BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370 - Computer Architecture

Project2 Report

Multi Cycle RISC Processor

Prepared By:

Partners Name	*Partners ID*
Donia Yasin	1201831
Tala Alswaitti	1191068
Eman Naji	1200358

Instructor: Dr. Aziz Qaroush

Section 2

January 28, 2024

Abstract

This report highlights the development and execution of a Multi-Cycle Processor tailored for a particular RISC instruction set. The design phase began with an in-depth examination of each instruction, leading to the identification and creation of essential components for the processor's architecture. These components were subsequently integrated into the overall processor design. During the implementation stage, the focus was on generating control signals for each component, emphasizing the correct timing and sequencing to ensure seamless instruction execution. The processor underwent rigorous testing to confirm its functionality, with each instruction being individually tested for accuracy and scenarios being applied to assess overall performance. The entire process, from design and implementation to testing, aimed to ensure the processor's effectiveness in accurately and efficiently executing the RISC instruction set.

Table of Contents

List Of Figures

List Of Tables

1. Introduction

The Multi-Cycle RISC Processor represents a CPU design adhering to RISC (Reduced Instruction Set Computer) architecture principles. RISC processors are characterized by a streamlined set of instructions, which are simple and consistent, enhancing processing speed and efficiency. In a multi-cycle RISC processor, the execution of each instruction spans several clock cycles, with each cycle focusing on a distinct phase of the instruction execution, such as fetching, decoding, executing, accessing memory, and write-back. This segmented approach in the instruction execution enhances the design's modularity and flexibility. [1]

One of the primary benefits of the multi-cycle RISC architecture is the simplification of the processor's control logic. By breaking down the instruction execution into smaller, more manageable stages, the complexity of the processor design is reduced, potentially leading to higher operational speeds. Additionally, the multi-cycle structure allows for the incorporation of more intricate instructions within the RISC framework. Although RISC processors typically utilize a limited assortment of basic instructions, the multi-cycle model accommodates the incorporation of complex instructions by dispersing their execution across multiple cycles, thereby enhancing the processor's functionality and adaptability. [1]

In this project, develop and test a simple multi-cycle RISC processor was created using Verilog HDL. The processor is designed to support a limited set of instructions encompassing memory operations, logical operations, arithmetic computations, and Stack.

The control and the Datapath basically contains five stages, and between each single stage a buffer that stores, and transfer the only required signals to the next stage depending on the instruction set architecture. The number of clock cycles in this multi-cycle CPU varies depending on which instruction is fetched from memory and is not constant.

2. Design and Implementation

2.1. Overall design

The multi-cycle design strategy is utilized to develop a processor capable of handling diverse instructions. This design involves segmenting the data path into stages, each aligned with a specific clock cycle. During each clock cycle, a distinct part of the instruction is executed, and the outcomes are stored in the relevant register or memory location. To reduce redundancy, data required for upcoming clock cycles is stored in state elements at the conclusion of each clock cycle. Programmer-visible state elements, such as the register file, program counter (PC), memory, and Stack Pointer (SP), retain data necessary for future clock cycle instructions. However, data needed by the same instruction in subsequent cycles is stored in additional registers.

The approach to developing the data path for this multi-cycle RISC processor revolves around breaking down instruction execution into sequential steps rather than attempting to complete all actions within a single cycle. The comprehensive data path design integrates various components, including registers, an ALU, multiplexers, memory, a control unit, and a Stack Pointer. These components are chosen and interconnected to optimize hardware utilization and facilitate functional unit sharing within the execution of a single instruction.

Strategically placed additional registers are based on the alignment of combination units within a single clock cycle and the data required for correct instruction execution in subsequent cycles. In this multi-cycle design, each clock cycle is designed to accommodate at most one operation, such as instruction memory access, data memory access, register file access, or ALU operation. To prevent timing conflicts and ensure precision, data generated by these functional units is stored in temporary registers for use in subsequent cycles. This approach minimizes the necessity for redundant data copying and mitigates the risk of using incorrect values.

2.2. Instruction Set and Formants

In this instruction set architecture (ISA), as depicted in table 1, there are a total of 17 instructions. These are categorized into four types: 3 R-type (register-type) instructions, 9 I-type (immediate-type) instructions, 3 J-type (jump-type) instructions, and 2 S-type (special-type) instructions. A corresponding table details each instruction's classification, along with its opcode value and description in Register Transfer Notation (RTN). This ISA, which the Datapath is designed to support, encompasses a diverse range of instructions as outlined.

Table 1: Instruction Set

No.	Instr	Meaning	Opcode Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	000000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	000001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	000010
I-Type Instructions			
4	ANDI	Reg(Rd) = Reg(Rs1) & Imm^{16}	000011
5	ADDI	Reg(Rd) = Reg(Rs1) + Imm^{16}	000100
6	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16})	000101

7	LW.POI	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16})
		Reg[Rs1] = Reg[Rs1] + 1
			000110

8	SW	Mem(Reg(Rs1) + Imm^{16}) = Reg(Rd)	000111

9	BGT	if (Reg(Rd) > Reg(Rs1))
		Next PC = PC + sign_extended (Imm^{16})
		else PC = PC + 1
			001000

10	BLT	if (Reg(Rd) < Reg(Rs1))
		Next PC = PC + sign_extended (Imm^{16})
		else PC = PC + 1
			001001

11	BEQ	if (Reg(Rd) == Reg(Rs1))
		Next PC = PC + sign_extended (Imm^{16})
		else PC = PC + 1
			001010

12	BNE	if (Reg(Rd) != Reg(Rs1))
		Next PC = PC + sign_extended (Imm^{16})
		else PC = PC + 1
			001011

J-Type Instructions			
13	JMP	Next PC = {PC[31:26], Immediate^{26} }	001100
14	CALL	Next PC = {PC[31:26], Immediate^{26} } PC + 1 is pushed on the stack	001101
15	RET	Next PC = top of the stack	001110
S-Type Instructions			
16	PUSH	Rd is pushed on the top of the stack	001111
17	POP	The top element of the stack is popped, and it is stored in the Rd register	010000

The format of each type is as follows:

- R-Type (Register Type)

Opcode^{6}	Rd^{4}	Rs1^{4}	Rs2^{4}	Unused^{14}

Figure 2-1: R-Type Instruction Format

- I-Type (Immediate Type)

Opcode^{6}	Rd^{4}	Rs1^{4}	Immediate^{16}	Mode^{2}

Figure 2-2: I-Type Instruction Format

- J-Type (Jump Type)

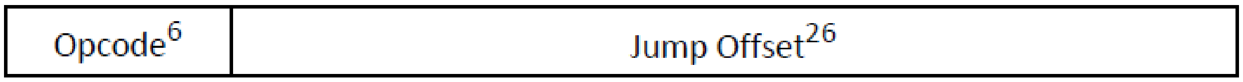


Figure 2-3: J-Type Instruction Format

- S-Type (Stack)

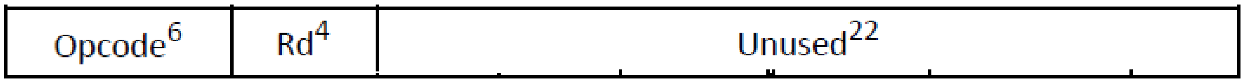


Figure 2-4: S-Type Instruction Format

2.3. Components Design

2.3.1. Memory

In constructing the memory for the multi-cycle processor, a Verilog module was developed to handle memory operations including read and write functions. The module is clocked, ensuring that memory read and write actions are synchronized with the system clock, enhancing stability and reliability during data transactions. The memory is segmented into instruction, data, and stack regions, with predefined sizes, to organize different types of information systematically. The memory array is word-addressable, allowing for 32-bit data handling, and is initialized with predefined values to represent instructions, data, and stack contents, demonstrating a preloaded state. This design allows for efficient memory utilization and facilitates the execution of various instructions by the processor's Datapath. The memory's read and write operations are controlled by dedicated signals, ensuring that data is correctly written to or read from the memory array, based on the incoming address and the state of the control signals. This approach provides a robust and flexible memory system tailored to the needs of the multi-cycle processor.

