Design and Verification of a Pipelined RISC-V (RV32IM) Processor with RTOS Integration

submitted in partial fulfillment of the requirements

for the research internship

RESEARCH INTERNSHIP

in

VLSI & Embedded Systems Lab

by

K.CHARAN

Supervisor(s)

Dr. VikramKumar Pudi

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

May 2025

DECLARATION

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

Place: Tirupati
Date: 07-05-2025

Signature
K.Charan

Note: If more than one authors mentioned in the cover page, modify each 'I' by 'we' and then include remaining author's name at the bottom of the page. Signature of each author in this page is must.

ACKNOWLEDGMENTS

We would like to express our special thanks of gratitude to Dr. Vikramkumar Pudi for giving his valuable suggestions during the planning and development of this work. His guidance was extremely helpful and motivating.

I would like to thank Dr Jaynarayan T Tudu, Assistant Professor Indian Institute of Technology. Tirupati, in guiding me through some key insights. I would like to thank the VLSI lab JTS Mr. Kumar for his help and Vanama Sai Srinivas(EE22B052) .Lastly, I would also like to thank our parents and friends, who were supportive throughout the project.

Place: Tirupati
Date: 07-05-2025

Signature
K.Charan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ISA	Instuction set architecture
RISC-V	fifthe verion ISA developed based on Reduced Instruction Set Architecture principles
FPGA	Feild Programable Gate Array
UART	Universal asynchronous Receving and Transmison

NOTATION

rd	Destination register address of size 5 bitsto store the computed value
rs1,rs2	Data address of size 5 bits of the data needed to be computed
Imm	32 bit Immediate value decode form the INstruction

CHAPTER 1

INTRODUCTION

RISC(Reduced Instruction Set Computer) offers advantages over CISC (Complex Instruction Set Computer) through simpler instructions, streamlined execution, and improved performance, enabling faster clock speeds and more efficient use of resources. RISC-V (Reduced Instruction Set Computer - Five) Instruction Set Architecture, a 5th major open-source RISC-based Instruction Set Architecture, originated from the efforts of researchers at the University of California, Berkeley, in 2010.

Unlike proprietary ISAs such as ARM or x86, RISC-V is freely available for anyone to use, modify, and distribute. It's modular design allows for flexibility in implementation, enabling developers to customize processors for specific applications or performance requirements. The architecture is designed to be extensible, allowing developers to add custom instructions or extensions to meet specific application requirements. It is a simple load and store architecture that supports 32-bit and 64-bit base integer instruction formats, with optional extensions for specific use cases, such as floating-point arithmetic, atomic operations, and vector processing which gives flexibility in terms of designing application-specific processors

RISC-V stands for "**Reduced Instruction Set Computer - Five**," where the "Five" refers to the fifth version RISC ISA designed by designes from the University of California, Berkeley. follows a **load-store architecture**, which is a type of Reg-Reg/Load-Store ISA. It has a relatively simple instruction set

Base	**about**	**Status**
RV32I	32 bit Integer GPR's	standard
RV64I	64 bit Integer GPR's	Ratified
RV32E	only 16 GPR's,used for embedded applications	Draft
RV128I	128 bit Integer GPR's	Draft
Extension	**about**	**Status**
M	Includes Direct Multiplication, Division ability	Ratified
A	Memory synchronization, Multi thread processing	Ratified
F	supports single precison Floting ponit operations	Ratified
Zicsr	enable software,to manage processor behaviour via CSR's	Ratified
C	supports compressed instructions	Ratified

Table 1.1: Table: Description of RISC-V Base and Extension Types

1.1 Objectives

The objective of this work is to design the RISC-V-based pipelined processor with a 32bit base ISA of RV32I along with M extension for direct multiplication and Division operations supportability and Its verification using different assembly programs, along with the help of RISC-V GNU Toolchain for for converting the c and c++ programs into the required instructions to run and test on the Designed processor

CHAPTER 2

LITERATURE REVEIW

For the Design purpose, I have referred to the Computer Architecture RISC-V edition Harris book for designing principles of single-cycle and pipeline processors.I have referred the **?**, and NEO32_Processor for software firmware to use the gnu toolchain for getting the instructions to run on the Designed processor.

2.1 RV32I ISA

Any RISC-V isa-based instruction would be in the six formats shown in Figure2.1. The major are R, I, S, and U, while B and J are the same as S and U except for the immediate encoding.

There are 46 instructions in the RV32 base ISA in Specifications Waterman (2017) in 2017 version. later, they separated 6 instructions from it as an extension of Zicsr. According to the new version SpecificationWaterman and Asanovic (2019), there are 40 base instructions in the RV32I base ISA, whereas 2 are system instructions that play an important role in environment executions.

There are 9 types of instructions in 32-bit base ISA, The purpose of each type of Instruction is shown in the Table 2.1 and their respective encoding formats are mentioned in Figure 2.2

There are 38 Instructions of our interest . Tables 2.2,2.3 2.4 and 2.5 show the computation instructions.Where the Table 2.6 shows the load and store instructions. Tables2.7 and 2.8 show the conditional and unconditional jumps, respectively.

Format	7'[31:25]	5'[24:20]	5'[19:15]	3'[14:12]	5'[11:7]	7'[6:0]
R	func7	rs2	rs1	func3	rd	opcode
I	imm		rs1	func3	rd	opcode
S	imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode
B	imm[12][10:5]	rs2	rs1	func3	imm[4:1][11]	opcode
J	imm[20][10:1][11][19:12]				rd	opcode
U	imm[20 bits]				rd	opcode

Figure 2.1: Instruction formats r1

Type	Purpose
R	Only Registers data computation
RI	Computation with constants
LD	Memory load
SR	Memory store
BR	Conditional Looped computations
JAL ,JALR	Calling routines ,subroutines
AUIPC	To jump to functions far from the current one
LUI	Loading larger constant values

Table 2.1: RV32I Instruction Purpose

Type	Format	7'[31:25]	5'[24:20]	5'[19:15]	3'[14:12]	5'[11:7]	7'[6:0] opcode
R	R	func7	rs2	rs1	func3	rd	0110011
RI	I	imm[11:0]		rs1	func3	rd	0010011
LD	I	imm[11:0]		rs1	func3	rd	0000011
SR	S	imm[11:5]	rs2 (Data)	rs1	func3	imm[4:0]	0100011
BR	B	imm[12][10:5]	rs2	rs1	func3	imm[4:1][11]	1100011
JAL	J	imm[20][10:1][11][19:12]				rd	1101111
JALR	I	imm[11:0]		rs1	func3	rd	1100111
AUIPC	U	imm[31:12]				rd	0010111
LUI	U	imm[31:12]				rd	0110111

Figure 2.2: RV32I Instruction Types r1

Instruction	Computation
ADD, ADDI	Addition
SUB	Subtraction
SLL, SLU	Shift logical data left
SRL, SRU	Shift logical data right
SRA, SRAI	Shift Arithmetic data right

Table 2.2: Arithmetic Instructions

Instruction	Computation
AND, ANDI	Bit-wise AND operation
OR, ORI	Bit-wise OR operation
XOR, XORI	Bit-wise XOR operation (also used for 1's complement computation)

Table 2.3: Logical Instructions

Instruction	Computation
SLT, SLTI	Set 1, if signed comparisons less than
SLTU, SLTIU	Set 1, if unsigned comparisons less than

Table 2.4: Comparison Instructions

Instruction	Computation
LUI	Load upper 20-bit immediate
AUIPC	Add upper 20-bit immediate to PC (jump to any address in the address space)

Table 2.5: Special Computation Instructions

Instruction	Computation
LB, SB	Load/store byte
LH, SH	Load/store half-word
LW, SW	Load/store word
LBU, LHU	Load/store unsigned byte, half-word

Table 2.6: Memory Access Instructions

Instruction	Operation
BEQ, BNE	Jump if (a=b), (a!=b) respectively
BGE, BLT	Jump if signed (a>=b), (a<b) respectively
BGEU, BLTU	Jump if unsigned (a>=b), (a<b) respectively

Table 2.7: Conditional Control Flow Instructions

Instruction	Operation
JAL	Jump to PC+imm (typically used for function calls)
JALR	jump to (rs1)+(imm) (typically used for function returns)

Table 2.8: Unconditional Control Flow Instructions

CHAPTER 3

PROCESSOR DESIGN

3.1 Basic Functionality Modules

Program Counter: The first mode of communication with the processor is instruction, which we can get from the Instruction memory, with the proper address, which is obtained from the program counter, .The program counter holds the current address of the instruction, and its value is decided by the Mux, which has inputs of incremented PC value or branch or jump address value.

Immediate Generator The module shown in the figure 3.2b converts the encoded immediate value in the instruction to the required 32-bit Value according to the respective instruction format shown in the 2.1. **ALU 2nd input mux**(aluIn2_Mux)" We need to choose the data to be sent to the ALU unit using a mux.It chooses between the rs2data read from the GPRs or the immediate value decoded from the instruction; based on the control signal, it chooses one.

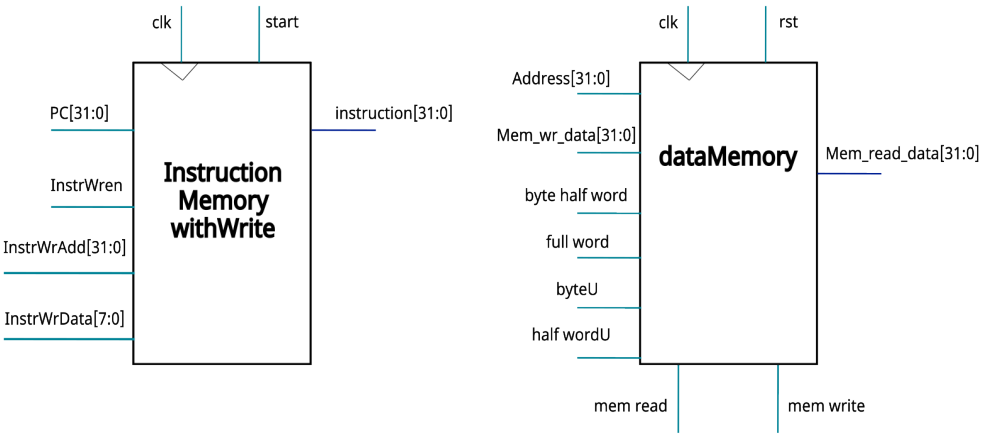
Data Memory: The Data memory shown in the figure 3.1b hold the required data must support the read and write operations when appropriate signals are enabled. We can use the asynchronous reading data memory in a single-cycle processor design with less data memory, which will be converted into the LUTRAMs in the FPGA when synthesized. When our data memory size is larger, we need to use the BRAM in the FPGA, making the memory synchronous read and synchronous write.

General Purpose registers(GPR's): Its dual port 32bit 32 number of registers stores the temporal data of the data memory. The first register is always zero. To write this module, we need a Demux and a Mux to access the register's data. The ports for this are shown in the the figure 3.3a

Arthamatic and logic unit(ALU): The ALU module shown in the figure3.3bcan be useful for performing the computation and branch instructions. We can use the existing comparison functionality used for SLT(set less than)instruction and XOR instruction for checking greater or less than and equality comparison branch instructions respectively.

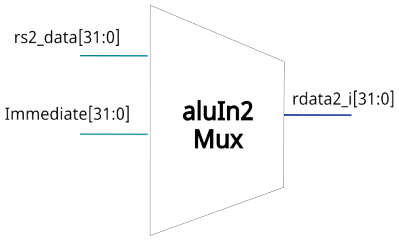
Instruction Memory(InstructionMemory_with_write) module shown in figure 3.1a gives us the instructions based on the Instruction address. I have designed this module suitable for writing its instruction byte-wise so that we can load it with required instructions using a peripheral protocol. **result write back mux** (rslt_Mux) in figure 3.4a is usefed to select the data, write it to the respective GPR's location based on the control signal coming from the control unit.

The Modules required to design for the types of instructions mentioned in the above table are as follows.



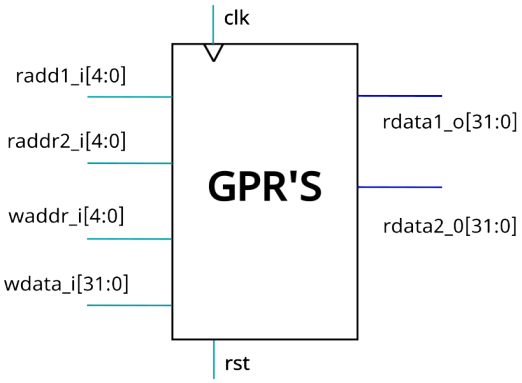
(a) Instruction Memory

(b) Data Memory

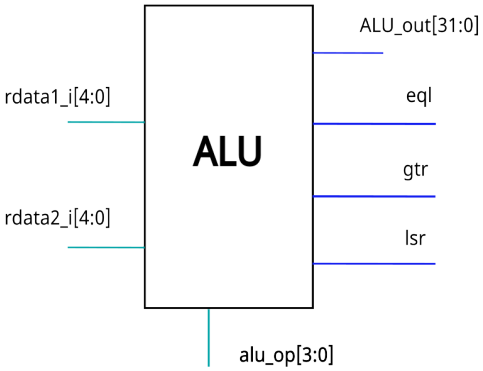


(a) ALU 2nd input mux

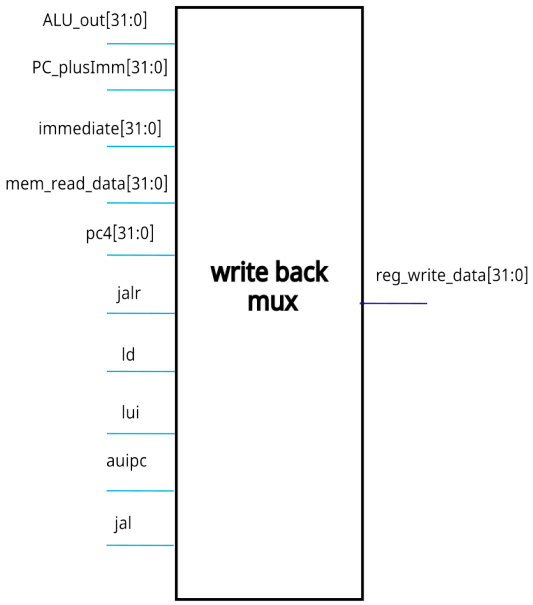
(b) Immediate Generator



(a) General Purpose Registers (GPRs)

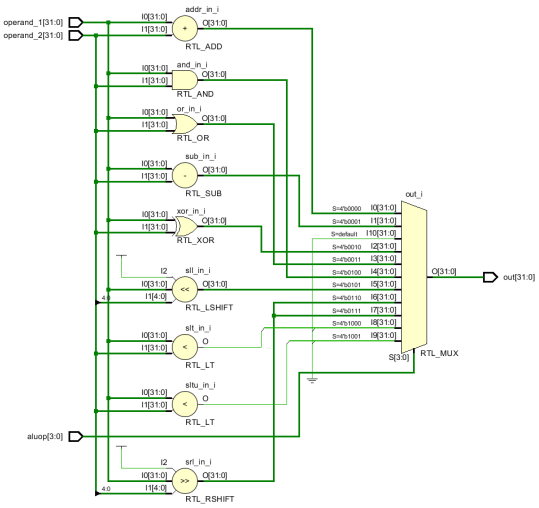


(b) Arithmetic Logic Unit (ALU)



