

MULTI CYCLE PIPELINED RISCv BASED PROCESSOR

TEAM 11

**CA Project Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Document Version** | **Remarks** | **Drafted by** |
| 10/09/ 2024 | Version 1.0 | Design Overview | Team 11 |
| 11/09/2024 | Version 2.0 | Architecture | Team 11 |
| 12/09/2024 | Version 3.0 | Simulation results and waveforms | Team 11 |

**PROJECT DOCUMENT- MULTI CYCLE PIPELINED RISCv BASED PROCESSOR**

Table of Contents

[**CHAPTER 1 – DESIGN OVERVIEW** 5](#_Toc177050693)

[1.1 INTRODUCTION TO RISC-V 5](#_Toc177050694)

[1.3 ADVANTAGES OF RISC-V 5](#_Toc177050695)

[1.4 DISADVANTAGES OF RISC-V 5](#_Toc177050696)

[1.2 KEY FEATURES 5](#_Toc177050697)

[1.5 PROJECT OVERVIEW 6](#_Toc177050698)

[1.7 INSTRUCTION FORMAT 6](#_Toc177050699)

[**CHAPTER 2 – ARCHITECTURE** 7](#_Toc177050700)

[2.1 MICRO - ARCHITECTURE 7](#_Toc177050701)

[2.2 ARCHITECTURE COMPONENTS 8](#_Toc177050702)

[2.2.1 PROGRAM COUNTER 8](#_Toc177050703)

[2.2.2 PROGRAM MEMORY 8](#_Toc177050704)

[2.2.3 INSTRUCTION DECODER 9](#_Toc177050705)

[2.2.4 REGISTER FILE 9](#_Toc177050706)

[2.2.5 ALU 10](#_Toc177050707)

[2.2.6 LOAD BLOCK 10](#_Toc177050708)

[**CHAPTER 3 – PIPELINE STAGES** 11](#_Toc177050709)

[3.1. FETCH STAGE: 11](#_Toc177050710)

[3.2. Decode Stage: 11](#_Toc177050711)

[3.3 EXECUTE STAGE : 11](#_Toc177050712)

[3.4 WRITE BACK STAGE : 11](#_Toc177050713)

[**CHAPTER 4 – TESTBENCH IMPLEMENTATION** 12](#_Toc177050714)

[4.1 DESIGN CODES 12](#_Toc177050715)

[4.1.1 PROGRAM COUNTER RTL CODE 12](#_Toc177050716)

[4.1.2 PROGRAM MEMOERY RTL CODE 12](#_Toc177050717)

[4.1.3 INSTRUCTION DECODER RTL CODE 12](#_Toc177050718)

[4.1.4 REGISTER FILE RTL CODE 12](#_Toc177050719)

[4.1.5 ALU RTL CODE 12](#_Toc177050720)

[4.1.6 LOAD BLOCK 12](#_Toc177050721)

[4.2 TESTBENCH CODE 12](#_Toc177050722)

[**CHAPTER 5 - SIMULATION RESULTS** 13](#_Toc177050723)

[5.1 LOAD IMMEDIATE 13](#_Toc177050724)

[5.2 ADD OPERATION 13](#_Toc177050725)

[5.3 LOGICAL OPERATION (AND OR XOR) 14](#_Toc177050726)

[5.4 ALL THE OPERATIONS 14](#_Toc177050727)

# **CHAPTER 1 – DESIGN OVERVIEW**

## 1.1 INTRODUCTION TO RISC-V

## 

The RISC-V instruction set has a 32-bit fixed length and follows a little-endian format. It includes 32 general-purpose registers, with reg0 fixed at 0, and a 32-bit program counter that increments by one for word-addressable instruction memory. Chosen for its efficiency, RISC-V is pipeline-friendly and requires minimal hardware and power. Techniques like loop unrolling and compiler scheduling are used to optimize performance. Supports six instruction formats: R-type, U-type, I-type, B-type, J-type, and S-type.

