Lab Oriented Project Report

On

Final Specification

Submitted in partial fulfilment

of

Computer Architecture

Under the Guidance of :

Prof. Dr. A. Siggelkow

Submitted By :

Jayant Patil (18041745)

Aditya Grewal (29541041)

Avanindra Kumar Mishra (20241804)

Sagar Shreeshailappa Hosmani (27741741)

Omkarnimith Muthabyraiah(16841822)

Jash Shah (23741845)

MASTER OF ELECTRICAL ENGINEERING AND EMEBDDED SYSTEMS

HOCHSCHULE RAVENSBURG-WEINGARTEN

June 26, 2024

Contents

List of Figures

List of Tables

1. Functional Requirements

Requirement	ID	Importance	Verifiable	Description
Program Counter (PC)	G01	High	System Verilog-TB	The PC holds the address of the next instruction to be executed. It increments by 4 bytes each clock cycle, corresponding to the size of each instruction.
Instruction Memory	G02	High	System Verilog-TB	Stores the program's instructions and supplies them to the processor. It retrieves instructions based on the address provided by the PC.
Instruction Decoder	G03	High	System Verilog-TB	Interprets the binary instruction fetched from the instruction memory. It generates control signals to guide the processor components.
Register File	G05	High	System Verilog-TB	Consists of 32 general-purpose registers for storing intermediate data and results. It supports two read ports and one write port.
Arithmetic Logic Unit (ALU)	G06	High	System Verilog-TB	Performs arithmetic and logical operations. Receives operands from the register file or immediate values and executes operations specified by control signals.
Data Memory	G07	High	System Verilog-TB	Used for reading from and writing to memory locations during load and store instructions. It interacts with memory beyond the register file.
Control Unit	G08	High	System Verilog-TB	Generates control signals based on the decoded instruction. Directs operations of the ALU, data memory, and register file.

Instruction Execution Flow	G08	High	System Verilog-TB	Each instruction starts by using the PC to fetch the instruction. Operands are read from the registers. The ALU is used for address calculation, operation execution, or equality check. Results are written back to the register file or used for further processing.
Memory-Reference Instructions	G08	High	System Verilog-TB	Includes load doubleword (ld) and store doubleword (sd). Uses the ALU for address calculation and accesses memory for data operations.
Arithmetic-Logical Instructions	G08	High	System Verilog-TB	Includes add, sub, and, and or. Uses the ALU for executing operations and writes results back to registers.
Conditional Branch Instructions	G08	High	System Verilog-TB	Includes branch if equal (beq). Uses the ALU for equality tests and may alter the next instruction address based on the comparison.
Multiplexors	G08	High	System Verilog-TB	Regulate the flow of data to various components based on the instruction class. Controlled by control signals decoded from the instruction.
RISC-V RV64I Compliance	G08	High	Application Program	The processor must support all RV64I instructions (e.g., ADD, LOAD, STORE, BEQ) and execute them correctly as per the RISC-V specifications.

2. History

Rev. No.	Rev. Date	Description of change in Current Version	Author
V 1.0	15.04.2024	Functional Requirements	Jayant Patil Aditya Grewal Avanindra Kumar Mishra Omkarnimith Muthabyraiah Jash Shah Sagar Shreeshailappa Hosmani
V 2.0	05.06.2024	Specifications	Jayant Patil Aditya Grewal Avanindra Kumar Mishra Omkarnimith Muthabyraiah Jash Shah Sagar Shreeshailappa Hosmani
V 3.0	15.06.2024	Final Specifications, Design Descriptions and Simulation	Jayant Patil Aditya Grewal Avanindra Kumar Mishra Omkarnimith Muthabyraiah Jash Shah Sagar Shreeshailappa Hosmani

3. Document Overview

3.1 Glossary

Term	Description
ALU (Arithmetic Logic Unit)	A digital circuit used to perform arithmetic and logical operations. In the RISC-V implementation, the ALU performs operations such as addition, subtraction, AND, and OR based on the control signals provided.
ALU Control	The unit that generates the control signals for the ALU based on the ALUOp field and the function codes (funct7 and funct3) of the instruction. It determines which operation the ALU will perform.
ALUOp	A 2-bit control signal that indicates the type of operation to be performed by the ALU. The values of ALUOp determine whether the operation is an addition, subtraction, or a function determined by the funct7 and funct3 fields for R-type instructions.
Branch	A control signal indicating whether the current instruction is a branch instruction. If the condition for the branch is met, the program counter (PC) is updated to the target address.
Control Unit	The component that generates control signals based on the opcode of the instruction. It orchestrates the flow of data within the CPU, enabling the execution of instructions by controlling the operation of other components such as the ALU, registers, and memory.
Data Memory	A memory unit used to store data during program execution. The data memory is accessed using addresses computed by the ALU, and it supports read and write operations.
Immediate Generator (ImmGen)	A unit that generates immediate values from the instruction fields. Immediate values are often used for arithmetic operations or as addresses for load and store instructions.
Instruction Memory	A memory unit that stores the program instructions. The instruction memory is accessed sequentially using the program counter (PC), which holds the address of the current instruction.
MUX (Multiplexer)	A digital switch that selects one of several input signals and forwards the selected input to the output. In the RISC-V implementation, MUXes are used to select between different data sources based on control signals.
Program Counter (PC)	A register that holds the address of the next instruction to be executed. The PC is updated after each instruction fetch to point to the next instruction in memory.

Term	Description
Register File	A set of registers used to store operands and intermediate results. The register file has read and write ports to access and update the registers during instruction execution.
Zero Flag	A flag set by the ALU to indicate whether the result of an operation is zero. This flag is often used for branch instructions to determine if a condition is met (e.g., branch if equal).
Funct3 and Funct7	Fields within the R-type instruction format that specify the specific operation to be performed by the ALU. Funct3 is a 3-bit field, and funct7 is a 7-bit field.
MemRead	A control signal that enables reading data from memory.
MemWrite	A control signal that enables writing data to memory.
MemtoReg	A control signal that determines whether the data to be written to a register comes from memory or the ALU result.
RegWrite	A control signal that enables writing data to a register in the register file.
ALUSrc	A control signal that determines whether the second operand for the ALU comes from a register or an immediate value.
Add	An arithmetic operation where two values are summed. In the context of the PC, adding 4 to the current PC value points to the next instruction.
Shift Left 1	An operation that shifts a binary value left by one position, effectively multiplying it by 2. This operation is often used in calculating branch target addresses.

Table 3.1: Glossary of Terms for RISC-V Implementation

4. Product Overview

The RV64I single-cycle RISC-V processor is a fundamental design that exemplifies simplicity and efficiency in modern computer architecture. This processor, which adheres to the RISC-V instruction set architecture (ISA), is characterized by its 64-bit width and single-cycle execution, where each instruction is completed in one clock cycle.

Fundamental design elements in RV64I are:

- **The Program Counter (PC)** is a crucial component that holds the address of the next instruction to be executed in the sequence. On each clock cycle, the PC increments by 4 bytes, corresponding to the size of each instruction in the RISC-V architecture.
- **Instruction Memory** is responsible for storing the program's instructions and supplying them to the processor. When the PC provides an address, the instruction memory retrieves the corresponding instruction and forwards it to the instruction decoder. This read-only memory is typically implemented as a ROM or a similar non-volatile storage, ensuring the integrity and availability of the program instructions throughout the execution cycle.
- **The Instruction Decoder** plays a pivotal role in interpreting the binary instruction fetched from the instruction memory. It breaks down the instruction into its constituent parts, such as opcode, source registers, destination register, and immediate values. The decoder then generates the necessary control signals that guide the other components of the processor to perform the specified operation, whether it be an arithmetic calculation, memory access, or control flow alteration.
- **The Register File** consists of a set of 32 general-purpose registers that store intermediate data and results during instruction execution. It includes two read ports and one write port, allowing two source operands to be read simultaneously and one result to be written back each cycle.
- **The Arithmetic Logic Unit (ALU)** is the computational heart of the processor, responsible for performing arithmetic operations (such as addition, subtraction) and logical operations (such as AND, OR, XOR). The ALU receives its operands from the register file or immediate values decoded from the instruction and executes the operation specified by the control signals from the instruction decoder. The result is then either written back to a register or used for further processing.
- **Data Memory** is used for reading from and writing to memory locations during load and store instructions. The memory address is calculated by the ALU and sent to the data memory along with the data to be stored (for store operations) or a request to retrieve data (for load operations). This allows the processor to interact with memory to access data beyond the limited capacity of the register file, enabling larger and more complex computations.

