Design and Implementation of a 64-bit RISC-V Single-Cycle Processor

Yusuf Qwareeq
Advanced Processor Systems (ECE 5612)
Professor Son Nguyen
12/10/2024

Temple University

Objectives

The objective of this project is to design and implement a processor using Verilog. The processor must execute a specific subset of instructions, encompassing arithmetic, logical, memory access, and control flow operations, within a single clock cycle.

The processor will implement the following **R-type instructions**: `ADD`, `ADDU`, `SUB`, `SUBU`, `MUL`, `MULU`, `MULH`, `MULHU`, `DIV`, `DIVU`, `REM`, `REMU`, `AND`, and `OR`. Additionally, it will support **load and store instructions** (`LD`, `SD`) and the **branch instruction** (`BEQ`).

To achieve this, the project involves designing each module in the processor, including the arithmetic logic unit (ALU), control unit, register file, data memory, and support modules, ensuring modularity and correctness. The ALU is implemented using structural modeling, building it hierarchically from basic components such as half adders to a 64-bit adder/subtractor, and incorporating support for both signed and unsigned multiplication and division. Behavioral modeling is employed for components like the control unit and logical operations to streamline development. All modules are then integrated into a fully functional processor.

The processor's functionality is validated through comprehensive testing at both the module and system levels. Step-by-step simulations of instruction execution demonstrate correctness, ensuring that the design meets the project's objectives.

Tools

The design and simulation of the processor utilized the following tools and software:

Verilog (Vivado) was used for hardware design entry and simulation, and the Vivado Waveform Viewer was used to analyze and debug simulation results. No hardware implementation was required for this project, as the focus was entirely on simulation and verification in Vivado.

Introduction

Single-cycle processors execute each instruction in one clock cycle, making them simple yet effective for understanding instruction set architecture and processor design. In such designs, all necessary operations, including fetching, decoding, execution, and memory access, are performed within one cycle. This approach simplifies control logic but may require longer cycle times to accommodate the slowest instruction.

The processor for this project is designed around a block diagram (shown in Figure 1) that includes fundamental components like the ALU, control unit, register file, and data memory. Each module plays a critical role: the ALU performs arithmetic and logical operations, the control unit generates signals based on instruction opcodes, and the register file facilitates data storage and transfer. The integration of these modules ensures the seamless execution of all supported instructions.

The supported instructions can be grouped into three types. Load and store instructions enable interaction with memory, facilitating data transfer between registers and memory locations. R-type instructions, such as `ADD`, `MUL`, and `AND`, perform arithmetic and logi-

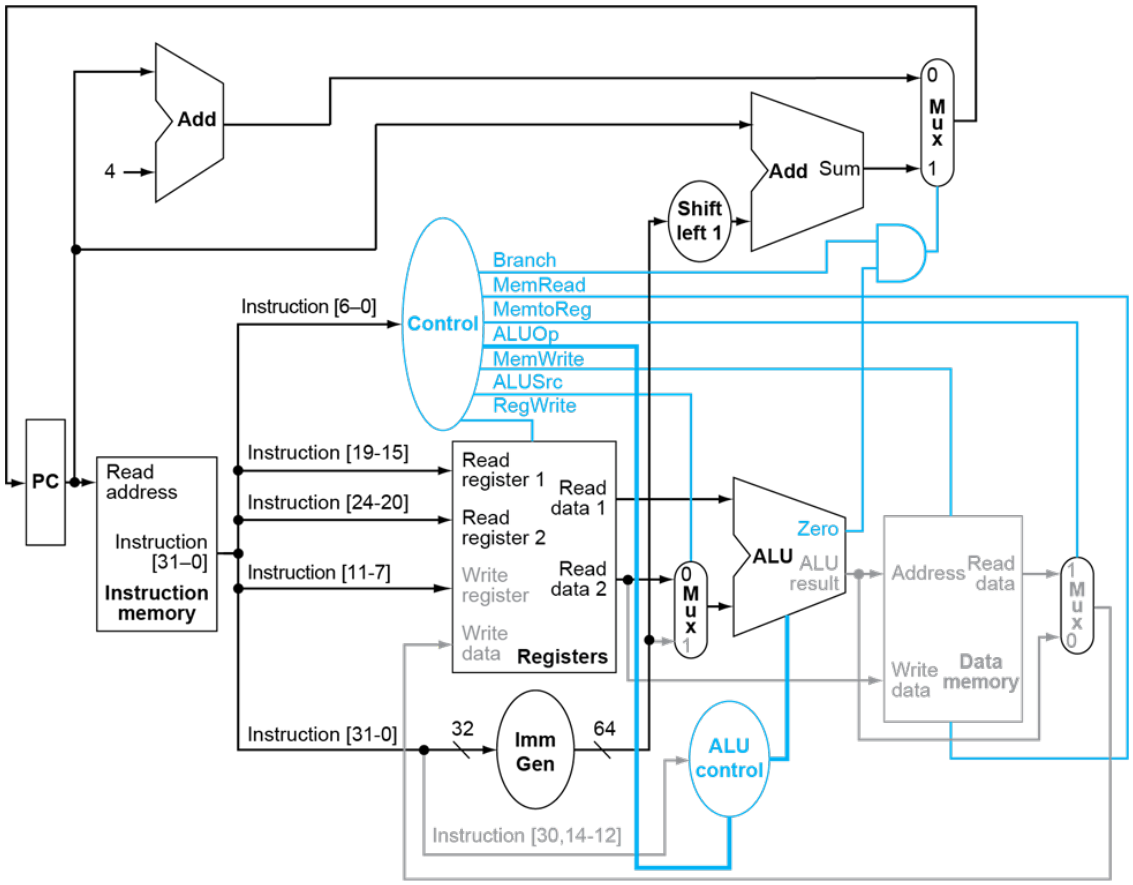

Figure 1: Block diagram of the single-cycle processor. This diagram outlines the core components, including the ALU, control unit, register file, and data memory, and shows their interconnections for seamless execution of instructions.

cal operations on data stored in registers. Finally, branch instructions like `BEQ` allow for conditional program control by altering the flow of execution based on comparisons.