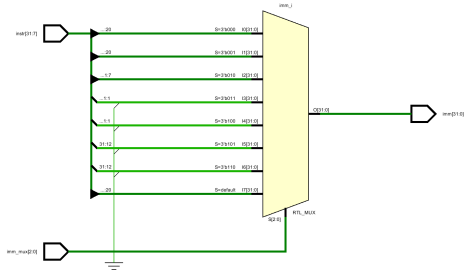
(a) Result Write Back Mux

Figure 3.4: Result Write Back Mux



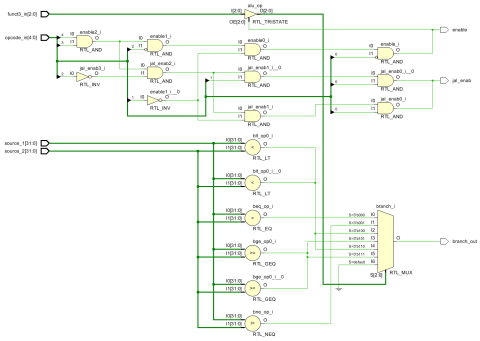
(a) ALU RISCV I/O planning

Figure 3.5: ALU RISCV I/O planning



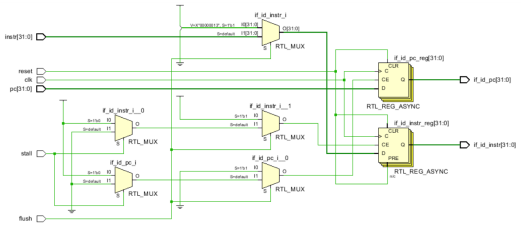
(b) IMMEDIATE GENERATION I/O planning

Figure 3.6: IMMEDIATE GENERATION I/O planning



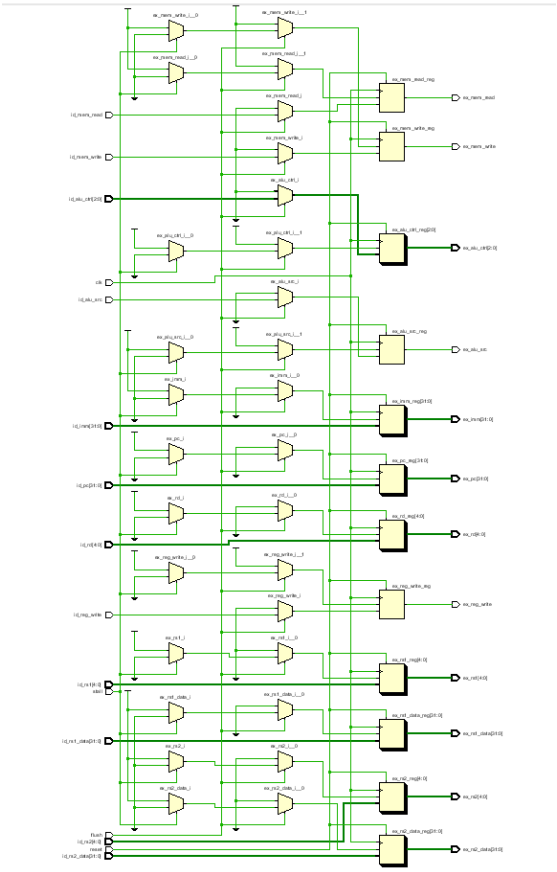
(c) Branch Unit I/O planning

Figure 3.7: Branch Unit I/O planning



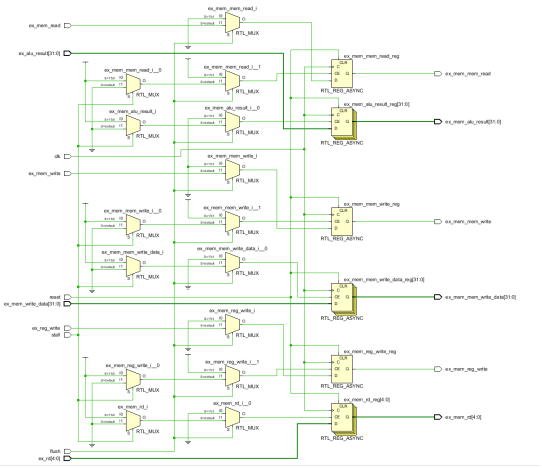
(d) if-id-pipeline I/O planning

Figure 3.8: if-id-pipeline I/O planning



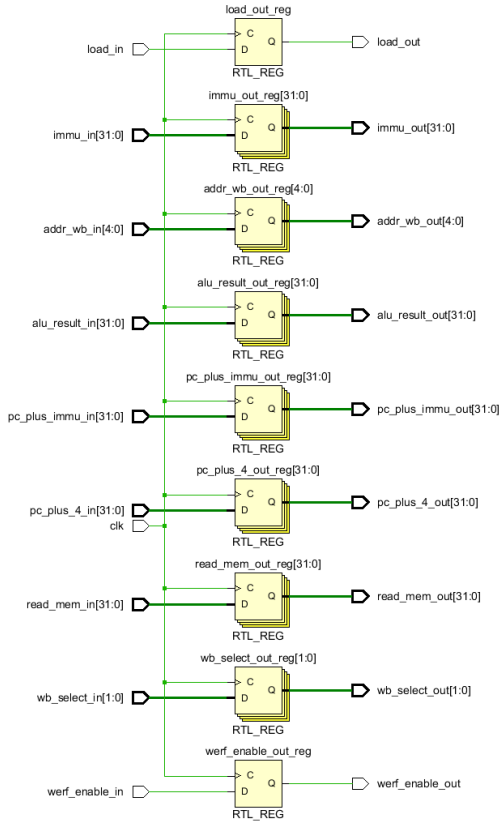
(e) id-ex-pipeline I/O planning

Figure 3.9: id-ex-pipeline I/O planning



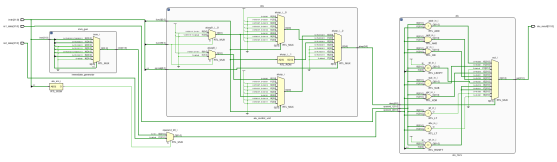
(f) ex-mem-pipeline I/O planning

Figure 3.10: ex-mem-pipeline I/O planning



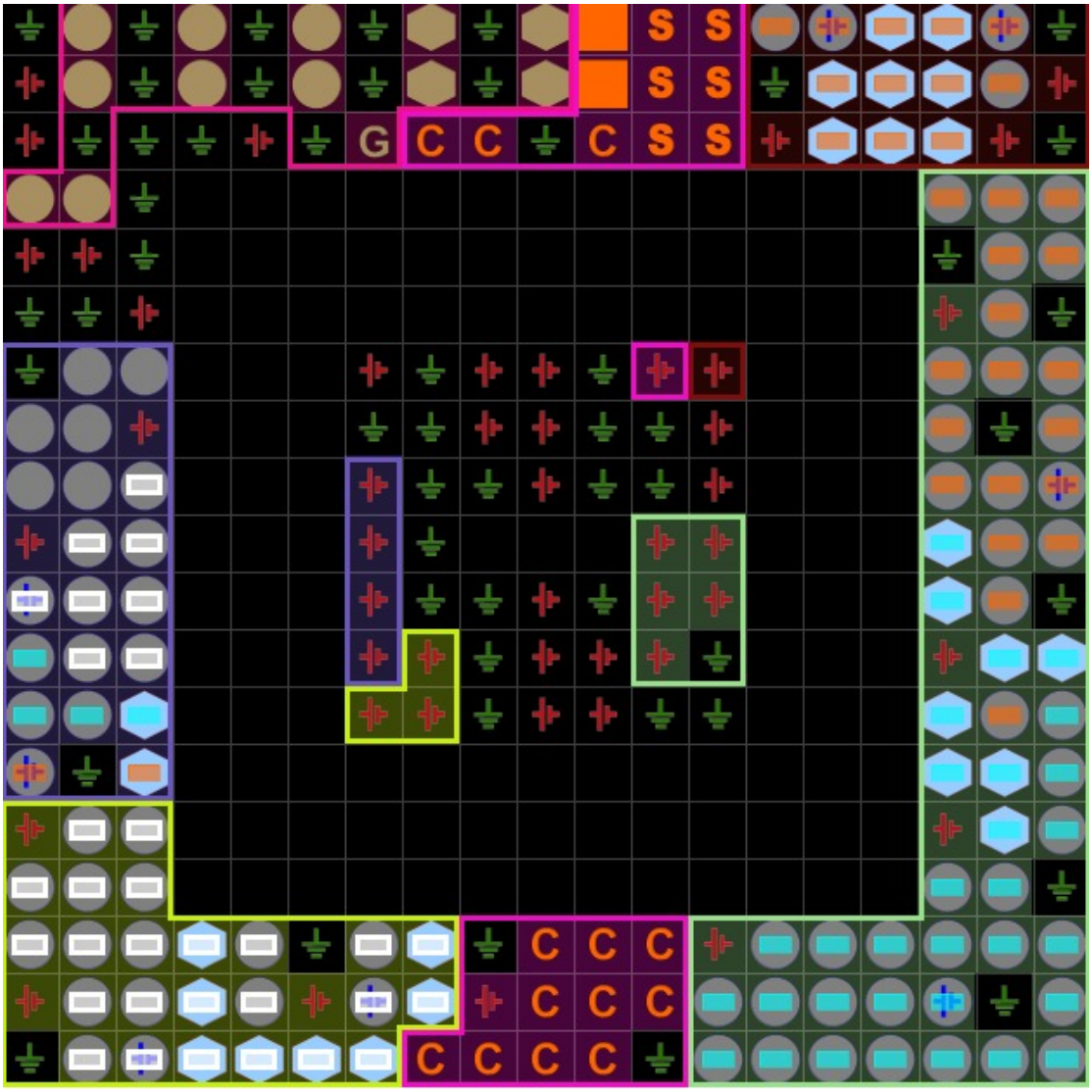
(g) wb-pipeline I/O planning

Figure 3.11: wb-pipeline I/O planning



(h) RV32I I/O planning

Figure 3.12: RV32I I/O planning



(I) ALU RISCV I/O planning of core

Figure 3.13: ALU RISCV I/O planning of core

3.2 Single Cycle Architecture

For Singel Cycle Design We can realize it in two ways ,In one way we have to use the asynchronous Instruction Memory ,only for an Positive edge trigerred GPRs , in a Positive edge triggered Program Counter, with a synchronous Data Memory. In another one , for a positve edge trigered program counter , along with asynchronous instruction memory and synchronous Data Memory , we have to use the negative trigerred GPRs.

The overall data Path is shown in the figure 3.14 The Respective Data Paths for each type of instruction are as below. The supportability of the data path for the all types instructions of RV32I mentioned in 2.2 are shown in the diagrams 3.15, 3.16,3.17,3.18,3.19,3.20, 3.21,3.22,3.23,