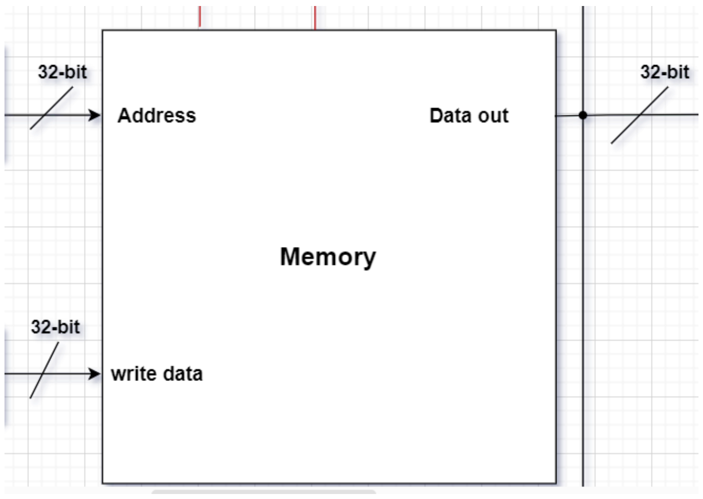


Figure 2-5: Memory block diagram

The memory module was rigorously tested using a Verilog test bench designed to simulate memory read and write operations across its instruction, data, and stack segments. The test bench initializes the system by setting the clock and control signals, and then proceeds through a series of steps: writing a test instruction to the instruction segment, reading it back, writing test data to the data segment, reading it back, writing test data to the stack segment, and finally, reading it back. The test bench employs a monitor to output the current time, the address accessed, the data written, and the data read, providing a comprehensive view of the memory's performance in real-time. The simulation results indicate that the memory behaves as expected, with correct data being stored and retrieved in each segment, thereby validating the functionality of the memory within the context of the processor's operation.

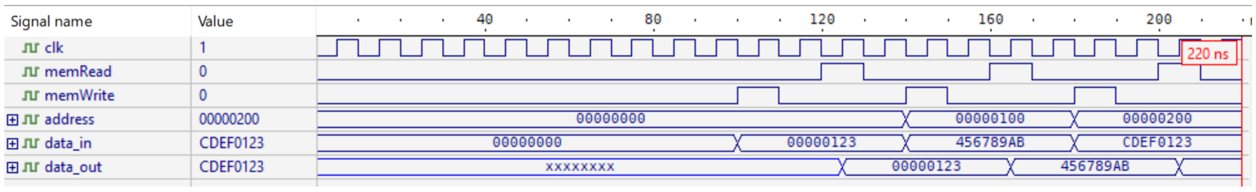
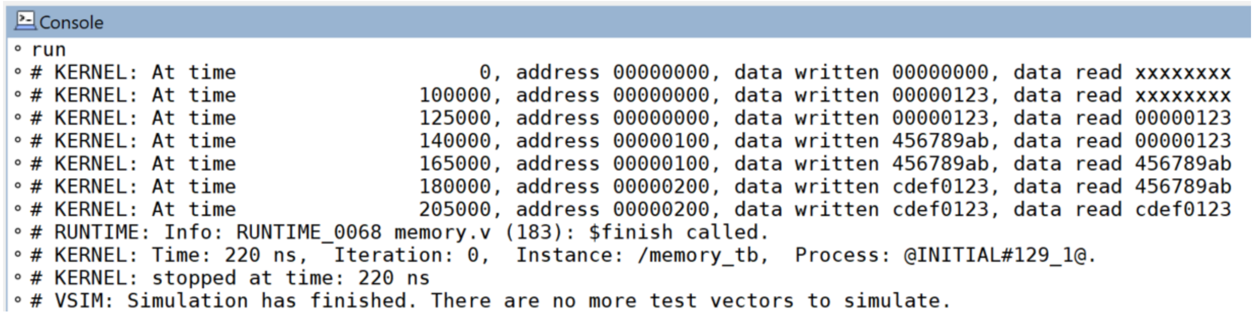