This project demonstrates the design and testing of a processor that efficiently integrates these operations while adhering to single-cycle constraints. The modular design approach ensures that each component is independently verified before integration, maintaining correctness and scalability.

Module Descriptions

Arithmetic Logic Unit (ALU)

The ALU is a central component of the processor, responsible for performing arithmetic, logical, and comparison operations. It supports a range of operations determined by a 4-bit control signal, including addition, subtraction, multiplication, division, logical AND, and OR. The module produces additional flags, such as zero, overflow, carry out, and division by zero.

The ALU was implemented using hierarchical structural modeling. For example, addition was built up from a half adder, progressing to a 64-bit ripple carry adder. Multiplication and division operations were implemented to handle both signed and unsigned numbers.

Inputs and Outputs The ALU accepts two 64-bit operands (`operand_a`, `operand_b`) and a 4-bit control signal (`alu_control`) as inputs. It produces a 64-bit result (`alu_result`) and flags (`zero_flag`, `overflow_flag`, `carry_out_flag`, and `divide_by_zero_flag`) as outputs. The operations performed by the ALU are determined by the control signals, as shown in Table 1.

Control Signal	Operation
0000	Logical AND
0001	Logical OR
0010	Signed addition (ADD)
0011	Unsigned addition (ADDU)
0110	Signed subtraction (SUB)
0111	Unsigned subtraction (SUBU)
1000	Signed multiplication (MUL)
1001	Unsigned multiplication (MULU)
1010	Signed division (DIV)
1011	Unsigned division (DIVU)
1100	Signed remainder (REM)
1101	Unsigned remainder (REMU)
1110	High bits of signed multiplication (MULH)
1111	High bits of unsigned multiplication (MULHU)

Table 1: ALU control signals and their corresponding operations.

Testbench Description The functionality of the ALU was validated using a testbench that applied various combinations of inputs and control signals. Each operation was tested using predetermined inputs and expected outputs, which were displayed and verified. A waveform (shown in Figure 2) was generated to visualize the operation of the ALU over time.

Results Table 2 summarizes the results of the testbench. These results demonstrate the ALU's correct behavior across all tested operations. For example, in signed addition (`ctrl = 0010`), adding `0x0000000000000005` and `0x0000000000000003` produced the expected result `0x0000000000000008` without setting any flags. In unsigned addition (`ctrl = 0011`), adding `0xFFFFFFFFFFFFFFFF` and `0x0000000000000001` correctly resulted in `0x0000000000000000` with the carry-out flag set to 1. Logical operations like AND (`ctrl = 0000`) produced accurate results, as did multiplication and division operations.

ctrl	A	B	result	zero	overflow	c_out
0010	0000000000000005	0000000000000003	0000000000000008	0	0	0
0011	FFFFFFFFFFFFFFFF	0000000000000001	0000000000000000	0	0	1
0110	000000000000000A	000000000000000A	0000000000000000	1	0	0
0110	0000000000000010	0000000000000010	0000000000000000	1	0	0
1000	0000000000000003	0000000000000002	0000000000000006	0	0	0
1010	000000000000000A	0000000000000003	0000000000000003	0	0	0
0000	FFFFFFFFFFFFFFFF	000000000000000F	000000000000000F	0	0	0

Table 2: ALU test results in hexadecimal. The table shows the control signal (`ctrl`), operands (`A`, `B`), result (`result`), and flag states (`zero`, `overflow`, and `c_out`) for each operation.

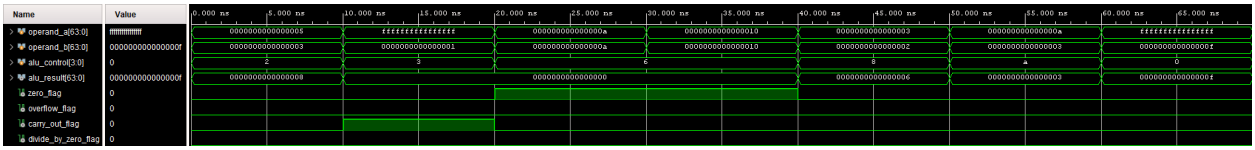

Figure 2: Waveform generated during ALU testing. The figure illustrates the transitions of inputs and outputs for operations like addition, subtraction, and multiplication, verifying the correctness and timing of the ALU implementation.

Ripple Carry Adder (RCA)

The Ripple Carry Adder (RCA) is a key component of the ALU, responsible for performing both signed and unsigned addition. The design uses a hierarchical structure, starting from a half adder and building up to a 64-bit ripple carry adder. It also supports the detection of overflow for signed addition, which is indicated by a dedicated flag.

