6:0] == 7'b0110011) begin
                op_ALU_ctrl[2:0] = SUB; // subtraction involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = ip_funct_3[1]; // unsigned data involved/not
                op_imm_en = 1'b0; // immediate idtruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b1; // branch instrcution involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved

                if(ip_funct_3[2:0] == 3'b000) begin // BEQ is selected
                        op_br_ctrl[1:0] = 2'b00; // output BEQ control signal
                end

                else if(ip_funct_3[2:0] == 3'b001) begin // BNE is selected
                        op_br_ctrl[1:0] = 2'b01; // output BNE control signal
                end

                else if(ip_funct_3[2:0] == 3'b100) begin // BLT is selected
                        op_br_ctrl[1:0] = 2'b10; // output BLT control signal
                end

                else if(ip_funct_3[2:0] == 3'b101) begin // BGE is selected
                        op_br_ctrl[1:0] = 2'b11; // output BGE control signal
                end

                else if(ip_funct_3[2:0] == 3'b110) begin // BLTU is selected
                        op_br_ctrl[1:0] = 2'b10; // output BLT control signal
                end

                else if(ip_funct_3[2:0] == 3'b111) begin // BGEU is selected
                        op_br_ctrl[1:0] = 2'b11; // output BGE control signal
                end
        end

        // load instruction
        else if(ip_opcode[6:0] == 7'b0000011) begin
                op_ALU_ctrl[2:0] = ADD; // addition involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = ip_funct_3[2]; // unsigned data involved/not
                op_imm_en = 1'b0; // immediate idtruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
                op_ld_ctrl = 1'b1; // load data involved
                op_st_ctrl = 1'b0; // store data not involved
        end

        // store instruction
        else if(ip_opcode[6:0] == 7'b0100011) begin
                op_ALU_ctrl[2:0] = ADD; // addition involved
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_imm_ctrl = 1'b1; // immediate data involved
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_imm_en = 1'b0; // immediate idtruction not involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b1; // store data involved
        end

        // ALU operation with immediate
        else if(ip_opcode[6:0] == 7'b0010011) begin
                op_imm_ctrl = 1'b1; // immediate data involved
                op_imm_en = 1'b1; // immediate idtruction involved
                op_AUIPC_en = 1'b0; // AUIPC not involved
                op_JAL_en = 1'b0; // JAL not involved
                op_JALR_en = 1'b0; // JALR not involved
                op_br_en = 1'b0; // branch instrcution not involved
                op_ld_ctrl = 1'b0; // load data not involved
                op_st_ctrl = 1'b0; // store data not involved

                if(ip_funct_3[2:0] == 3'b000) begin // ADDI is selected
                        op_ALU_ctrl[2:0] = ADD; // output ADD control signal
                        op_result_ctrl = 2'b00; // output result from ALU control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                end

                else if(ip_funct_3[2:0] == 3'b001) begin // SLLI is selected
                        op_sh_ctrl = 1'b1; // output shift left control signal
                        op_result_ctrl = 2'b10; // output result from barrel shifter control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                end

                else if(ip_funct_3[2:0] == 3'b010) begin // SLTI is selected
                        op_ALU_ctrl[2:0] = SUB; // output SUB control signal
                        op_result_ctrl = 2'b01; // output result from ALU control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b1; // SLT involved
                end

                else if(ip_funct_3[2:0] == 3'b011) begin // SLTIU is selected
                        op_ALU_ctrl[2:0] = SUB; // output SUB control signal
                        op_result_ctrl = 2'b01; // output result from SLT control signal
                        op_uns_ctrl = 1'b1; // usigned data involved
                        op_SLT_ctrl = 1'b1; // SLT involved
                end

                else if(ip_funct_3[2:0] == 3'b100) begin // XORI is selected
                        op_ALU_ctrl[2:0] = XOR; // output XOR control signal
                        op_result_ctrl = 2'b00; // output result from ALU control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                end

                else if(ip_funct_3[2:0] == 3'b101) begin // SRLI/SRAI is selected
                        op_sh_ctrl = 1'b1; // output shift right control signal
                        op_result_ctrl = 2'b10; // output result from barrel shifter control signal
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                        op_uns_ctrl = ~ip_funct_7[5];  // SRLI is selected {1}, SRAI is selected {0}
                end

                else if(ip_funct_3[2:0] == 3'b110) begin // ORI is selected
                        op_ALU_ctrl[2:0] = OR; // output OR control signal
                        op_result_ctrl = 2'b00; // output result from ALU control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                end

                else if(ip_funct_3[2:0] == 3'b111) begin // ANDI is selected
                        op_ALU_ctrl[2:0] = AND; // output AND control signal
                        op_result_ctrl = 2'b00; // output result from ALU control signal
                        op_uns_ctrl = 1'b0; // unsigned data not involved
                        op_SLT_ctrl = 1'b0; // SLT not invloved
                end
        end

        // ALU operation with register data
```

```
else if(ip_opcode[6:0] == 7'b0110011) begin
        op_imm_ctrl = 1'b0; // immediate data not involved
        op_imm_en = 1'b0; // immediate idtruction not involved
        op_AUIPC_en = 1'b0; // AUIPC not involved
        op_JAL_en = 1'b0; // JAL not involved
        op_JALR_en = 1'b0; // JALR not involved
        op_br_en = 1'b0; // branch instrcution not involved
        op_ld_ctrl = 1'b0; // load data not involved
        op_st_ctrl = 1'b0; // store data not involved

        if(ip_funct_3[2:0] == 3'b000) begin // ADD/SUB is selected
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved

                if(ip_funct_7[5] == 1'b0) begin // ADD is selected
                        op_ALU_ctrl[2:0] = ADD; // output ADD control signal
                end

                else if(ip_funct_7[5] == 1'b1) begin // SUB is selected
                        op_ALU_ctrl[2:0] = SUB; // output SUB control signal
                end
        end

        else if(ip_funct_3[2:0] == 3'b001) begin // SLL is selected
                op_sh_ctrl = 1'b0; // output shift left control signal
                op_result_ctrl = 2'b10; // output result from barrel shifter control signal
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        else if(ip_funct_3[2:0] == 3'b010) begin // SLT is selected
                op_ALU_ctrl[2:0] = SUB; // output SUB control signal
                op_result_ctrl = 2'b01; // output result from SLT control signall
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_SLT_ctrl = 1'b1; // output SLT control signal
        end

        else if(ip_funct_3[2:0] == 3'b011) begin // SLTU is selected
                op_ALU_ctrl[2:0] = SUB; // output SUB control signal
                op_result_ctrl = 2'b01; // output result from SLT control signal
                op_SLT_ctrl = 1'b1; // output SLT control signal
                op_uns_ctrl = 1'b1; // usigned data involved
        end

        else if(ip_funct_3[2:0] == 3'b100) begin // XOR is selected
                op_ALU_ctrl[2:0] = XOR; // output XOR control signal
                op_result_ctrl = 2'b00; // output result from ALU control signal
                op_uns_ctrl = 1'b0; // unsigned data not involved
                op_SLT_ctrl = 1'b0; // SLT not invloved
        end

        else if(ip_funct_3[2:0] == 3'b101) begin // SRL/SRA is selected
                op_sh_ctrl = 1'b1; // output shift right control signal
                op_result_ctrl = 2'b10; // output result from barrel shifter control signal
                op_SLT_ctrl = 1'b0; // SLT not invloved
                op_uns_ctrl = ~ip_funct_7[5];  // SRL is selected {1}, SRA is selected {0}
        end

        else if(ip_funct_3[2:0] == 3'b110) begin // OR is selected
```

```
                    op_ALU_ctrl[2:0] = OR; // output OR control signal
                    op_result_ctrl = 2'b00; // output result from ALU control signal
                    op_uns_ctrl = 1'b0; // unsigned data not involved
                    op_SLT_ctrl = 1'b0; // SLT not invloved
          end