Figure 2-6: Memory simulation



```
Console
◦ run
◦ # KERNEL: At time                    0, address 00000000, data written 00000000, data read xxxxxxxx
◦ # KERNEL: At time               100000, address 00000000, data written 00000123, data read xxxxxxxx
◦ # KERNEL: At time               125000, address 00000000, data written 00000123, data read 00000123
◦ # KERNEL: At time               140000, address 00000100, data written 456789ab, data read 00000123
◦ # KERNEL: At time               165000, address 00000100, data written 456789ab, data read 456789ab
◦ # KERNEL: At time               180000, address 00000200, data written cdef0123, data read 456789ab
◦ # KERNEL: At time               205000, address 00000200, data written cdef0123, data read cdef0123
◦ # RUNTIME: Info: RUNTIME_0068 memory.v (183): $finish called.
◦ # KERNEL: Time: 220 ns,  Iteration: 0,  Instance: /memory_tb,  Process: @INITIAL#129_1@.
◦ # KERNEL: stopped at time: 220 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Figure 2-7: Memory simulation console

2.3.2. Instruction Register (IR)

The instruction register module, an integral component of the multi-cycle processor, was built to decode incoming 32-bit instructions and extract relevant parts based on the instruction type. The module is triggered on the positive edge of the clock and utilizes a control signal “ir_write” to enable the write operation. It supports R-type, I-type, J-type, and S-type instructions, with each type having unique opcode values defined as parameters within the module. Upon receiving an instruction, the module deciphers the opcode and assigns the corresponding parts of the instruction to output registers like opcode, rd, rs1, rs2, immediate, and jump_offset. This structured approach allows for the isolation of each component of an instruction, such as the destination register, source registers, immediate value, and jump offset, which are crucial for the execution phase in the processor's datapath.

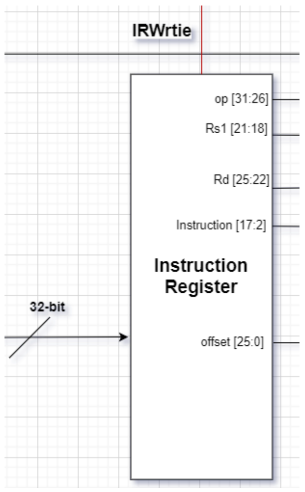
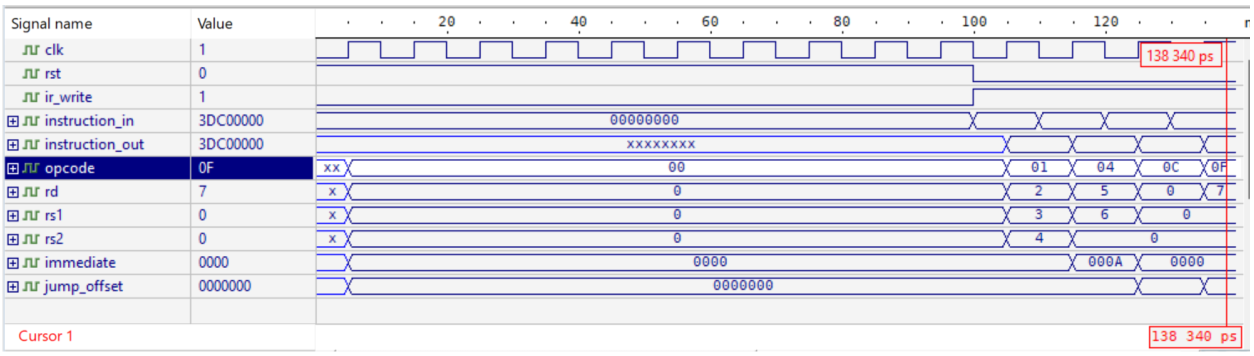


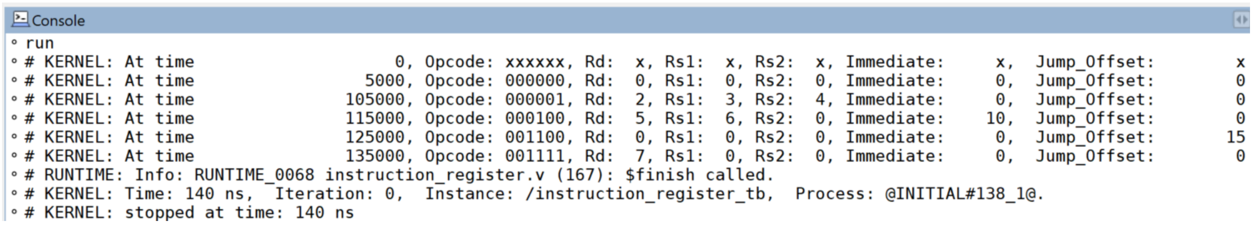
Figure 2-8: IR block diagram

The instruction register (IR) module of the multi-cycle processor was validated using a comprehensive test bench that simulates the IR's behavior in response to various types of instructions. The test bench generates a clock signal and initializes the module with a reset signal, followed by a sequence of test instructions encompassing R-type, I-type, J-type, and S-type formats. Each instruction type is inputted with specific opcode values and operands, and the test bench observes the corresponding decoded outputs, such as the opcode, register identifiers, immediate values, and jump offsets. The test bench actively monitors and reports the state of the IR outputs at every significant simulation time step, ensuring that the IR correctly interprets and outputs the components of each instruction type.



Signal name	Value
clk	1
rst	0
ir_write	1
instruction_in	3DC00000
instruction_out	3DC00000
opcode	0F
rd	7
rs1	0
rs2	0
immediate	0000
jump_offset	0000000
Cursor 1	

Figure 2-9: IR simulation



```
◦ run
◦ # KERNEL: At time                    0, Opcode: xxxxxx, Rd:  x, Rs1:  x, Rs2:  x, Immediate:     x,  Jump_Offset:       x
◦ # KERNEL: At time                 5000, Opcode: 000000, Rd:  0, Rs1:  0, Rs2:  0, Immediate:     0,  Jump_Offset:       0
◦ # KERNEL: At time               105000, Opcode: 000001, Rd:  2, Rs1:  3, Rs2:  4, Immediate:     0,  Jump_Offset:       0
◦ # KERNEL: At time               115000, Opcode: 000100, Rd:  5, Rs1:  6, Rs2:  0, Immediate:    10,  Jump_Offset:       0
◦ # KERNEL: At time               125000, Opcode: 001100, Rd:  0, Rs1:  0, Rs2:  0, Immediate:     0,  Jump_Offset:      15
◦ # KERNEL: At time               135000, Opcode: 001111, Rd:  7, Rs1:  0, Rs2:  0, Immediate:     0,  Jump_Offset:       0
◦ # RUNTIME: Info: RUNTIME_0068 instruction_register.v (167): $finish called.
◦ # KERNEL: Time: 140 ns,  Iteration: 0,  Instance: /instruction_register_tb,  Process: @INITIAL#138_1@.
◦ # KERNEL: stopped at time: 140 ns
```

Figure 2-10: IR simulation console

2.3.3. Memory Data Register

The Memory Data Register (MDR) module serves as a temporary storage unit within the multi-cycle processor architecture, designed to hold data that is being transferred between the processor and memory. This module is composed of a simple register that captures the input data on the rising edge of the clock signal. The 32-bit data_in port receives the data, and the 32-bit data_out port outputs the stored data. The MDR is crucial for operations such as fetching a data from memory or writing data back to memory, as it ensures data integrity by latching the data during a specific clock cycle. This design aligns with the multi-cycle nature of the processor, where different stages of instruction execution are distributed across multiple clock cycles, necessitating intermediate storage like the MDR to maintain data coherence throughout the execution process.

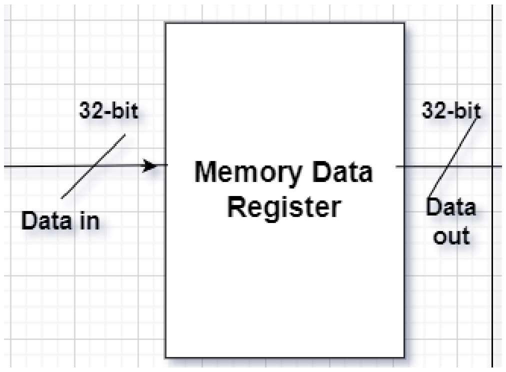


Figure 2-11: Memory Data Register block diagram

The Memory Data Register (MDR) was tested for its functionality through a structured test bench that simulates clock cycles and input data signals. The test bench applies a sequence of predefined 32-bit values to the data input, while a generated clock signal at 50MHz ensures the MDR latches the input data at the appropriate clock edges.

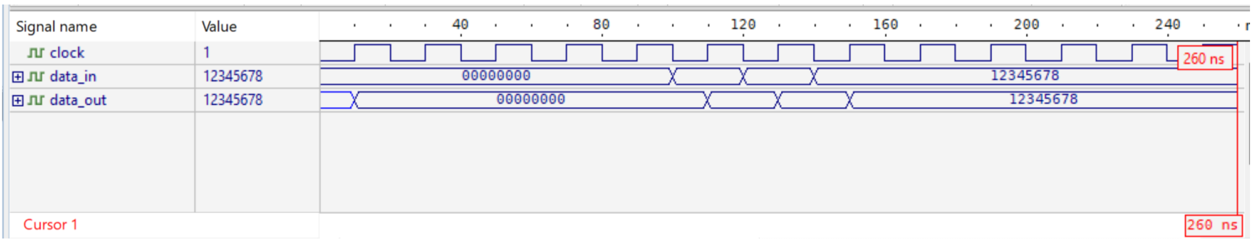
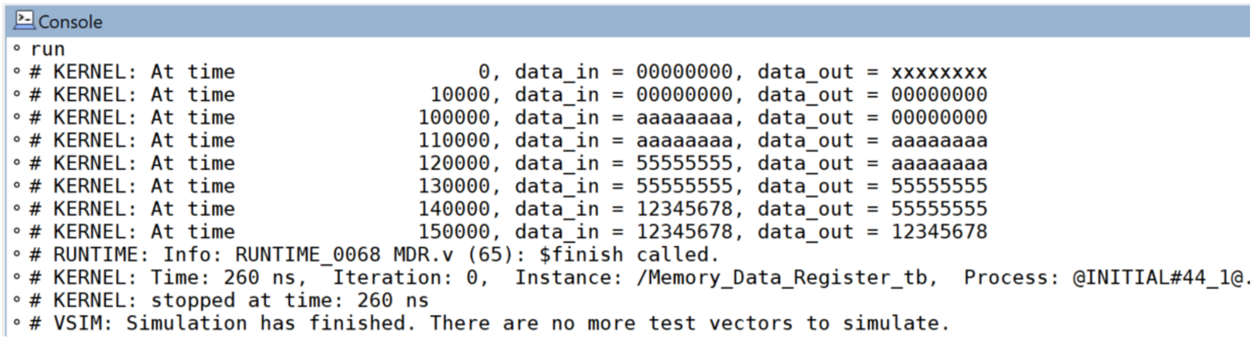


Figure 2-12: Memory Data Register simulation