Inputs and Outputs The RCA accepts two 64-bit operands (`input_a`, `input_b`), a carry-in (`carry_in`), and a signal (`is_signed_add`) indicating whether the addition is signed or unsigned. It produces a 64-bit sum (`sum_out`), a carry-out (`carry_out`), and an overflow flag (`overflow`).

Testbench Description The RCA was tested using a comprehensive testbench that validated its functionality across various cases, including edge scenarios such as positive and negative overflow. Each test case was designed to provide a mix of signed and unsigned additions with specific input values. A waveform was generated to confirm the correct operation over time, as shown in Figure 3.

Results The testbench results, summarized in Table 3, demonstrate the RCA's ability to handle all tested scenarios. For instance, in unsigned addition, adding `0x0000000000000005` and `0x000000000000000A` produces the expected result of `0x000000000000000F` with no carry-out or overflow. In signed addition, adding the maximum positive value `0x7FFFFFFF FFFFFFFF` with `0x0000000000000001` correctly triggers the overflow flag.

A	B	c_in	signed	result	c_out	overflow
0000000000000005	000000000000000A	0	0	000000000000000F	0	x
FFFFFFFFFFFFFFFF	0000000000000001	1	0	0000000000000001	1	x
7FFFFFFFFFFFFFFF	0000000000000001	0	1	8000000000000000	x	1
8000000000000000	FFFFFFFFFFFFFFFF	0	1	7FFFFFFFFFFFFFFF	x	1
0000000000003039	FFFFFFFFFFFCFDAF	0	1	FFFFFFFFFFFD2DE8	x	0
FFFFFFFFFFFFFFFF	0000000000000001	0	0	0000000000000000	1	x
0000000000000000	0000000000000000	0	0	0000000000000000	0	x
FFFFFFFFFFFFFF9C	FFFFFFFFFFFFFF38	0	1	FFFFFFFFFFFFFED4	x	0

Table 3: RCA test results in hexadecimal. The table shows the input operands (A and B), the carry-in (c_in), the signed flag (signed), the sum result (result), the carry-out (c_out), and the overflow flag (overflow).

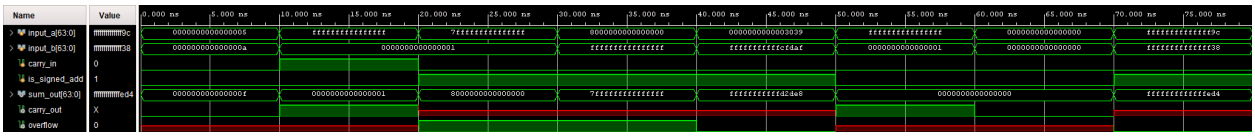

Figure 3: Waveform generated during RCA testing. The figure illustrates the transitions of inputs and outputs for operations like addition, verifying the correctness and timing of the implementation.

Ripple Carry Subtractor (RCS)

The Ripple Carry Subtractor (RCS) is an integral component of the processor, designed to perform both signed and unsigned subtraction. It implements subtraction as $A + (-B)$, leveraging the ripple carry adder to compute the sum of the minuend and the two's complement of the subtrahend. This design efficiently reuses the addition logic while maintaining accuracy for both signed and unsigned operations.

Inputs and Outputs The RCS accepts two 64-bit operands (input_a, input_b), a borrow-in signal (borrow_in), and a control signal (is_signed_sub) that specifies whether the subtraction is signed or unsigned. It produces a 64-bit difference (diff_out), a borrow-out signal (borrow_out) for unsigned operations, and an overflow flag (overflow) for signed operations.

Testbench Description The functionality of the RCS was validated through a comprehensive testbench. The testbench applied various combinations of inputs to test scenarios such as unsigned subtraction with borrow, signed subtraction with overflow, and edge cases involving maximum and minimum values. A waveform was generated to verify the transitions of inputs and outputs over time, ensuring correctness in the module's operation.

Results The results of the testbench, detailed in Table 4, confirm the RCS's accuracy across all tested scenarios. For instance, in unsigned subtraction, subtracting 0x00000000

00000002 from 0x0000000000000005 yields 0x0000000000000003 with no borrow-out or overflow. In signed subtraction, subtracting 0xFFFFFFFFFFFFFFFF from the maximum positive value 0x7FFFFFFFFFFFFFFF correctly triggers the overflow flag.

A	B	b_in	signed	result	b_out	overflow
0000000000000005	0000000000000002	0	0	0000000000000003	0	x
0000000000000001	0000000000000002	1	0	0000000000000000	1	x
7FFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	0	1	8000000000000000	x	1
8000000000000000	0000000000000001	0	1	7FFFFFFFFFFFFFFF	x	1
0000000000003039	FFFFFFFFFFFCFDAF	0	1	000000000003328A	x	0
0000000000000000	FFFFFFFFFFFFFFFF	0	0	0000000000000001	1	x
0000000000000000	0000000000000000	0	0	0000000000000000	0	x
FFFFFFFFFFFFFF9C	FFFFFFFFFFFFFF38	0	1	0000000000000064	x	0

Table 4: RCS test results in hexadecimal. The table shows the input operands (A and B), the borrow-in (b_in), the signed flag (signed), the difference result (result), the borrow-out (b_out), and the overflow flag (overflow).