Data Paths

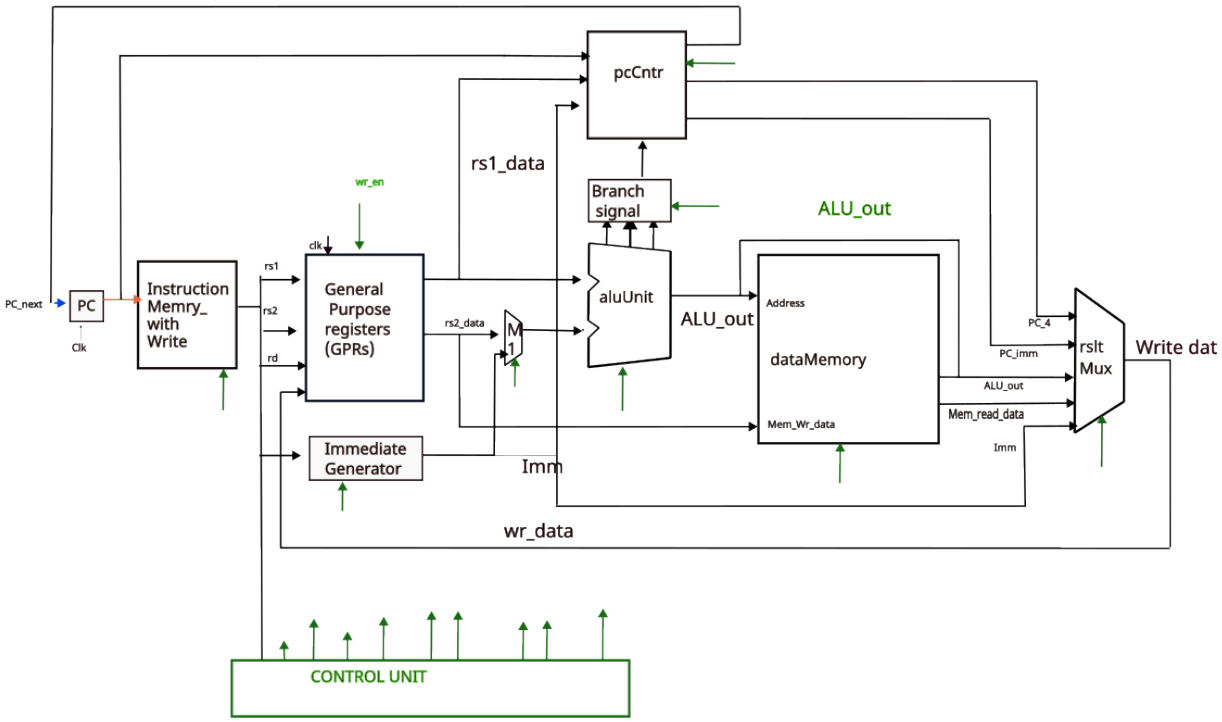


Figure 3.14: RV32I Data Path

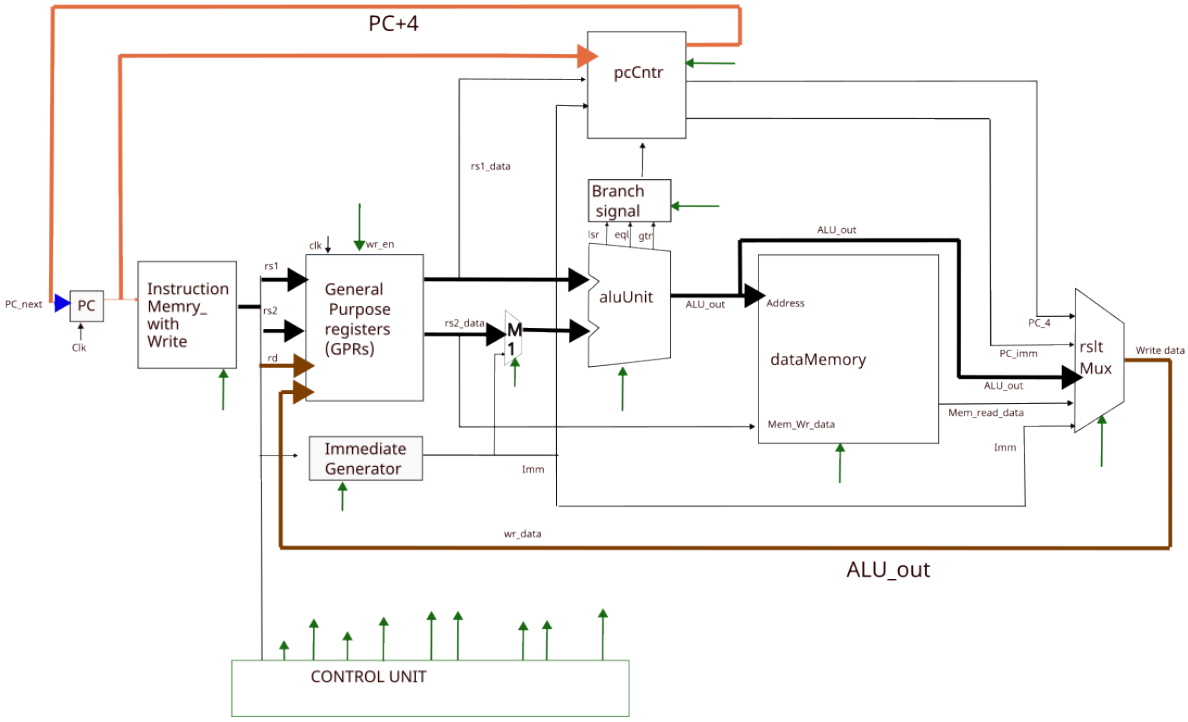


Figure 3.15: R type Instruction data Path

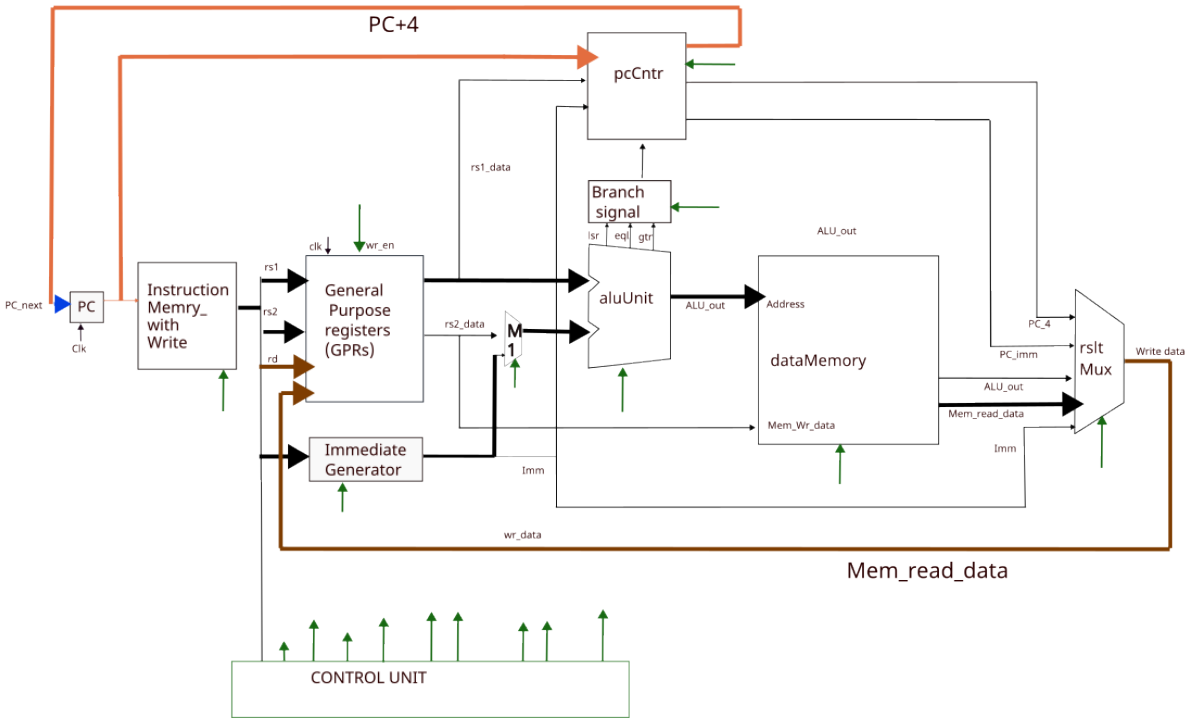


Figure 3.16: L type Instruction data Path

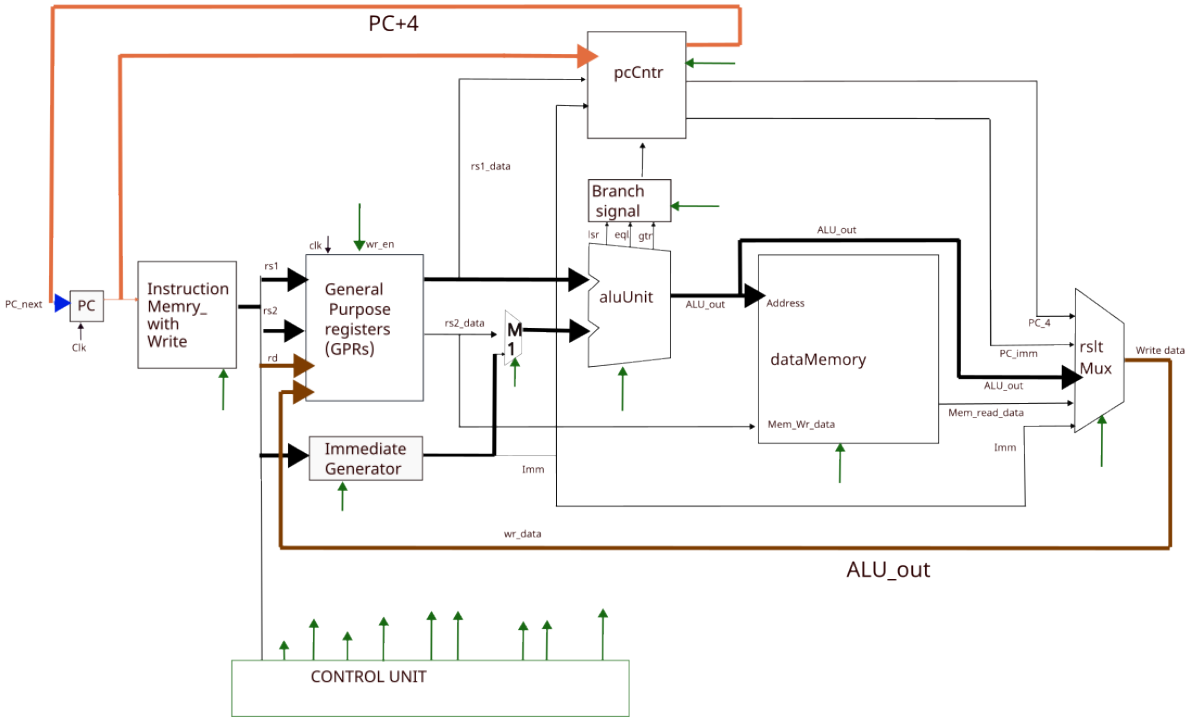


Figure 3.17: RI type Instruction data Path

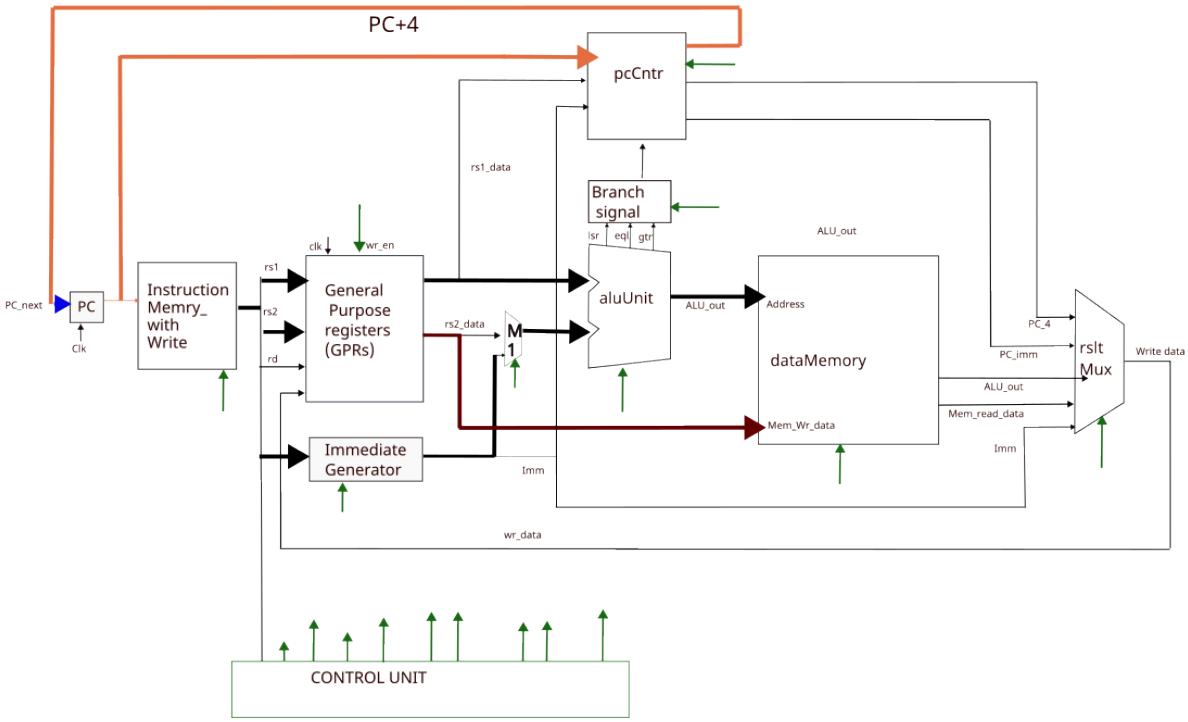


Figure 3.18: S type Instruction data Path

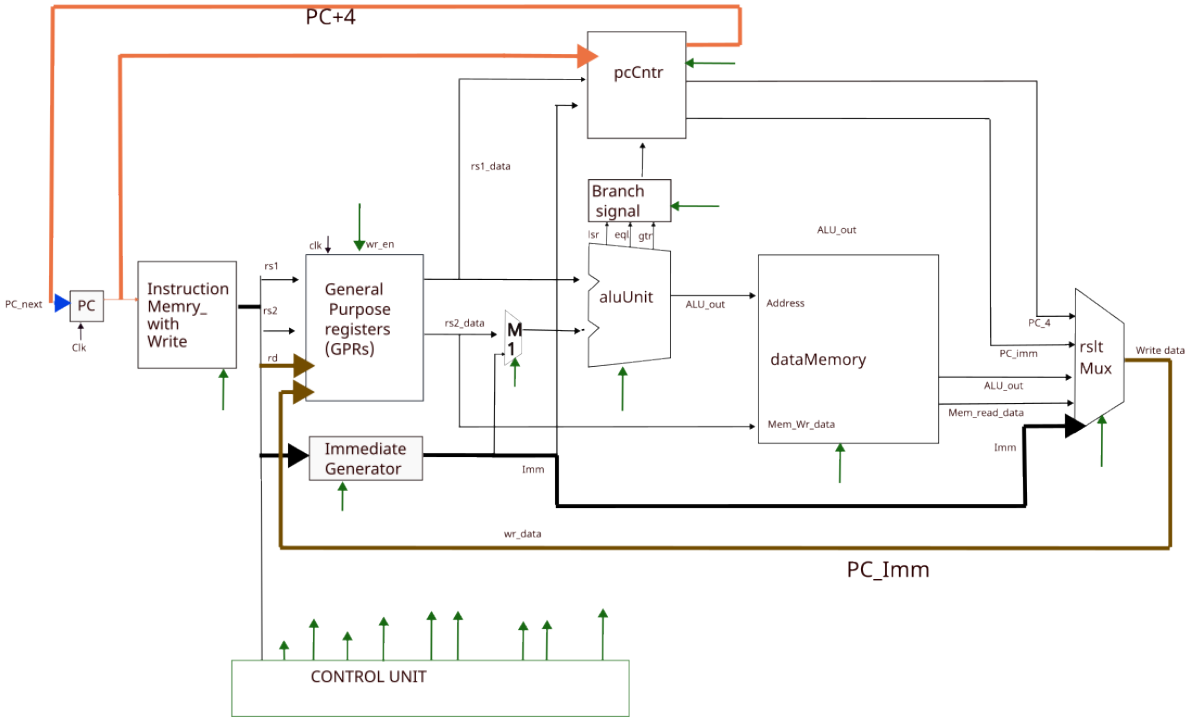


Figure 3.19: LUI type Instruction data Path

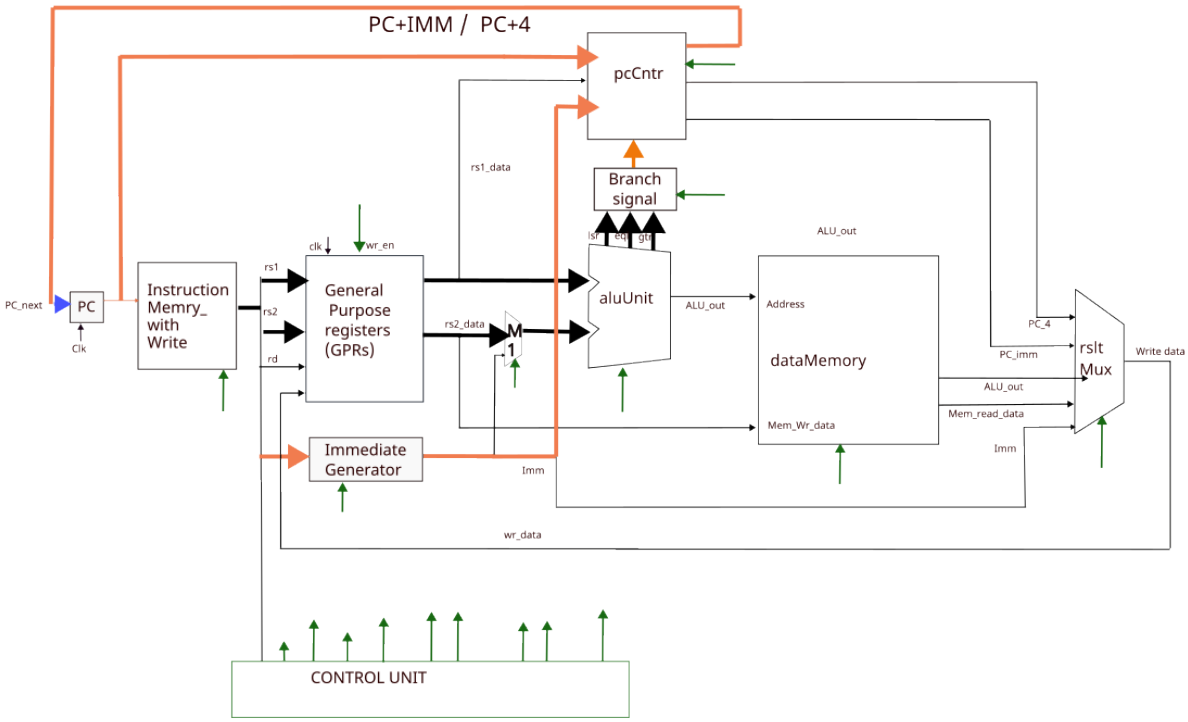


Figure 3.20: BR type Instruction data Path

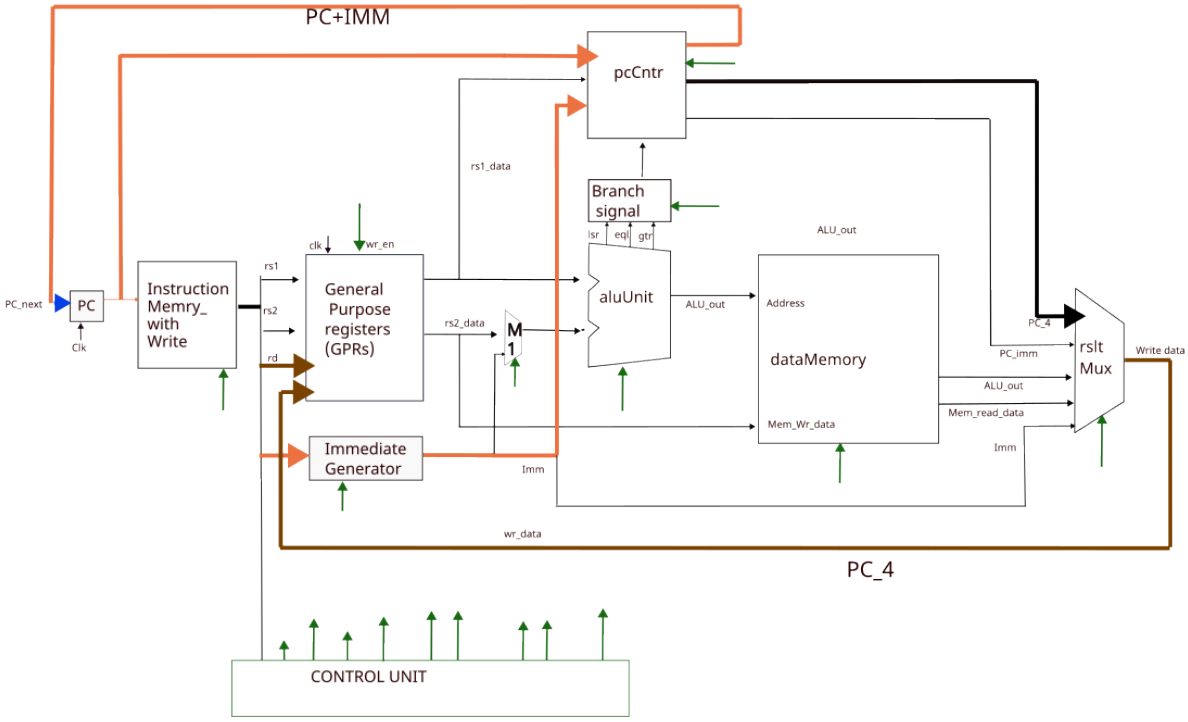


Figure 3.21: JAL type Instruction data Path

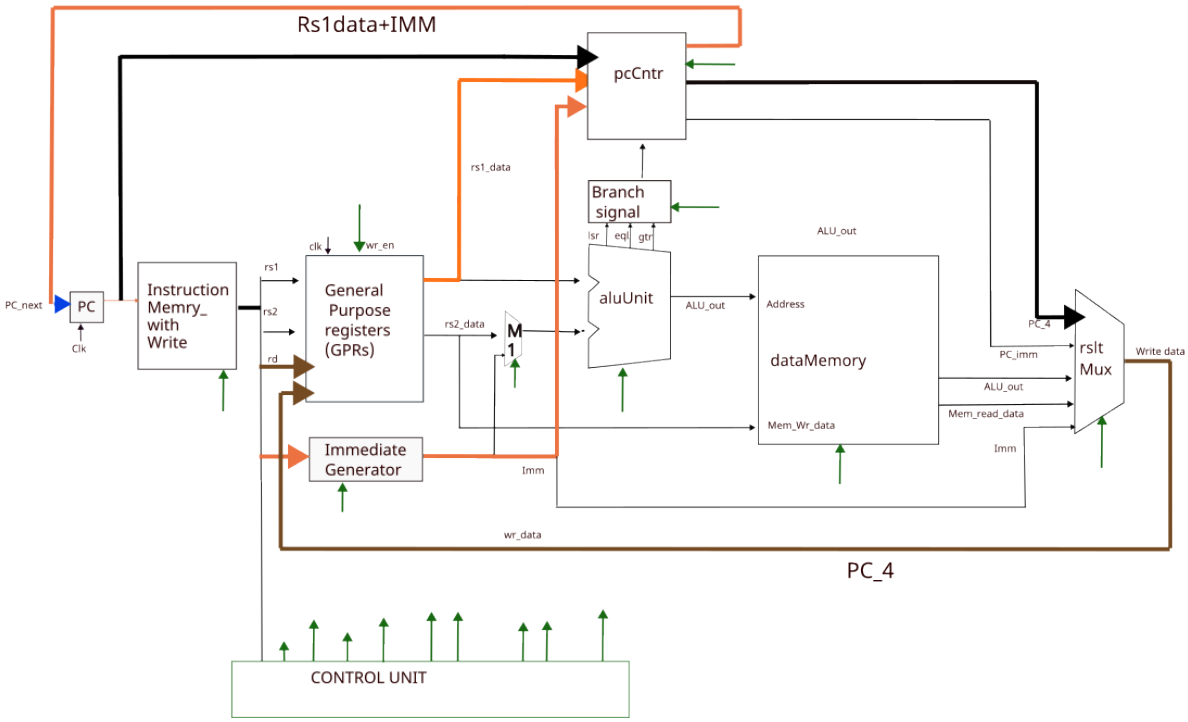


Figure 3.22: JARL type Instruction data Path

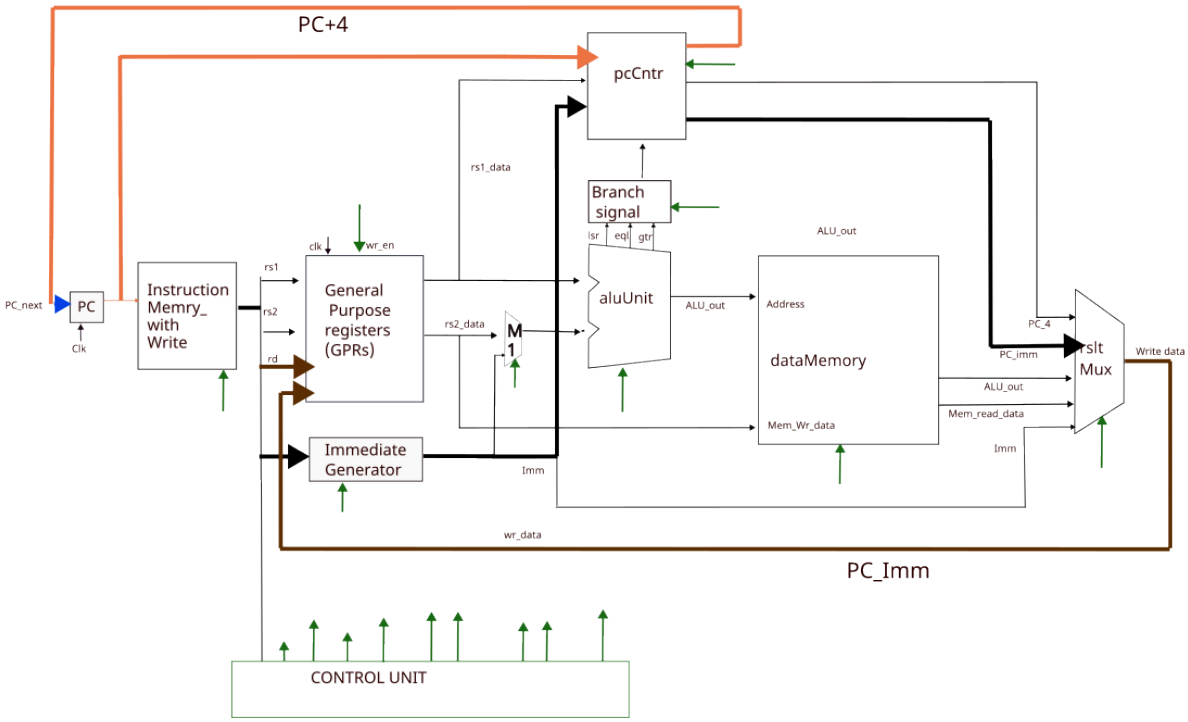


Figure 3.23: AUIPC type Instruction data Path

3.3 Adding M extension to RV32I

RISC-V architecture can incorporate acceleration capabilities, which makes it appropriate for a wide range of applications and sophisticated computing requirements'-V processors like the RV32IM, used as a Digital Signal Processor, that make tasks like AI processing, AI models execution more efficiently compared to the RV32I ISA.

M extensioo Instructions: Building upon the RV32I base, the RV32M Extension introduces specialized ALU instructions tailored for integer multiplication and division operations. This extension enhances the computational capabilities of RISC-V processors by incorporating instructions such as "MUL" "MULH" , "MULHSU" "MULHU", "DIV" and "REM" These instructions enable efficient handling of complex arithmetic tasks, particularly useful in applications requiring intensive mathematical Computation containing Multiplication and division.The Instructions are listed in Table 3.1.The specific operations of the instructions are mentioned in the Table 3.3 , 3.2

M extension Computation Module :As shown in the Figure 3.24 It is a extra hardware module required to attach it to the existing ALU of base isa, The The multiplication module used is Array Multiplier and Division the module used is based on the repetitive subtracting and comparing method.