## 1.3 ADVANTAGES OF RISC-V

* RISC-V is an open-source ISA, allowing for free use and modification1.
* It has a clean and simple design, making it easier to implement and understand2.
* The architecture is modular and extensible, enabling custom extensions for specific applications1.
* It reduces development costs by leveraging shared tools and resources1.
* RISC-V fosters innovation by providing a flexible platform for developing new processor designs

## 1.4 DISADVANTAGES OF RISC-V

* The software ecosystem for RISC-V is still developing and may not be as mature as those for established ISAs like x86 or ARM1.
* There is a risk of fragmentation due to the open-source nature, which can lead to compatibility issues1.
* RISC-V processors may require more memory to store additional instructions needed for complex tasks2.
* Developing for RISC-V can require more effort compared to more established architectures3.
* The development and manufacturing of RISC-V processors can be more expensive

## 1.2 KEY FEATURES

* Operations: ADD, SUB, AND, OR, XOR ,NOP.
* Develop a multi-cycle 5 stage pipelined processor
* For each operation implement the corresponding instruction of RISC-V

## 

## 1.5 PROJECT OVERVIEW

* Designing Micro-architecture, define the project's micro-architecture and functional blocks.
* Implement ADD, SUB, AND, OR, XOR and NOP operations using corresponding RISC-V instructions.
* Create a testbench to verify the functionality of the processor and each ALU operation
* Develop a multi-cycle 5-stage pipelined processor architecture.
* Pipeline Stages: The processor pipeline is divided into different stages, Instruction Fetch (IF), Instruction Decode (ID), Execute (EX) and Write Back (WB)
* Developing various stages which includes Program Counter, Program memory, Instruction decoder, ALU and Load Block and developing their HDL codes using Verilog