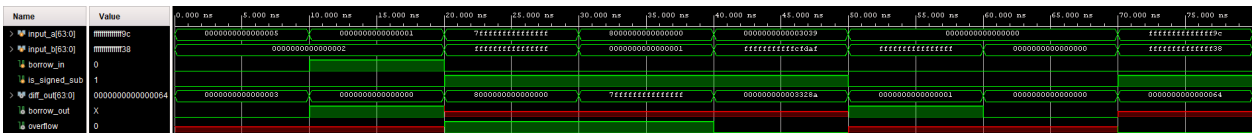

Figure 4: Waveform generated during RCS testing. The figure illustrates the transitions of inputs and outputs for operations like subtraction, verifying the correctness and timing of the implementation.

Multiplier

The 64-bit Multiplier module performs signed and unsigned multiplication based on the control signal (is_signed_mult). It uses an iterative bit-shifting approach for efficient hardware implementation. This process involves updating the product register based on the least significant bit (LSB) of the multiplier, shifting the multiplicand left and the multiplier right on each iteration. Signed multiplication is handled by converting inputs to their absolute values and applying a correction to the result based on the original signs. The steps of this algorithm are summarized in Algorithm 1.

Inputs and Outputs The multiplier accepts two 32-bit operands (multiplicand_in, multiplier_in) and a control signal (is_signed_mult) indicating whether the operation is signed or unsigned. It produces a 64-bit product (product_out).

Testbench Description The multiplier was tested using a comprehensive testbench that validated its functionality across various scenarios, including signed and unsigned multiplication, edge cases like multiplication by zero, and typical cases involving both large and

Algorithm 1 64-bit Multiplication Algorithm

Require: $a, b \in \mathbb{Z}_{32}$, is_signed $\in \{0, 1\}$
Ensure: $p \in \mathbb{Z}_{64}$
1: **if** is_signed **then**
2: Convert a, b to positive; set flags.
3: **end if**
4: $p \leftarrow 0$
5: **for** $i \leftarrow 0$ to 31 **do**
6: **if** $b_0 = 1$ **then**
7: $p \leftarrow p + a$
8: **end if**
9: $a \leftarrow a \ll 1$, $b \leftarrow b \gg 1$
10: **end for**
11: **if** is_signed **then**
12: Adjust sign.
13: **end if**
14: $p_{\text{out}} \leftarrow p$

small values. Each test confirmed the correct operation of the algorithm and its adherence to expected behavior.

Results The results, detailed in Table 5, confirm the accuracy of the multiplier. For example, the product of `0x00000005` and `0x0000000A` in unsigned mode was correctly computed as `0x0000000000000032`. Similarly, signed multiplication of `0x80000000` and `0xFFFFFFFF` resulted in `0x0000000080000000`, demonstrating correct handling of signed operations.

multiplicand	multiplier	signed	p_out
00000005	0000000A	0	0000000000000032
FFFFFFFF	00000001	0	00000000FFFFFFFF
7FFFFFFF	00000002	1	00000000FFFFFFFE
80000000	FFFFFFFF	1	0000000080000000
00003039	FFFFFDAF	1	FFFFFFFFFF904BF7
00000000	FFFFFFFF	0	0000000000000000
FFFFFF9C	FFFFFF38	1	0000000000004E20
00000000	00000000	0	0000000000000000

Table 5: Multiplier test results in hexadecimal. The table shows the input operands (`multiplicand`, `multiplier`), the signed control flag (`migned`), and the computed product (`p_out`).

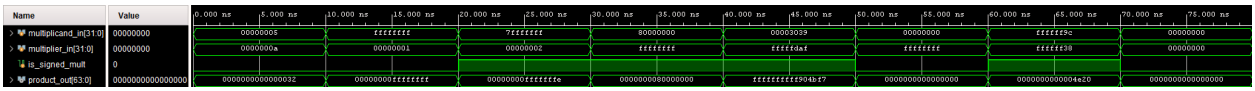

Figure 5: Waveform generated during multiplier testing. The figure illustrates the transitions of inputs and outputs for various test cases, confirming the correctness of the implementation.

Divider

The 64-bit Divider module performs signed and unsigned division based on the control signal (`is_signed_div`). It uses an iterative hardware-based algorithm that mimics manual division by shifting and subtracting. The divisor is shifted right on each iteration while the quotient is built one bit at a time. Signed division is handled by converting inputs to their absolute values and applying corrections to the result based on the original signs. The steps of this algorithm are summarized in Algorithm 2.

Inputs and Outputs The divider accepts a 64-bit dividend (`dividend_in`), a 32-bit divisor (`divisor_in`), and a control signal (`is_signed_div`) indicating whether the operation is signed or unsigned. It produces a 32-bit quotient (`quotient_out`), a 64-bit remainder (`remainder_out`), and a divide-by-zero flag (`divide_by_zero_flag`).

Testbench Description The divider was tested using a comprehensive testbench that validated its functionality across various scenarios, including signed and unsigned division, edge cases such as division by zero, and typical cases involving large and small values. Each test confirmed the correct operation of the algorithm and its adherence to expected behavior.

Results The results, detailed in Table 6, confirm the accuracy of the divider. For example, the division of `0x0000000000000064` by `0x00000005` in unsigned mode was correctly computed as `quotient = 0x00000014` and `remainder = 0x0000000000000000`. Similarly, signed division of `0xFFFFFFFFFFFFFF9C` by `0xFFFFFFFF` resulted in `quotient = 0x00000014` and `remainder = 0x0000000000000000`, demonstrating correct handling of signed operations.