          else if(ip_funct_3[2:0] == 3'b111) begin // AND is selected
                    op_ALU_ctrl[2:0] = AND; // output AND control signal
                    op_result_ctrl = 2'b00; // output result from ALU control signal
                    op_uns_ctrl = 1'b0; // unsigned data not involved
                    op_SLT_ctrl = 1'b0; // SLT not invloved
          end
     end
end
endmodule
```

4.8 M Extension

4.8.1 Functionality and Feature

The M extension performs multiplication and division operation and stores the result back to the register file. The common multiplication is to multiply a set range of digit where from unsigned bit to signed bit multiplication, same as well as division. All the data for computing all will be transfer to positive if it is signed data. Multiplication using methods of "add and shift" algorithm to do multiplication. There will be separate into several parts to assist on doing including addition, shift register and a counter for computing result of two operands. It needs 32 clock cycles to compute the result. The result will write back to target registers. The upper 32-bits result will be output when called by MULH, MULHSU or MULHU and the lower 32-bits result will be output when called by MUL. Besides that, division is using method of "Subtract and compare" algorithm to perform a division. There will be separate into several parts to assist on doing including subtraction, comparing and a switch for stalling the data path for computing result of two operands. The division has ability to come quotient, and remainder after complete computing. Since, the M Extension is required more than 1 cycle to compute, the extension will stop the data path to prevent data crashing

4.8.2 Interface and I/O Pin Description

- **Interface**

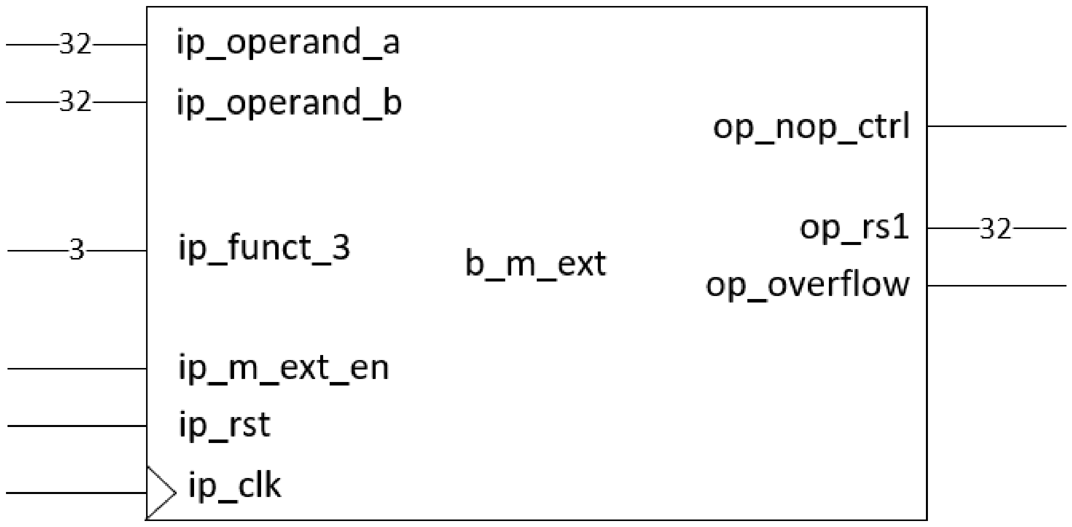


Figure 4.8.2 – M extension interface

- **I/O Pin Description**

Pin name: ip_operand_a[31:0] **Pin class:** data **Pin function:** A 32 bits data for computing, it will be multiplicand if multiplication, and be dividend if division.	**Source:** Register File **Destination:** M Extension
Pin name: ip_operand_b[31:0] **Pin class:** data **Pin function:** A 32 bits data for computing, it will be multiplier if multiplication, and be dividend if divisor	**Source:** Register File **Destination:** M Extension
Pin name: ip_funct_3[2:0] **Pin class:** control **Pin function:** A 3 bits function signal for defined the multiplication or division used by instruction and determine output result from which register.	**Source:** Control Unit **Destination:** M Extension
Pin name: ip_rst **Pin class:** global **Pin function:** A pin to reset the register of M extension.	**Source:** Datapath **Destination:** M Extension
Pin name: ip_clk **Pin class:** global **Pin function:** A clock signal for the system running	**Source:** Datapath **Destination:** M Extension
Pin name: op_result[31:0] **Pin class:** data **Pin function:** A 32 bits data of result after computing in extension	**Source:** M Extension **Destination:** Register File
Pin name: op_overflow **Pin class:** control **Pin function:** A pin to determine the calculation is overflow or error. The division is possible happen overflow when largest negative number divided by -1, or occurred error when divide zero.	**Source:** M Extension **Destination:** Datapath

Table 4.8.2 – M extension pin description

4.3.3 Internal Operation

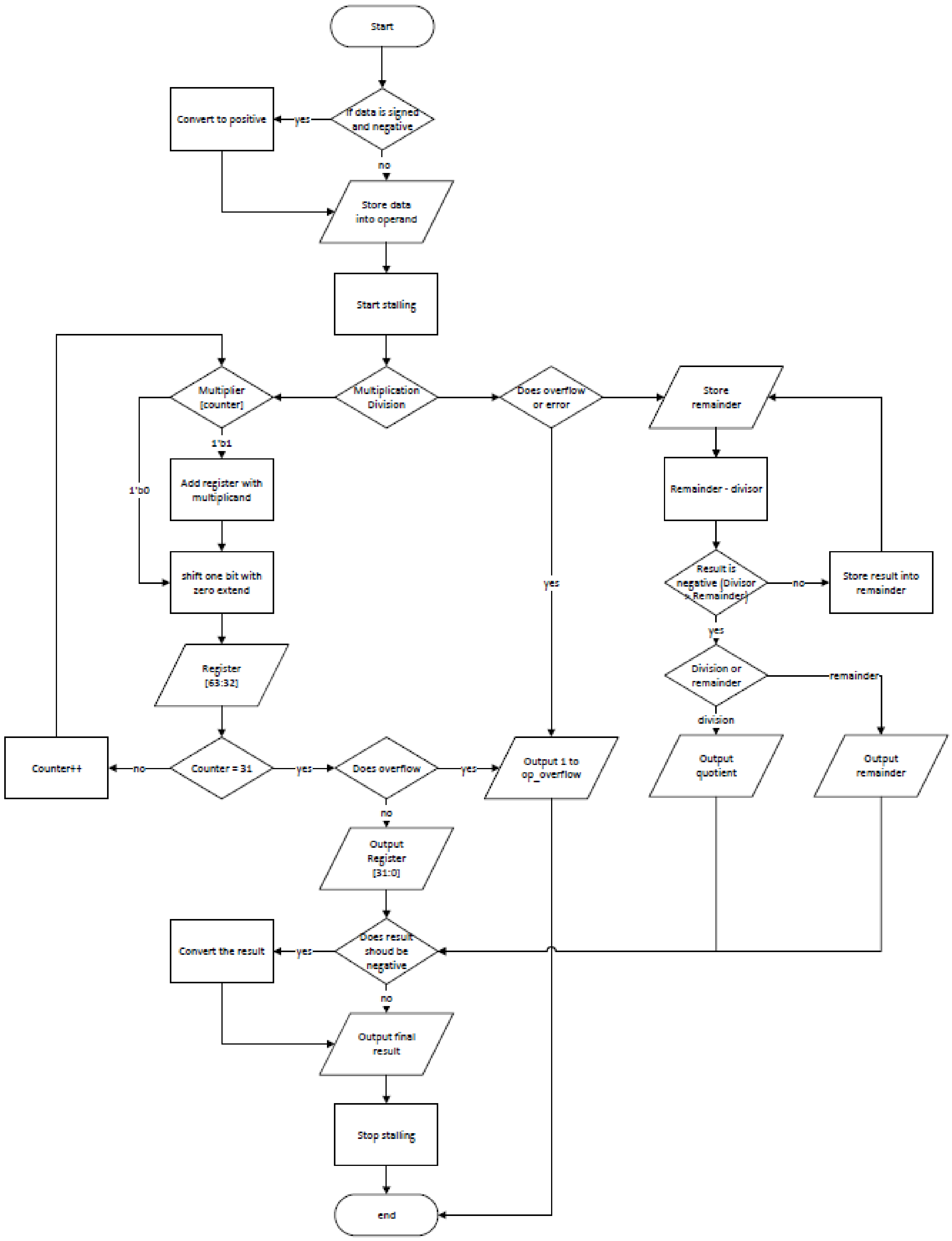


Figure 4.8.3 – Flowchart of M extension operation

4.8.4 Schematic Design

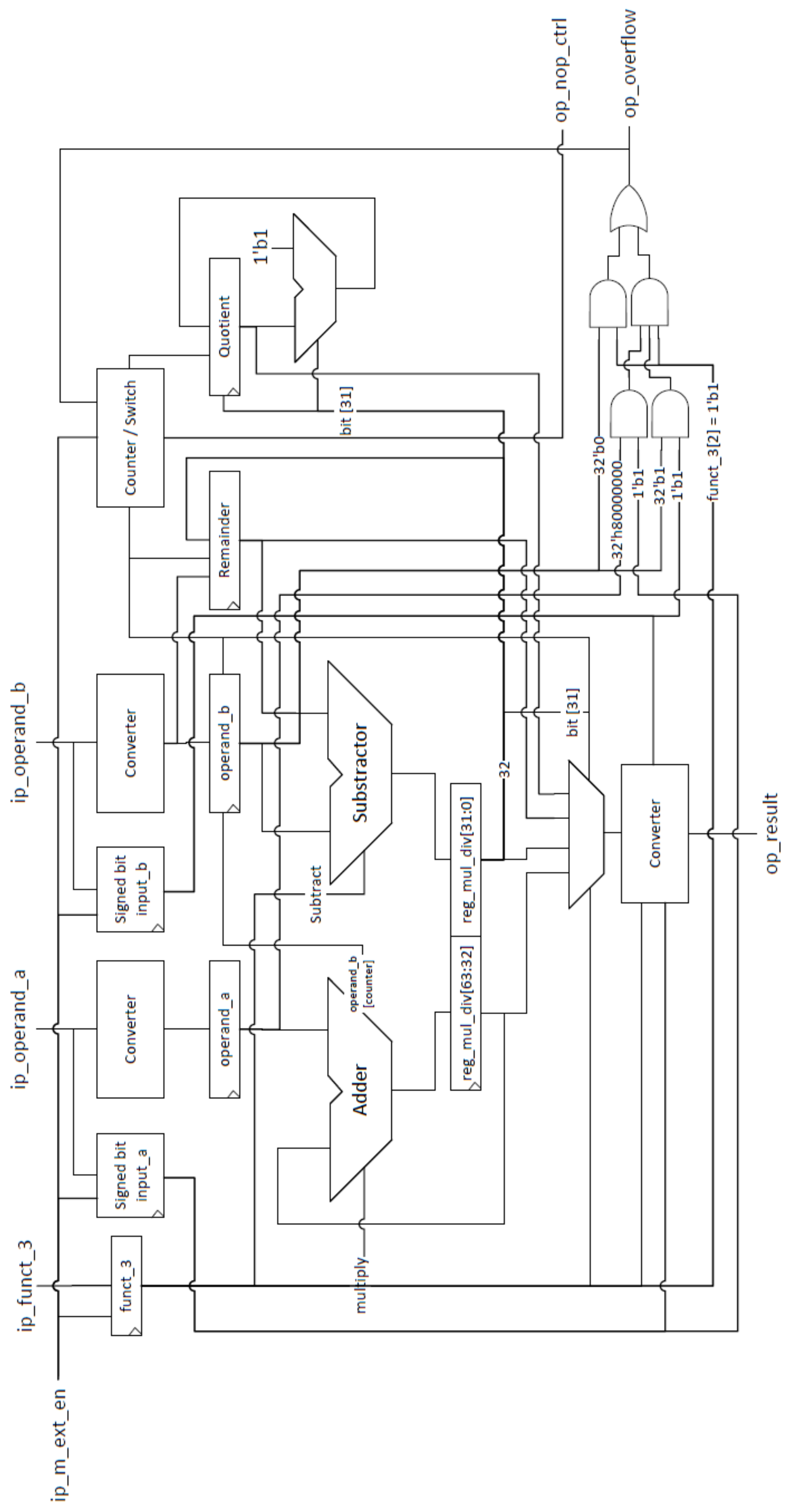


Figure 4.8.4 – Schematic design of M extension

4.2.5 Verilog Model

```
/********************************************************************
Project: Develop Extended ISA on RISC-V Based Processor
Module: b_m_ext.v
Version: 1
Date Created: 4/8/2022
Created By: Lee Ang
`Code Type: Verilog
Description: M Extension
********************************************************************/

`default_nettype none // to catch typing errors due to typo of signal names

module b_m_ext
#( // declare all the parameter needed
parameter ONE32 = 32'b11111111111111111111111111111111, // 32 of one bits
ONE64 = 64'b1111111111111111111111111111111111111111111111111111111111111111 // 64 of
one bits
)
( // declare all the input and output pin needed
        input wire [31:0] ip_operand_a, ip_operand_b,
        input wire [2:0] ip_funct_3,
        input wire ip_rst, ip_clk, ip_m_ext_en,
        output reg [31:0] op_result,
        output reg op_nop_ctrl, op_overflow
);

reg [64:0] carry_bit_reg_mul_div; // carry bit of reg_mul_div converter
reg [63:0] reg_mul_div; // 64-bits register
reg [32:0] carry_bit_operand_a, carry_bit_operand_b; // carry bit of operand converter
reg [32:0] carry_bit_quotient, carry_bit_remainder; // carry bit of quotient and remainder converter
reg [31:0] operand_a, operand_b; // operand a and b
reg [31:0] quotient, remainder; // quotient and remainder of division
reg [2:0] funct_3; // selection of function
reg sign_operand_a, sign_operand_b; // signed bit of operand
reg [5:0] carry_bit_counter; // carry bit of counter
reg [4:0] counter; // 5-bits counter
reg switch; // a switch to control operation
reg ready_div_op; // control to output divsion

integer i; // for loop

// operand a, operand b, quotient, remainder, sign_operand_a, sign_operand_a and funct_3 registers
always @(posedge ip_clk) begin
        if (ip_rst == 1'b1) begin // if reset
                // reset to default value
                operand_a[31:0] = 32'b0;
                operand_b[31:0] = 32'b0;
                carry_bit_operand_a[32:0] = 33'b0;
                carry_bit_operand_b[32:0] = 33'b0;
                quotient[31:0] = 32'b0;
                remainder[31:0] = 32'b0;
                sign_operand_a = 1'b0;
                sign_operand_b = 1'b0;
                carry_bit_quotient[32:0] = 33'b0;
                carry_bit_remainder[32:0] = 33'b0;
                funct_3[2:0] = 3'b0;
        end
```