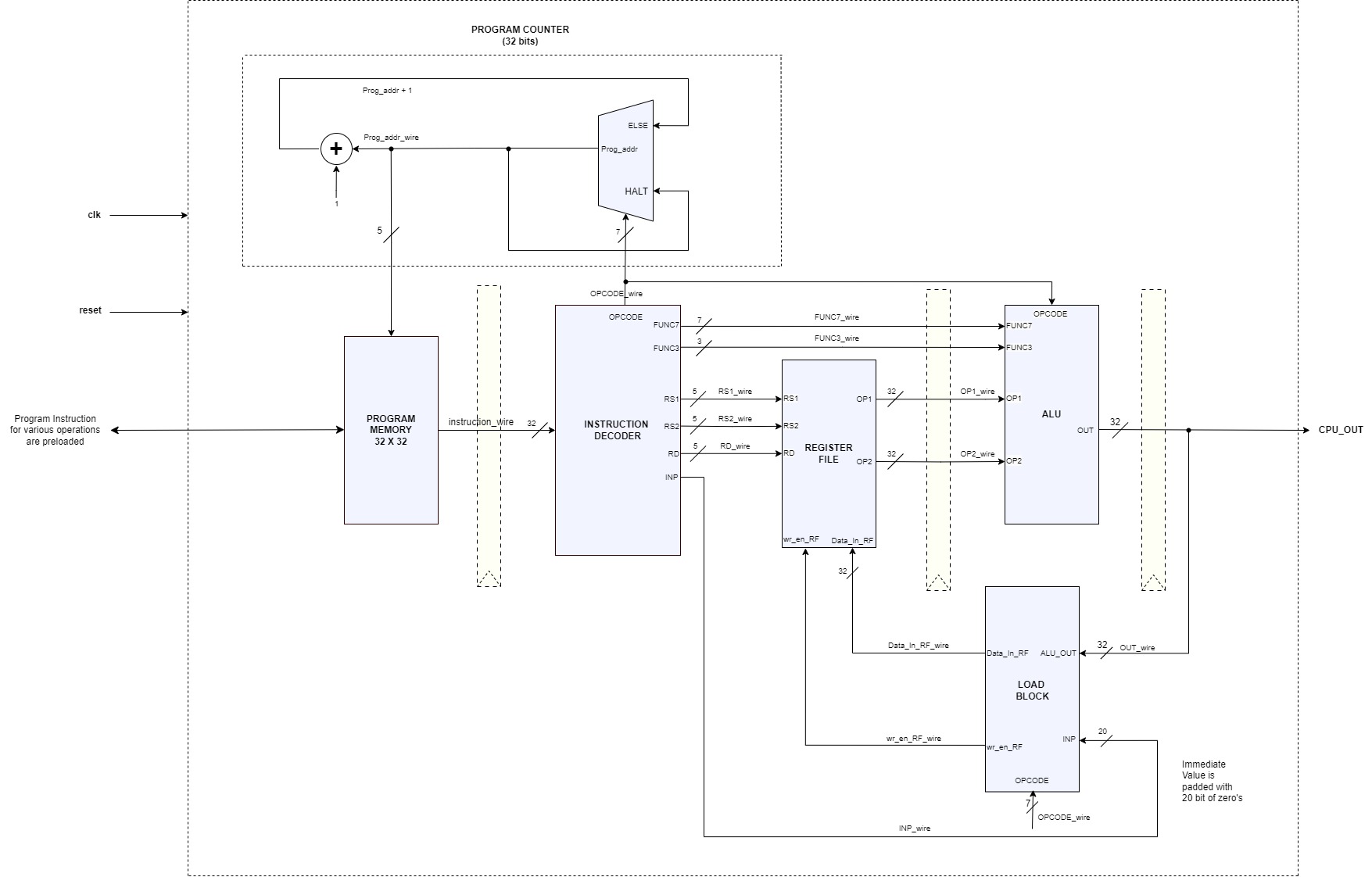
## 1.7 INSTRUCTION FORMAT

A table with numbers and symbols

Description automatically generated

# **CHAPTER 2 – ARCHITECTURE**

## 2.1 MICRO - ARCHITECTURE



## 2.2 ARCHITECTURE COMPONENTS

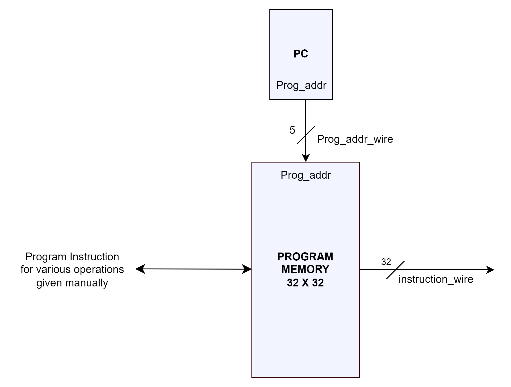
### 2.2.1 PROGRAM COUNTER

A screen shot of a computer

Description automatically generated

* The PC, or program counter, is a key CPU register that keeps track of the order in which instructions are executed.
* The PC is in sync with the system clock and stores the memory address of the subsequent instruction that needs to be retrieved.
* Every time there is a clock pulse, the CPU retrieves the instruction from the PC's address and advances the PC to the subsequent instruction.
* Through this procedure, the CPU is guaranteed to carry out instructions in the proper order, facilitating seamless and effective system operation.

### 2.2.2 PROGRAM MEMORY



* The instructions that the CPU needs to carry out are kept in the program memory.
* The next instruction is fetched and transmitted to the control unit and decoder after the Program Counter (PC) locates its precise memory address.
* By disassembling the instruction into its constituent parts, such as register addresses, opcodes, and functions, the decoder is able to understand it. In order to guarantee that the command is carried out properly, the control unit subsequently routes this data to the relevant CPU components. Through the translation and orderly execution of instructions, this process makes it possible for the CPU to function smoothly.

### 2.2.3 INSTRUCTION DECODER

A screen shot of a video game

Description automatically generated

* The instruction decoder disassembles the fetched instruction into its constituent parts, such as register addresses, opcodes, and functions.
* After decoding, the control unit routes the decoded information to the relevant CPU components to ensure proper execution of the command.
* This decoding process facilitates the orderly translation and execution of instructions, enabling the CPU to function smoothly.
* The instruction decoder works in conjunction with the Program Counter (PC), which locates the precise memory address of the next instruction to be fetched and decoded.

### 2.2.4 REGISTER FILE

A close-up of a register file

Description automatically generated

* The register file consists of 32 registers, each 32 bits wide, designed to store operands and operation results.
* It uses 5-bit addresses for Rs1 and Rs2, allowing the processor to swiftly read from or write to these registers.
* Data can be read from any register at any time, and data can be written to a register in sync with the clock signal.
* When the processor is reset, all registers are set to zero, ensuring efficient data management and utilization during processor operations

### 2.2.5 ALU

A screen shot of a cell phone

Description automatically generated

* The Arithmetic Logic Unit (ALU) of a CPU is where all arithmetic and logic operations are carried out.
* The resultant data must be stored for later use after these processes are completed.
* Through a procedure known as "write-back," the outcome is restored into the register set.
* The CPU's registers are quick, compact storage spaces that momentarily store data.