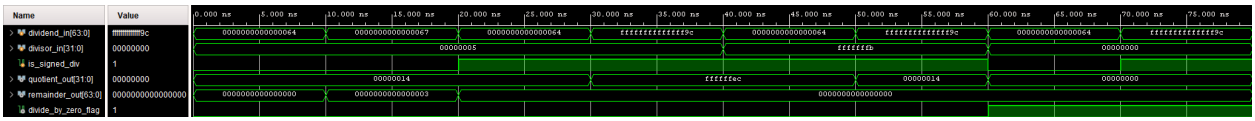

Figure 6: Waveform generated during divider testing. The figure illustrates the transitions of inputs and outputs for various test cases, confirming the correctness of the implementation.

Algorithm 2 64-bit Division Algorithm

Require: $d \in \mathbb{Z}_{64}, r \in \mathbb{Z}_{32}, \text{is\_signed} \in \{0, 1\}$
Ensure: $q \in \mathbb{Z}_{32}, \text{remainder} \in \mathbb{Z}_{64}, \text{divide\_by\_zero} \in \{0, 1\}$

if $r = 0$ **then**
  divide_by_zero $\leftarrow 1$
  $q \leftarrow 0$, remainder $\leftarrow 0$
else
  if is_signed **then**
    Convert d, r to positive; set flags.
  end if
  remainder $\leftarrow d$
  $r \leftarrow r \ll 32$
  $q \leftarrow 0$
  for $i \leftarrow 0$ to 32 **do**
    if remainder $\geq r$ **then**
      remainder $\leftarrow$ remainder $- r$
      $q \leftarrow (q \ll 1)|1$
    else
      $q \leftarrow q \ll 1$
    end if
    $r \leftarrow r \gg 1$
  end for
  if is_signed **then**
    Adjust sign of q and remainder.
  end if
end if
$q_{\text{out}} \leftarrow q$, $\text{remainder}_{\text{out}} \leftarrow$ remainder, $\text{divide\_by\_zero}_{\text{out}} \leftarrow$ divide_by_zero

dividend	divisor	signed	quotient	remainder	divide_by_zero
0000000000000064	00000005	0	00000014	0000000000000000	0
0000000000000067	00000005	0	00000014	0000000000000003	0
0000000000000064	00000005	1	00000014	0000000000000000	0
FFFFFFFFFFFFFF9C	00000005	1	FFFFFFEC	0000000000000000	0
0000000000000064	FFFFFFFB	1	FFFFFFEC	0000000000000000	0
FFFFFFFFFFFFFF9C	FFFFFFFB	1	00000014	0000000000000000	0
0000000000000064	00000000	0	00000000	0000000000000000	1
FFFFFFFFFFFFFF9C	00000000	1	00000000	0000000000000000	1

Table 6: Divider test results in hexadecimal. The table shows the input operands (`dividend`, `divisor`), the signed control flag (`signed`), the computed quotient (`quotient`), the remainder (`remainder`), and the divide-by-zero flag (`divide_by_zero`).

AND Gate

The 64-bit AND Gate is a fundamental digital logic component designed to perform bitwise AND operations on two 64-bit input vectors. It plays a crucial role in the ALU of the RISC-V

64-bit Single-Cycle CPU, enabling efficient execution of logical operations.

Inputs and Outputs The AND Gate module accepts two 64-bit input vectors, `A` and `B`, which represent the operands for the logical AND operation. It produces a single 64-bit output vector, `result`, which contains the result of the bitwise AND operation applied to the inputs.

Testbench Description The functionality of the AND Gate was verified using a testbench that explored a variety of scenarios, including edge cases and typical use cases. The inputs were varied to include all zeros, all ones, alternating bit patterns, and randomly selected values. The outputs were validated against expected results to ensure accuracy.

Results The testbench results, summarized below, confirm the correctness of the AND Gate module. For instance, when both inputs were all ones (`0xFFFFFFFFFFFFFFFF`), the output was correctly computed as `0xFFFFFFFFFFFFFFFF`. Similarly, for alternating input patterns (`0xAAAAAAAAAAAAAAAA` and `0x5555555555555555`), the output was `0x00000000 00000000`, demonstrating accurate logical behavior.

A	B	result
0000000000000000	0000000000000000	0000000000000000
FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
AAAAAAAAAAAAAAAA	5555555555555555	0000000000000000
123456789ABCDEF0	0FEDCBA987654321	0224422882244220
FFFFFFFF00000000	00000000FFFFFFFF	0000000000000000

Table 7: AND Gate test results in hexadecimal. The table shows the input vectors (`A` and `B`) and the computed AND result (`result`).

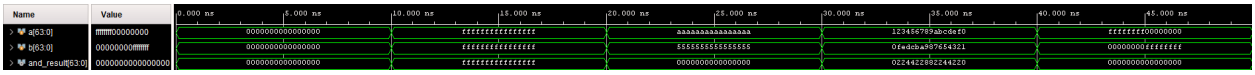

Figure 7: Waveform generated during AND Gate testing. The figure illustrates the transitions of inputs and outputs, validating the correctness of the bitwise AND operation.