```
    else if (ip_rst == 1'b0) begin // if no reset
        if ((ip_m_ext_en == 1'b1) && (switch == 1'b0)) begin // if m extension is selected,
then store all the nessary values
            // store original signed bit
            sign_operand_a <= ip_operand_a[31];
            sign_operand_b <= ip_operand_b[31];

            // store input a
            // if instruction of rs1 is a negetive sign value
            if ((((ip_funct_3[2] == 1'b0) && (ip_funct_3[1:0] != 2'b11)) || ((ip_funct_3[2]
== 1'b1) && (ip_funct_3[0] == 1'b0))) && (ip_operand_a[31] == 1'b1)) begin
                // 32-bit negative to positive converter
                operand_a[31:0] = ip_operand_a[31:0] ^ ONE32; // invert the value
                // add 1 by using adder
                carry_bit_operand_a[0] = 1'b1;
                for (i = 0; i < 32; i = i + 1) begin
                    carry_bit_operand_a[i   +   1'b1]   =   operand_a[i]   &
carry_bit_operand_a[i]; // calculation of carry bit
                    operand_a[i] <= operand_a[i] ^ carry_bit_operand_a[i]; //
store result affter convert

                    // store the remainder for division from operand a
                    if ((ip_funct_3[2] == 1'b1) && (switch == 1'b0)) begin // if
division
                        remainder[i]        <=        operand_a[i]        ^
carry_bit_operand_a[i];
                    end
                end
            end

            else begin // if instruction of rs1 is unsigned value
                operand_a[31:0] <= ip_operand_a[31:0]; // store positive input a

                // store the remainder for division from operand a
                if ((ip_funct_3[2] == 1'b1) && (switch == 1'b0)) begin // if division
                    remainder[31:0] <= ip_operand_a[31:0];
                end
            end

            // store input b
            if (((ip_funct_3[2:1] == 2'b00) || ((ip_funct_3[2] == 1'b1) && (ip_funct_3[0]
== 1'b0))) && (ip_operand_b[31] == 1'b1)) begin // if instruction of rs2 is a negetive sign value
                // 32-bit negative to positive converter
                operand_b[31:0] = ip_operand_b[31:0] ^ ONE32; // invert the value
                // add 1 by using adder
                carry_bit_operand_b[0] = 1'b1;
                for (i = 0; i < 32; i = i + 1) begin
                    carry_bit_operand_b[i   +   1'b1]   =   operand_b[i]   &
carry_bit_operand_b[i]; // calculation of carry bit
                    operand_b[i] <= operand_b[i] ^ carry_bit_operand_b[i]; //
store result affter convert
                end
            end

            else begin
                operand_b[31:0] <= ip_operand_b[31:0]; // store positive input b
            end

            quotient[31:0] <= 32'b0; // empty the quotient for calculation
            funct_3[2:0] <= ip_funct_3[2:0]; // store the selected function
```

```
                    end
          end
end

// multiplication adder and shifter, division adder
always @(posedge ip_clk) begin
          if (switch == 1'b1) begin // if m extension is used
                    if (funct_3[2] == 1'b0) begin // if multiplication is selected
                              if (operand_b[counter] == 1'b1) begin // if number of operand_b bit = 1
                                        // 32-bit Adder
                                        carry_bit_reg_mul_div[32] = 0;
                                        for (i = 0; i < 32; i = i + 1) begin
                                                  // calculation of carry bit
                                                  carry_bit_reg_mul_div[i + 33] = (reg_mul_div[i + 32] &
operand_a[i]) | (reg_mul_div[i + 32] & carry_bit_reg_mul_div[i + 32]) | (operand_a[i] &
carry_bit_reg_mul_div[i + 32]);
                                                  // calculation of add result
                                                  reg_mul_div[i + 32] = reg_mul_div[i + 32] ^ operand_a[i]
^ carry_bit_reg_mul_div[i + 32];
                                        end

                                        reg_mul_div[63:0]            =            {carry_bit_reg_mul_div[64],
reg_mul_div[63:1]}; // shift 1 bit with last carry bit extend after addition
                              end

                              else if (operand_b[counter] == 1'b0) begin // if number of operand_b bit = 0
                                        reg_mul_div[63:0] = {1'b0, reg_mul_div[63:1]}; // shift 1 bit with 0
extend after addition
                              end

                    end

                    else if (funct_3[2] == 1'b1) begin // if division is selected
                              if (ready_div_op == 1'b0) begin
                                        // if divisor = 0 or the largest negative number divide by -1 (special
case)
                                        if (operand_b[31:0] == 32'b0 || ((operand_a[31] == 1'b1) &&
(operand_a[30:0] == 31'b0) && (sign_operand_a == 1'b1) && (operand_b[31:0] == 32'b1) &&
(sign_operand_b == 1'b1))) begin
                                                  ready_div_op <= 1'b1;
                                                  remainder[31:0] <= 32'b0;
                                        end

                                        else begin
                                                  // compare remainder with divisor
                                                  reg_mul_div[31:0] = operand_b[31:0] ^ ONE32; // invert
the value
                                                  // add 1 by using adder
                                                  carry_bit_reg_mul_div[0] = 1'b1;
                                                  for (i = 0; i < 32; i = i + 1) begin
                                                            // calculation of carry bit
                                                            carry_bit_reg_mul_div[i + 1'b1] = (remainder[i]
& reg_mul_div[i]) | (remainder[i] & carry_bit_reg_mul_div[i]) | (reg_mul_div[i] &
carry_bit_reg_mul_div[i]);
                                                            // calculation of add result
                                                            reg_mul_div[i] = remainder[i] ^ reg_mul_div[i] ^
carry_bit_reg_mul_div[i];
                                                  end
                    if (reg_mul_div[31] == 1'b1) begin // if remainder less than
divisor
                        ready_div_op <= 1'b1;
                    end

                    else if (reg_mul_div[31] == 1'b0) begin // if remainder >=
divisor
                        remainder[31:0] = reg_mul_div[31:0];

                        // quotient add 1 by using adder
                        carry_bit_quotient[0] = 1'b1;
                        for (i = 0; i < 32; i = i + 1) begin
                            // calculation of carry bit
                            carry_bit_quotient[i + 1'b1] = quotient[i]
& carry_bit_quotient[i];
                            // calculation of add result
                            quotient[i]      <=      quotient[i]      ^
carry_bit_quotient[i];
                        end
                    end
                end
            end
        end
    end
end

// multiplication / division register
always @(posedge ip_clk) begin
    if (ip_rst == 1'b1) begin // if reset
    // reset to default value
        reg_mul_div[63:0] = 64'b0;
        carry_bit_reg_mul_div[64:0] = 65'b0;
        op_result[31:0] = 32'b0;
        op_overflow = 1'b0;
    end

    else if (ip_rst == 1'b0) begin // if no reset
        if (switch == 1'b1) begin // if the start calculate
            if (funct_3[2] == 1'b0) begin // if multiplication selected
                if (counter[4:0] == 5'b11111) begin // if done calculate (counter
counts to 31)
                    // convert answer of result should be negative
                    if (((funct_3[1] == 1'b0) && (sign_operand_a ^
sign_operand_b)) || (funct_3[1:0] == 2'b10) && (sign_operand_a == 1'b1)) begin
                        // 64-bit positive to negetive converter
                        reg_mul_div[63:0]   =   reg_mul_div[63:0]   ^
ONE64; // invert the value
                        // add 1 by using adder
                        carry_bit_reg_mul_div[0] = 1'b1;
                        for (i = 0; i < 64; i = i + 1) begin
                            carry_bit_reg_mul_div[i   +   1'b1]   =
reg_mul_div[i] & carry_bit_reg_mul_div[i]; // calculation of carry bit
                            reg_mul_div[i]   =   reg_mul_div[i]   ^
carry_bit_reg_mul_div[i]; // store result affter convert
                        end
                    end

                    // output result
                    if(funct_3[1:0] == 2'b00) begin // if MUL
```

```
                        op_result[31:0] <= reg_mul_div[31:0]; // output
lower 32-bit of result
                    end

                    else if (funct_3[1:0] != 2'b00) begin // if MULH, MULHU,
MULHSU
                        op_result[31:0] <= reg_mul_div[63:32]; // output
higher 32-bit of result
                    end

                    op_overflow <= 1'b0; // no overflow occured
                end
            end

            if (funct_3[2] == 1'b1) begin // if division is selected
                if (ready_div_op == 1'b1) begin // if division is ready to output
                    if (funct_3[1:0] == 2'b00) begin // if DIV
                        if ((sign_operand_a ^ sign_operand_b) == 1'b1)
begin // if signed bit of inputs are different (one positive and one negative)
                            // convert quotient from positive to
negative
                            quotient[31:0] = quotient[31:0] ^ ONE32;
                            // quotient add 1 by using adder
                            carry_bit_quotient[0] = 1'b1;
                            for (i = 0; i < 32; i = i + 1) begin
                                // calculation of carry bit
                                carry_bit_quotient[i + 1'b1] =
quotient[i] & carry_bit_quotient[i];

                                // calculation of add result
                                quotient[i] = quotient[i] ^
carry_bit_quotient[i];
                            end

                            op_result[31:0] <= quotient[31:0]; //
output quotient
                        end

                        else if ((sign_operand_a ^ sign_operand_b) ==
1'b0) begin // if signed bit of inputs are same (two positive or negative)
                            op_result[31:0] <= quotient[31:0]; //
output quotient
                        end
                    end

                    else if (funct_3[1:0] == 2'b01) begin // if DIVU
                        op_result[31:0] <= quotient[31:0]; // output
quotient
                    end

                    else if (funct_3[1:0] == 2'b10) begin // if REM
                        if (sign_operand_a == 1'b1) begin // if signed bit
of dividend is negative
                            // convert remainder from positive to
negative
                            remainder[31:0] = remainder[31:0] ^
ONE32;