The Datapath modification can be done by replacing the base ISA ALU with Modular ALU for this RV32IM. It is the combination of the base ALU and the M extension module along with some muxing at the output port as shown in the Figure 3.25.The multiplication module gives the 64bit output, which is split into MSB results and LSB results, in the same way the division the module also gives the 32-bit Remainder and quotient, so by using muxes, we can select the required result

The extra Module required to generate the required control signals for the extra M extension module is as shown in Figure 3.26. Its respective control signals are listed in tables 3.4 and 3.5 for Multiplication and Divisions respectively.

Instruction	Funct7	rs2	rs1	Funct3	rd	Opcode Extension
MUL	0000001	rs2	rs1	000	rd	0110011
MULH	0000001	rs2	rs1	001	rd	0110011
MULHSU	0000001	rs2	rs1	010	rd	0110011
MULHU	0000001	rs2	rs1	011	rd	0110011
DIV	0000001	rs2	rs1	100	rd	0110011
DIVU	0000001	rs2	rs1	101	rd	0110011
REM	0000001	rs2	rs1	110	rd	0110011
REMU	0000001	rs2	rs1	111	rd	0110011

Table 3.1: RV32M Standard Extension Instructions

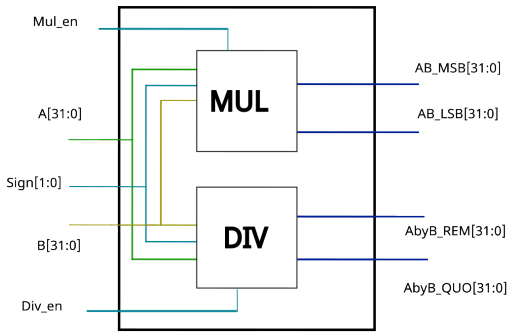


Figure 3.24: M extension Computation Module

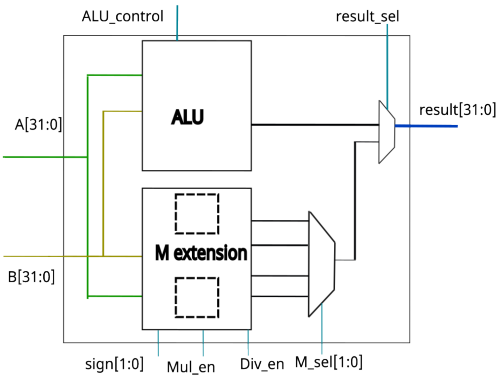


Figure 3.25: ALU Module for RV32IM

The Data Path required for the RV32IM is shown in the below figure3.27.

Instruction	(Funct3)	Rd_data	Data (32-bit)	
			Rs1 (Multiplicand)	Rs2 (Multiplicand)
MUL	000	LSB 32-bit result	Signed	Signed
MULH	001	MSB 32-bit result	Signed	Signed
MULHSU	010	MSB 32-bit result	Signed	Unsigned
MULHU	011	MSB 32-bit result	Unsigned	Unsigned

Table 3.2: Multiplication Instructions operation

Instruction	(Funct3)	Rd	Data (32-bit)	
			Rs1 (Dividend)	Rs2 (Divisor)
DIV	100	Quotient	Signed	Signed
DIVU	101	Quotient	Unsigned	Unsigned
REM	110	Remainder	Signed	Signed
REMU	111	Remainder	Unsigned	Unsigned

Table 3.3: Division Instructions Operation

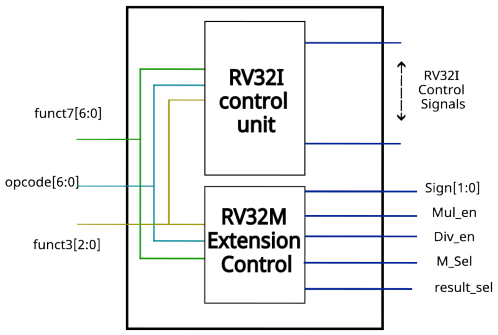


Figure 3.26: M extension Control Module

Instruction (Funct3)	Sign	Mul_en	Div_en	M_sel	result_
MUL (000)	11	1	0	10	1
MULH (001)	11	1	0	11	1
MULHSU (010)	10	1	0	11	1
MULHU (011)	00	1	0	11	1

Table 3.4: Multiplication Instructions Control signals

Instruction (Funct3)	Sign	Mul_en	Div_en	M_sel	result_
DIV (100)	11	0	1	00	1
DIVU (101)	00	0	1	00	1
REM (110)	11	0	1	01	1
REMU (111)	00	0	1	01	1