```
Console
◦ run
◦ # KERNEL: At time                    0, data_in = 00000000, data_out = xxxxxxxx
◦ # KERNEL: At time                10000, data_in = 00000000, data_out = 00000000
◦ # KERNEL: At time               100000, data_in = aaaaaaaa, data_out = 00000000
◦ # KERNEL: At time               110000, data_in = aaaaaaaa, data_out = aaaaaaaa
◦ # KERNEL: At time               120000, data_in = 55555555, data_out = aaaaaaaa
◦ # KERNEL: At time               130000, data_in = 55555555, data_out = 55555555
◦ # KERNEL: At time               140000, data_in = 12345678, data_out = 55555555
◦ # KERNEL: At time               150000, data_in = 12345678, data_out = 12345678
◦ # RUNTIME: Info: RUNTIME_0068 MDR.v (65): $finish called.
◦ # KERNEL: Time: 260 ns,  Iteration: 0,  Instance: /Memory_Data_Register_tb,  Process: @INITIAL#44_1@.
◦ # KERNEL: stopped at time: 260 ns
◦ # VSIM: Simulation has finished. There are no more test vectors to simulate.
```

Figure 2-13: Memory Data Register simulation console

2.3.4. Register File

RegFile which is an essential component in digital systems, particularly in CPUs and other processing units. The Register File consists of 16 individual registers, each 32 bits wide, allowing for the storage and retrieval of 32-bit data values. The module is designed with two read ports (`RA1` and `RA2`) and two write ports (`RW1` and `RW2`), enabling simultaneous read and write operations in a single clock cycle. Read and write operations are controlled by separate addresses and enable signals (`RegWrite1` and `RegWrite2`), ensuring flexible and concurrent access to the register contents. During write operations, data is written to the specified registers (`Bus_W1` and `Bus_W2`) at the rising edge of the clock signal if the corresponding write enable signals are asserted. For read operations, data from the requested registers is made available immediately on the output lines (`Bus_A1` and `Bus_A2`) due to the combinational logic used, providing low latency access to register values. The initial block in the module ensures all registers are zeroed at startup, providing a known state for system initialization. A testbench (`RegFile_tb`) accompanies the Register File module to verify its functionality through various test cases, including writing to and reading from different registers, simultaneous reads and writes, and accessing registers that have not been written to, ensuring the module behaves as expected under different conditions.

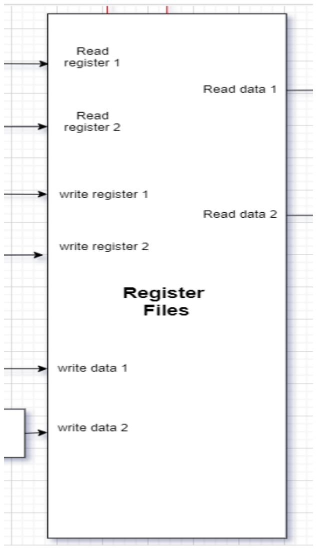
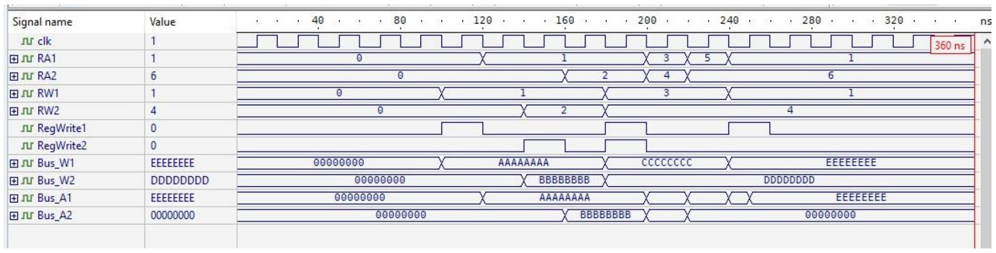


Figure 2-14: Register File block diagram

The testing results of our Register File unit are shown in the waveform bellow.



Signal name	Value
clk	1
RA1	1
RA2	6
RW1	1
RW2	4
RegWrite1	0
RegWrite2	0
Bus_W1	EEEEEEEE
Bus_W2	DDDDDDDD
Bus_A1	EEEEEEEE
Bus_A2	00000000

Figure 2-15: Register File simulation

2.3.5. Temporary Registers

- **Instruction registers (IR)** – holds the instruction after it's been pulled from memory.
- **Memory data register (MDR)** – temporarily holds data grabbed from memory until the next cycle.
- **A** – temporarily holds the contents of read register 1 until the next cycle.
- **B** – temporarily holds the contents of read register 2 until the next cycle.
- **ALUout** – temporarily holds the contents of the ALU until the next cycle. Note: every register is written on every cycle except for the instruction register.

2.3.6. Extender

The Extender component plays a pivotal role in digital systems and processors, facilitating the bit-width extension of immediate instruction values through sign-extension or zero-extension based on the ExtOp control signal. Sign-extension replicates the most significant bit of the 16-bit input (imm16) to maintain the value's sign in the extended 32-bit output (imm32), whereas zero-extension pads the upper bits with zeros, ensuring value preservation. This Verilog-implemented module efficiently adapts immediate values for wider-bit operations, with its functionality and reliability confirmed by a comprehensive testbench that assesses its handling of both positive and negative numbers through various test scenarios.

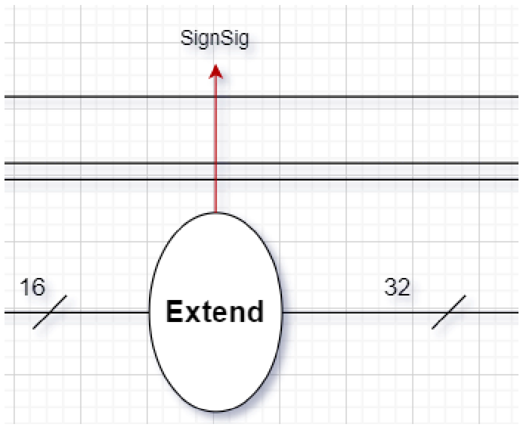


Figure 2-16: Extender block diagram

The testing results of our Extender unit are shown in the waveform bellow.

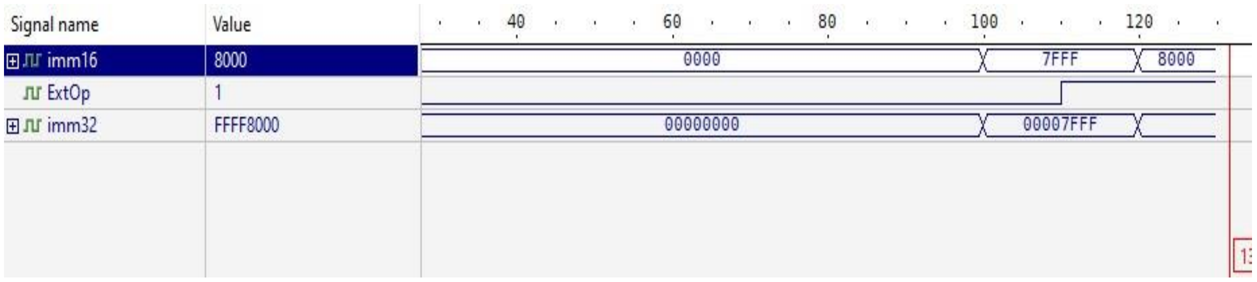


Figure 2-17: Extender simulation

2.3.7. Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a critical component of a computer's CPU responsible for performing arithmetic and logical operations. It handles operations like addition, subtraction, multiplication, division, as well as logical operations such as AND, OR, NOT, and XOR. The ALU operates on binary data and produces a result that is stored in a register or used as memory location. It is optimized for high-speed operation through specialized circuits and logic gates. The ALU communicates with other CPU components through input and output buses, allowing data to be fetched, processed, and written back. In this implementation, the ALU's components work continuously, but the specific output sent to the result bus depends on the instruction opcode being executed. The ALU also features zero and negative flags, which are set or cleared based on the zeroes and sign of the result, respectively. Additionally, the ALU includes carry and