                            // remainder add 1 by using adder
                            carry_bit_remainder[0] = 1'b1;
                            for (i = 0; i < 32; i = i + 1) begin
                                // calculation of carry bit
                                carry_bit_remainder[i + 1'b1] =
remainder[i] & carry_bit_remainder[i];
                                // calculation of add result
                                remainder[i] = remainder[i] ^
carry_bit_remainder[i];
                            end

                            op_result[31:0] <= remainder[31:0]; //
output remainder
                        end

                        else if (sign_operand_a == 1'b0) begin // if signed
bit of dividend is positve
                            op_result[31:0] <= remainder[31:0]; //
output remainder
                        end
                    end

                    else if(funct_3[1:0] == 2'b11) begin // if REMU
                        op_result[31:0] <= remainder[31:0]; // output
remainder
                    end

                    // output control sign of overflow
                    // if divisor = 0 or the largest negative number divide by -1
(special case)
                    if (operand_b[31:0] == 32'b0 || ((operand_a[31] == 1'b1)
&& (operand_a[30:0] == 31'b0) && (sign_operand_a == 1'b1) && (operand_b[31:0] == 32'b1) &&
(sign_operand_b == 1'b1))) begin
                        op_overflow <= 1'b1; // error occured
                    end

                    else begin
                        op_overflow <= 1'b0; // error occured
                    end
                end
            end
        end

        else if (switch == 1'b0) begin // if not calculate then reset value
            reg_mul_div[63:0] <= 64'b0;
            carry_bit_reg_mul_div[64:0] <= 65'b0;
            op_result[31:0] <= 32'b0;
            op_overflow <= 1'b0;
        end
    end
end

// counter and switch
always @(posedge ip_clk) begin
    if (ip_rst == 1'b1) begin // if reset
    // reset to default value
        op_nop_ctrl = 1'b0;
        counter[4:0] = 5'b0;
        carry_bit_counter[5:0] = 6'b0;
        switch = 1'b0;
        ready_div_op = 1'b0;
    end

    else if (ip_rst == 1'b0) begin // if no reset
```

```
                if (ip_m_ext_en == 1'b1) begin // m extandtion is selected
                        switch <= 1'b1; // to start calculation
                        counter[4:0] <= 5'b0; // reset counter for multiplication
                        op_nop_ctrl <= 1'b1; // enable stalling
                end

                if (switch == 1'b1) begin // if calculation is running
                        if (funct_3[2] == 1'b0) begin // if multiplication is selected
                                // counter + 1 by using adder
                                carry_bit_counter[0] = 1'b1;
                                for (i = 0; i < 5; i = i + 1) begin
                                        // calculation of carry bit
                                        carry_bit_counter[i    +    1'b1]    =    counter[i]    &
carry_bit_counter[i];
                                        // calculation of add result
                                        counter[i] <= counter[i] ^ carry_bit_counter[i];
                                end

                                // if multiplication done calculate
                                if (counter[4:0] == 5'b11111) begin // if counter counts to 31
                                        counter[4:0] <= 5'b0; // set back to default value
                                        switch <= 1'b0; // to stop calculation
                                        op_nop_ctrl <= 1'b0; // disable stalling
                                end
                        end

                        else if (funct_3[2] == 1'b1) begin // if division is selected
                                // if division done calculate
                                if (ready_div_op == 1'b1) begin
                                        switch <= 1'b0; // to stop calculation
                                        op_nop_ctrl <= 1'b0; // disable stalling
                                        ready_div_op <= 1'b0; // only output one clock cycle
                                end
                        end
                end
        end
end
endmodule
```

CHAPTER 5
Result and Discussion

5.1 Testbench

In this project only focus on the M extension of functionality and testing result. This test has fully test out all the scenario case that might occurred error.

- **Test Plan**

No	Description	Status
1.	**Test Case: MUL with two positive signed value** • Instruction MUL will be performed with two positive data when ip_m_ext_en is asserted and start compute multiplication of the result with 33 clock cycles. **Input Requirement:** • ip_operand_a = 32'h00015C7B (89211) • ip_operand_b = 32'h0000058A (1418) • ip_funct_3 = 3b'000 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'h078A414E (126501198) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle. • op_nop_ctrl will be high start from 2nd until 33th clock cycle and be low at 34th clock cycle.	**P**
2.	**Test Case: MUL with one positive and one negative signed value** • Instruction MUL will be performed with one positive and one negative data when ip_m_ext_en is asserted and start compute multiplication of the result with 33 clock cycles **Input Requirement:**	**P**

	• ip_operand_a = 32'hFFFEA385 (-89211) • ip_operand_b = 32'h0000058A (1418) • ip_funct_3 = 3b'000 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'hF875BEB2 (-126501198) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle. • op_nop_ctrl will be high start from 2nd until 33th clock cycle and be low at 34th clock cycle.	
3.	**Test Case: MUL with two negative signed value** • Instruction MUL will be performed with two negative data when ip_m_ext_en is asserted and start compute multiplication of the result with 33 clock cycles **Input Requirement:** • ip_operand_a = 32'hFFFEA385 (-89211) • ip_operand_b = 32'hFFFFFA76 (-1418) • ip_funct_3 = 3b'000 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'h078A414E (126501198) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle. • op_nop_ctrl will be high start from 2nd until 3th clock cycle and be low at 34th clock cycle.	**P**
4.	**Test Case: MULH with two positive signed value** • Instruction MULH will be performed with two positive data when ip_m_ext_en is asserted and start compute	**P**

	multiplication of the upper 32-bit of 64-bit result with 33 clock cycles. **Input Requirement:** • ip_operand_a = 32'h00015C7B (89211) • ip_operand_b = 32'h000426C4 (272068) • ip_funct_3 = 3b'001 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'h00000005 (5) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 33th clock cycle • op_nop_ctrl will be high start from 2nd until 34th clock cycle and be low at 34th clock cycle	
5.	**Test Case: MULH with one positive and one negative signed value** • Instruction MULH will be performed with one positive and one negative data when ip_m_ext_en is asserted and start compute multiplication of the upper 32-bit of 64-bit result with 33 clock cycles. **Input Requirement:** • ip_operand_a = 32'hFFFEA385 (-89211) • ip_operand_b = 32'h000426C4 (272068) • ip_funct_3 = 3b'001 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'hFFFFFFFA (-6) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 33th clock cycle.	**P**

	• op_nop_ctrl will be high start from 2nd until 34th clock cycle and be low at 34th clock cycle.	
6.	**Test Case: MULHSU with one positive signed and one unsigned value** • Instruction MULHSU will be performed with one positive signed and one unsigned data when ip_m_ext_en is asserted and start compute multiplication of the upper 32-bit of 64-bit result with 33 clock cycles **Input Requirement:** • ip_operand_a = 32'h00015C7B (89211) • ip_operand_b = 32'h9EC4BA46 (2663693870) • ip_funct_3 = 3b'001 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'h0000D81F (55327) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle • op_nop_ctrl will be high start from 2nd until 33th clock cycle and be low at 34th clock cycle.	**P**
7.	**Test Case: MULHSU with one negative signed and one unsigned value** • Instruction MULHSU will be performed with one negative signed and one unsigned data when ip_m_ext_en is asserted and start compute multiplication of the upper 32-bit of 64-bit result with 33 clock cycles **Input Requirement:** • ip_operand_a = 32'hFFFEA385 (-89211) • ip_operand_b = 32'h9EC4BA46 (4294911968) • ip_funct_3 = 3b'001 • ip_m_ext_en = 1'b1	**P**

	• Hold for 1 clock cycles. **Expected Output:** • op_result = 32'hFFFF27E0 (55327) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle • op_nop_ctrl will be high start from 2nd until 33th clock cycle and be low at 34th clock cycle.	
8.	**Test Case: MULHU with one positive signed and one unsigned value** • Instruction MULHU will be performed with two unsigned data when ip_m_ext_en is asserted and start compute multiplication of the upper 32-bit of 64-bit result with 33 clock cycles **Input Requirement:** • ip_operand_a = 32'hCAF1B84E (3404838990) • ip_operand_b = 32'h8841A4E9 (2286003433) • ip_funct_3 = 3b'011 • ip_m_ext_en = 1'b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32'h6C047404 (1812231172) • op_overflow = 1'b0 • the result only output for 1 clock cycle when reach 34th clock cycle. • op_nop_ctrl will be high start from 2nd until 33th clock cycle and be low at 34th clock cycle.	**P**
9.	**Test Case: DIV with two positive signed value** • Instruction DIV will be performed with two positive signed data when ip_m_ext_en is asserted and start compute quotient by using division with 10 clock cycles. **Input Requirement:**	**P**

	• ip_operand_a = 32’h000000BF (191) • ip_operand_b = 32’h00000017 (23) • ip_funct_3 = 3b’100 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h00000008 (8) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 12th clock cycle. • op_nop_ctrl will be high start from 2nd until 11th clock cycle and be low at 12th clock cycle.	
10.	**Test Case: DIV with one positive and one negative signed value** • Instruction DIV will be performed with one positive and one negative signed data when ip_m_ext_en is asserted and start compute quotient by using division with 10 clock cycles **Input Requirement:** • ip_operand_a = 32’h000000BF (191) • ip_operand_b = 32’hFFFFFFE9 (-23) • ip_funct_3 = 3b’100 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’hFFFFFFF8 (-8) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 12th clock cycle. • op_nop_ctrl will be high start from 2nd until 11th clock cycle and be low at 12th clock cycle.	**P**
11.	**Test Case: DIV with two negative signed value**	**P**

	• Instruction DIV will be performed with two negative signed data when ip_m_ext_en is asserted and start compute quotient by using division with 10 clock cycles **Input Requirement:** • ip_operand_a = 32’hFFFFFF41 (-191) • ip_operand_b = 32’hFFFFFFE9 (-23) • ip_funct_3 = 3b’100 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h00000008 (8) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 12th clock cycle. • op_nop_ctrl will be high start from 2nd until 11th clock cycle and be low at 12th clock cycle.	
12.	**Test Case: DIVU with two unsigned value** • Instruction DIV will be performed with two unsigned data when ip_m_ext_en is asserted and start compute quotient by using division with 11 clock cycles **Input Requirement:** • ip_operand_a = 32’hC7485D8D (3343408525) • ip_operand_b = 32’h15A51D1A (363142426) • ip_funct_3 = 3b’101 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h00000009 (9) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 13th clock cycle.	**P**

	• op_nop_ctrl will be high start from 2nd until 12th clock cycle and be low at 13th clock cycle.	
13.	**Test Case: REM with two positive signed value** • Instruction REM will be performed with two positive signed data when ip_m_ext_en is asserted and start compute remainder by using division with 11 clock cycles **Input Requirement:** • ip_operand_a = 32’h00001C72 (7282) • ip_operand_b = 32’h00000272 (626) • ip_funct_3 = 3b’110 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h0000018C (396) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 13th clock cycle. • op_nop_ctrl will be high start from 2nd until 12th clock cycle and be low at 13th clock cycle.	**P**
14.	**Test Case: REM with one positive and one negative signed value** • Instruction REM will be performed with one positive and one negative signed data when ip_m_ext_en is asserted and start compute remainder by using division with 11 clock cycles **Input Requirement:** • ip_operand_a = 32’h00001C72 (7282) • ip_operand_b = 32’hFFFFFD8E (-626) • ip_funct_3 = 3b’110 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:**	**P**

	• op_result = 32’h0000018C (396) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 13th clock cycle. • op_nop_ctrl will be high start from 2nd until 12th clock cycle and be low at 13th clock cycle.	
15.	**Test Case: REM with two negative signed value** • Instruction REM will be performed with two negative signed data when ip_m_ext_en is asserted and start compute remainder by using division with 11 clock cycles **Input Requirement:** • ip_operand_a = 32’hFFFFE38E (-7282) • ip_operand_b = 32’hFFFFFD8E (-626) • ip_funct_3 = 3b’110 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’hFFFFFE74 (-396) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 13th clock cycle. • op_nop_ctrl will be high start from 2nd until 12th clock cycle and be low at 13th clock cycle.	**P**
16.	**Test Case: REMU with two unsigned value** • Instruction REMU will be performed with two unsigned data when ip_m_ext_en is asserted and start compute remainder by using division with 11 clock cycles **Input Requirement:** • ip_operand_a = 32’hC7485D8D (3343408525) • ip_operand_b = 32’h15A51D1A (363142426) • ip_funct_3 = 3b’111 • ip_m_ext_en = 1’b1	**P**

	• Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h047A57A3 (75126691) • op_overflow = 1’b0 • the result only output for 1 clock cycle when reach 13th clock cycle. • op_nop_ctrl will be high start from 2nd until 12th clock cycle and be low at 13th clock cycle.	
17.	**Test Case: Divisor is zero when division (Special Case)** • When a division is occurred, however the divisor is zero which is an error in a division. Hence, a signal to indicate an error is required. **Input Requirement:** • ip_operand_a = 32’h003AE27C (3859068) • ip_operand_b = 32’h00000000 (0) • ip_funct_3 = 3b’100 • ip_m_ext_en = 1’b1 • Hold for 1 clock cycles. **Expected Output:** • op_result = 32’h00000000 (0) • op_overflow = 1’b1 • it required 2 clock cycle to output the overflow signal • the overflow signal only be high for 1 clock cycle when reach 3rd clock cycle. • op_nop_ctrl will be high 2nd clock cycle and be low at 3rd clock cycle.	**P**

Table 5.1 – Test Plan of M Extension

- **Testbench Model**