OR Gate

The 64-bit OR Gate module performs a bitwise OR operation on two 64-bit input vectors. This simple yet essential operation is crucial in many arithmetic and logical operations, particularly in control logic and data masking. The module is implemented with a direct assignment, ensuring efficient hardware synthesis and operation.

Inputs and Outputs The OR gate accepts two 64-bit input vectors (`A` and `B`) and produces a single 64-bit output vector (`result`). Each bit in the output vector represents the logical OR of the corresponding bits in the two input vectors.

Testbench Description The OR gate module was tested using a comprehensive testbench that verified its behavior across various scenarios, including all zeros, all ones, alternating bits, and edge cases. Each test case provided distinct inputs to ensure the module's correctness. The results were displayed in hexadecimal format for compactness and clarity.

Results The testbench results confirm the module's correctness across all scenarios. For instance, when the inputs are `0xAAAAAAAAAAAAAAAA` and `0x5555555555555555`, the output is correctly computed as `0xFFFFFFFFFFFFFFFF`. Similarly, when inputs are `0xFFFFFFFF 00000000` and `0x00000000FFFFFFFF`, the output is correctly computed as `0xFFFFFFFF FFFFFFFF`. These results demonstrate the accuracy and reliability of the module.

A	B	result
0000000000000000	0000000000000000	0000000000000000
FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
AAAAAAAAAAAAAAAA	5555555555555555	FFFFFFFFFFFFFFFF
123456789ABCDEF0	0FEDCBA987654321	1FFDDFF99FFDDFF1
FFFFFFFF00000000	00000000FFFFFFFF	FFFFFFFFFFFFFFFF

Table 8: 64-bit OR Gate test results in hexadecimal format. Each row shows the inputs (`A` and `B`) and the corresponding OR result.

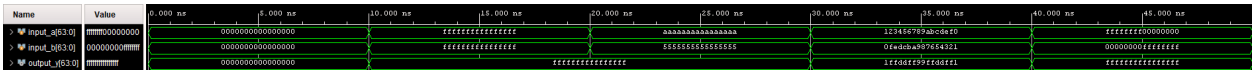

Figure 8: Waveform generated during 64-bit OR Gate testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

ALU Control Unit

The ALU Control Unit is a critical module responsible for decoding the operation type based on the instruction fields `alu_op`, `funct7`, and `funct3`. It produces a 4-bit `alu_control` signal, enabling the ALU to execute operations such as addition, subtraction, logical AND, OR, multiplication, division, and remainder computations. The unit differentiates between signed and unsigned operations, ensuring proper decoding for all supported instructions.

Inputs and Outputs The ALU Control Unit accepts three inputs: `alu_op` (2 bits), `funct7` (7 bits), and `funct3` (3 bits). It generates a 4-bit `alu_control` signal to guide the ALU in selecting the correct operation.

ALU Operations The mapping of `alu_op`, `funct7`, and `funct3` to the `alu_control` signal is shown in Table 9. Each entry corresponds to a specific ALU operation.

alu_op	funct7	funct3	alu_control	**Operation**
00	XXXXXXX	XXX	0010	LD/SD (Signed ADD)
01	XXXXXXX	XXX	0110	BEQ (Signed SUBTRACT)
10	1001000	111	0000	AND (Bitwise)
10	1001001	111	0001	OR (Bitwise)
10	0000000	000	0010	ADD (Signed)
10	1000001	000	0011	ADDU (Unsigned)
10	0100000	000	0110	SUB (Signed)
10	1000011	100	0111	SUBU (Unsigned)
10	0000001	000	1000	MUL (Signed)
10	0110001	001	1001	MULU (Unsigned)
10	0000001	100	1010	DIV (Signed)
10	0110101	010	1011	DIVU (Unsigned)
10	0111000	011	1100	REM (Signed)
10	0111001	011	1101	REMU (Unsigned)
10	0111100	110	1110	MULH (Signed)
10	0111101	110	1111	MULHU (Unsigned)

Table 9: Mapping of `alu_op`, `funct7`, and `funct3` fields to `alu_control` and their corresponding operations.

Testbench Description The testbench evaluated all combinations of `alu_op`, `funct7`, and `funct3` fields, confirming that the module correctly generated the expected `alu_control` signal for each input configuration. The results were displayed in a structured format for verification.

Results The testbench results confirm the module's correctness. For instance, when `alu_op = 10`, `funct7 = 1001000`, and `funct3 = 111`, the module correctly generates `alu_control = 0000` for the AND operation. Table 10 summarizes the test results.

Figure 9: Waveform generated during ALU Control Unit testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

Control Unit

The Control Unit is a critical component of the 64-bit RISC-V single-cycle CPU. It is responsible for generating control signals that orchestrate the execution of instructions by other components such as the ALU, memory, and register file. Based on the 7-bit `opcode` field of an instruction, the Control Unit ensures the correct behavior for various instruction types, including load, store, branch, and arithmetic operations.

alu_op	funct7	funct3	alu_control
10	1001000	111	0000
10	1001001	111	0001
00	XXXXXXX	XXX	0010
10	0000000	000	0010
10	1000001	000	0011
01	XXXXXXX	XXX	0110
10	0100000	000	0110
10	1000011	100	0111
10	0000001	000	1000
10	0110001	001	1001
10	0000001	100	1010
10	0110101	010	1011
10	0111000	011	1100
10	0111001	011	1101
10	0111100	110	1110
10	0111101	110	1111

Table 10: ALU Control Unit test results in binary format. The table shows the `alu_op`, `funct7`, `funct3`, and the resulting `alu_control`.