The CPU can swiftly retrieve the data for next operations by storing the results in registers,

guaranteeing effective instruction processing and execution

### 2.2.6 LOAD BLOCK

A screen shot of a computer

Description automatically generated

* The LOAD\_BLOCK module handles three key operations: LOAD\_IMMEDIATE, NO\_OPERATION (NOP), and ALU\_OPERATION. It is responsible for either loading immediate values into registers, performing no operation (NOP), or transferring ALU results to the register file.
* In the case of a LOAD\_IMMEDIATE instruction (OPCODE = 7'b1111111), the module zero-extends the immediate input (INP) to 32 bits and writes it into the register file (Data\_In\_RF). The write enable signal (wr\_en\_RF) is asserted to ensure the data is written
* When the NOP instruction (OPCODE = 7'b0000000) is detected, the module disables the write operation by de-asserting wr\_en\_RF, ensuring no data is written to the register file.
* For ALU operations (OPCODE = 7'b0110011), the module captures the ALU output (ALU\_OUT) and writes it to the register file. The wr\_en\_RF signal is asserted to indicate that the ALU result should be stored in the destination register.

# **CHAPTER 3 – PIPELINE STAGES**

## 3.1. FETCH STAGE:

Program Counter (PC):

The Program Counter (PC) keeps track of the address of the instruction currently being executed. With each clock cycle, it advances to the address of the next instruction. It resets when the reset signal is active.

## 3.2. DECODE STAGE :

The register file comprises a collection of registers that temporarily store data for the CPU. During the decode phase, the RegFile reads two source registers (Rs1 and Rs2) based on specific fields from the instruction (instruction[19:15] and instruction[24:20]). It also includes a write-back mechanism (Reg\_write) to update the destination register (Rd) after the execution stage.

## 3.3 EXECUTE STAGE :

ALU (Arithmetic Logic Unit):

The ALU performs arithmetic and logical operations based on the control signals from the control unit (alu\_op). It takes two operands (op1 and op2), which could either come from the register file or include an immediate value, and produces an output (alu\_out).

## 3.4 WRITE BACK STAGE :

Load Block

The LOAD\_BLOCK module handles three key operations: LOAD\_IMMEDIATE, NO\_OPERATION (NOP), and ALU\_OPERATION. It is responsible for either loading immediate values into registers, performing no operation (NOP), or transferring ALU results to the register file

# **CHAPTER 4 – TESTBENCH IMPLEMENTATION**

## 4.1 DESIGN CODES

### 4.1.1 PROGRAM COUNTER RTL CODE



### 4.1.2 PROGRAM MEMOERY RTL CODE



### 4.1.3 INSTRUCTION DECODER RTL CODE



### 4.1.4 REGISTER FILE RTL CODE



### 4.1.5 ALU RTL CODE



### 4.1.6 LOAD BLOCK



## 4.2 TESTBENCH CODE



# **CHAPTER 5 - SIMULATION RESULTS**

## 5.1 LOAD IMMEDIATE

Instructions:

PROG\_MEM[0] <= 32'b0000000\_00000\_00000\_001\_01000\_1111111; // LOAD\_IMM LOAD\_IMM R8, #1

PROG\_MEM[1] <= 32'b0000000\_00000\_00000\_100\_01001\_1111111; // LOAD\_IMM LOAD\_IMM R9, #4

A screenshot of a computer

Description automatically generated

The Immediate value 1 and 4 is stored in R8 and R9

## 5.2 ADD SUB OPERATION

Instructions:

PROG\_MEM[3] <= 32'b0000000\_01001\_01000\_000\_00001\_0110011; // ADD ADD R1, R8, R9

PROG\_MEM[4] <= 32'b0100000\_01001\_01000\_000\_00010\_0110011; // SUB SUB R2, R8, R9

A screenshot of a computer

Description automatically generated

In the above waveform , operand1 and operand 2 indicates the 2 operand for the ADD and SUB operation

## 5.3 LOGICAL OPERATION (AND OR XOR)

Instructions:

PROG\_MEM[5] <= 32'b0000000\_01001\_01000\_110\_00011\_0110011; // AND AND R3, R8, R9

PROG\_MEM[6] <= 32'b0000000\_01001\_01000\_111\_00100\_0110011; // OR OR R4, R8, R9

PROG\_MEM[7] <= 32'b0000000\_01001\_01000\_100\_00101\_0110011; // XOR XOR R5, R8, R9

A computer screen shot of a black screen

Description automatically generated

The AND and OR operation result is stored in the specified destination register.

The XOR operation result is stored in the specified destination register.

## 5.4 ALL THE OPERATIONS

A computer screen shot of a black background

Description automatically generated