- **The Control Unit** generates the necessary control signals based on the decoded instruction. These signals direct the operation of other components like the ALU, data memory, and register file. It determines the type of operation (arithmetic, logical, load, store, branch) and configures the processor's pathways accordingly.

Our design is capable of executing memory-reference instructions load doubleword (ld) and store doubleword (sd), the arithmetic-logical instructions add, sub, and, and or, and the conditional branch instruction branch if equal (beq).

For every instruction, the first two steps are identical:

- Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
- Read one or two registers, using fields of the instruction to select the registers to read. For the ld instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. The three instruction classes involved in this design are memory-reference, arithmetic-logical, and branches. All instruction classes use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic logical instructions for the operation execution, and conditional branches for the equality test. After using the ALU, the actions required to complete various instruction classes differ. A memory reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a conditional branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction.

Below figure shows the high-level view of a RISC-V implementation, focusing on the various functional units and their interconnection along with the required multiplexors as well as control lines for the major functional units.

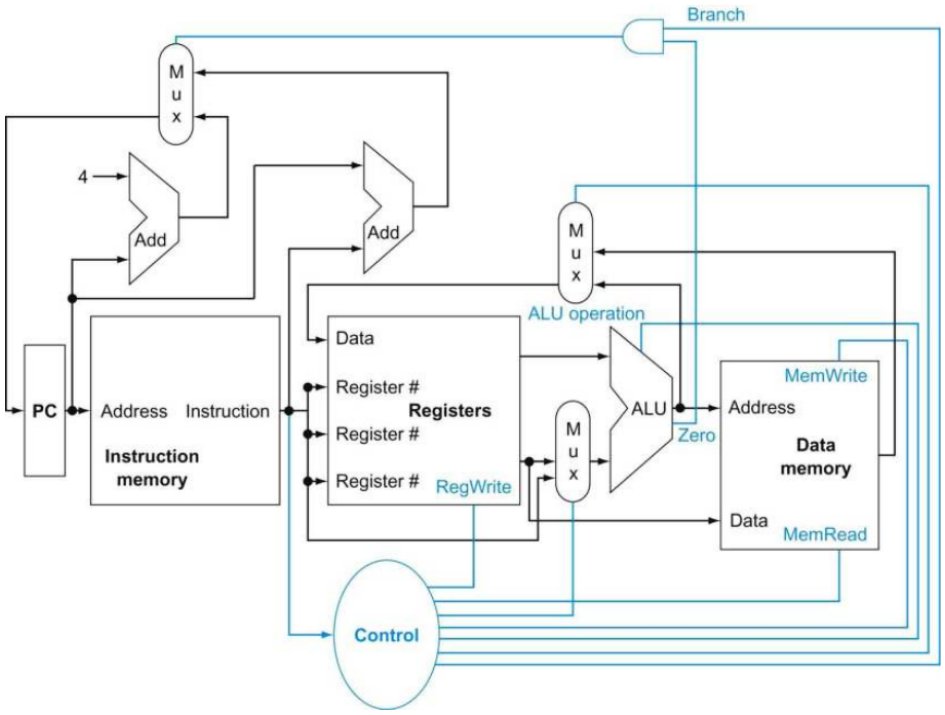

Figure 4.1: Abstract view of RV64I

All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

The multiplexors are used to regulate the flow of the data to various design elements depending on the class of the instruction that is being executed. These multiplexors are controlled by various control signals decoded from the instruction itself. A detailed description of the instruction decoding, control signals, datapath and other necessary design elements will be given in detail in Chapter 6.

5. Architecture Concepts

5.1 RISC-V ISA

The RISC-V ISA (Instruction Set Architecture) is a modern, open standard developed to support a wide range of computing devices. The RV64I is a specific implementation of the RISC-V ISA, focusing on a 64-bit integer base instruction set. The RV64I ISA is a simple yet powerful 64-bit instruction set designed with a reduced instruction set computing (RISC) philosophy. It includes a basic set of instructions for integer computation, memory operations, and control flow, which form the foundation for more complex instructions and extensions.

RV64I defines 32 general-purpose registers (x0 to x31), each 64 bits wide. Register x0 is hard-wired to zero, providing a convenient way to zero out registers or perform no-ops in certain instructions. The other registers (x1 to x31) are used for standard data manipulation and addressing.

5.1.1 Instruction Formats

RV64I uses a fixed 32-bit instruction format with several encoding types to support various instruction needs:

Name (Field size)	**7 bits**	**5 bits**	**5 bits**	**3 bits**	**5 bits**	**7 bits**	**Comments**
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
U-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
UJ-type	immediate[31:12]				rd	opcode	Upper immediate format

Table 5.1: RISC-V Instruction Formats

- **R-Type (Register-Register):** Used for arithmetic and logical operations between registers.

- **I-Type (Immediate):** Used for arithmetic operations with an immediate value, loads, and certain control instructions.
- **S-Type (Store):** Used for storing data from a register to memory.
- **SB-Type (Branch):** Used for conditional branching based on comparisons between register values.
- **U-Type (Upper Immediate):** Used for instructions that operate with a 20-bit upper immediate.

In our design, we will only be implementing datapaths and logic elements for R-Type, I-Type, S-Type and SB-Type instruction classes.

6. Description of the Design Elements

6.1 Register File

6.1.1 Overview

The register file is a fundamental component in the RISC-V processor architecture, responsible for providing fast storage and access to the CPU's general-purpose registers. This module is integral to the CPU's operation, enabling efficient data handling and manipulation during instruction execution.

6.1.2 Architectural Design

A register file consists of a set of registers that can be read and written by specifying the register number. The RISC-V architecture typically uses a register file with 32 registers, each 64 bits wide in the RV64I subset. This allows for rapid data access and storage, crucial for maintaining high performance in processing tasks.
The register file is designed to support multiple read and write operations. Specifically, it includes:

- Two read ports, allowing simultaneous reading from two registers.
- One write port, enabling the writing of data to a single register per clock cycle.

Figure 6.1: Register File Block Diagram

6.1.3 Functional Operation

In the context of the RISC-V CPU, the register file interacts closely with other components such as the ALU (Arithmetic Logic Unit) and instruction decoder. During the execution of R-format instructions, for example, the register file provides the operands for the ALU and stores the result of the computation.
The typical operations involving the register file include:

1. **Read Operations**: Two source registers are read, and their values are provided to the ALU or other computational units.

2. **Write Operations**: The result of an ALU operation or a value loaded from memory is written back to a destination register.

The design ensures that reads are combinational operations, meaning the data is immediately available based on the provided register addresses. Writes, however, are sequential operations controlled by a clock signal to ensure data integrity and proper synchronization.

6.1.4 Integration with CPU Architecture

The register file is crucial for the overall performance of the CPU as it stores the operands and results of arithmetic and logic operations. It interfaces directly with the instruction memory, instruction decoder, and the ALU:

- **Instruction Memory**: The instruction memory fetches instructions that specify which registers to read from and write to.

- **Instruction Decoder**: The decoder interprets the instructions and generates control signals required to access the register file.

- **ALU**: The ALU performs computations using the data read from the register file and stores the results back into the register file.

6.1.5 Initialization and Read/Write Synchronization

In practical implementations, some registers in the register file may be initialized to specific values for various purposes, such as testing or specific algorithmic requirements. This initialization can be crucial for certain applications or during the boot process of the CPU.
The read operations are usually instantaneous, providing the required data to the ALU or other processing units without delay. Write operations are synchronized with the clock to ensure data integrity and prevent race conditions.