Inputs and Outputs The Control Unit accepts a single 7-bit input, `opcode`, which encodes the instruction type. It generates seven output signals that control various processor functions. The `branch` signal is asserted for branch instructions to activate the branching logic. The `mem_read` signal is asserted during load instructions to enable reading from memory. The `mem_to_reg` signal selects whether data from memory or the ALU result is written back to the register file. The `alu_op` signal is a 2-bit signal passed to the ALU Control Unit to indicate the required operation type. The `mem_write` signal is asserted during store instructions to enable writing to memory. The `alu_src` signal determines whether the ALU's second operand is a register or an immediate value. The `reg_write` signal is asserted to enable writing results back to a register. The table below shows how the control signals are generated based on the opcode:

Opcode	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite
0000011	0	1	1	00	0	1	1
0100011	0	0	0	00	1	1	0
1100011	1	0	0	01	0	0	0
0110011	0	0	0	10	0	0	1

Table 11: Control signal mapping for different opcodes.

Testbench Description The testbench for the Control Unit evaluated the module's response to different `opcode` values. Each test case checked whether the module correctly asserted or de-asserted the output signals in accordance with the expected behavior. The results were displayed in a structured format to confirm the correctness of the control signal generation.

Results The testbench results confirm the accuracy of the Control Unit's behavior. For example, when `opcode = 0000011`, the module correctly enabled `mem_read` and `mem_to_reg`, while setting `alu_op = 00`. For `opcode = 0100011`, the module asserted `mem_write` and `alu_src`, while disabling all other signals. These results validate the module's functionality across various instruction types.

opcode	branch	mem_read	mem_to_reg	alu_op	mem_write	alu_src	reg_write
0000011	0	1	1	00	0	1	1
0100011	0	0	0	00	1	1	0
1100011	1	0	0	01	0	0	0
0110011	0	0	0	10	0	0	1
1111111	0	0	0	00	0	0	0

Table 12: Control Unit test results for various `opcode` values. The table lists the input `opcode` and the corresponding control signal outputs.

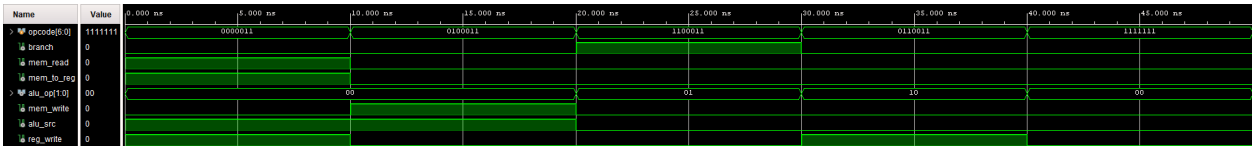

Figure 10: Waveform generated during Control Unit testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

Data Memory

The Data Memory module is a critical component in the single-cycle RISC-V CPU. It facilitates data storage and retrieval during program execution. This module supports synchronous write operations on the rising edge of the clock and asynchronous read operations for immediate data availability. Data is stored in 8-bit locations, with each 64-bit word distributed across eight consecutive memory locations in big-endian format. The memory layout and design ensure compatibility with the CPU's single-cycle constraints.

Inputs and Outputs The Data Memory module accepts several inputs and produces a single output. The `address` input specifies the memory location being accessed. The `write_data` input provides the 64-bit data to be stored. Control signals, `mem_read` and

`mem_write`, enable read and write operations, respectively. The `clk` input ensures synchronization for write operations. The module's output, `read_data`, reflects the content of the addressed memory location when `mem_read` is asserted.

Testbench Description The testbench for the Data Memory module thoroughly verified its functionality by simulating a variety of read and write operations. Initial values were preloaded into the memory to test the correctness of both reading and writing processes. Each test case verified that the module behaved as expected, with the results presented in a structured format for clarity.

Results The testbench results confirm the module's correct operation across all tested scenarios. For instance, during the second cycle, when `address = 00` and `write_data = DEADBEEF12345678`, the data was successfully written to the specified memory location. Subsequent read operations confirmed that the stored values could be retrieved accurately. Table 13 summarizes the results, detailing the cycle-by-cycle behavior of the module.

cycle	address	write_data	mem_write	mem_read	read_data
2	00	DEADBEEF12345678	1	0	0000000000000000
3	01	CAFEBABE87654321	1	0	0000000000000000
4	02	123456789ABCDEF0	1	0	0000000000000000
5	00	-	0	1	DEADBEEF12345678
6	01	-	0	1	CAFEBABE87654321
7	02	-	0	1	123456789ABCDEF0
8	03	AABBCCDDEEFF0011	1	0	0000000000000000
9	03	-	0	1	AABBCCDDEEFF0011

Table 13: Data Memory operation summary. This table shows the cycle-by-cycle behavior of the Data Memory module, including read and write operations.

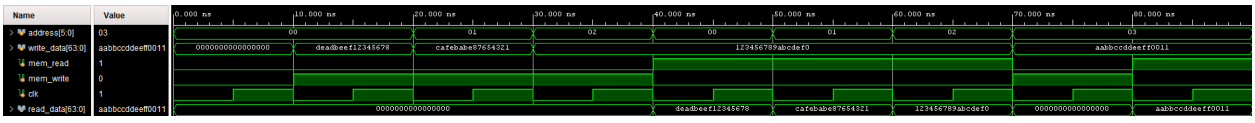

Figure 11: Waveform generated during Data Memory testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality for various read and write scenarios.