```
/*******************************************************************
Project: Developing Extended ISA on RISC-V Based Processor
Module: b_mul_tb.v
Version: 1
Date Created: 18/04/2023
Created By: Lee Ang
Code Type: Verilog
Description: RV32 M Extension Test Bench
*******************************************************************/

`include "macro.v"
`default_nettype none

module tb_b_m_ext
();
reg [31:0] ip_operand_a_tb, ip_operand_b_tb;
reg [2:0] ip_funct_3_tb;
reg ip_rst_tb, ip_clk_tb, ip_m_ext_en_tb;
wire [31:0] op_result_tb;
wire op_nop_ctrl_tb, op_overflow_tb;

b_m_ext
dut_b_m_ext(
        .ip_operand_a(ip_operand_a_tb),
        .ip_operand_b(ip_operand_b_tb),
        .ip_funct_3(ip_funct_3_tb),
        .ip_m_ext_en(ip_m_ext_en_tb),
        .ip_rst(ip_rst_tb),
        .ip_clk(ip_clk_tb),
        .op_result(op_result_tb),
        .op_nop_ctrl(op_nop_ctrl_tb),
        .op_overflow(op_overflow_tb)
);

initial ip_clk_tb <= 1'b1;
always #(`PERIOD_HALF) ip_clk_tb = ~ip_clk_tb;

initial begin
        @(posedge ip_clk_tb) // initialize the value (at sim time 1)
                ip_operand_a_tb[31:0] <= 32'b0;
                ip_operand_b_tb[31:0] <= 32'b0;
                ip_funct_3_tb[2:0] <= 3'b0;
                ip_m_ext_en_tb <= 1'b0;
                ip_rst_tb <= 1'b1;

        // test case 1 (MUL with two positive signed value)
        @(posedge ip_clk_tb) // insert value
                ip_operand_a_tb[31:0] <= 32'h00015C7B; // 89211
                ip_operand_b_tb[31:0] <= 32'h0000058A; // 1418
                ip_funct_3_tb[2:0] <= 3'b000;
                ip_m_ext_en_tb <= 1'b1;
                ip_rst_tb <= 1'b0;

        repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
                ip_operand_a_tb[31:0] <= 32'b0;
                ip_operand_b_tb[31:0] <= 32'b0;
                ip_funct_3_tb[2:0] <= 3'b000;
                ip_m_ext_en_tb <= 1'b0;
```

```
end

// test case 2 (MUL with one positive and one negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFEA385; // -89211
        ip_operand_b_tb[31:0] <= 32'h0000058A; // 1418
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 3 (MUL with two negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFEA385; // -89211
        ip_operand_b_tb[31:0] <= 32'hFFFFFA76; // -1418
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 4 (MULH with two positive signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h00015C7B; // 89211
        ip_operand_b_tb[31:0] <= 32'h000426C4; // 272068
        ip_funct_3_tb[2:0] <= 3'b001;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 5 (MULH with one positive and one negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFEA385; // -89211
        ip_operand_b_tb[31:0] <= 32'h000426C4; // 272068
        ip_funct_3_tb[2:0] <= 3'b001;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
```

```
end

// test case 6 (MULHSU with one positive signed and one unigned value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h00015C7B; // 89211
        ip_operand_b_tb[31:0] <= 32'h9EC4BA46; // 2663692870
        ip_funct_3_tb[2:0] <= 3'b010;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 7 (MULHSU with one negetive signed and one unigned value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFEA385; // -89211
        ip_operand_b_tb[31:0] <= 32'h9EC4BA46; // 2663692870
        ip_funct_3_tb[2:0] <= 3'b010;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 8 (MULHU with two unigned value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hCAF1B84E; // 3404838990
        ip_operand_b_tb[31:0] <= 32'h8841A4E9; // 2286003433
        ip_funct_3_tb[2:0] <= 3'b011;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(34) @(posedge ip_clk_tb) begin // 32 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 9 (DIV with two positive signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h000000BF; // 191
        ip_operand_b_tb[31:0] <= 32'h00000017; // 23
        ip_funct_3_tb[2:0] <= 3'b100;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(12) @(posedge ip_clk_tb) begin // 10 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
```

```
end

// test case 10 (DIV with one positive and one negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h000000BF; // 191
        ip_operand_b_tb[31:0] <= 32'hFFFFFFE9; // -23
        ip_funct_3_tb[2:0] <= 3'b100;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(12) @(posedge ip_clk_tb) begin // 6 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 11 (DIV with two negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFFFF41; // -191
        ip_operand_b_tb[31:0] <= 32'hFFFFFFE9; // -23
        ip_funct_3_tb[2:0] <= 3'b100;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(12) @(posedge ip_clk_tb) begin // 10 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 12 (DIVU with two unsigned value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hC7485D8D; // 3343408525
        ip_operand_b_tb[31:0] <= 32'h15A51D1A; // 363142426
        ip_funct_3_tb[2:0] <= 3'b101;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(13) @(posedge ip_clk_tb) begin // 11 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 13 (REM with two positive signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h00001C72; // 7282
        ip_operand_b_tb[31:0] <= 32'h00000272; // 626
        ip_funct_3_tb[2:0] <= 3'b110;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(15) @(posedge ip_clk_tb) begin // 13 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
```

```
end

// test case 14 (REM with one positive and one negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h00001C72; // 7282
        ip_operand_b_tb[31:0] <= 32'hFFFFFD8E; // -626
        ip_funct_3_tb[2:0] <= 3'b110;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(15) @(posedge ip_clk_tb) begin // 13 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 15 (REM with two negative signed value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hFFFFE38E; // -7282
        ip_operand_b_tb[31:0] <= 32'hFFFFFD8E; // -626
        ip_funct_3_tb[2:0] <= 3'b110;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(15) @(posedge ip_clk_tb) begin // 13 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 16 (REMU with two unsigned value)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'hC7485D8D; // 3343408525
        ip_operand_b_tb[31:0] <= 32'h15A51D1A; // 363142426
        ip_funct_3_tb[2:0] <= 3'b111;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(13) @(posedge ip_clk_tb) begin // 11 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end

// test case 17 (Special Case - divide by zero)
@(posedge ip_clk_tb) // insert value
        ip_operand_a_tb[31:0] <= 32'h003AE27C; // 3859068
        ip_operand_b_tb[31:0] <= 32'h00000000; // 0
        ip_funct_3_tb[2:0] <= 3'b100;
        ip_m_ext_en_tb <= 1'b1;
        ip_rst_tb <= 1'b0;

repeat(4) @(posedge ip_clk_tb) begin // 4 cycle to compute the result
        ip_operand_a_tb[31:0] <= 32'b0;
        ip_operand_b_tb[31:0] <= 32'b0;
        ip_funct_3_tb[2:0] <= 3'b000;
        ip_m_ext_en_tb <= 1'b0;
end
```

```
        // test case 18 (Special Case - the largest negative number divide by -1)
        @(posedge ip_clk_tb) // insert value
                ip_operand_a_tb[31:0] <= 32'h80000000; // -4294967296
                ip_operand_b_tb[31:0] <= 32'hFFFFFFFF; // -1
                ip_funct_3_tb[2:0] <= 3'b100;
                ip_m_ext_en_tb <= 1'b1;
                ip_rst_tb <= 1'b0;

        repeat(5) @(posedge ip_clk_tb) begin // 4 cycle to compute the result
                ip_operand_a_tb[31:0] <= 32'b0;
                ip_operand_b_tb[31:0] <= 32'b0;
                ip_funct_3_tb[2:0] <= 3'b000;
                ip_m_ext_en_tb <= 1'b0;
        end