6.1.6 Signal Descriptions

The key signals associated with the register file include:

- **clk**: The clock signal that synchronizes the write operations.

- **regWriteFlag**: A control signal indicating whether a write operation should be performed.

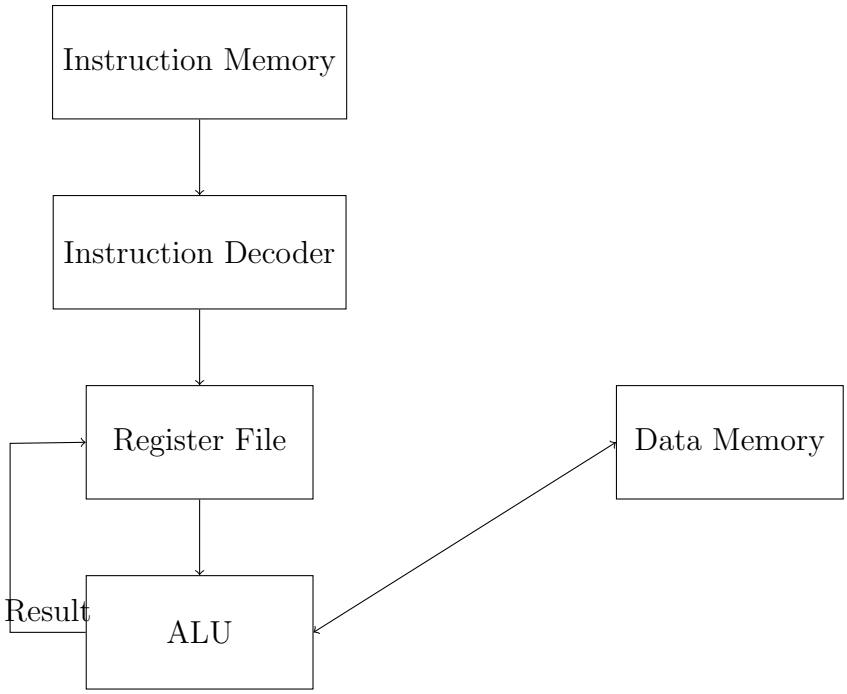

Figure 6.2: Integration of Register File within the CPU

- **readAddr01, readAddr02**: Addresses of the registers to be read.
- **writeAddr01**: Address of the register to be written.
- **writeResult**: Data to be written to the specified register.
- **readResult01, readResult02**: Data read from the specified registers.

6.1.7 Conclusion

The register file is a crucial part of the RISC-V processor, enabling efficient and fast access to frequently used data. Its design supports simultaneous reads and controlled writes, ensuring high performance and reliability. Understanding the operation and integration of the register file within the processor architecture is essential for implementing and optimizing RISC-V systems.

6.2 ALU

This module is a behavioral model of an Arithmetic Logic Unit (ALU) for a RISC-V processor implemented in SystemVerilog. This ALU performs various arithmetic and logic operations based on the provided control signals. The module takes two 64-bit input operands and produces a 64-bit result. It also generates a zero flag to indicate whether the result is zero.

6.2.1 Ports

- data01 (input, 64 bits): The first operand for the ALU operation.
- data02 (input, 64 bits): The second operand for the ALU operation.
- aluCtl (input, 4 bits): The control signal to specify the type of operation to be performed.
- aluOp (input, 2 bits): The high-level operation type signal, used for decoding the instruction class.
- zeroFlag (output, 1 bit): A flag that indicates whether the result is zero.
- result (output, 64 bits): The result of the ALU operation.

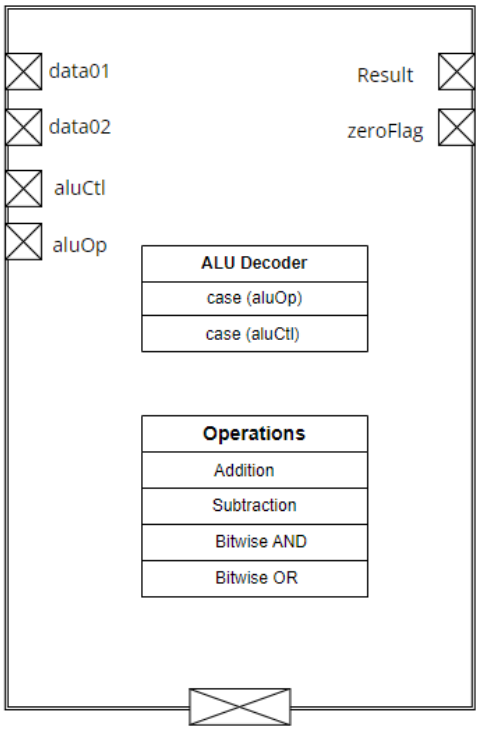

Figure 6.3: ALU block diagram

6.2.2 Parameters

- **loadOrStoreOp (2 bits):** Specifies load or store operation (2'b00).
- **branchOp (2 bits):** Specifies branch operation (2'bx1).
- **rTypeOp (2 bits)**: Specifies R-type operation (2'b1x).
- **store (4 bits):** Specifies store operation (4'b0010).
- **load (4 bits):** Specifies load operation (4'b0010).
- **branchIfEqual (4 bits):** Specifies branch if equal operation (4'b0110).
- **ariAdd (4 bits):** Specifies addition operation (4'b0010).
- **ariSub (4 bits):** Specifies subtraction operation (4'b0110).
- **bitAnd (4 bits):** Specifies bitwise AND operation (4'b0000).
- **bitOr (4 bits):** Specifies bitwise OR operation (4'b0001).

6.2.3 ALU Operation

The ALU operation is decoded based on the aluOp and aluCtl signals. The following table summarizes the operations.

`aluOp`	`aluCtl`	Operation	Description
`2'b00`	`4'b0010`	Load/Store	`result = data01 + data02`
`2'b01`	`4'b0110`	Branch if Equal	`result = data01 - data02`
`2'b10`	`4'b0010`	Addition (R-type)	`result = data01 + data02`
`2'b10`	`4'b0110`	Subtraction (R-type)	`result = data01 - data02`
`2'b10`	`4'b0000`	Bitwise AND (R-type)	`result = data01 & data02`
`2'b10`	`4'b0001`	Bitwise OR (R-type)	`result = data01 | data02`
Default	N/A	Default Operation	`result = 64'hxxxxxxxxxxxxxxxx`

Figure 6.4: ALU operation truth table

6.2.4 Detailed Functionality

1. Load/Store Operation (aluOp = 2'b00)
    - Performs addition of data01 and data02.
    - Sets zeroFlag if the result is zero.
2. Branch Operation (aluOp = 2'bx1)
    - Performs subtraction of data01 and data02.
    - Sets zeroFlag if the result is zero, indicating equality.
3. R-type Operation (aluOp = 2'b1x)
    - Addition (aluCtl = 4'b0010): Adds data01 and data02.

- Subtraction (aluCtl = 4'b0110): Subtracts data02 from data01.
- Bitwise AND (aluCtl = 4'b0000): Performs bitwise AND between data01 and data02.
- Bitwise OR (aluCtl = 4'b0001): Performs bitwise OR between data01 and data02.
- Sets zeroFlag if the result of any operation is zero.

4. Default Operation
   - Sets result to 64'hxxxxxxxxxxxxxxxx.
   - Sets zeroFlag to 0.

6.3 Instruction memory

Overview

To execute any operation, the first step involves fetching the instruction from memory. To prepare for the next instruction, we increment the program counter so it points to the subsequent instruction's address, typically advancing by 4 bytes. This setup forms a datapath that handles instruction fetching and program counter incrementation for sequential instruction execution.