overflow flags, which indicate if there was a carry-out or borrow during arithmetic operations, and if the result of a signed arithmetic operation exceeds the representable range, respectively. These flags provide important status information for subsequent instructions or conditional branching, ensuring proper handling of carry and overflow conditions and enhancing the accuracy and reliability of arithmetic and logical computations. The ALU module in our project performs arithmetic (ADD, SUB) and logical (AND) operations on 32-bit inputs A and B using a 3-bit opcode (01 for ADD,10 for SUB,00 for AND). It produces a 32-bit result along with carry, zero, negative, and overflow flags. These flags provide information about carry-out, zero result, negative result, and overflow conditions. The ALU efficiently computes results and status flags based on the input values and opcode, enabling various arithmetic and logical operations in a processor.

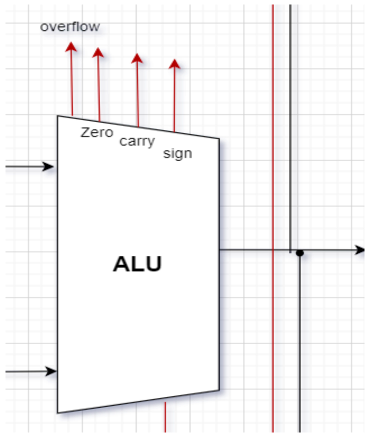


Figure 2-18: ALU block diagram

The testing results of our ALU unit are shown in the waveform bellow.

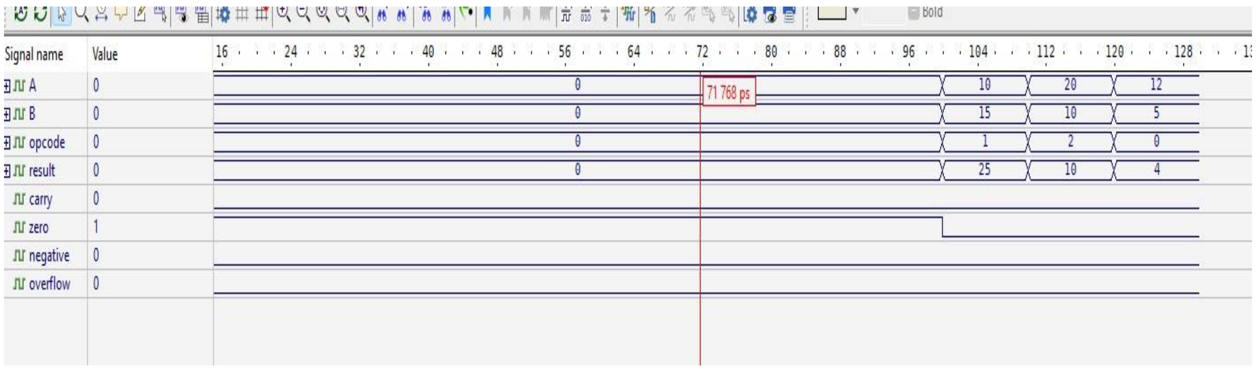


Figure 2-19: ALU simulation

2.3.8. Programmable Counter (PC)

A Programmable Counter (PC) in digital systems is a fundamental component used to keep track of execution order, typically by holding the address of the next instruction to be executed in a processor. It is an essential part of control units and microprocessors, enabling sequential execution of instructions, branching, and control flow management. In Verilog code of pc describes a 32-bit programmable counter (`pc` module) used in digital systems, primarily within processors, to manage the sequence of executed instructions. It updates based on a clock signal and can be reset or loaded with a specific address, facilitating both sequential execution and controlled jumps or branches. The counter operates on the rising edge of the clock, resets to zero when a reset signal is activated, and updates to a new value when a write enable signal is asserted. The testbench (`pc_tb` module) validates this functionality by simulating various operational scenarios. This setup exemplifies a fundamental component in digital design, enabling controlled program flow and instruction sequencing in computing architectures.

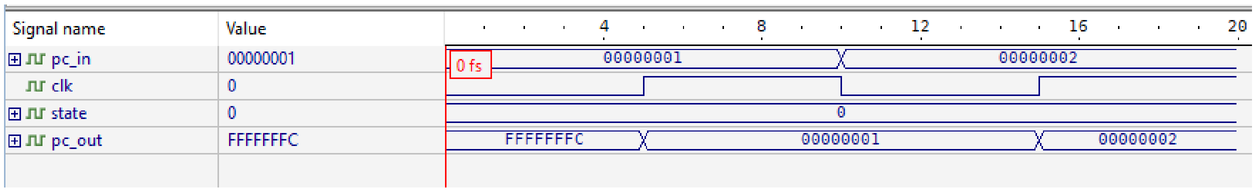


Figure 2-20: Programmable Counter simulation

2.3.9. Stack Pointer (SP)

a stack pointer in digital systems, specifically within a computing context where a stack is used to manage data in a Last In, First Out (LIFO) manner. The stack pointer (SP_OUT) is a crucial component in managing the stack's top position, indicating where the next push or pop operation should occur.in Verilog code defines a stack pointer module (`sp`) that controls the top position of a stack with push and pop operations in a computing system. The stack pointer (`SP_OUT`) is initialized to `512` on reset and can vary between `512` and `767`. Push operations (`stack_sig = 1`) increment the pointer, while pop operations (`stack_sig = 0`) decrement it, with reset conditions applied at the boundaries to prevent overflow and underflow. The testbench (`sp_tb`) demonstrates the module's functionality by simulating various scenarios, showcasing how the stack pointer responds to different operations and resets under specific conditions.

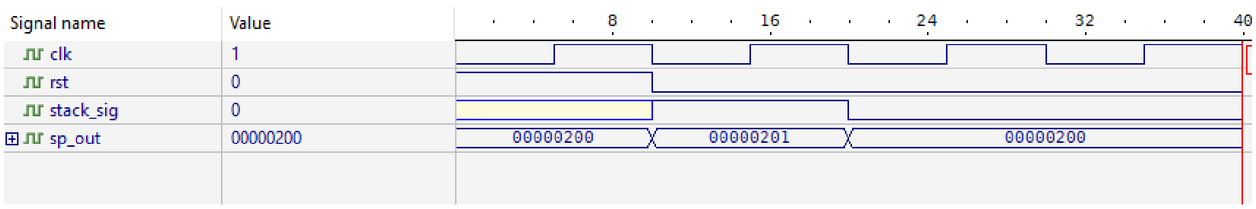


Figure 2-21: stack pointer simulation

2.4. Control Signals

Table 2: 1-bit signals

1-Bit Signal Name	*Effect When Dsserted*	*Effect When Asserted*
IRWrite	None	The output of the memory is written into the Instruction Register (IR)
PCWrite	None	The PC is written the source is controlled by PC-Source.
MemRead	None	Content of memory at the location specified by the Address input is put on the Memory data output.
MemWrite	None	Memory contents of the location specified by the Address input is replaced by the value on the Write data input.
RegWrite1,2	None	Write register is written with the value of the Write data input.
signSig of Extend	Zero_extention.	Sign_extention.
Stack sig	Push (+1)	Pop (-1)
Data Write	The value fed to the write data Memory input comes from pc+4.	The value fed to the write data Memory input comes from RD.
ReadDest	The value fed to the read register2 of register file comes from RD field.	The value fed to the read register2 of register file comes from RS2 field.

Memto Reg	The value fed to the register file input comes from Memory data reg.	The value fed to the register file input is ALUout.
ALUSrcA	The first ALU operand is PC.	The first ALU operand is A register.

Table 3: 2-bit Signals

2-Bit Signal Name	***Value/Effect***
ALUOP	01 The ALU performs an add operation. 10 The ALU performs a sub operation. 00 The ALU performs an and operation.
IorD	00 The PC supplies the Address to the Memory element. 01 The sp supplies the Address to the Memory element. 10 ALUOut is used to supply the address to the memory unit.
ALUSrcB	00 The second input to ALU comes from the B register. 01 The second input to ALU is 4. 10 The second input to the ALU is the sign-extended, lower 16 bits of the Instruction Register (IR). 11 The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left by 2 bits.