Table 3.5: Division Instructions Control signals

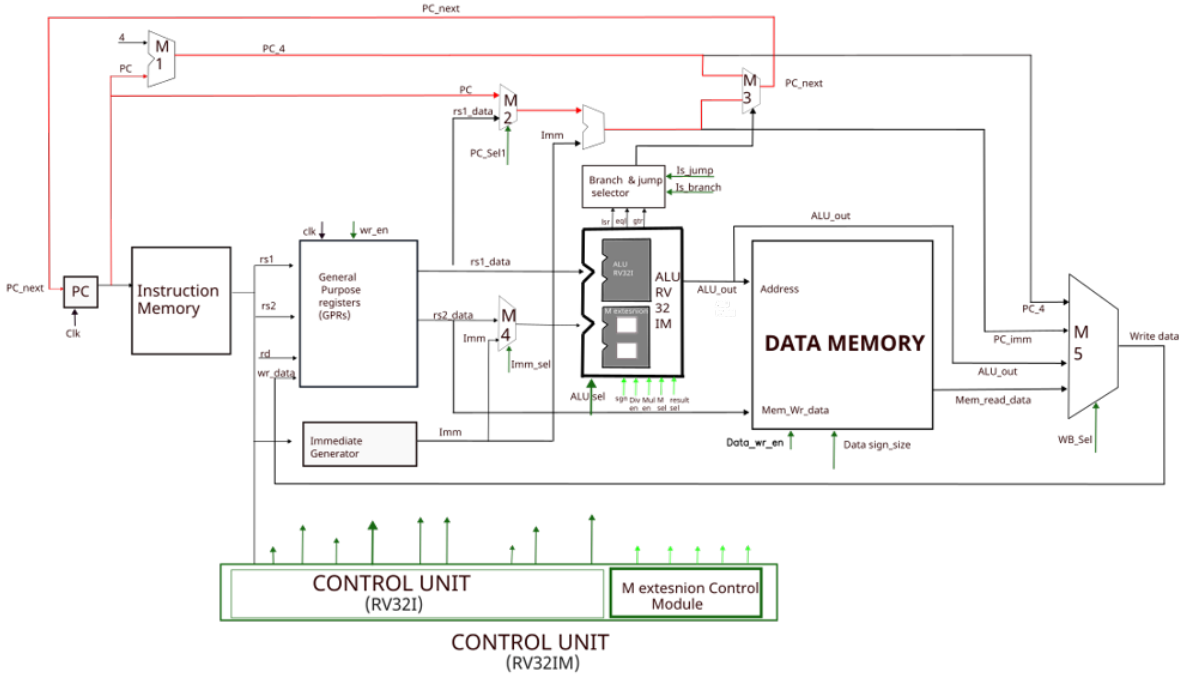


Figure 3.27: rv32im data path

3.4 Pipelining Design OF RV32IM

3.4.1 Designing

The whole process can be seen in five different stages: Instruction Fetch, Instruction Decode, Computation Execution, and Result Memory Write. We need to add a few buffers, such as IF_ID buffer, ID_EX buffer, EX_MEM buffer, and MEM_EX buffer, as shown in the 3.28, between each state to increase the clock frequency of our processor design.

Here, the Branch Detection is happening in the Decoding stage with a cost of a slight increase in delay, In this kind of design there is no need of a Branch predictor

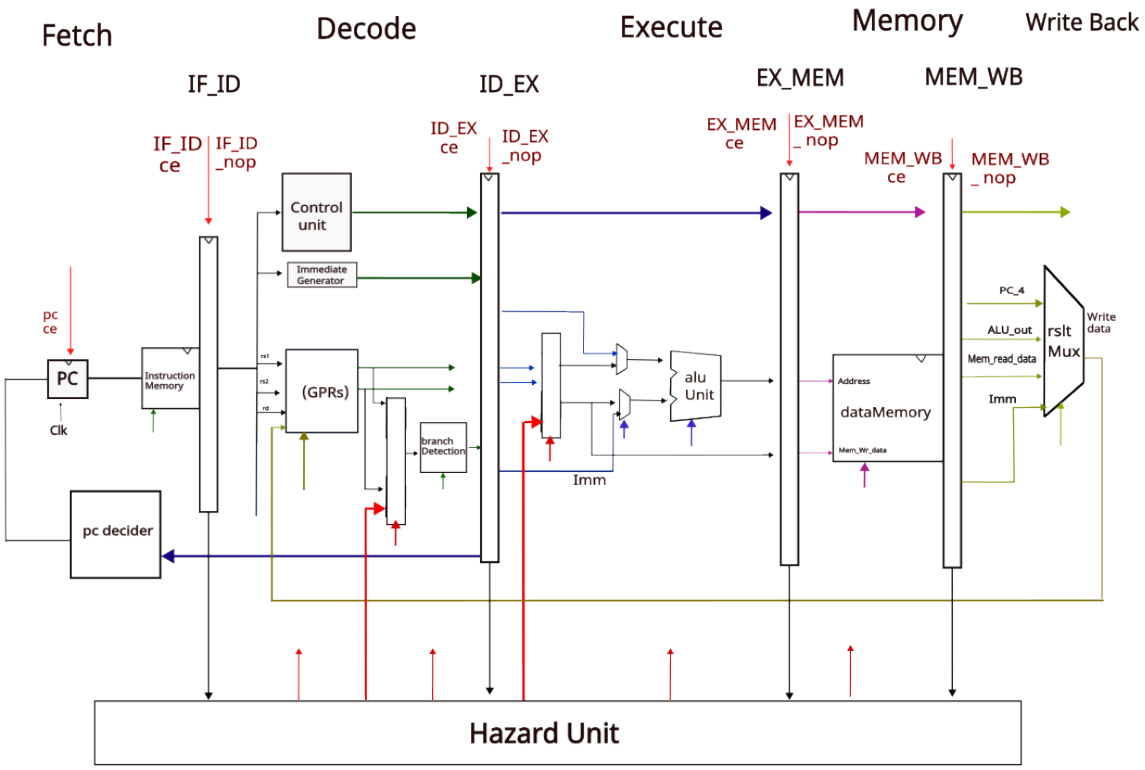


Figure 3.28: Pipelined Microarchitecture Design

To Design Pipelining Design We need to deal with Hazards that occur due to Instruction Dependencies on Data, Hardware and Control .

3.4.2 Hazards

Data Hazards occur when there are Data Dependencies between the instuctions. In the figure 3.29 , the outputs of the first instructions and the 2nd instruction are needed to be the inputs of the 3rd instructions , bu tin pipelingn design , the write back happens inn the last stage , the 3 rd instcutions which is supposed to load the required data from the GPRs , could load due to the **RAW**(Read after Write) Hazrd .

So one way of resolving this is using a Data forwarder in the execution stage, so that when a later instruction needs data computed by the former instructions then after the computation of the former instructions computation, those required data by the later instruction would be forwarded using a data forwarding unit as shown in the figure.the required control signals for the data forwarding unit would come form the hazard unit

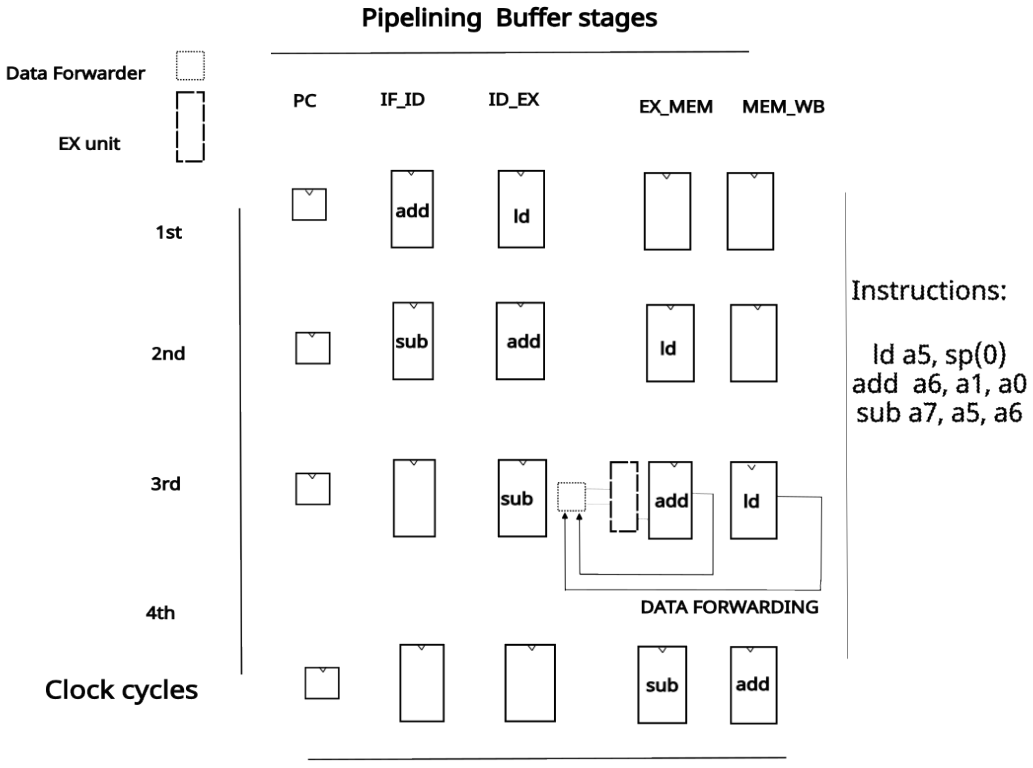


Figure 3.29: Data Hazard Handling

Control Hazard occurs due to the branch and jump instructions, which are used to call instructions of father addresses. In the figure 3.30 if we see in the instruction decode stage,we will know that the type of instructions is jump. at that time there would be the following instruction **in1, in2** in the pipeline, so in the next clock cycle,the instruction data in the buffers IF_ID, ID_EX would be set to zero as**nop**(no operation) and the

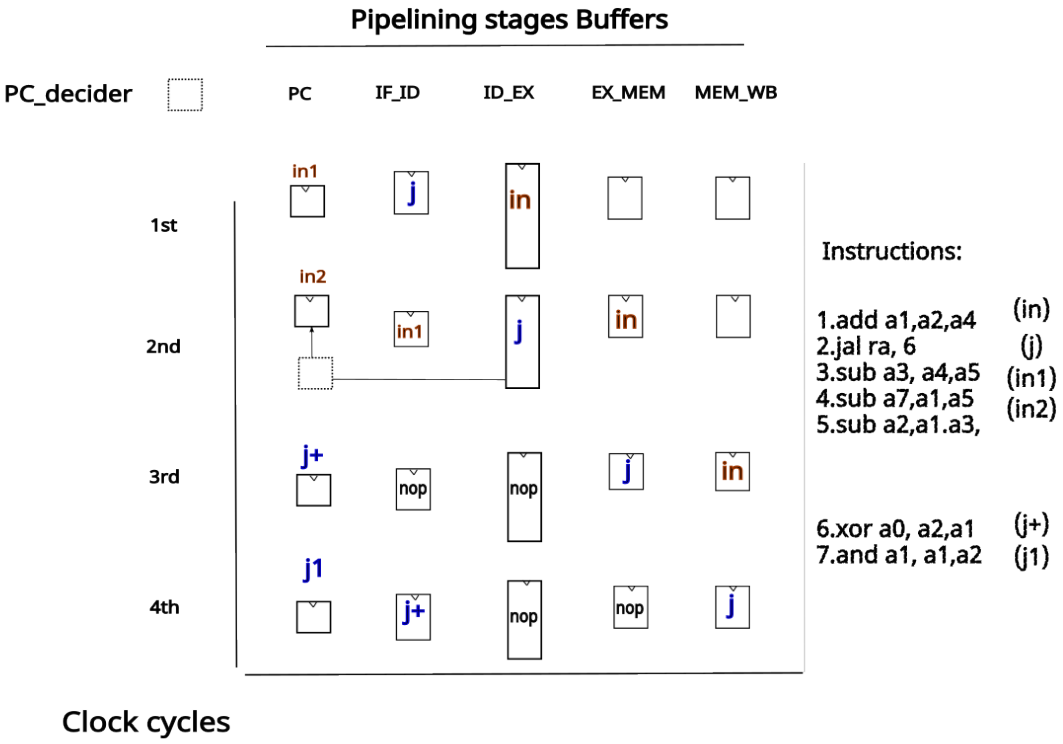


Figure 3.30: Control Hazard Handling

instruction that the jump instructions is directing to would be stored in the program counter. meanwhile, the **in** instruction would pass through the pipeline normally

CHAPTER 4

VERIFICATION METHODOLOGY

I have used the assembly codes having different test cases for individual instructions from this githuib source riscv_tests. then converted them to hexadecimal data using the toolchain and placed those instructions into the processor instruction memory, in vivado,then verified the Processor functionality to support all instructions.

4.1 Softwares used

RISC-V_gnu_toolchain It is used to convert the C/C++ code to the RISC-V instructions according to our required ISA, along with extension Instructions if we require them .

Xilinx Vivado :The whole Designing of single cycle and pipelined processor is designed and simulated, synthesized using the Vivado 2022.2 version

4.2 Simulation Based Verification

Below are the screenshots of waveforms of the processor signals when control and data hazards occurs. The waveform 4.2 is a simulation of the data-dependent consecutive instructions highlighted in the fuigure4.4,when they occur the FOrwding unit forwards the data from the pipelined available data to the execution unit .The forwarding unit forwards a)Data from the Memory available in the Write back stage,b)ALU computation data available in the Memory stage c) computed ALU output data writing back to the GPR's in the Write back stage.In the same way, the waveform 3.30 is the simulation of the few instructions, which has jump instructions in it as highlighted in the figure 4.3. When a jump instruction is detected in the execution stage, then the execution Execution stage and the decoding stage will not operation(nop-all data is reset to zero) in the the next clock cycle. After another clock cycle, the instructions at the jump address come to the decoding stage.