// To stop simulation.
$stop;
end
endmodule
```

5.2 Result of M Extension

- Simulation Result

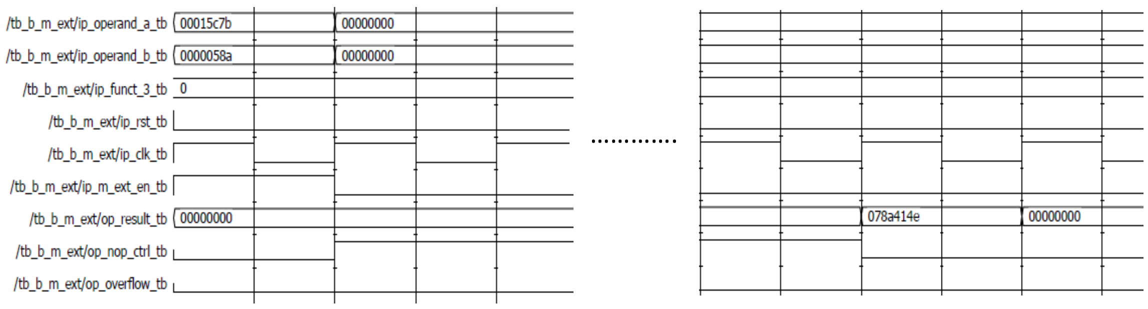


Figure 5.2.1 – Test case 1 Wave form

Figure 5.2.1 show that the simulation wave form of test case 1. Instruction MUL will be performed with two positive data when ip_m_ext_en is asserted and start compute multiplication of the result with 33 clock cycles. First, assert two data (32’h00015C7B (89211) and 32’h0000058A (1418)) into ip_operand_a[31:0] and ip_operand_b[31:0] and also insert 3’b000 into ip_funct_3[2:0] for select MUL as operation, and 1’b1 into ip_m_ext_en for 1 clock cycle to start the multiplication. After insert the data, set ip_operand_a[31:0], ip_operand[31:0], ip_funct_3[2:0] and ip_m_ext_en as default value which are all 0 value for 33 clock cycles. In between 2nd and 33th clock cycle, the switch which is in the counter turn on and op_nop_ctrl outputs high signal (1’b1) to datapath for 32 clock cycles for stalling the progam counter. The counter also starts count from 0 to 31. After 32 clock cycles, the output result pin (op_result[31:0]) outputs 32’h078A414E (126501198) for 1 clock cycle which the result (64’h00000000078A414E (126501198)) comes from the M extension general registers (reg_mul_div[63:0]). The overflow pin will be 0 for all the time.

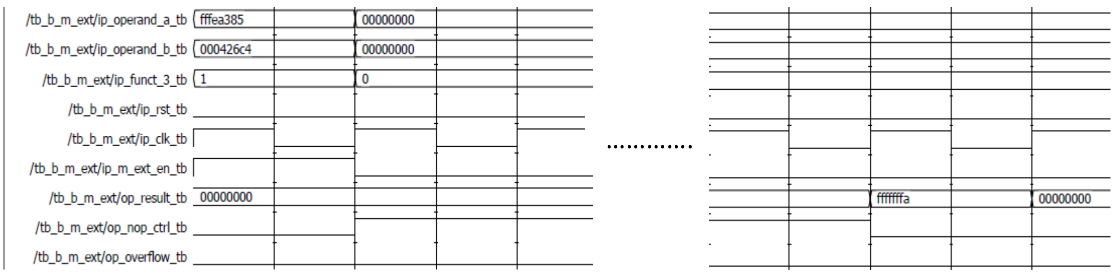


Figure 5.2.2 – Test case 5 Wave form

Figure 5.2.2 show that the simulation wave form of test case 5. Instruction MULH will be performed with one negative and one positive data when ip_m_ext_en is asserted and start compute multiplication of the result with 33 clock cycles. First, assert two data (32'h FFFEA385 (-89211) and 32'h000426C (272068)) into ip_operand_a[31:0] and ip_operand_b[31:0] and also insert 3'b001 into ip_funct_3[2:0] for select MULH as operation, and 1'b1 into ip_m_ext_en for 1 clock cycle to start the multiplication. After insert the data, set ip_operand_a[31:0], ip_operand[31:0], ip_funct_3[2:0] and ip_m_ext_en as default value which are all 0 value for 33 clock cycles. In between 2nd and 33th clock cycle, the switch which is in the counter turn on and op_nop_ctrl outputs high signal (1'b1) to datapath for 32 clock cycles for stalling the progam counter. The counter also starts count from 0 to 31. After 32 clock cycles, the output result pin (op_result[31:0]) outputs 32'hFFFFFFFA (-6) for 1 clock cycle which the result (64'hFFFFFFFA594EEFD4 (-24271458348)) comes from the M extension general registers (reg_mul_div[63:0]). The overflow pin will be 0 for all the time.

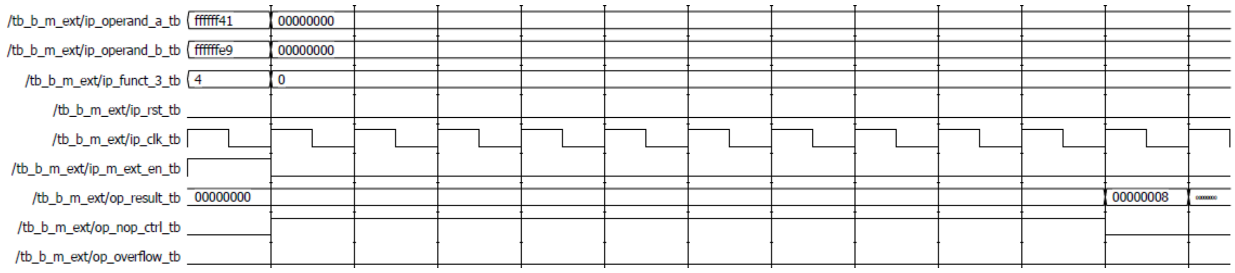


Figure 5.2.3 – Test case 11 Wave form

Figure 5.2.3 show that the simulation wave form of test case 11. Instruction DIV will be performed with two negative data when ip_m_ext_en is asserted and start compute division of the result with 10 clock cycles. First, assert two data (32'hFFFFFF41 (-191) and 32'hFFFFFFE9 (-23)) into ip_operand_a[31:0] and ip_operand_b[31:0] and also insert 3'b100 into ip_funct_3[2:0] for select DIV as operation, and 1'b1 into ip_m_ext_en for 1 clock cycle to start the multiplication. After insert the data, set ip_operand_a[31:0], ip_operand[31:0], ip_funct_3[2:0] and ip_m_ext_en as default value which are all 0 value for 10 clock cycles. In between 2nd and 11th clock cycle, the switch which is in the counter turn on and op_nop_ctrl outputs high signal (1'b1) to datapath for 10 clock cycles for stalling the progam counter. The switch also will be one to start the calculation until remainder is less than divisor. If the remainder is greater than divisor, the quotient will plus one on it to calculate result.

After 10 clock cycles, the output result pin (op_result[31:0]) outputs 32’h00000008 (8) for 1 clock cycle which the result comes from the quotient registers (quotient [31:0]). The overflow pin will be 0 for all the time.

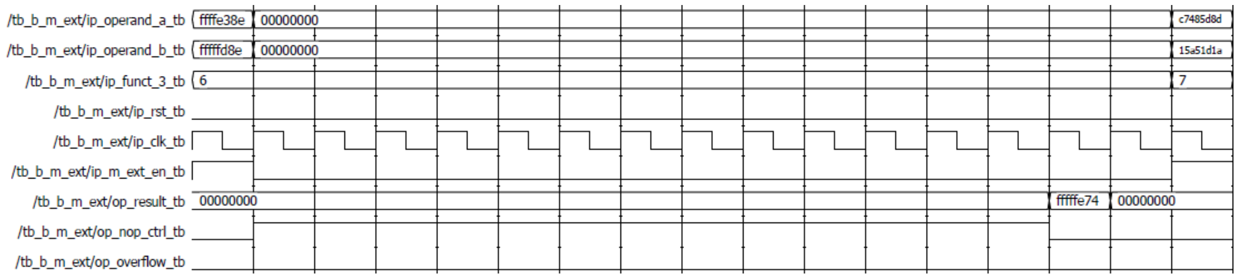

/tb_b_m_ext/ip_operand_a_tb
ffffe38e
00000000
c7485d8d
/tb_b_m_ext/ip_operand_b_tb
ffffd8e
00000000
15a51d1a
/tb_b_m_ext/ip_funct_3_tb
6
7
/tb_b_m_ext/ip_rst_tb
/tb_b_m_ext/ip_clk_tb
/tb_b_m_ext/ip_m_ext_en_tb
/tb_b_m_ext/op_result_tb
00000000
fffffe74
00000000
/tb_b_m_ext/op_nop_ctrl_tb
/tb_b_m_ext/op_overflow_tb

Figure 5.2.4 – Test case 15 Wave form

Figure 5.2.4 show that the simulation wave form of test case 15. Instruction REM will be performed with two negative data when ip_m_ext_en is asserted and start compute division of the result with 10 clock cycles. First, assert two data (32’hFFFFE38E (-7282) and 32’hFFFFFD8E (-626)) into ip_operand_a[31:0] and ip_operand_b[31:0] and also insert 3’b110 into ip_funct_3[2:0] for select REM as operation, and 1’b1 into ip_m_ext_en for 1 clock cycle to start the multiplication. After insert the data, set ip_operand_a[31:0], ip_operand[31:0], ip_funct_3[2:0] and ip_m_ext_en as default value which are all 0 value for 11 clock cycles. In between 2nd and 11th clock cycle, the switch which is in the counter turn on and op_nop_ctrl outputs high signal (1’b1) to datapath for 11 clock cycles for stalling the progam counter. The switch also will be one to start the calculation until remainder is less than divisor. If the remainder is greater than divisor, the quotient will plus one on it to calculate result. After 11 clock cycles, the output result pin (op_result[31:0]) outputs 32’h FFFFFE74 (-396) for 1 clock cycle which the result comes from the remainder registers (remainder[31:0]). The overflow pin will be 0 for all the time.

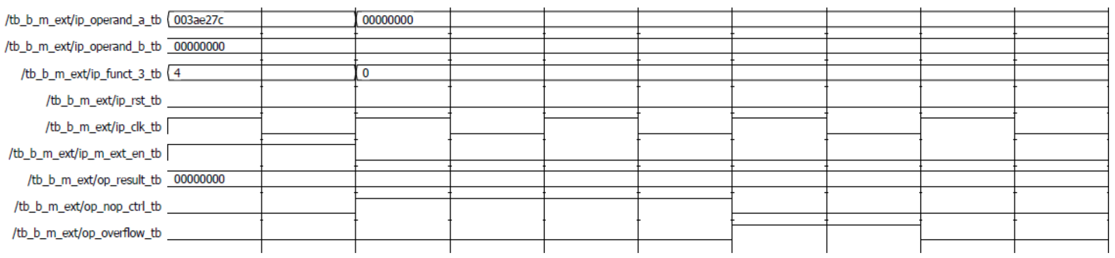

/tb_b_m_ext/ip_operand_a_tb
003ae27c
00000000
/tb_b_m_ext/ip_operand_b_tb
00000000
/tb_b_m_ext/ip_funct_3_tb
4
0
/tb_b_m_ext/ip_rst_tb
/tb_b_m_ext/ip_clk_tb
/tb_b_m_ext/ip_m_ext_en_tb
/tb_b_m_ext/op_result_tb
00000000
/tb_b_m_ext/op_nop_ctrl_tb
/tb_b_m_ext/op_overflow_tb

Figure 5.2.5 – Test case 17 Wave form

Figure 5.2.5 show that the simulation wave form of test case 15. This test case is a special that the input value might occurred error when calculating a division. When a divisor is zero, means it is a error for a division, then the error signal will be high to indicate is an error there. First, assert two data (32'h003AE27C (3859068) and 32'h00000000 (0)) into ip_operand_a[31:0] and ip_operand_b[31:0] and also insert 3'b100 into ip_funct_3[2:0] for select DIV as operation, and 1'b1 into ip_m_ext_en for 1 clock cycle to start the division. After insert the data, set ip_operand_a[31:0], ip_operand[31:0], ip_funct_3[2:0] and ip_m_ext_en as default value which are all 0 value for 2 clock cycles. the switch which is in the counter turn on and op_nop_ctrl outputs high signal (1'b1) to datapath for 2 clock cycle for stalling the progam counter. The switch also will be one to start the calculation however the divisor is zero. In 4th clock cycle, it will output the output result pin (op_result[31:0]) as 32'h 00000000 (0) and error signal (op_overflow) become high for 1 clock cycle. Hence, it will directly end the operation and light up the error signal.

- **Performance**

By applying M extension, it has help to reduce clock cycles needed for perform a multiplication or division. For multiplication it required few instructions like SRL, AND, BEQ and ADD to performe a for loop for add and shift in 32 cycle. Below shown how to perform a multiplication by using I instruction set only:

```
#Input
li t2, 10
li t4, 3
loop:
  srli  t3, t2, 0   # get the least significant bit of a
  andi t3, t3, 1  # mask off all but the least significant bit
  beq t3, zero, skip   # if the bit is 0, skip the addition
  add  t0, t0, t4  # if the bit is 1, add b to the result
skip:
  srli  t2, t2, 1    # shift a right by 1
  add  t1, t1, 1  # increment a counter
  bne t1, 32, loop   # loop until all 32 bits have been multiplied
  mv   s0, t0      # move the result to c
```

Accourding the assembly code above, it has used 8 instruction to do a multiplication. However, M extension instruction is designed to reduced the instruction into a single instruction. Hence, it at least increase 6 times faster in multiplication performance.

For division, below shown how a to use I instruction set only to perform a division:

```
# Initialize the dividend and divisor in registers
li a0, 100    # Dividend = 100
li a1, 10     # Divisor = 10
# Compute the sign of the quotient
xor a2, a0, a1   # Check if the signs of the dividend and divisor are different
bltz a2, negate  # If the signs are different, negate the quotient at the end
li a2, 1         # Otherwise, the quotient is positive
# Initialize the quotient and remainder to 0
li a3, 0    # Quotient = 0
li a4, 0    # Remainder = 0
# Compute the quotient and remainder using shift and subtract
loop:
   sll a4, a4, 1    # Shift remainder left by 1 bit
   srl a5, a0, 31   # Get the sign bit of the dividend
   or a4, a4, a5    # Add sign bit to remainder
   sll a0, a0, 1    # Shift dividend left by 1 bit
   sub a6, a4, a1   # Compute the difference between remainder and divisor
   bge a6, zero, subtract  # If difference is non-negative, subtract divisor from
remainder
   or a3, a3, a2     # Add quotient sign bit to quotient
   srl a5, a3, 31    # Get the sign bit of the quotient
   bne a5, a2, negate # If the signs of quotient and dividend are different,
negate quotient
   jal end           # Otherwise, division is complete
subtract:
   addi a4, a4, -a1  # Subtract divisor from remainder
   ori a3, a3, 1     # Add 1 to quotient
   jal loop
negate:
   neg a3, a3        # Negate quotient
end:
```

As can found that it required couple line of instructions. However, the division part of M extension might require more clock cycle to compute a result. When there is a very large dividend divide by a very small divisor, the consume more clock cycle to do subtraction compare with a fix number of instructions needed. Hence, It is able to help reducing number of instructions needed when expected quotient is a small value number, but might decrease performance when expected quotient is a very big value.

As result, there are improvement on multiplication, but increment or decrement performance of division is depending on the gap between two inputs.

5.3 Implementation Issues and Challenges

In the progress of developing extended ISA, there are few difficulties. First, the unsigned and signed data compute in M extension is a major challenge to get a correct result. Signed data need to consider the negative value and positive value because it will cause different result. The operand includes negative value like a negative multiply a positive should consider shifting with zero extend or signed extend of the rs1 value. Differential of extension will cause different result in the most 32 significant bits in 64 bit result register that related to instruction of MULH and MULHSU. Furthermore, the division of in M extension even be more challenges to design.

Comparing with multiplication, division is using subtraction to implement the division. It need consider more logic rules such as signed division, division with zero values, clock cycle requirement. Signed division need to consider sequence of positive value and negative division because it effects to consider using addition or subtraction on reducing dividend to get quotient. For example, positive value divides positive value required using subtraction because positive value should minus positive values is correct way to reduce dividend; for positive value divides negative value required using addition because positive value should minus positive values however the divisor in a negative value. If using subtraction, the dividend will become larger and larger and unable to get the answer. Hence, using addition on a negative value can be seen as doing subtraction on division. In above method, there are a bug of division with zero values because dividend subtract with zero will remain the same value and cause the infinite loop for doing division. So, it required to design a logic gate to detect there is a division with zero value. In multiplication, it is fixed to required 32 clock to implement a multiplication, but division is another story. The division execution clock cycle depends on time of subtraction, so there is not a fix clock cycle to come out a result. Therefore, it required to design comparator to detect dividend is unable to minus anymore to prove that division is complete.

CHAPTER 6
Conclusion

In a nutshell, this project aims to develop extended ISA on RISC based processor. The progress will be separate into several part. First, design a 5-stage instruction execution processor that reference to RV32I processor that computation with 32-bit width data. The stages are instruction fetch, instruction decode, execute, memory access and write back stage. So far the project has designed the instruction fetch unit that contain a function to compute data in the data path, registers file that store 32 of 32-bit values, ALU that compute data with the function selected like, arithmetic, logical, bit shifting, branch or jump address, etc, data memory that have more larger space, and control unit of processor that controlling the component function output the correct result.

In additions, the development of Standard Extension for Integer Multiplication and Division (M) also has been designed. However, there are some challenges when design the multiply and divide function. Signed data with negative value and positive value calculate multiplication cause different result in most significant bit. It is solved by zero extend or signed extend of the multiplicand. In division part, it need consider more logic rules such as signed division that need consider to compute addition or subtraction to achieve dividend minus divisor one by one to get quotient, division with zero values that need design a detector for fixing division bugs, and clock cycle requirement that need detect unpredictable cycle to stop division by comparing dividend and divisor.

As result of this extension has reduced the clock cycle, improve the performance of multiplication, but not in division due to gap of clock cycle requirement can be very large. Nevertheless, all 8 instruction of M extension are functional. After analyse the project can found that there are few future work can be implemented. For example, multiplication part can using plenty of mux to further decrease clock cycle required into 1 clock cycle. For division can change operating arithmetic method into “subtract and shift” method to solve the unpredictable clock cycle requirement. Since, there is only one extension be done, future development can be floating point or vector extension even though customize extension like computing cryptography.

In the end, this work focuses on developing an extended ISA on RISC-V based processors to provide potential customers, such as chipset manufacturers and IoT device companies, with powerful, high-performance processors that meet specific requirements. The flexibility of customizing the ISA helps different IT areas to be more satisfied with the processor they receive, and the growth of RISC-V ISA has the potential to disrupt the dominance of proprietary architectures in the market. RISC-V offers a free and open alternative that can be customized for specific use cases, making it more accessible for smaller companies and startups and enabling new applications and use cases, such as low-power IoT devices and specialized machine learning hardware.

REFERENCES

1. Codsip. “Extending RISC-V ISA With a Custom Instruction Set Extension,” design-reuse.com. https://www.design-reuse.com/articles/46237/extending-risc-v-isa-with-a-custom-instruction-set-extension.html (accessed Apr 10, 2022)