PCSource	00 top of stack poped 01the jump target address are sent to the pc for writing. 10 Output of the ALU (PC+4) is sent to the PC for writing. 11 The contents of ALUOut (the branch target address) are sent to the PC for writing.

Table 4: 1-Bit Signals for Branches

1-Bit Signals for Branches	***Effect***
BLTcond	is Asserted if the instruction is BLT
BLTcond	is Asserted if the instruction is BGT
PcWriteCond	is Asserted if the instruction is BEQ and BNE
ZeroCond	is Asserted if the instruction is BEQ and BNE (1 in BEQ and 0 in BNE)

2.5. Finite State Machine (FSM)

The figure below represented the FSM for the processor, which depicts how control signals are derived and transitioned between stages for each type of instruction. The FSM is the heart of the control unit, which governs the sequencing of operations in the datapath. It ensures that the correct set of control signals is generated at each step, directing the processor to correctly fetch, decode, execute, and write back the results of instructions.

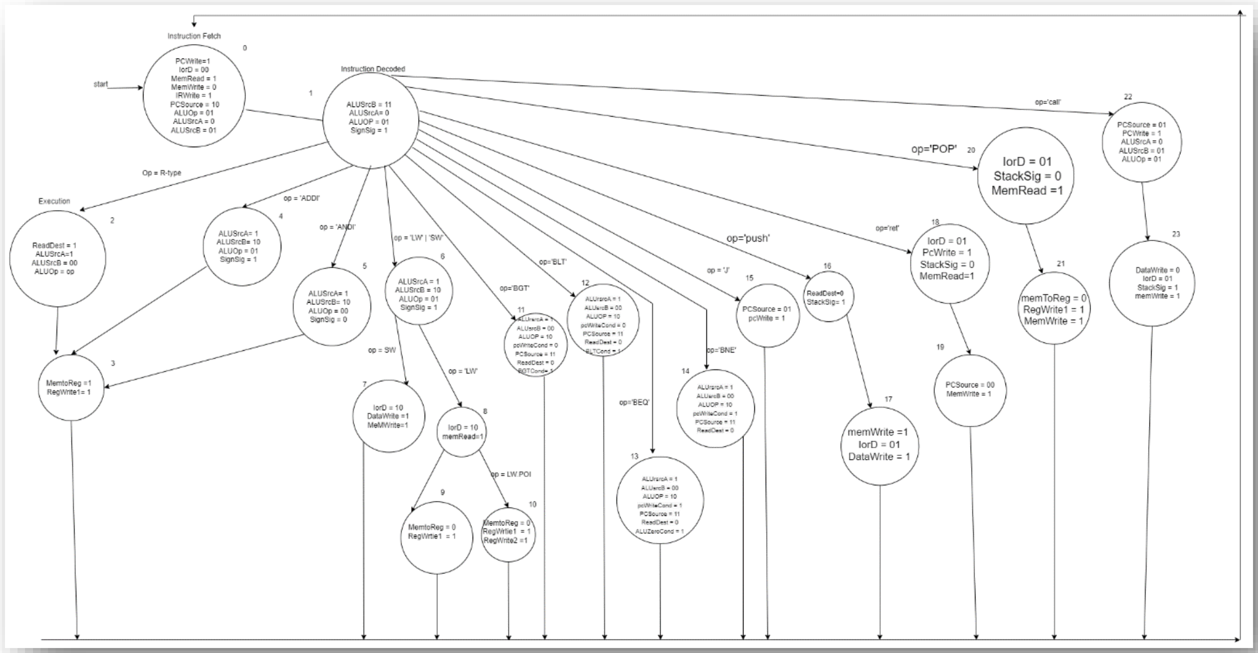


Figure 2-22: Finite State Machine (FSM)

2.6. Control Units

After completion of the Datapath and FSM construction, the corresponding control signals were meticulously determined. The subsequent tables illustrate the variations in each control signal contingent upon the type of instruction and the specific opcodes associated with them.

2.6.1. Mani Control Unit

Table 5: Main Control Signals

OpCode	Reg Write1	Reg Write2	IorD	Mem Read	Mem Write	IR Write	Read Dest	Mem toReg	StackSig	Data Write	SignSig
R-type	1	0	00	0	0	1	1	1	x	x	x
ANDI	1	0	00	0	0	1	x	1	x	x	zero =0
ADDI	1	0	00	0	0	1	x	1	x	x	sign =1
LW	1	0	10	1	0	0	x	0	x	x	sign =1
LOW.PI	1	1	10	1	0	0	x	0	x	x	sign =1
SW	0	0	10	0	1	0	x	x	x	1	sign =1
Branch	0	0	00	0	0	0	0	x	x	x	x
JMP	0	0	00	0	0	0	x	x	x	x	x
CALL	0	0	01	0	1	0	x	x	push =1	0	x
RET	0	0	01	0	1	0	x	x	pop =0	x	x
PUSH	0	0	01	0	1	0	0	0	push =1	1	x

RegWrite1 = R-type + ANDI + ADDI + LW + LOW.PI

RegWrite2 = LOW.PI

IorD[1] = LW + LOW.PI + SW // IorD[0] = CALL + RET + PUSH

MemRead = LW + LOW.PI

MemWrite = SW + CALL + RET + PUSH

IRWrite = R-type

ReadDest = R-type

MemtoReg = ANDI + ADDI

StackSig = 1 for CALL + PUSH // StackSig = 0 for RET

SignSig_sign = ADDI + LW + LOW.PI + SW // SignSig_zero = ANDI

DataWrite = PUS

2.6.2. PC Control Unit

Table 6: PC Control Signals

OpCode	PCSource	PCWrite	PCWriteCond
R-type	10	0	0
ANDI	10	0	0
ADDI	10	0	0
LW	10	0	0
LOW.PI	10	0	0
SW	10	0	0
Branch	11	0	BEQ/BNE = 1
JMP	01	1	0
CALL	01	1	0
RET	00	1	0
PUSH	10	0	0
POP	10	0	0

PCSource[1] = R-type + ANDI + ADDI + LW + LOW.PI + SW + PUSH + POP

PCSource[0] = Branch

PCWrite = JMP + CALL + RET

PCWriteCond = Branch && (BEQ || BN

2.6.3. ALU Control Unit

Table 7: ALU Control Signals

OpCode	ALUOp	ALUSrcA	ALUSrcB
R-type	R-type	1	00
ANDI	10	1	10
ADDI	00	1	10
LW	00	1	10
LOW.PI	00	1	10
SW	00	1	10
Branch	01	1	00
JMP	x	x	x
CALL	00	0	01
RET	x	x	x
PUSH	x	x	x
POP	x	x	x

ALUOp[1] = R-type

ALUOp[0] = Branch

ALUSrcA = R-type + ANDI + ADDI + LW + LOW.PI + SW + Branch

ALUSrcB[1] = ANDI + ADDI + LW + LOW.PI + SW

ALUSrcB[0] = CALL

3. Datapath

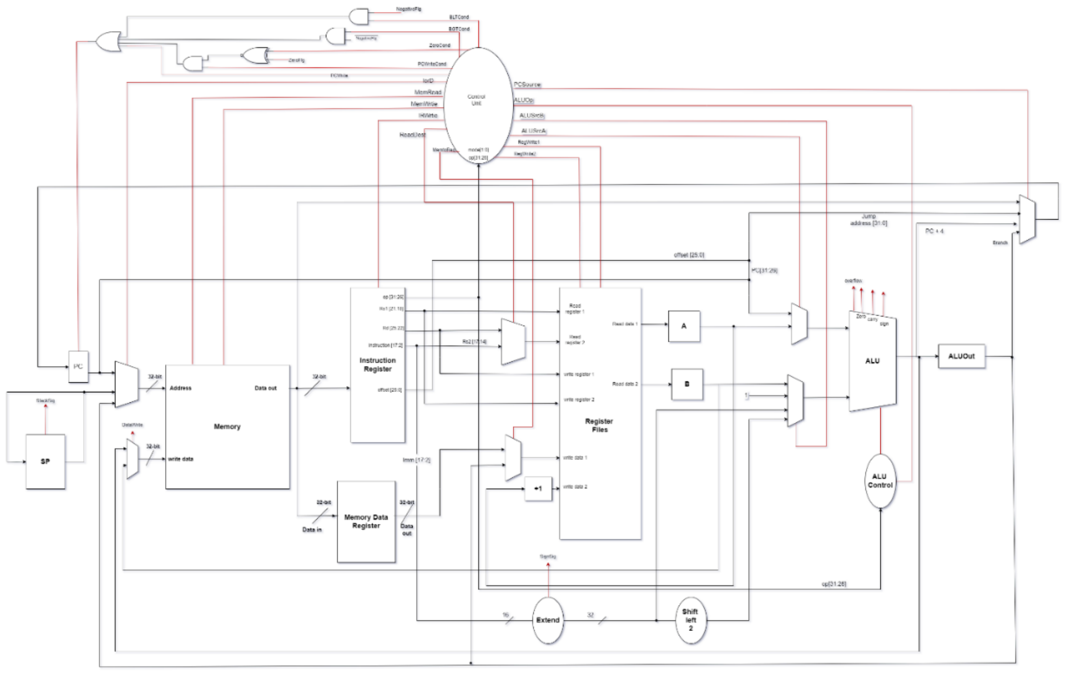


Figure 3-3-1: Datapath

Figure above shows the Datapath, which can be divided into 5 stages.

3.1. CPU-Stage

3.1.1. Instruction Fetch

The instruction fetching process involves accessing the instruction memory based on the address provided by the program counter (PC). The PC address is updated at the positive clock edge when the state is IF (Instruction Fetch). It can be updated to four different values, depending on the instruction being executed: If the PC address is 00, it means that the instruction is RET and requires the program counter to be updated with the value from the stack pointer (SP). If the PC address is 01, it signifies that the instruction is a jump instruction. In this case, the program counter is updated with the jump destination, respectively. If the PC address is 11, the program counter is updated with the branch jump address. Branch instructions are used to conditionally alter the program flow, allowing for decisions and loops in the code execution. On the other hand, jump instruction is usually used to handle subroutines and exceptions. If the PC address is 10, it implies a normal sequential execution of instructions. The program counter is updated by adding 1 to its current value, incrementing it to the next instruction address. This ensures the program proceeds to the subsequent instruction in memory. By updating the program counter to these different values, the instruction fetching mechanism enables the processor to correctly fetch and execute instructions based on the control flow requirements of the program. It allows for branching, looping, and sequential execution, ensuring the program runs in a controlled and organized manner.

3.1.2. Instruction Decode