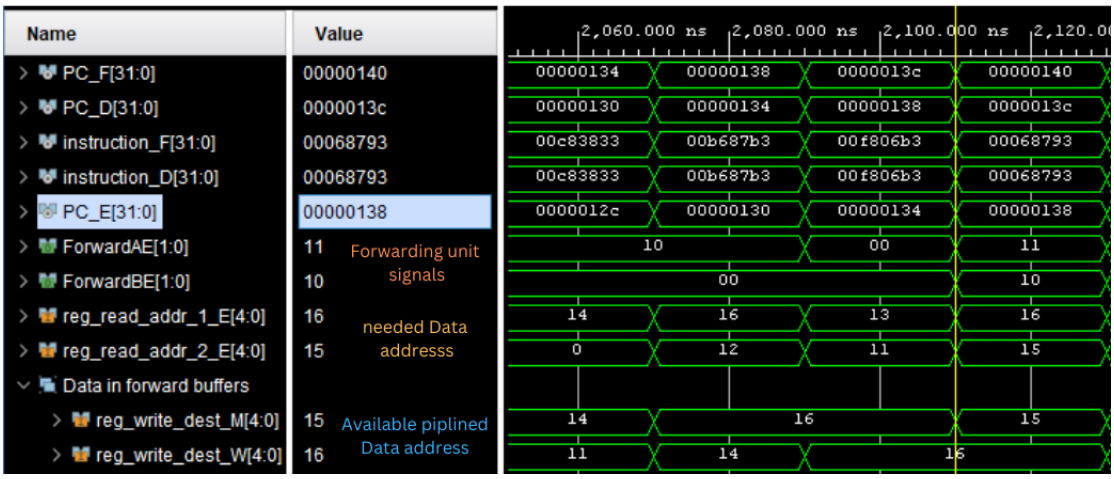


Figure 4.1: Data Hazards

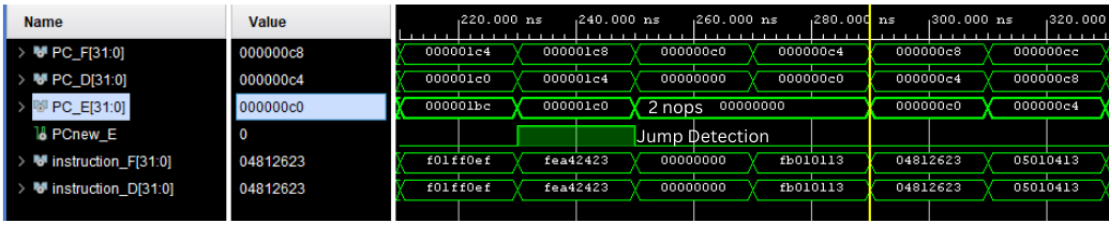


Figure 4.2: Control Hazards

```
000001ac <main>:
 1ac:    addi sp,sp,-32
 1b0:    sw ra,28(sp)
 1b4:    sw s0,24(sp)
 1b8:    addi s0,sp,32
 1bc:    addi a0,zero,25
 1c0:    jal  ra,c0 <fibonacci_sequence>
 1c4:    sw a0,-24(s0)
 1c8:    sw a1,-20(s0)

000000c0 <fibonacci_sequence>:
  c0:    addi sp,sp,-80
  c4:    sw s0,76(sp)
  c8:    addi s0,sp,80
  cc:    sw a0,-68(s0)
  d0:    addi a5,zero,0
```

Figure 4.3: Instructions Causing Control Hazard

```
12c:    addi  a6,a4,0
130:    sltu  a6,a6,a2
134:    add a5,a3,a1
138:    add a3,a6,a5
13c:    addi  a5,a3,0
140:    sw  a4,-40(s0)
144:    sw  a5,-36(s0)
148:    lw  a4,-32(s0)
```

Figure 4.4: Instructions Causing Data Hazards

4.2.1 Arithmetic operation programe

The code in figure 4.5 , has multiplication ,Division, addition , subtraction operations on the given data, The final output of the Computation is 70 , which can seen in the the simulation

```cpp
#include <iostream>

using namespace std;

int main() {
    int num1 = -20, num2 = 10, num3 = -5;
    // Input numbers -20  10 -5
    int result_mult, result_div,result_add,result_sub;

    // Perform multiplication
    result_mult = num1 * num2;        //-200

    // Perform division
    result_div = result_mult / num3;  //40
    //perform addition
    result_add = result_div+num2;     //50
    //perform subtraction
    result_sub = result_add-num1;     //70

    return result_sub;
}
```

Figure 4.5: Arthamatic operations c++ code

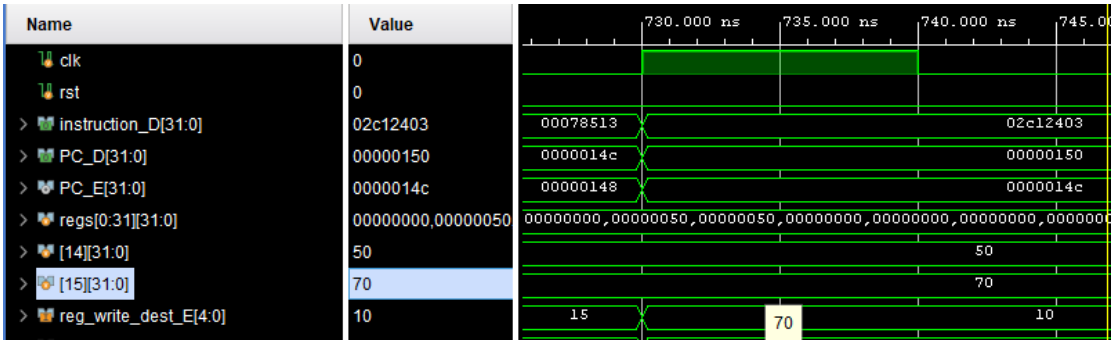


Figure 4.6: Arthamatic operations output simulation

4.2.2 Fibonacci program

The simulation show inn the figure 4.8 is obtained from c code written to compute the 25th number in the Fibanachi series ,which is compiled using the risc-v gnu toolchain, then extracted the hexadecimal instructions from it , and then simulated on the Designed Pipelined processor.

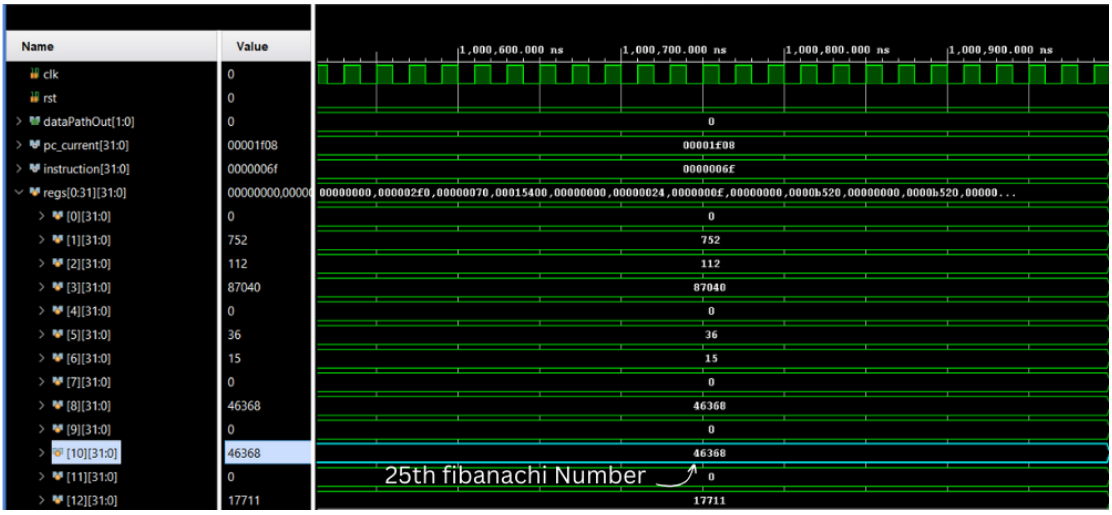


Figure 4.7: 25th Fibanachi number calculation

4.3 FPGA Prototyping and Testing with Arty A7 on RISC-V Core

The FPGA-based verification of the RISC-V (RV32IM) pipelined processor core was performed on the Xilinx Arty A7-100T FPGA platform, utilizing its reconfigurable fabric to validate the complete RTL-to-GDSII flow. The Arty A7, featuring the Xilinx Artix-7 XC7A100T FPGA, offers 101,440 logic cells, 240 DSP slices, and 4,860 Kb of block RAM (BRAM), making it well-suited for deploying the custom 5-stage pipelined RISC-V processor with RV32IM support (base integer, multiplication/division extensions).

The processor design supported instruction fetch, decode, execute, memory access, and write-back stages, and included a hazard detection unit, forwarding logic, and branch prediction mechanism. Verification involved executing bare-metal test programs and compiled RISC-V binaries, including system calls and arithmetic benchmarks, transferred to the FPGA via a UART interface and stored in DDR3 memory.

A custom Python-based loader transmitted the binaries and interfaced with the UART to monitor processor outputs, including program counter traces, memory accesses, and register file changes. The RV core correctly executed a suite of compliance and stress tests, with successful handling of load-store operations, arithmetic instructions, CSR accesses, and interrupt handling (via a mock RTOS scheduler). The processor achieved a maximum clock frequency of 75 MHz, with an average execution latency of 12.5 cycles per instruction across test workloads.

Resource utilization on the FPGA reached 68% LUTs, 64% flip-flops, and 72% DSP slices, with an average power consumption of 1.8W. Challenges such as control hazards and pipeline stalls due to memory access latency were addressed through pipelining optimizations and a basic cache controller. Timing closure was achieved using floorplanning and register retiming strategies.

4.3.1 Arty A7 FPGA Platform Specifications

The Xilinx Arty A7-100T development board was selected for its balance of computational resources, energy efficiency, and peripheral support:

The Xilinx Arty A7-100T development board was selected for its balanced computational resources and peripheral support. Key specifications include:

- Device: Artix-7 XC7A100T-1CSG324C
- Logic Cells: 101,440
- DSP Slices: 240 (arithmetic acceleration)
- Block RAM (BRAM): 4,860 Kb (CNN weights/image buffers)
- Clock: 100 MHz default system clock (450 MHz maximum)

Memory Subsystem:

- DDR3L SDRAM: 256 MB (large dataset storage)
- Non-volatile Storage: 128 Mb QSPI Flash (bitstream storage)

I/O Capabilities:

- 16 PMOD expansion interfaces (camera/display connectivity)
- USB-UART bridge (host communication)
- 4 user-programmable LEDs (status indication)

Physical Characteristics:

- Package: CSG324
- Power Consumption: <3 W (typical operation)

Note: Specifications comply with Xilinx Artix-7 Technical Reference Manual (DS181) and Digilent documentation.

4.3.2 FPGA based verification

A C/C++ program written in such a way that it would return the result , using the risc-v gnu toolchain is converted into a text file containing the assembly and hexadecimal instructions data . Now, a Python

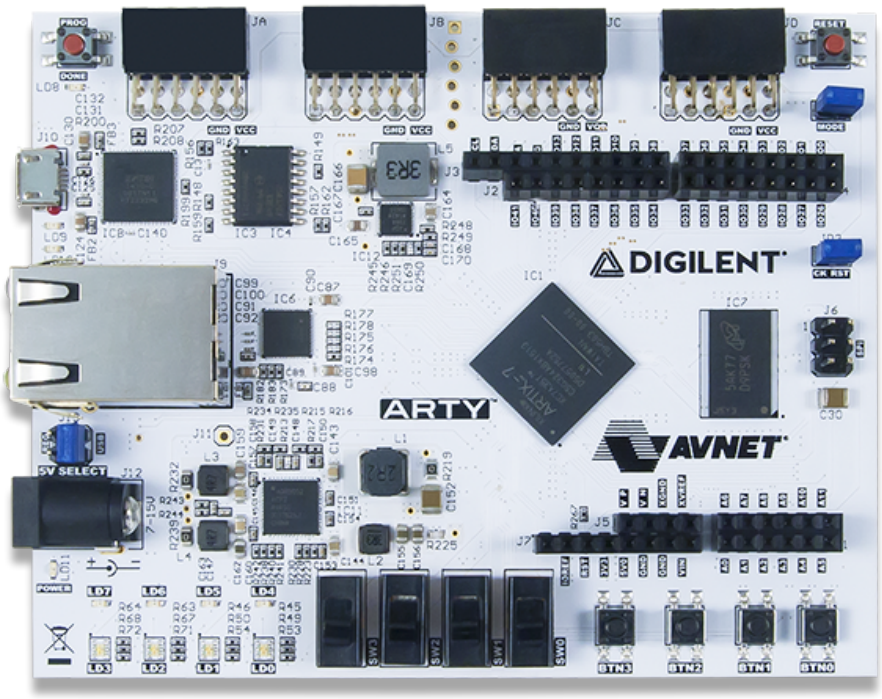


Figure 4.8: Arty A7 FPGA Board used to carry out the entire experiment

script is used to extract the hexadecimal instructions data and write them in a new text file.Now using a UART transmission protocol Python, the hexadecimal data is sent to the FPGA implemented process. Now after enabling the start push button in the processor to enable it to run the program instructions loaded in it. After a After executing all the instructions on the board, We can send the data memory contents to our laptop while running a Python script for UART receiving ptotocal in our laptop. Now, we can see the contents of the data memory using a text file.

Based on our program returning value, we can check whether that computed data is there or not in the data memory contents.

VERIFICATION METHODOLOGY:

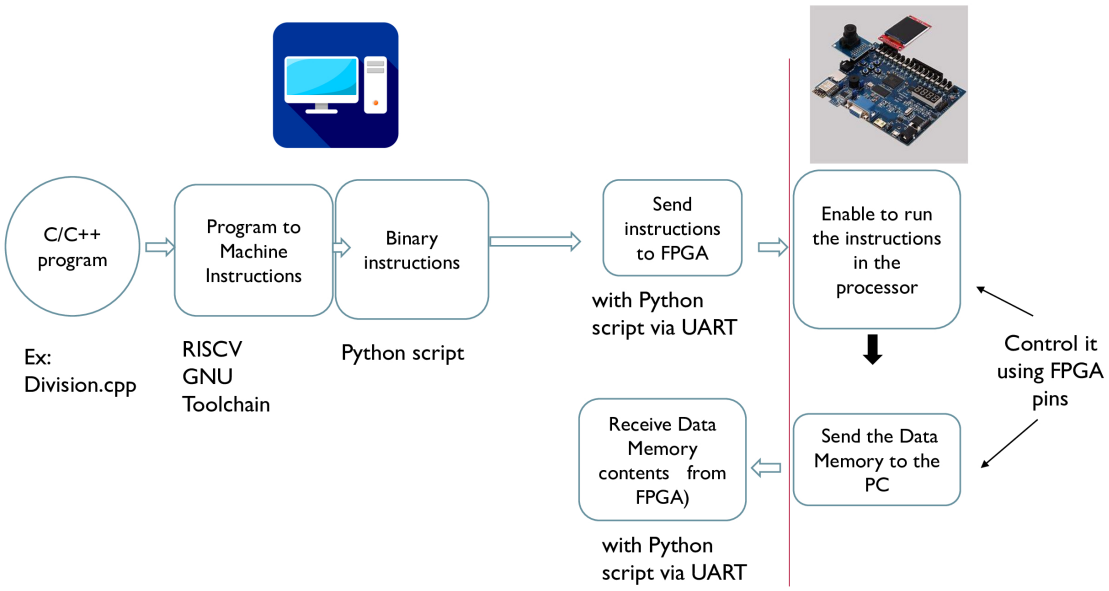


Figure 4.9: FPGA Verification Flow

CHAPTER 5

RESULTS AND DISCUSSION

I have successfully Designed and run the programs on the RV32IM-based pipelined RISC-V processor. Below are the results of in terms of delay

Singel Cycle	Pipelined
90.082ns	59.525ns

Table 5.1: Delay results of RV32IM single cycle vs Piplined Designs

The reason for nearly only half reduction in the delay even in the five stage pipelined design is the Major delay contributor is ALU unit with MUltiplicaiton and DIvision Support whose delay is nearly equal to the pipelined delay which we got, so due to the larger delay contributor in execution stage , we can only see 2 fold delay reduction , but not 5 fold

Processsor	LUT	LUTRAM	BRAM
Single Cycle	8336	5168	0
Pipelined	5431	1855	17

Table 5.2: FPGA resource utlization

The reason for the No BRAM utilization in the single cycle, is due to lack of asynchronous reading memory modules in the design, where BRAM would only be utilized for synchronous memory read and writes. Meanwhile n pipelined design , the synchronous read can be used

CHAPTER 6

Summary and Conclusion

This thesis presents the comprehensive design, implementation, and verification of a custom **RISC-V RV32I pipelined processor** and its integration with a lightweight Real-Time Operating System (RTOS), culminating in FPGA-based validation and deployment.