Design Requirements

- **Memory Size:** 128 bytes (32 32-bit instructions).
- **Data Bus Widths:**
  - **Program Counter (PC):** 64 bits
  - **Flash Instruction Input (flashInstruction):** 8 bits
  - **Instruction Output:** 32 bits
  - **Program Counter Bypass Output (pcBypass):** 64 bits

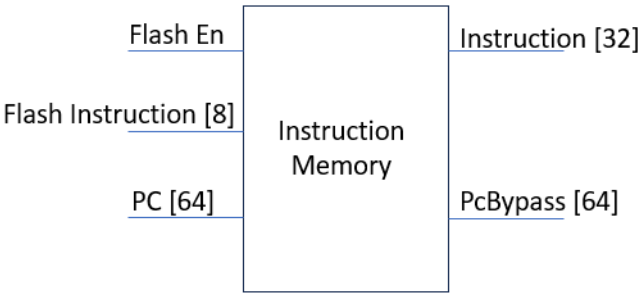

Figure 6.5: Instruction Memory

Implementation Approach

The implementation approach for the instructionMemory module involves the following key components and strategies:

- **Memory Array:** A 128-byte internal memory array (instructionMem) is used to store the instructions. Each byte in this array represents a portion of the stored instructions.
- **Conditional Operation:** The module uses conditional logic to switch between flash programming mode and instruction fetch mode based on the state of the flashEn signal. This is achieved through a combination of always blocks and conditional assignments.
- **Byte Concatenation:** During instruction fetch mode, the module concatenates four consecutive bytes from the memory array to form a 32-bit instruction. This ensures that the CPU can retrieve complete instructions for execution.
- **Address Management:** The pcBypass output provides an alternative address for the program counter when flash programming is active. This helps in managing the program flow during memory updates.

Detailed Functioning

- **Program Counter Bypass (pcBypass):** Set to 64'dX when flashEn is high, indicating an undefined state and halting normal PC operation.
- **Conditional Operation:** Writes flashInstruction[7:0] into instructionMem at the PC-specified address.Sets Instruction[31:0] to 32'dX during programming.
- **Instruction Memory Read (flashEn low):** Populates instruction with a 32-bit instruction fetched from instructionMem.
  Constructs the fetched instruction by concatenating four 8-bit segments from instructionMem.

6.4 Data Memory

Overview

In computer architecture, data memory is used to store data that is read from or written to by the processor. Data memory is typically implemented using an array of memory cells, each capable of holding a fixed amount of data, such as a byte or word. Key operations associated with data memory include:

- **Read Operation:** Retrieving data from a specific memory address.
- **Write Operation:** Storing data at a specific memory address.
- **Addressing:** Using an address to specify the location of data within the memory.

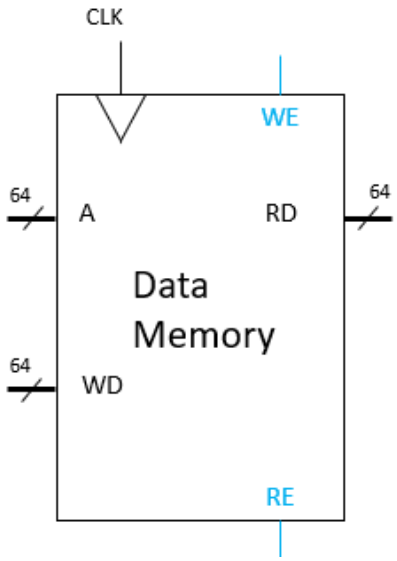

Figure 6.6: Data Memory

Implementation

Parameterization:

- **Data Width:** The width of the data bus, which determines how much data can be read from or written to the memory in one operation. In the provided code, this is set to 64 bits.
- **Address Width:** The width of the address bus, which determines the number of addressable memory locations. Here, it's set to 64 bits.

Input/Output Interfaces:

- **clk (input):** Clock signal to synchronize operations.
- **memWriteEnable (input):** Control signal to enable writing to memory.
- **memReadEnable (input):** Control signal to enable reading from memory.
- **address (input):** 64-bit address bus specifying the memory location for read/write operations.
- **writeData (input):** 64-bit data bus carrying the data to be written to memory.
- **readData (output):** 64-bit data bus outputting the data read from memory.

Memory Array:

- The memory array is 128 bytes, organized as an array of 8-bit registers.

Read and Write Operations:

- Write operation should be synchronous with the clock signal.
- Read operation should be combinational based on the control signal.

- Data should be written and read in chunks of 64 bits, spread across 8 consecutive memory locations.

Initialization:

- The memory array should be initialized to zero during simulation.

6.5 Decoder

The decoder module is a crucial component in the control unit of a processor, responsible for interpreting 32-bit instructions and generating control signals that manage the processor's operations. The inputs to the decoder module include a 32-bit instruction and a 64-bit program counter (pc). The instruction to be decoded comprises the opcode, function codes, and operand addresses, while the program counter indicates the address of the instruction. The outputs of the decoder include several control signals (`branch`, `memRead`, `memWrite`, `memToReg`, `aluSrc`, `writeReg`), ALU control signals (`aluOp`, `aluCtl`), register addresses (`readAddr01`, `readAddr02`, `writeAddr01`), the updated program counter (`newPC`, `curretPC`), and a 64-bit immediate value (`immidiateValue`).

6.5.1 Truth Table

The truth table below summarizes the control signals generated by the decoder module for each instruction type.

Instruction	**ALUSrc**	**Mem to Reg**	**Reg Write**	**Mem Read**	**Mem Write**	**Branch**	**ALU Op1**	**ALU Op0**
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Table 6.1: Setting of the control lines is completely determined by the opcode fields of the instruction.

6.5.2 Instruction Types

The decoder module handles four instruction types: R-Type, Memory Load, Memory Store, and Branch instructions.

R-Type Instructions R-Type instructions perform arithmetic and logical operations between registers. The control signals generated for R-Type instructions indicate no branching or memory access, with the ALU using register operands and writing back to the register file. Examples of R-Type instructions include ADD, SUB, AND, and OR. The ALU operations for R-Type instructions are specified by `aluOp` and `aluCtl` based on the function codes.

Memory Load Instructions Memory Load instructions load data from memory into a register. The control signals for these instructions enable memory read, with the ALU performing address calculation and the data being written back to the register. An example of a Memory Load instruction is LW (load word). The immediate value for Memory Load instructions is extracted and sign-extended from the instruction.

Memory Store Instructions Memory Store instructions store data from a register into memory. The control signals for these instructions enable memory write, with the ALU performing address calculation and no register write back. An example of a Memory Store instruction is SW (store word). The immediate value for Memory Store instructions is extracted from the instruction and combined into two parts.
Branch Instructions Branch instructions perform conditional jumps based on register comparisons. The control signals for these instructions enable branching, with the ALU performing subtraction to compare register values. An example of a Branch instruction is BEQ (branch if equal). The immediate value for Branch instructions is extracted and sign-extended from the instruction.
Control Signal Generation Control signals are generated based on the instruction's opcode. For R-Type instructions, the signals are set for arithmetic or logical operations with register operands. For Memory Load instructions, the signals enable memory read and data transfer to a register. For Memory Store instructions, the signals enable memory write from a register. For Branch instructions, the signals configure the ALU for comparison and enable branching. The ALU control signals (`aluOp` and `aluCtl`) are derived from the opcode and function codes. For R-Type instructions, ALU control is based on `funct7` and `funct3`. For Memory Load and Store instructions, the ALU is set to perform address calculation (addition). For Branch instructions, the ALU is set to perform subtraction for comparison.
Immediate Value Extraction Immediate values are extracted and sign-extended to 64 bits for memory access and branch instructions, facilitating address calculation or branch target computation. This process ensures that the instructions are accurately interpreted and executed by the processor.

6.5.3 Decoder Functioning

- The decoder module is responsible for interpreting the instruction and generating control signals for other components within a CPU. This functionality is depicted in the block diagram.
- **Instruction Fetch:** The 32-bit instruction, fetched from memory, serves as the input that needs to be decoded.
- **Program Counter (PC):** Holds the current address of the instruction being executed.
- **Opcode Decoder:** Extracts the opcode from the instruction to determine the type of instruction and generate the appropriate control signals.
- **Function Decoder:** Extracts the function codes (`funct7` and `funct3`) from the instruction to specify the particular operation (e.g., ADD, SUBTRACT, AND, OR) for R-type instructions.