The instruction decode stage is responsible for decoding the fetched instruction and extracting its different components. The instruction is typically decoded into its type, function code, source one register (Rs1), source two register (Rs2), destination register (Rd), immediate values (Immediate16, Immediate26), mode, and other relevant fields. Once the instruction is decoded, the Control Unit starts generating control signals depending on the available instruction type and function. These control signals determine the actions to be taken in subsequent stages of the processor. One important task during the instruction decode stage is reading the registers from the Register File. The source registers (Rs1, Rs2, and Rd) are identified based on the decoded instruction, and their corresponding values are fetched from the register file. This retrieval of register values ensures that the necessary operands are available for subsequent stages. After obtaining the register values and immediate data, the instruction decode stage is completed. The values of the registers and immediate data are then passed on to the execution stage for further processing. These values serve as inputs for the execution of arithmetic, logical, or control operations specified by the instruction.

In this stage, we compute the speculative branching target and store it the ALU output register, even though we will not need it. We have nothing better to do while we decode the instruction so we might as well.

3.1.3. Execute

The execution stage is where the actual operations specified by the instruction are performed using the Arithmetic Logic Unit (ALU) unit. During the execution stage, the ALU takes the input values from the previous stages, such as register values and immediate data, and performs the necessary calculations or operations. This may include arithmetic operations like addition and subtraction or logical operations such as AND or comparison operations. Once the ALU completes the operation, the resulting value or status is passed on to the next stage. The output from the execution stage may include the result of the operation, a flag indicating the status of the operation (e.g., overflow or carry), or any other relevant information. The produced flags are fed to the control unit to determine, for example, if a branch will be taken.

3.1.4. Memory

In the memory stage of a processor, the control signals derived from previous stages determine whether a read operation, a write operation, or no operation (also known as a "no-op") will occur. These control signals govern the interaction between the processor and the memory system. Here's a breakdown of each possibility:

• Read Operation: If the control signals indicate a read operation, it means that the processor needs to fetch data from memory. The memory address is typically provided by the instruction being executed. The control signals activate the necessary circuits and signals within the memory system to read the data stored at the specified memory address. The retrieved data is then passed along to the subsequent stages of the processor for further processing.

• Write Operation: When the control signals indicate a write operation, it means that the processor needs to store data in memory. Like a read operation, the memory address is typically provided by the instruction being executed. The control signals enable the necessary circuits and signals within the memory system to write the data to the specified memory address.

• No Operation (No-Op): In certain cases, the control signals derived from previous stages may indicate that no memory operation is necessary. This could happen when an instruction does not involve accessing or modifying data in memory, such as arithmetic or logic instructions that operate solely on register values. In such cases, the control signals effectively bypass the memory stage, allowing the processor to proceed to the next stage without any memory-related operations.

The determination of whether a read, write, or no-op occurs is typically based on the instruction being executed and the specific design and implementation of the processor's control unit. The control signals derived from earlier stages help coordinate the interactions between the processor and the memory system, ensuring the correct flow of data and instructions during the execution of a program.

3.1.5. Write Back

The write-back stage, also known as the WB stage, is a crucial step in the instruction execution process within a processor. Its primary responsibility is to write data back to the appropriate destination, typically the register file, after performing necessary computations or retrieving data from memory.

The specific data to be written back depends on the type of instruction being executed:

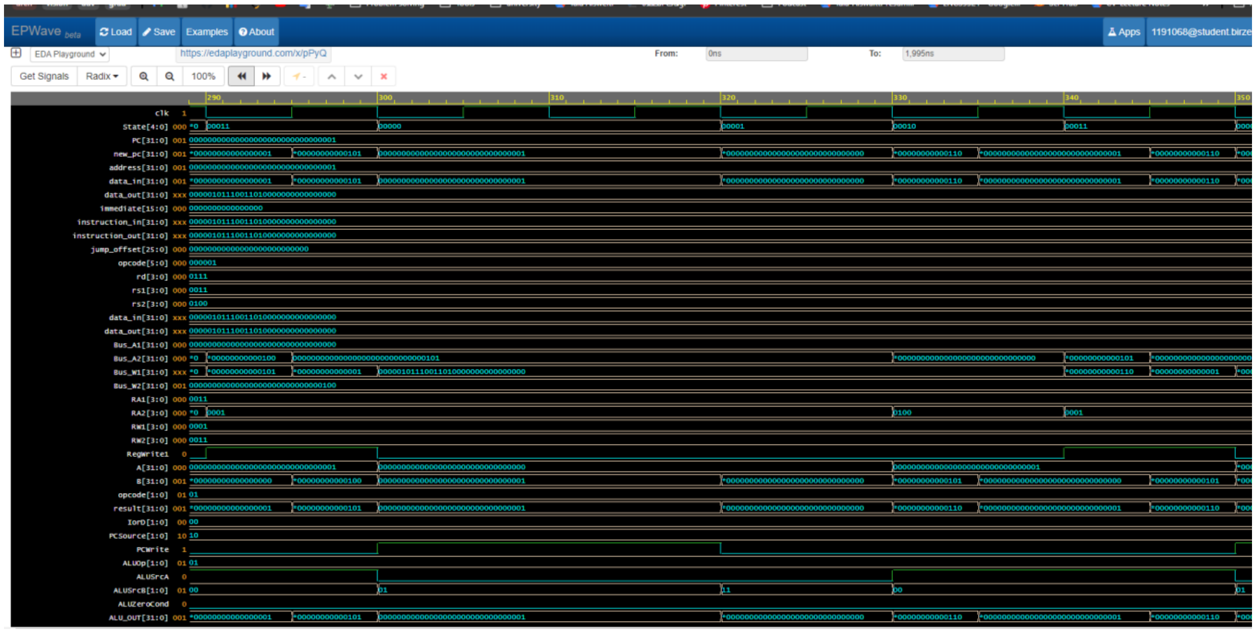
ALU Result: In many arithmetic and logical instructions, the ALU (Arithmetic Logic Unit) performs computations, such as addition and subtraction, etc. The result of these computations is typically written back to the register file.

Memory Data: In load instructions (e.g., LW - Load Word), data is fetched from memory and then written back to the register file. The memory data could be the value stored at a particular memory address or a portion of it.

4. Process Simulation and Testing

Unit testing has been performed for each component and instruction successfully, as shown previously.

We formulated test scenarios for individual instructions and stored them in the memory. While the system simulation proceeded without errors, the output outcomes were inconsistent. Unfortunately, time constraints prevented us from thoroughly debugging the code to identify the underlying issue. Our hypothesis is that the problem lies in the timing delays. We are inclined to think that introducing appropriate delays between stages could potentially rectify the discrepancies in the output and yield accurate results.