6.1 Summary

The project began with the architectural development of the **RV32I single-core processor** using a standard five-stage pipeline—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-back (WB). The pipeline was enhanced with data forwarding, hazard detection, and basic branch prediction techniques to manage data, control, and structural hazards.

The core supports the RV32I base instruction set with planned extensions including M (Multiply/Divide), A (Atomic), and Zicsr/Zifencei for control and synchronization. The core operates across **privileged modes** (User and Machine mode) as governed by RISC-V CSRs.

The processor's datapath integrates essential components such as the ALU, register file, memory controller, control logic, and pipeline registers. The processor was developed in Verilog and simulated using Verilator and ModelSim.

6.1.1 RTOS Integration

An RTOS was implemented atop the RV32I core, incorporating core real-time features:

- **Context Switching** via software interrupts using CSR instructions and `mret`.
- **Scheduling** through preemptive round-robin or priority-based mechanisms.
- **System Tick Timer** using `MTIME` and `MTIMECMP`:

```
#define MTIME  (*(volatile uint64_t*)0x200BFF8)
#define MTIMECMP (*(volatile uint64_t*)0x2004000)
#define TIMER_FREQ 1000000
void set_systick(uint64_t interval) {
   MTIMECMP = MTIME + interval;
}
```

- **Task Scheduler**:

```
void scheduler() {
   current_task = (current_task + 1) % NUM_TASKS;
   context_switch(tasks[current_task]);
}
```

- **Interrupt Handling** using a PLIC-based model:

```
void external_interrupt_handler() {
   uint32_t irq = PLIC_CLAIM;
   if (irq == UART_IRQ) {
      uart_handle_irq();
   }
   PLIC_COMPLETE = irq;
}
```

- **Peripheral Drivers** for UART and GPIO:

```
void uart_write(char c) {
   while (!(UART_STATUS & TX_READY));
   UART_DATA = c;
}

void gpio_write(uint32_t pin, uint8_t value) {
   if (value)
      GPIO_SET = (1 << pin);
   else
      GPIO_CLEAR = (1 << pin);
}
```

6.1.2 FPGA Verification and Sapphire SoC

The custom core, named **Sapphire**, was implemented on FPGA platforms such as Artix-7 (Arty A7) and Genesys-2, and verified via both simulation and hardware debugging tools.

Key features of Sapphire SoC:

- RV32I-compliant 5-stage pipeline.
- AXI4-Lite interconnect for memory-mapped I/O.
- Memory Map:
  A Hello World application was successfully executed:

```
#include "sapphire.h"
int main() {
  uart_init(115200);
  uart_puts("Sapphire SoC Booted!\n");
  while(1) {
    led_toggle();
    delay_ms(500);
  }
  return 0;
}
```

6.1.3 Performance Comparison

Major contributions include:

- Custom Verilog-based RV32I processor with 5-stage pipeline.
- RTOS integration using context switching and MTIME scheduling.
- AXI4-lite-based SoC design with memory-mapped peripherals.
- Verification via QEMU, Verilator, and Vivado on Artix-7.

Future Work:

- Extend to RV64GC with multi-core support.
- Integrate formal verification using Symbiyosys or JasperGold.
- Enhance cache hierarchy and introduce dynamic memory.
- Port full FreeRTOS for broader task management support.

Address Range	Description
0x00000000–0x0000FFFF	Boot ROM (64KB)
0x20000000–0x20000FFF	GPIO
0x30000000–0x300000FF	UART
0x40000000–0x4FFFFFFF	AXI4-Lite Memory

Table 6.1: Sapphire SoC Memory Map

- PLIC with 32 interrupt priority levels.

Metric	Sapphire	SHAKTI C-Class	PicoRV32
ISA Support	RV32I	RV64IMAC	RV32I
Pipeline Stages	5	3	-
FPGA Freq (MHz)	75	100	150
LUT Utilization	1,200	2,500	750
Verification	UVM + FPGA	Formal	Direct

Table 6.2: Performance Comparison

6.2 Conclusion

This project demonstrates a complete RTL-to-RTOS hardware-software co-design workflow based on the open-source RISC-V ISA. The work validates that a minimal RV32I core with pipelined architecture can support RTOS functionalities such as task switching, interrupt handling, and real-time scheduling, and be deployed on FPGA hardware for embedded applications.

This thesis serves as a foundational effort in RISC-V based real-time embedded systems and paves the way for scalable and customizable SoC design.

REFERENCES

1. **S. L. H. M. Harris**, *Digital_design_and_computer_Architecture_RISCV_editio*. ELSEVIER, .

2. **NEO32_Processor** (). Neorv32_processor. https://github.com/stnolting/neorv32.

3. **r1** (). https://drive.google.com/drive/u/0/home.

4. **RISC-V_gnu_toolchain** (). Risc-v_gnu_toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain.

5. **riscv_tests** (). riscv tests for individual instructions. https://github.com/AngeloJacobo/RISC-V/tree/main/test/extra.

6. **A. Waterman** (2017). The risc-v instruction set manual: Volume i user-level isa version 2.2. https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

7. **A. Waterman** and **K. Asanovic** (2019). Unprivileged specification version 20191213. https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

APPENDIX A

Processor ISA Types

A.0.1 Background Intro

Generally, a processor performs a specific computation on the required data. Any computation we require is communicated to the processor through INSTRUCTIONS, which the processor supports. The Only language a processor speaks is in terms of instructions. The only language that a processor can understand is its instruction language.

Any programming language that we have to translate(complied exactly)into the instructions of that processor on the PC.

a) Stack-based ISA: Data comes from Memory and forms a data stack. Like a LIFO (Last In, First Out) way, elements added last are the ones removed first. The top elements of the stack are used as operands for arithmetic or logical operations.
Here, with a type of instruction (Push A, Push B), the data that comes from Memory is stored temporarily as a data stack. Then, computation is performed with an instruction (Add) on the data stack, and finally, the output is sent to the memory with an instruction (Pop C).
Example:Java Virtual Machines work on stack based ISA

a) Accumulator-based ISA: An accumulator-based ISA involves arithmetic and logic operations using the accumulator as one of the operands and storing the result back in the accumulator.
Here, the computation happens between the data from direct memory and data in the accumulator with an instruction (Add B) after computation, the result would occur in the accumulator itself, and then finally, the result in the accumulator could be stored back in memory with an instruction(Store C).
Example: The Intel 8085 microcontroller is on accumulator-based ISA.

a) Register-to-memory ISA: In a register-to-memory ISA, data is transferred between registers and memory. Instructions in this type of ISA typically involve loading data from memory into registers or storing data from registers into memory.

   For example, For a computation mentioned in the Figure, operand one is temporarily stored in registers initially with an instruction(Load R1, A), and then the computation is performed with an instruction(Add R1, B) on operand one from registers and operand two from Memory. The output is stored in a register only, and then, finally, the result in the register could be stored back in memory with an instruction (Store R1, C).
   Example: x86 architecture.is based on reg to memory ISA

a) Register-to-register(Load/Store) ISA: In a register-to-register ISA, operations are performed directly between registers without involving memory. Initially, instructions load the required data from the memory to the registers and manipulate data within registers, then store it back with instructions. In the figure, the initial instructions(Load R1, A, Load R2, B) loaded the required data from memory to the registers, and then computation was performed on the date in the respective registers with an instruction(Add R3, R2, R1), then the output is stored in the register R3. Finally, the result is stored in the memory via an instruction(Store R3, C).
   Example: RISC-V,MIPS, and ARM architectures are based on reg-to-reg ISA

Each type of ISA has its own characteristics and influences the design and behaviour of the processor it governs. The selection of ISA type affects the processor architecture's complexity, speed, and capabilities.

APPENDIX B

Data Path Modules Description and functionalities

Port	**Description**
clk	Clock signal
InstrWrAdd	Instruction write address
InstrWrData	Instruction write data (byte)
InstrWrEn	Instruction write enable
pc	Program counter
instruction	Instruction output
start	Start control signal
InstrWrEn (Control)	Instruction write enable (control signal)

Table B.1: Instruction Memory Ports Description

Start	Operation
1	instruction = Instruction_Memory[PC];
0	instruction = 32'd0

Table B.2: Instruction Memory Asynchronous Reading for a single cycle

Start	clk	Operation
1	posedge	instruction <= Instruction_Memory[PC];
0	posedge	instruction <= 32'd0

Table B.3: Instruction Memory Synchronous Reading for Multi-cycle or Pipelined Design

InstrWrEn	clk	Operation
1	posedge	Instruction_Memory[InstrWrAdd] <= InstrWrData;
0	posedge	X

Table B.4: Instruction Memory Synchronous Writing

"<=" is a synchronous operation with the clock, which takes a positive edge triggered clock cycle for the operation to perform

"=" Asynchronous operation , independent of the clock

Signal	**Description**
clk	Clock signal
mem_access_addr[31:0]	Memory access address
mem_write_data[31:0]	Data to be written to memory
mem_read_data[31:0]	Data read from memory
mem_write	Memory write control signal
mem_read	Memory read control signal
byte_half_word	Byte/half-word selection control signal
full_word	Full-word selection control signal
byteU	Unsigned byte selection control signal
half_wordU	Unsigned half-word selection control signal

Table B.5: Data Memory Ports

byte	half_word	full_word	mem_access_addr[1:0]	data_to_be_stored
1	X	X	00	=mem_write_data[7:0]
1	X	X	01	=mem_write_data[15:8]
1	X	X	10	=mem_write_data[23:16]
1	X	X	11	=mem_write_data[31:24]
0	1	X	X0	=mem_write_data[15:0]
0	1	X	X1	=mem_write_data[31:16]
0	0	X	XX	=mem_write_data[31:0]

Table B.6: Data Memory data to be stored control signals

mem write	clk	Operation
1	posedge	dataMemory[mem_access_addr[31:2]] <= data_to_be_stored;
0	posedge	X

Table B.7: Data Memory Synchronous Writing

Data_Reading_signals = {byte, half_word, byteU, half_wordU}

Data_Reading_signals	mem_access_addr[1:0]	reading_data
X X X X	XX	No read operation
1 X X X	00	{memory[ram_addr][7:0]}
1 X X X	01	{memory[ram_addr][15:8]}
1 X X X	10	{memory[ram_addr][23:16]}
1 X X X	11	{memory[ram_addr][31:24]}
0 1 X X	X0	{memory[ram_addr][15:0]}
0 1 X X	X1	{memory[ram_addr][31:16]}
0 0 1 X	00	{24'd0, memory[ram_addr][7:0]}
0 0 X 1	00	{16'd0, memory[ram_addr][15:0]}
0 0 0 X	01	{24'd0, memory[ram_addr][15:8]}
0 0 X 0	01	{16'd0, memory[ram_addr][31:16]}
0 0 X X	10	{24'd0, memory[ram_addr][23:16]}
0 0 X X	11	{24'd0, memory[ram_addr][31:24]}

Table B.8: DataMemory reads operation control signals

mem read	clk	Operation
1	posedge	mem_read_data <= reading_data;
0	posedge	mem_read_data <= 32'd0

Table B.9: Data Memory Synchronous Reading

clk	gprsWren	Read	Write
x	0	rdata1_o= GPRs_[raddr1_i], rdata2_o= GPRs_[raddr2_i]	x
posedge	1	rdata1_o= GPRs_[raddr1_i], rdata2_o= GPRs_[raddr2_i]	GPRs[waddr_i] <= wdata_i

Table B.10: GPR's synchronous write & asynchronous read

ImmOp	Operation
000	I-type
001	S-type
010	B-type
011	J-type
100	U-type
Default	Undefined

Table B.11: ImmOp and Operation

Operation	ImmExtD Output
I-type	{{20{ImmEncd[24]}}, ImmEncd[24:13]}
S-type	{{20{ImmEncd[24]}}, ImmEncd[24:18], ImmEncd[4:0]}
B-type	{{20{ImmEncd[24]}}, ImmEncd[0], ImmEncd[23:18], ImmEncd[4:1], 1'b0}
J-type	{{12{ImmEncd[24]}}, ImmEncd[12:5], ImmEncd[13], ImmEncd[23:14], 1'b0}
U-type	{ImmEncd[24:5], 12'b000000000000}
Undefined	32'd0

Table B.12: Immediate Generator Functionality

alu_op	Operation	ALU_out	lsr	gtr	eql
0000	ADD	$in_1 + in_2$	0	0	0
1000	SUB	$in_1 - in_2$	0	0	0
0010	SLT	1 if $in_1 < in_2$	ALU_out[0]	!(ALU_out[0])	0
0011	SLTU	1 if unsigned($in_1 < in_2$)	ALU_out[0]	!(ALU_out[0])	0
0111	AND	$in_1 \& in_2$	0	0	0
0110	OR	$in_1 \mid in_2$	0	0	0
0100	XOR	$in_1 \wedge in_2$	0	0	!(ALU_out)
0001	SLL	$in_1 << in_2[4:0]$	0	0	0
0101	SRL	$in_1 >> in_2[4:0]$	0	0	0
1101	SRA	$in_1 >>> in_2[4:0]$	0	0	0
Default	No operation	0	0	0	0

Table B.13: ALU unit control signals

ld	jal	jalr	auipc	lui	Output (reg_write_data)
0	0	0	0	0	ALU_out
1	0	0	0	0	mem_read_data
0	1	0	0	0	pc4
0	0	1	0	0	pc4
0	0	0	1	0	PC_plusImm
0	0	0	0	1	immediate

Table B.14: write back :rsltMux control signals

APPENDIX C

Code snippets

Below is the Python code to extract the hexadecimal instructions form the text file generated form toolchain:

Listing C.1: Python code to extract data

```
import re
import os

def extract_data(input_file_path):
    try:
        input_file_name, input_file_extension =
            ↪ os.path.splitext(input_file_path)
        output_file_path =
            ↪ f"{input_file_name}_extracted_macine_n_Assembly.txt"
        machine_code_output_file_path =
            ↪ f"{input_file_name}_machine.txt"

        with open(input_file_path, 'r') as file:
            data = file.read()
            hex_data =
                ↪ re.findall(r'\b[0-9a-fA-F]+:\s+([0-9a-fA-F]+)\s+',
                ↪ data)

        with open(output_file_path, 'w') as file:
            file.write('\n'.join(hex_data))

        with open(machine_code_output_file_path, 'w') as file:
            file.write('\n'.join(hex_data))

        return True, output_file_path,
            ↪ machine_code_output_file_path # Return True to
```

```
            ↪ indicate the operation was successful and the
            ↪ paths of the output files
    except Exception as e:
        print(f"An error occurred: {e}")
        return False, None, None # Return False to indicate that
            ↪ an error occurred and no output files were created

# Taking input file path as input
input_path = input("Enter the input file path: ")
extraction_result, output_file, machine_code_output_file =
    ↪ extract_data(input_path)

if extraction_result:
    print(f"Filtered data has been saved to the output files:
        ↪ {output_file} and {machine_code_output_file}")
else:
    print("An error occurred while processing the file.")
```