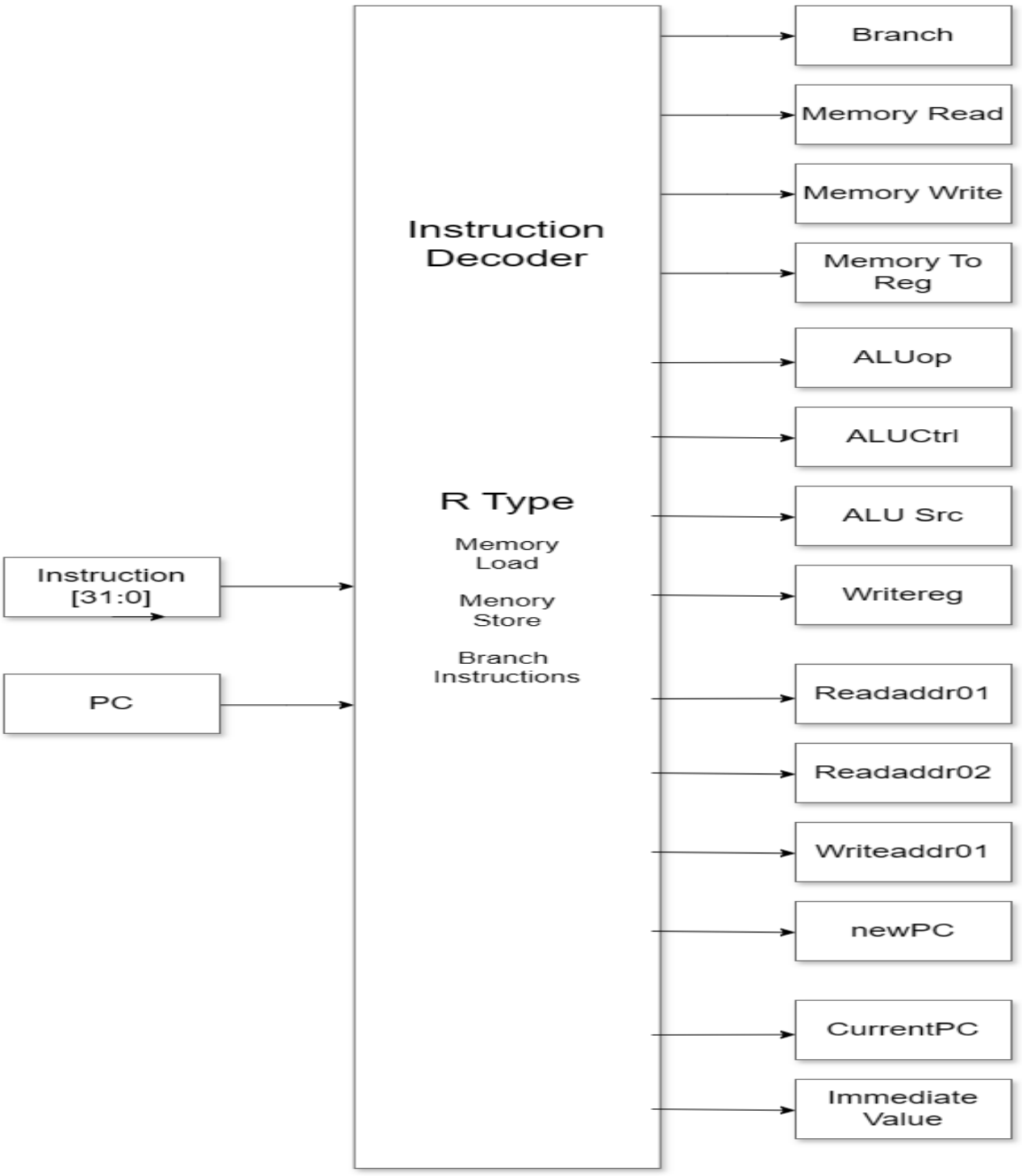

Figure 6.7: Instruction Decoder

- **Read Address 1 (rs1) and Read Address 2 (rs2):** Extract the respective fields from the instruction, specifying the source registers for the operation.
- **Write Address (rd):** Extracts the field indicating the destination register for the operation's result.
- **Immediate Value Extraction:** Handles the extraction of immediate values from the in-

struction for relevant operations.

- **Current PC:** Represents the address of the current instruction.
- **New PC:** Incremented by 4, points to the next instruction.
- **Control Unit:** Generates control signals based on the opcode, including signals for Branch, MemRead, MemWrite, MemToReg, ALUSrc, and WriteReg.
- **ALU Control Unit:** Produces ALU control signals (`ALUOp` and `ALUCtl`) derived from the opcode and function codes. Specific control signals such as Branch Control manage branching operations, Memory Read Control oversees memory read operations, Memory Write Control handles memory write operations, Memory to Register Control determines whether data from memory or the ALU is written to the register, ALU Source Control decides if the second ALU operand is a register value or an immediate value, and Write Register Control governs whether the result is written to a register.
- **Arithmetic Logic Unit (ALU):** Performs arithmetic or logical operations as specified by the control signals.
- **Register File:** Holds the CPU registers, interfacing with read and write addresses to access or store data.
- This comprehensive setup ensures the decoder module functions effectively, interpreting instructions and coordinating the necessary control signals for proper CPU operation.

6.6 Branch PC Generator

The `branchPCGenerator` module is a crucial component in the control unit of a processor, implemented using Verilog, a hardware description language. This module determines the next value of the program counter (PC), which is essential for guiding the processor's instruction flow. It takes several inputs, including control signals and current state values, to decide whether to branch to a new address or continue sequentially. The primary inputs to this module are `branchControl`, `zeroFlag`, `newPC`, `curretPC`, and `immidiateValue`, while the output is the `nextPC`.

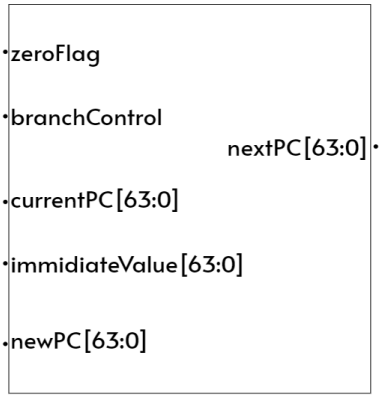

Figure 6.8: Branch PC generator

The inputs `branchControl` and `zeroFlag` are critical for branch decision-making. `branchControl` indicates whether a branch instruction is active, and `zeroFlag` reflects the outcome of a condition check, typically whether a computed result is zero. These signals are combined using a logical AND operation to produce the `branchDecision` signal. This signal is high only when both the branch control is asserted and the zero flag condition is met, indicating that a branch should be taken.

The `nextPC` value is calculated based on the `branchDecision` signal. If `branchDecision` is high, indicating that a branch should be taken, `nextPC` is set to the current PC (`curretPC`) plus an immediate value (`immidiateValue`) shifted left by one bit. The left shift operation effectively multiplies the immediate value by two, a common technique used in instruction set architectures for byte addressing with word-aligned instructions. This operation ensures that the processor can jump to the correct instruction address specified by the branch.

If `branchDecision` is low, indicating that the branch condition is not met, the `nextPC` is set to `newPC`, which represents the next sequential instruction address. This conditional assignment, resembling a multiplexer function, ensures that the processor continues to execute instructions in the correct order, either by branching to a new address or moving to the next sequential address. Thus, the `branchPCGenerator` module enables efficient control flow management within the processor, ensuring that branch instructions are correctly handled based on the given conditions.