```
clk                   1
State[4:0]          000 *0 |00011                              |00000
PC[31:0]            001 00000000000000000000000000000001
new_pc[31:0]        001 *0000000000000001 |*000000000000101    |00000000000000000000000000000001
address[31:0]       001 00000000000000000000000000000001
data_in[31:0]       001 *0000000000000001 |*000000000000101    |00000000000000000000000000000001
data_out[31:0]      xxx 00000101110011010000000000000000
immediate[15:0]     000 0000000000000000
instruction_in[31:0] xxx 00000101110011010000000000000000
instruction_out[31:0] xxx 00000101110011010000000000000000
jump_offset[25:0]   000 00000000000000000000000000
opcode[5:0]         000 000001
rd[3:0]             000 0111
rs1[3:0]            000 0011
rs2[3:0]            000 0100
data_in[31:0]       xxx 00000101110011010000000000000000
data_out[31:0]      xxx 00000101110011010000000000000000
Bus_A1[31:0]        000 00000000000000000000000000000000
Bus_A2[31:0]        000 *0 |*00000000000100 |00000000000000000000000000000101
Bus_W1[31:0]        xxx *0 |*00000000000101 |*00000000000001 |00000101110011010000000000000000
Bus_W2[31:0]        001 00000000000000000000000000000100
RA1[3:0]            000 0011
RA2[3:0]            000 *0 |0001
RW1[3:0]            000 0001
RW2[3:0]            000 0011
Regwrite1             0
A[31:0]             000 00000000000000000000000000000001 |00000000000000000000000000000000
B[31:0]             001 *0000000000000000 |*000000000000100 |00000000000000000000000000000001
opcode[1:0]          01 01
result[31:0]        001 *0000000000000001 |*000000000000101 |00000000000000000000000000000001
IorD[1:0]            00 00
PCSource[1:0]        10 10
PCWrite               1
ALUOp[1:0]           01 01
ALUSrcA               0
ALUSrcB[1:0]         01 00 |01 |11 |00 |01
ALUZeroCond           0
ALU_OUT[31:0]       001 *0000000000000001 |*000000000000101 |00000000000000000000000000000001
```

Figure 4-1 CPU test

5. Conclusion

The project successfully designed and implemented a multi-cycle processor capable of executing a RISC instruction set with a total of 17 instructions across four types. The modular design allows for increased flexibility, modularity, and efficiency. The thorough testing phase confirmed the processor's accuracy and functionality, ensuring its potential for practical application in a computing environment.

6. References

[1] "ieeexplore.ieee.org," [Online]. Available: https://ieeexplore.ieee.org/document/8364013. [Accessed 28 1 2024].

7. Appendix

[1] CPU Test – EDA_Playground