2. A. Waterman, “Design of the RISC-V Instruction Set Architecture,” Ph.D. dissertation, Dept. Elect. Univ. California, Berkeley, California, USA, 2016. [Online]. Available: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf
3. WikiChip, “ARMv8 - ARM” en.wikichip.org. https://en.wikichip.org/wiki/arm/armv8#:~:text=ARMv8%20(codename%20Oban)%20is%20the,of%2064%2Dbit%20operating%20capabilities. (accessed Apr 10, 2022)
4. R. Awati, “Scalable Processor Architecture (SPARC),” searchservervirtualization.techtarget.com. https://searchservervirtualization.techtarget.com/definition/SPARC (accessed Apr 10, 2022)
5. Wikipedia, “OpenRISC,” en.wikipedia.org. https://en.wikipedia.org/wiki/OpenRISC#:~:text=OpenRISC%20is%20a%20project%20to,project%20of%20the%20OpenCores%20community (accessed Apr 10, 2022)
6. R. E. Bryant, “Alpha Assembly Language Guide,” dissertation, Univ. Carnegie Mellon, Pittsburgh, Pennsylvania, USA, 1998. [Online]. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15213-f98/doc/alpha-guide.pdf
7. A. Waterman, “The RISC-V Instruction Set Manual,” Ph.D. dissertation, Dept. EECS. Univ. California, Berkeley, California, USA, 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
8. J. He, “SUPERSCALAR RISC-V PROCESSOR WITH SIMD VECTOR EXTENSION,” M.S. dissertation, Dept. Elect. Univ. Saskatchewan, Saskatoon, Saskatchewan, Canada, 2020. [Online]. Available: https://harvest.usask.ca/bitstream/handle/10388/13040/HE-THESIS-2020.pdf?sequence=1

9. Ginni, "What is SIMD Architecture?" tutorialspoint.com. https://www.tutorialspoint.com/what-is-simd-architecture#:~:text=SIMD%20represents%20single%2Dinstruction%20multiple,as%20displayed%20in%20the%20figure. (accessed Apr 10, 2022)
10. A. Tong, "Dynamic Scheduling," cs.umd.edu. https://www.cs.umd.edu/~meesh/cmsc411/website/projects/dynamic/tomasulo.html. (accessed Apr 10, 2022)
11. E. F. Gehringer, "Improved Branch Predictors," people.engr.ncsu.edu. https://people.engr.ncsu.edu/efg/521/f02/common/lectures/notes/lec16.pdf. (accessed Apr 10, 2022)
12. Wikipedia, "Verilog," en.wikipedia.org. https://en.wikipedia.org/wiki/Verilog (accessed Apr 10, 2022)
13. Model SE 2020. (2020). Siemens Accessed: April 8, 2022. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ s Software
14. G. Nişancı, P. G. Flikkema, and T. Yalçın, "Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms," *Cryptography*, vol. 6, no. 3, p. 41, Aug. 2022, doi: https://doi.org/10.3390/cryptography6030041.
15. Aoki, K.; Ichikawa, T.; Kanda, M.; Matsui, M.; Moriai, S.; Nakajima, J.; Tokita, T. Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms—Design andAnalysis. In Selected Areas in Cryptography; Stinson, D.R., Tavares, S., Eds. Springer: Berlin/Heidelberg, Germany, 2001, pp. 39–56. (accessed on 12 April 2023). [CrossRef]

APPENDIX

Final Year Project Weekly Report

(Project II)

Trimester, Year: Y3S3	**Study week no.:** 1-4
Student Name & ID: Lee Ang, 20ACB04056	
Supervisor: Ts. Ooi Joo On	
Project Title: Developing Extended ISA on RISC Based Processor	

1. WORK DONE - Multiplier development - Testing and debugging RV32I
2. WORK TO BE DONE - Testing and debugging multiplier in M extension - Study division architecture - Develop divider
3. PROBLEMS ENCOUNTERED - Wrong upper 32-bit result of signed data multiplication
4. SELF EVALUATION OF THE PROGRESS - still can follow up the progress

[illegible] [illegible]

Supervisor's signature Student's signature

APPENDIX

Trimester, Year: Y3S3	**Study week no.:** 5-9
Student Name & ID: Lee Ang, 20ACB04056	
Supervisor: Ts. Ooi Joo On	
Project Title: Developing Extended ISA on RISC Based Processor	

1. WORK DONE - Testing and debugging multiplier in M extension - Study division architecture - Develop divider
2. WORK TO BE DONE - Develop divider - Testing and debugging divider in M extension
3. PROBLEMS ENCOUNTERED - Wrong concept division design.
4. SELF EVALUATION OF THE PROGRESS - still can follow the progress

[illegible]

Supervisor's signature

LeeAng

Student's signature

APPENDIX

Trimester, Year: Y3S3	**Study week no.:** 10-13
Student Name & ID: Lee Ang, 20ACB04056	
Supervisor: Ts. Ooi Joo On	
Project Title: Developing Extended ISA on RISC Based Processor	

1. WORK DONE - Develop divider
2. WORK TO BE DONE - Testing and Debugging divider - FYP2 Report
3. PROBLEMS ENCOUNTERED - wrong clock cycle control cause unexpected output
4. SELF EVALUATION OF THE PROGRESS **-** a bit behind schedule

[illegible]

Supervisor's signature

LeeAng

Student's signature

POSTER

Final Year Project

DEVELOPING EXTENDED ISA ON RISC BASED PROCESSOR

UTAR UNIVERSITI TUNKU ABDUL RAHMAN

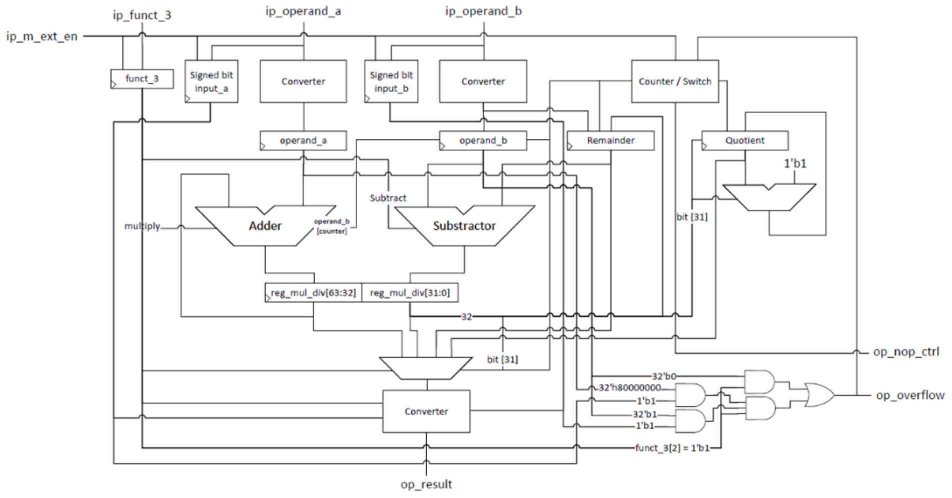
INTRODUCTION

This project aims to develop extended ISA on RISC-V based processor. Those ISA can help the application to accelerate progress during execution and improve the performance. Therefore, it is a useful product since it preserves software compatibility while also allowing for differentiation and innovation. This project will show how it develops a RISC-V RV32I processor and its extension by logic gates and simulates by using Verilog code in ModelSim.

OBJECTIVE

- Design RV32I RISC-V processor in execute 5 stage
- Design M extension that can do multiplication and division
- M extension is flexible to integrate in RV32I processor
- Reduce the number of clock cycles when performing multiplication or division

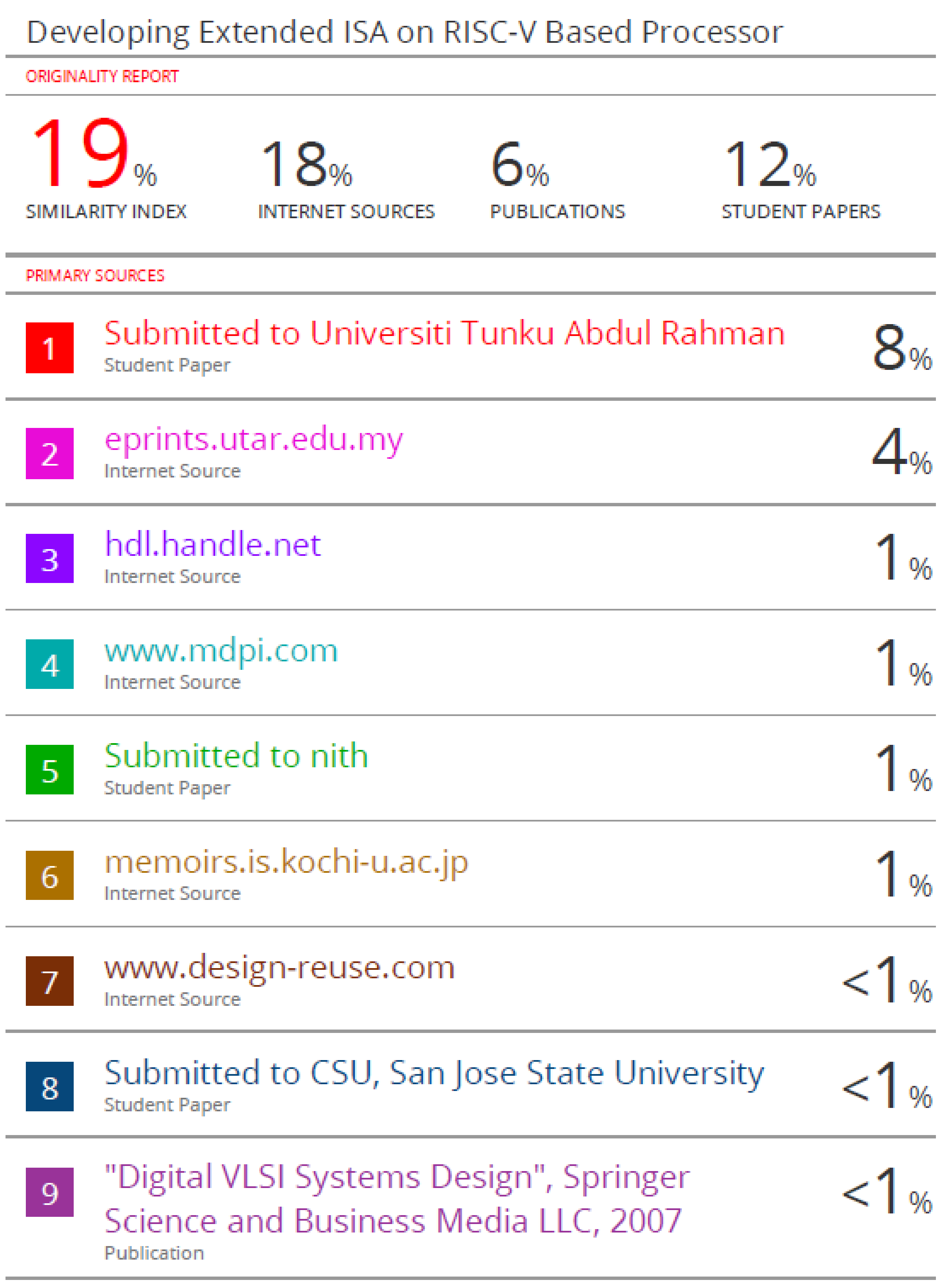
SYSTEM DESIGN



CONCLUSION

This project aims to develop an extended RISC-V ISA, with a focus on the Integer Multiplication and Division (M) extension. The project has designed a 5-stage instruction execution processor, which has reduced clock cycles and improved multiplication performance. Future work includes further optimizing the processor and developing new extensions. RISC-V offers a free and open alternative to proprietary architectures, with potential for disrupting the market and enabling new applications.

Plagiarism Check Result



Developing Extended ISA on RISC-V Based Processor

ORIGINALITY REPORT

19%	18%	6%	12%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Universiti Tunku Abdul Rahman Student Paper	8%
2	eprints.utar.edu.my Internet Source	4%
3	hdl.handle.net Internet Source	1%
4	www.mdpi.com Internet Source	1%
5	Submitted to nith Student Paper	1%
6	memoirs.is.kochi-u.ac.jp Internet Source	1%
7	www.design-reuse.com Internet Source	<1%
8	Submitted to CSU, San Jose State University Student Paper	<1%
9	"Digital VLSI Systems Design", Springer Science and Business Media LLC, 2007 Publication	<1%

PLAGIARISM CHECK RESULT

Universiti Tunku Abdul Rahman			
Form Title: Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Lee Ang
ID Number(s)	010513-01-0327
Programme / Course	Computer Engineering (CT)
Title of Final Year Project	Developing Extended ISA on RISC Based Processor

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)**
Overall similarity index: 19 **%** **Similarity by source** Internet Sources: 18 % Publications: 6 % Student Papers: 12 %	The percentage is acceptable.
Number of individual sources listed of more than 3% similarity: 2	The number of individual sources listed is acceptable.

Parameters of originality required and limits approved by UTAR are as Follows:
(i) Overall similarity index is 20% and below, and
(ii) Matching of individual sources listed must be less than 3% each, and
(iii) Matching texts in continuous block must not exceed 8 words
Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.

Note Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

[illegible]

Signature of Supervisor	Signature of Co-Supervisor
Name: Ts. Ooi Joo On	Name:
Date: 25/4/2023	Date:

FYP2 Report Checklists

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	20ACB04056
Student Name	Lee Ang
Supervisor Name	Ts. Ooi Joo On

TICK (√)	**DOCUMENT ITEMS** Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
	Front Plastic Cover (for hardcopy)
√	Title Page
√	Signed Report Status Declaration Form
√	Signed FYP Thesis Submission Form
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
	Appendices (if applicable)
√	Weekly Log
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)
√	I agree 5 marks will be deducted due to incorrect format, declare wrongly the ticked of these items, and/or any dispute happening for these items in this report.

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report.

[illegible]

(Signature of Student)
Date: 27/4/2023