6.7 Top Level

6.7.1 Introduction

In this section, we integrate all previously explained design elements to create a unified single-cycle processor. The top-level module includes specific multiplexers labeled as M1, M2, and M3. M1 controls the datapath for the second operand of the ALU, M2 manages the write-back operation to the register file, and M3 adjusts the datapath of the PC register during flash operations. These multiplexers are crucial as they enable the processor to adapt its datapaths based on the instruction being executed, allowing a variety of operations to be performed. We will also understand the instruction memory flash mechanism in detail.

This section focuses on implementing the single-cycle processor's datapath flow described in the product overview. It will detail how different instruction classes are executed, emphasizing the role of these multiplexers in enabling diverse operations within the processor.

- Datapath for R-Type instructions.
- Datapath for load/store instructions.
- Datapath for branch-if-equal instruction.

6.7.2 Datapath for R-Type instructions

The figure below shows the datapath operation (Highlighted in red) for an R-type instruction such as `add x1, x2, x3`. Although everything occurs in one clock cycle, we can think of several steps to execute the instruction.

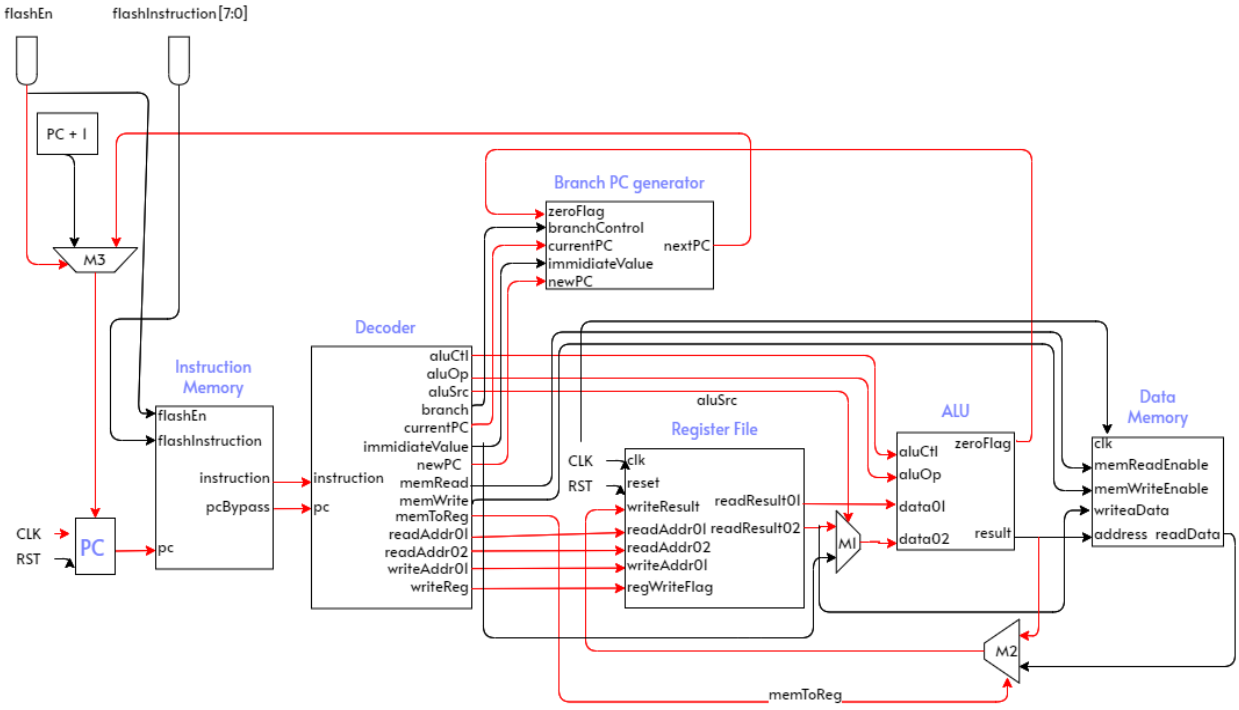

Figure 6.9: R-Type Datapath

- On each rising edge of the clock pulse, if the processor is not in flash mode (flashEn is zero), the PC increments by 4.
- The instruction located at the updated PC (oldPC + 4) is fetched from the instruction memory and sent to the decoder for decoding. This process generates necessary signals and addresses required for executing the instruction.
- For the current instruction "add x1, x2, x3", values from register file addresses x2 and x3 are fetched and sent to the ALU.
- As "add" is an R-Type instruction, aluSrc signal is zero, and M1 mux directs readResult02 to the ALU as the second operand.
- The ALU performs an arithmetic addition based on signals aluCtl and aluOp from the decoder. The resulting value is written back to register file location x1. memToReg flag is zero, and regWriteFlag is High, enabling this write operation.
- Since the branch signal is zero, the branch PC generator forwards the newPC (oldPC + 4) to nextPC. This updated value is loaded into the PC register at the next rising edge of the clock cycle.

6.7.3 Datapath for load/store instructions

The figure below shows the datapath operation (Highlighted in red) for a load instruction such as `ld x1, offset(x2)`. Although everything occurs in one clock cycle, we can think of several steps to execute the instruction.

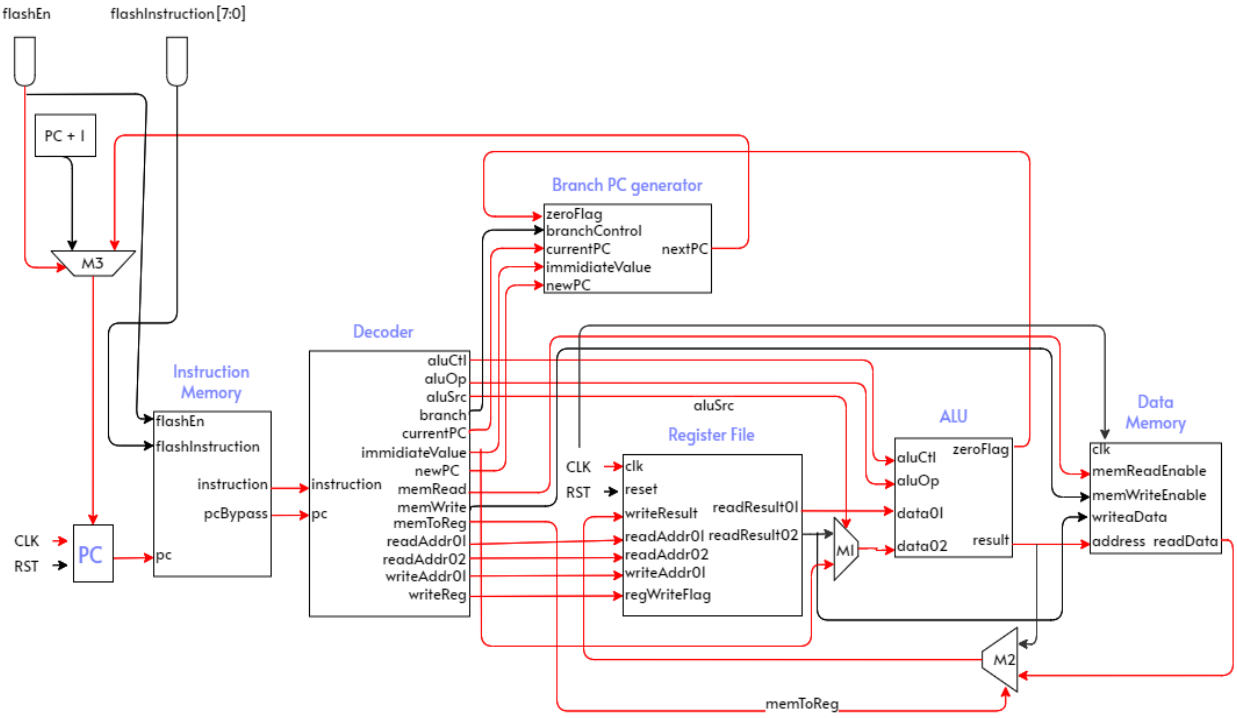

Figure 6.10: load/ store Datapath

- At each rising edge of the clock pulse, the PC increments by 4 when the processor is not in flash mode (flashEn is zero).
- The instruction located at the new PC (oldPC + 4) is fetched from the instruction memory and passed to the decoder for decoding. This process generates the necessary signals and addresses required for executing the instruction.
- For the current instruction "ld x1, offset(x2)", the value from register file address x2 is sent to the ALU. The ALU's second operand is a 12-bit immediateValue from the decoder, as aluSrc signal is High, and mux M1 adjusts the datapath accordingly compared to the previous instruction.
- The ALU adds these operands as directed by signals aluCtl and aluOp, producing the memory address from which data needs to be fetched.
- Since the memory read operation is clock-independent, the fetched data is written back to the register file during the falling edge of the same clock cycle. Here, mux M2 alters the datapath for the write-back operation to the register file, since memToReg signal is High. The fetched value is stored in register x1 of the register file.
- Meanwhile, with branch signal at zero, the branch PC generator forwards newPC to nextPC (oldPC + 4). This updated value is loaded into the PC register at the next clock cycle's rising edge.