Below is the Python code snippet for receiving data form FPGA using UART protocal:

Listing C.2: Python code for serial communication with FPGA

```
import serial
import binascii

# Replace "COM14" with the correct COM port where your FPGA is
    ↪ connected
serial_data = serial.Serial("COM14", 9600)

# Open a new text file for writing
with open("Data_Mem111.txt", "w") as file:
    consecutive_data = b"" # Initialize an empty byte string
        ↪ for consecutive data

    while True:
        dat = serial_data.read(1) # Read 1 byte of data from FPGA
```

```
        # Convert the received data to hexadecimal
        data_hex = dat.hex()

        # Append the data to the consecutive_data byte string
        consecutive_data += binascii.unhexlify(data_hex)

        # If we have received 4 consecutive bytes, print them in
            ↪ little-endian order
        if len(consecutive_data) == 4:
            little_endian_data = consecutive_data[::-1] # Reverse
                ↪ the order
            print("Little-endian data:", little_endian_data.hex())

            # Write the little-endian data to the text file
            file.write(little_endian_data.hex() + "\n")

            # Reset the consecutive_data for the next set of 4
                ↪ bytes
            consecutive_data = b""

# Close the serial connection (this line will not be reached
    ↪ as the loop runs indefinitely)
serial_data.close()
```

Below is the Python code snippet for send instructions from laptop to FPGA using UART protocal:

Listing C.3: Python code to send data via UART

```
import serial
import time

def send_data_via_uart(input_file_path):
    try:
        # Open the input file and read data
        with open(input_file_path, 'r') as file:
```

```
            data = file.read()

        # Open serial connection
        ser = serial.Serial('COM14', 9600, timeout=1) # Fixed
            ↪ COM port

        # Send data through UART
        ser.write(data.encode())
        print("Data sent successfully.")

        # Close serial connection
        ser.close()

        return True # Return True to indicate successful
            ↪ transmission
    except Exception as e:
        print(f"An error occurred: {e}")
        return False # Return False to indicate transmission
            ↪ failure

# Input file path
input_file_path = input("Enter the input file path: ")

# Send data through UART
send_result = send_data_via_uart(input_file_path)

if send_result:
    print("Data sent via UART successfully.")
else:
    print("Failed to send data via UART.")
```

APPENDIX D

Chisel Installation

D.1 Introduction

Chisel (Constructing Hardware in a Scala Embedded Language) and Verilog are both hardware description languages (HDLs), but they differ significantly in their approach and capabilities. This document provides an in-depth comparison with extensive code examples.

D.2 Key Differences

Aspect	Chisel	Verilog
Language Type	Embedded DSL in Scala	Standalone HDL
Abstraction Level	Higher (object-oriented, functional)	Lower (RTL/structural)
Code Reusability	High (generators, parameters)	Limited (macros, generate)
Meta-programming	Full Scala capabilities	Limited
Simulation	Scala testbenches	Verilog testbenches
Synthesis	Generates Verilog	Direct synthesis
Learning Curve	Steeper (requires Scala)	Easier (dedicated HDL)
Community	Growing (mainly RISC-V ecosystem)	Mature and widespread

Table D.1: Detailed Comparison between Chisel and Verilog

D.3 How Chisel Works

Chisel is a hardware construction language embedded in Scala. It allows hardware designers to leverage the full power of a modern programming language to write parameterized hardware generators.

D.4 Basic Constructs Comparison

D.4.1 Combinational Logic

Listing D.1: Combinational Logic in Verilog

```
module comb_logic(
```

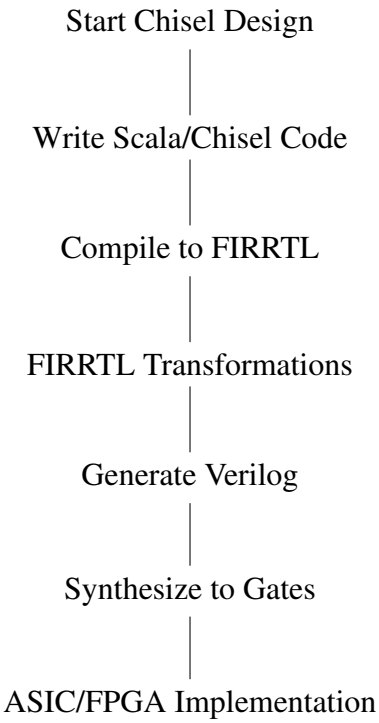


Figure D.1: Detailed Chisel Workflow

```
    input [3:0] a, b,
    output [3:0] y1, y2, y3
);
    assign y1 = a & b; // Bitwise AND
    assign y2 = a | b; // Bitwise OR
    assign y3 = a ^ b; // Bitwise XOR
endmodule
```

Listing D.2: Combinational Logic in Chisel

```
import chisel3._

class CombLogic extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val y1 = Output(UInt(4.W))
    val y2 = Output(UInt(4.W))
    val y3 = Output(UInt(4.W))
  })
```

```
  io.y1 := io.a & io.b // Bitwise AND
  io.y2 := io.a | io.b // Bitwise OR
  io.y3 := io.a ^ io.b // Bitwise XOR
}
```

D.4.2 Sequential Logic

Listing D.3: Register in Verilog

```
module register(
   input clk,
   input rst,
   input [7:0] d,
   output reg [7:0] q
);
always @(posedge clk or posedge rst) begin
   if (rst) q <= 8'h0;
   else q <= d;
end
endmodule
```

Listing D.4: Register in Chisel

```
import chisel3._

class Register extends Module {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U(8.W))
  reg := io.d
  io.q := reg
}
```

D.5 Advanced Examples

D.5.1 Finite State Machine

Listing D.5: FSM in Verilog

```
module fsm(
   input clk, rst, in,
   output reg out
);
   parameter S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   always @(posedge clk or posedge rst) begin
      if (rst) state <= S0;
      else state <= next_state;
   end

   always @(*) begin
      case (state)
         S0: next_state = in ? S1 : S0;
         S1: next_state = in ? S2 : S0;
         S2: next_state = in ? S2 : S0;
         default: next_state = S0;
      endcase
   end

   assign out = (state == S2);
endmodule
```

Listing D.6: FSM in Chisel

```
import chisel3._
import chisel3.util._

class FSM extends Module {
  val io = IO(new Bundle {
```

```
    val in = Input(Bool())
    val out = Output(Bool())
  })

  val s0 :: s1 :: s2 :: Nil = Enum(3)
  val state = RegInit(s0)

  io.out := (state === s2)

  switch(state) {
    is(s0) { state := Mux(io.in, s1, s0) }
    is(s1) { state := Mux(io.in, s2, s0) }
    is(s2) { state := Mux(io.in, s2, s0) }
  }
}
```

D.5.2 Memory Example

Listing D.7: Memory in Verilog

```
module memory(
    input clk,
    input we,
    input [3:0] addr,
    input [7:0] din,
    output [7:0] dout
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        if (we) mem[addr] <= din;
    end

    assign dout = mem[addr];
endmodule
```

Listing D.8: Memory in Chisel

```
import chisel3._

class Memory extends Module {
  val io = IO(new Bundle {
    val we = Input(Bool())
    val addr = Input(UInt(4.W))
    val din = Input(UInt(8.W))
    val dout = Output(UInt(8.W))
  })

  val mem = SyncReadMem(16, UInt(8.W))

  when(io.we) {
    mem.write(io.addr, io.din)
  }

  io.dout := mem.read(io.addr)
}
```

D.6 Testbench Comparison

D.6.1 Verilog Testbench

Listing D.9: Verilog Testbench

```
module testbench;
    reg clk, rst;
    reg [7:0] a, b;
    wire [7:0] sum;

    adder uut (.a(a), .b(b), .sum(sum));

    initial begin
        clk = 0;
```

```
        forever #5 clk = ~clk;
    end

    initial begin
        rst = 1;
        a = 0; b = 0;
        #20 rst = 0;

        a = 8'h12; b = 8'h34;
        #10 $display("Sum = %h", sum);

        a = 8'hFF; b = 8'h01;
        #10 $display("Sum = %h", sum);

        $finish;
    end
endmodule
```

D.6.2 Chisel Testbench

Listing D.10: Chisel Testbench

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

class AdderTest extends AnyFlatSpec with ChiselScalatestTester
    ↪ {
  "Adder" should "add numbers correctly" in {
    test(new Adder(8)) { dut =>
      dut.io.a.poke(0x12.U)
      dut.io.b.poke(0x34.U)
      dut.clock.step()
      dut.io.sum.expect(0x46.U)

      dut.io.a.poke(0xFF.U)
```

```
      dut.io.b.poke(0x01.U)
      dut.clock.step()
      dut.io.sum.expect(0x00.U)
    }
  }
}
```

D.7 Parameterization Examples

D.7.1 Verilog Parameterization

Listing D.11: Parameterized FIFO in Verilog

```
module fifo #(
    parameter WIDTH = 8,
    parameter DEPTH = 16
)(
    input clk, rst,
    input wr_en, rd_en,
    input [WIDTH-1:0] din,
    output [WIDTH-1:0] dout,
    output full, empty
);
    reg [WIDTH-1:0] mem [0:DEPTH-1];
    reg [$clog2(DEPTH):0] wr_ptr, rd_ptr;

    // Implementation omitted for brevity
endmodule
```

D.7.2 Chisel Parameterization

Listing D.12: Parameterized FIFO in Chisel

```
import chisel3._
import chisel3.util._
```

```
class Fifo(val width: Int, val depth: Int) extends Module {
  val io = IO(new Bundle {
    val wr_en = Input(Bool())
    val rd_en = Input(Bool())
    val din = Input(UInt(width.W))
    val dout = Output(UInt(width.W))
    val full = Output(Bool())
    val empty = Output(Bool())
  })

  val mem = SyncReadMem(depth, UInt(width.W))
  val wrPtr = RegInit(0.U(log2Ceil(depth).W))
  val rdPtr = RegInit(0.U(log2Ceil(depth).W))

  // Implementation omitted for brevity
}
```

D.8 Conclusion

Chisel offers powerful abstractions for hardware design through its integration with Scala, enabling more concise and reusable code. Verilog remains the industry standard with excellent tool support. The choice depends on:

- Project complexity (Chisel excels for complex, parameterized designs)
- Team expertise (Verilog is more accessible to traditional hardware engineers)
- Toolchain requirements (Verilog has universal support)
- Need for verification (Chisel integrates better with modern verification approaches)

APPENDIX E

Branch predictor Unit & Implementation by using Champsim

E.1 Introduction

Branch prediction is a fundamental part of modern pipelined processor design. It enhances instruction throughput by guessing the direction of conditional branch instructions before they are resolved. Accurate branch prediction is critical in reducing pipeline stalls and improving instruction-level parallelism.

E.2 Branch Predictor Architecture

The branch predictor is designed to detect control hazards and mitigate stalls using speculative execution. We implement a simple 2-bit saturating counter-based predictor.

E.2.1 Prediction States

- Strongly Taken (11)
- Weakly Taken (10)
- Weakly Not Taken (01)
- Strongly Not Taken (00)

E.3 Verilog Code: 2-bit Branch Predictor

Listing E.1: Verilog Implementation of 2-bit Branch Predictor

```
module branch_predictor (
  input wire clk,
  input wire rst,
  input wire branch_taken,
  input wire [4:0] branch_address,
  output reg prediction
);
  reg [1:0] bht [31:0]; // 32-entry Branch History Table
```

```
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      for (int i = 0; i < 32; i = i + 1)
        bht[i] <= 2'b10; // weakly taken on reset
    end else begin
      case (bht[branch_address])
        2'b00: prediction <= 0;
        2'b01: prediction <= 0;
        2'b10: prediction <= 1;
        2'b11: prediction <= 1;
      endcase

      // Update prediction state
      if (branch_taken && bht[branch_address] != 2'b11)
        bht[branch_address] <= bht[branch_address] + 1;
      else if (!branch_taken && bht[branch_address] != 2'b00)
        bht[branch_address] <= bht[branch_address] - 1;
    end
  end
endmodule
```

E.4 ChampSim: Branch Predictor Simulation

ChampSim is a trace-based simulator that allows detailed testing of branch predictors, cache hierarchies, and prefetchers. It's modular, making it ideal for evaluating the performance of your own predictor design.

E.4.1 Installation Guide

1. Clone the ChampSim repository:
   ```
   git clone https://github.com/ChampSim/ChampSim.git
   cd ChampSim
   ```

2. Install required packages (Ubuntu):
   ```
   sudo apt-get install build-essential libboost-all-dev
   ```

3. Build the simulator with a custom branch predictor:
   ```
   ./build_champsim.sh bimodal no no no no lru 1
   ```

4. Download and decompress the trace files from: https://www.ece.ubc.ca/~sasha/champsim-traces/speccpu
5. Run the simulation:

```
./champsim --warmup_instructions 10000000
    ↪ --simulation_instructions 100000000
    ↪ traces/600.perlbench_s-13241-43.trace.xz
\end{verbatim}
```

E.5 Conclusion

This work presents the design of a simple 2-bit branch predictor, integrated within a RISC-V pipeline processor. Its performance and accuracy can be benchmarked using trace-based simulations on ChampSim. Future work includes integrating a perceptron predictor and conducting performance comparisons with other state-of-the-art prediction techniques.

```
Generating dependencies for branch/branch_predictor.cc...
Compiling branch/branch_predictor.cc...
Generating dependencies for replacement/llc_replacement.cc...
Compiling replacement/llc_replacement.cc...
Generating dependencies for replacement/base_replacement.cc...
Compiling replacement/base_replacement.cc...
Generating dependencies for prefetcher/kpcp_util.cc...
Compiling prefetcher/kpcp_util.cc...
Generating dependencies for prefetcher/l2c_prefetcher.cc...
Compiling prefetcher/l2c_prefetcher.cc...
Generating dependencies for prefetcher/l1i_prefetcher.cc...
Compiling prefetcher/l1i_prefetcher.cc...
Generating dependencies for prefetcher/llc_prefetcher.cc...
Compiling prefetcher/llc_prefetcher.cc...
Generating dependencies for prefetcher/l1d_prefetcher.cc...
Compiling prefetcher/l1d_prefetcher.cc...
Linking bin/champsim...

ChampSim is successfully built
Branch Predictor: gshare
L1I Prefetcher: no
L1D Prefetcher: no
L2C Prefetcher: no
LLC Prefetcher: no
LLC Replacement: lru
Cores: 1
Binary: bin/gshare-no-no-no-no-lru-1core
```

Figure E.1: Branch Predictor Unit Successfully stored and Simulated on Champsim