Store operations will also have an almost similar datapath to store data from the reg file into the data memory.

6.7.4 Datapath for branch-if-equal instructions

The figure below shows the datapath operation (Highlighted in red) for a branch-if-equal instruction such as `beq x1, x2, offset`. It operates much like an R-format instruction but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address. Although everything occurs in one clock cycle, we can think of several steps to execute the instruction.

- During each rising edge of the clock pulse, the PC increments by 4 when the processor is not in flash mode (flashEn is zero).
- The instruction located at the updated PC (oldPC + 4) is fetched from the instruction memory and forwarded to the decoder for decoding. This step generates the necessary signals and addresses required to execute the instruction.
- Two registers, x1 and x2, are accessed and read from the register file.
- The ALU subtracts one data value from another, both obtained from the register file. If the result of this operation is zero, the zeroFlag is set High. If the branch flag is also High, indicating a branch instruction, the branch PC generator computes the branch target address by adding the current PC value to the sign-extended immediateValue (offset) extracted from the instruction (12 bits, left shifted by one).

- The new branch target address is written into the PC register during the rising edge of the next clock cycle. The instruction memory then retrieves and executes the instruction located at this branch target address. This process occurs independently of the clock due to the nature of instruction memory read operations.

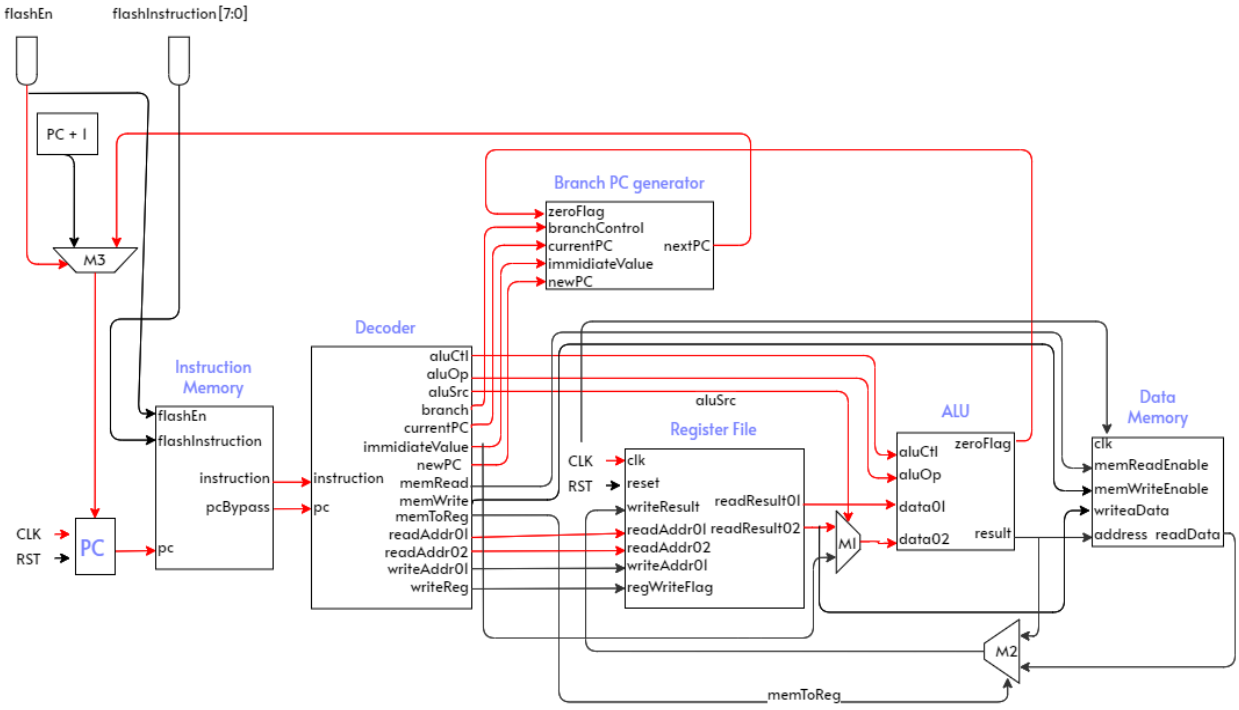

Figure 6.11: Branch-if-equal Datapath

These operations encompass all potential datapaths the processor can generate, providing a comprehensive understanding of the functional principles underlying the single-cycle RISC-V processor.

6.7.5 Datapath for processor flash operation

The diagram below illustrates the datapath operation (highlighted in red) for the processor's instruction memory during a flash operation. In the RISC-V architecture, each instruction is 32 bits wide, while the instruction memory operates on a byte-addressable basis. Initially, a hex file must be generated from high-level languages or assembly code using a dedicated toolchain. Once this hex file is prepared, the processor can be flashed by simulating the process typically handled by an onboard debugger like JTAG or a TAP Controller.

In this scenario, although no specific debugger is employed, the process involves writing one byte at the corresponding PC location in the instruction memory from the hex file. If the flashEn signal is High, the PC increments by one at the rising edge of each clock cycle, enabling the loading of one byte of instruction per clock cycle.

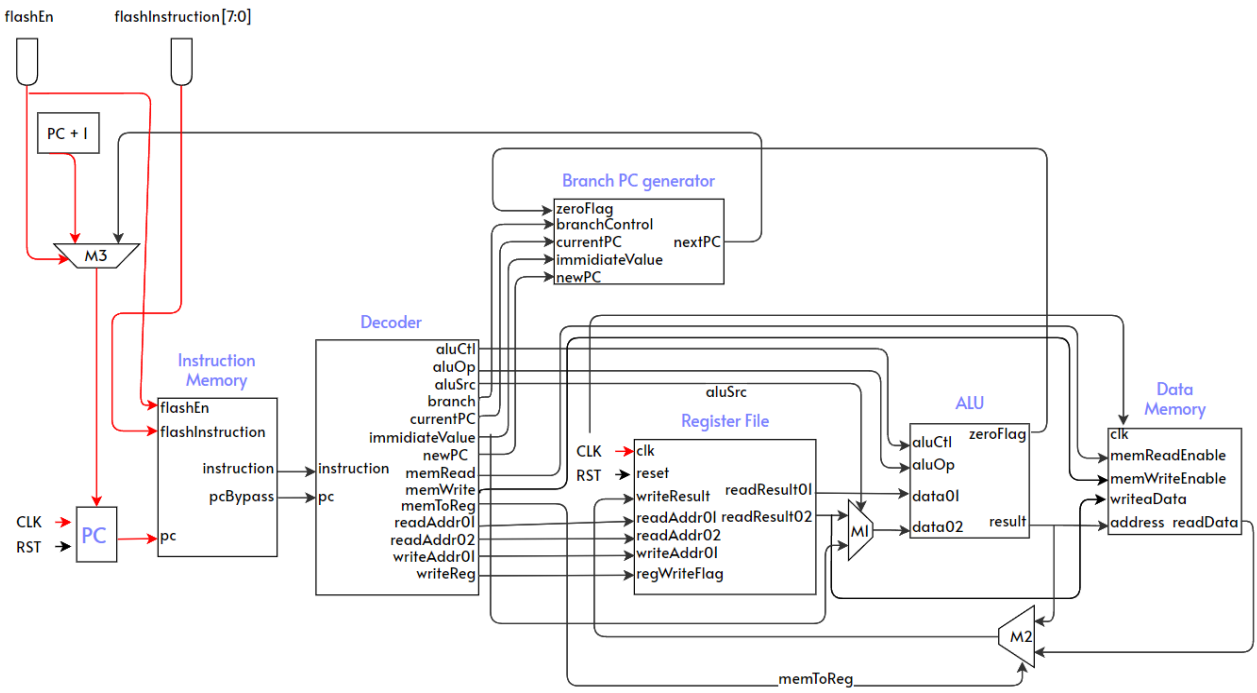

Figure 6.12: Flash Datapath

After completing the flashing operation, deactivate the flashEn signal and reset the processor. Subsequently, the processor will transition to normal mode and begin executing the instructions stored in the instruction memory that were loaded during the flashing process.

7. Simulation results

7.1 Flash instruction memory

The diagram below demonstrates the procedure of flashing the instruction memory. Throughout this process, the Program Counter (PC) advances by one on each clock cycle, and a byte of data from the hex file, containing instructions, is stored in the byte-addressable instruction memory. Flashing is active only when the flashEn signal is high and the reset signal is low, as indicated in the output. This operation of flashing takes about 290 nanoseconds to write 28 bytes of 7 RISC-V instructions into the instruction memory.

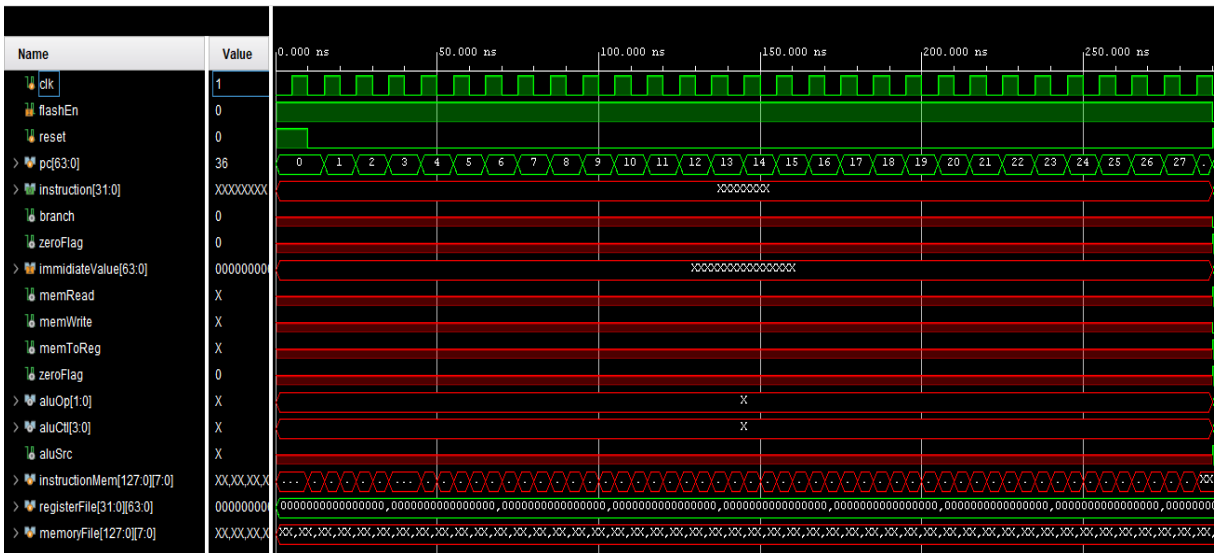

Figure 7.1: Instruction Memory Flashing Process

7.2 Processor simulation

After the instructions have been flashed into memory, the processor can be reset, causing the PC to reset to zero. Subsequently, the processor begins fetching instructions from the memory starting at address zero. This behavior is depicted in the image below. To facilitate explanation, the following timelines are considered:

- **Time Stamp (290ns – 300ns):** The reset signal gets pulsed, the PC becomes zero, and fetches the first instruction `ld r1 0(r0)` where `r0=0` with the hex code of `0x00003083` gets executed which fetches a double word from `0x00` data memory location into the `r1` register of the register file.

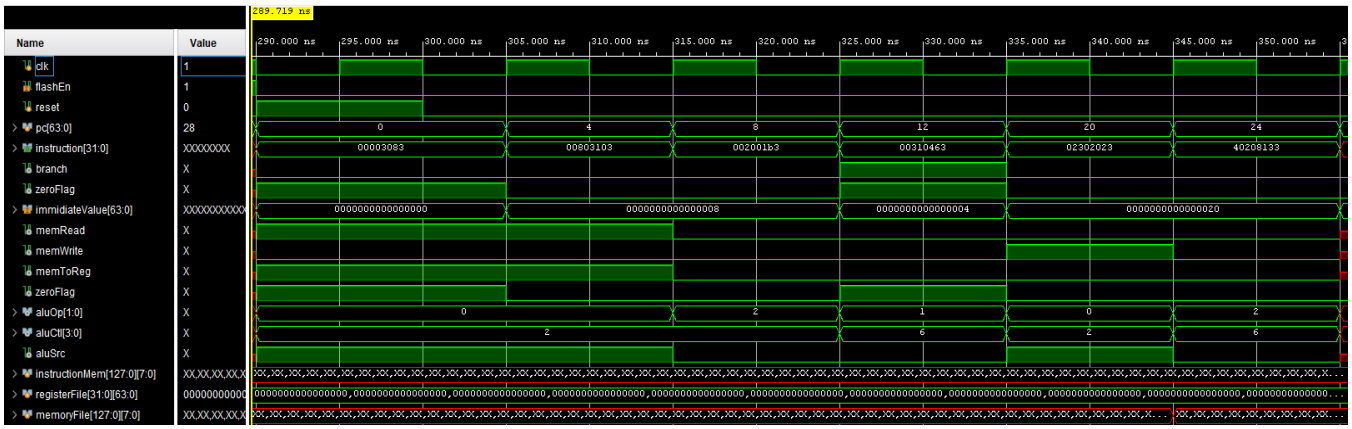

Figure 7.2: Processor Simulation

- **Time Stamp (300ns – 310ns):** Now `PC=4` and the next instruction gets fetched `ld r2 8(r0)` where `r0=0` with the hex code of `0x00803103` gets executed which fetches a double word from `0x08` data memory location into the `r2` register of the register file.
- **Time Stamp (310ns – 320ns):** Now `PC=8` and the next instruction gets fetched `add r3 r2 r0` with the hex code of `0x002001b3` gets executed which adds the data present in `r0` and `r2` registers and then stores the result in the `r3` register.
- **Time Stamp (320ns – 330ns):** Now `PC=12` and the next instruction gets fetched `beq r3 r2 4(offset)` with the hex code of `0x00310463` gets executed which checks if `r3==r2`, if yes then it gets branched to target address `(currentPC+4) + 4(offset)`. So `newPC = 20`. Also, observe imidiateValue during branch.
- **Time Stamp (330ns – 340ns):** Now `PC=20` and the next instruction gets fetched `sd r3 32(r0)` with the hex code of `0x02302023` gets executed which stores data present in `r3` at `r0+32`.
- **Time Stamp (340ns – 350ns):** Now `PC=24` and the next instruction gets fetched `sub r2 r2 r1` with the hex code of `0x40208133` gets executed which subtracts `r2 { r1` and stores back data into `r2`.

Bibliography

[1] Computer Organization and Design: The Hardware/Software Interface, February 27, 2017,RISC-V Edition,by Hennessy, J. L.

[2] Digital Design and computer architechture by David Money Harris, May 2, October 2008.