Register File

The Register File is a fundamental component of the 64-bit RISC-V single-cycle CPU. It consists of 32 registers, each 64 bits wide, and supports two simultaneous asynchronous read operations and one synchronous write operation per clock cycle. Register `x0` is hardwired to zero, ensuring compliance with the RISC-V ISA specification. Registers `x1` through `x11` are exposed for debugging and monitoring purposes.

Inputs and Outputs The Register File accepts three 5-bit inputs: `read_reg1`, `read_reg2`, and `write_reg`, which specify the registers for read and write operations. The 64-bit `write_data` input provides data to be written into the specified register, and the `reg_write` signal enables or disables write operations. The `clk` signal ensures synchronization for write operations. The module outputs two 64-bit read ports, `read_data1` and `read_data2`, and exposes the values of registers `x1` through `x11` for debugging purposes.

Testbench Description The testbench validated the Register File's functionality by simulating synchronous writes and asynchronous reads. Each test confirmed that data was written to the correct register and read correctly. Additionally, the hardwired behavior of `x0` was verified to ensure it remained constant at zero. Test results were displayed cycle-by-cycle to verify expected behavior.

Results The testbench confirmed the Register File's correct operation across all scenarios. For instance, in Cycle 2, a write to `x1` with the value `DEADBEEF12345678` was performed successfully. Attempts to write to `x0` were ignored as expected. Table 14 summarizes the observed behavior.

cycle	wr_en	wr_reg	wr_data	rd_data_1	rd_data_2
2	1	x1	DEADBEEF12345678	0000000000000000	0000000000000000
3	1	x2	CAFEBABE87654321	0000000000000000	0000000000000000
4	0	x2	CAFEBABE87654321	DEADBEEF12345678	CAFEBABE87654321
6	0	x3	123456789ABCDEF0	123456789ABCDEF0	CAFEBABE87654321
8	0	x0	FFFFFFFFFFFFFFFF	0000000000000000	CAFEBABE87654321

Table 14: Register File operation summary. This table shows the cycle-by-cycle behavior of the Register File module, including read and write operations.

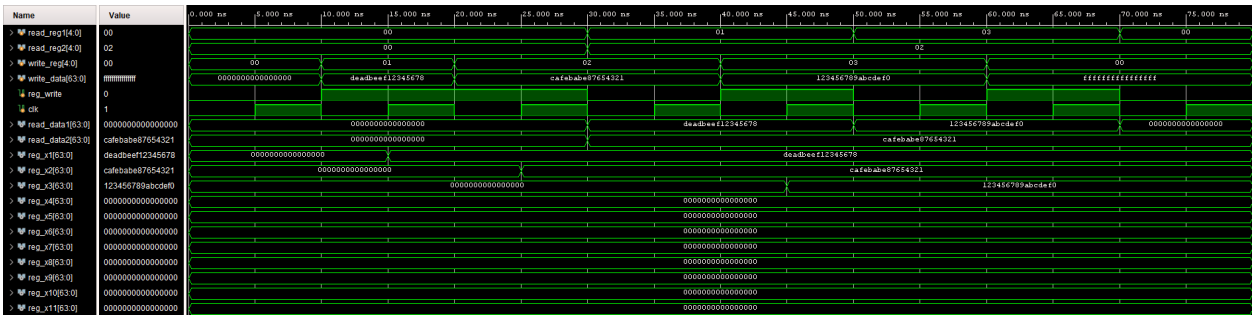

Figure 12: Waveform generated during Register File testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality for various scenarios.

Instruction Memory

The Instruction Memory module is a vital component of the 64-bit RISC-V single-cycle CPU. It stores program instructions and provides the instruction corresponding to the current

Program Counter (`pc`). The module is designed to latch the instruction output on the rising edge of the clock, ensuring synchronization with other components. Each instruction is 32 bits wide, and the `pc` increments by 4 bytes for each subsequent instruction to maintain alignment.

Inputs and Outputs The Instruction Memory module accepts a 64-bit `pc` input, representing the current address of the instruction to be fetched, and a clock signal (`clk`). The module outputs the 32-bit `instruction`, corresponding to the memory location specified by `pc`. The memory is preloaded with program instructions to evaluate the arithmetic expression

$$y = \frac{(a \cdot f - c \cdot d) + e}{b}.$$

Testbench Description The testbench validated the Instruction Memory's functionality by simulating sequential reads at different `pc` values. The Program Counter was incremented by 4 in each cycle to fetch consecutive instructions, verifying correct addressing and data retrieval. The results of each fetch were compared against the expected binary encoding of the instructions.

Results The testbench confirmed the correct operation of the Instruction Memory module. For each value of `pc`, the fetched instruction matched the expected binary encoding, verifying the correctness of the memory initialization and addressing logic. Table 15 summarizes the observed behavior.

pc	instruction
0000000000000000	00000000000000000011000010000011
0000000000000004	00000000010100000011000100000011
0000000000000008	00000010000100010000000110110011
000000000000000C	00000000001000000011001000000011
0000000000000010	00000000001100000011001010000011
0000000000000014	00000010010000101000001100110011
0000000000000018	01000000011000011000001110110011

Table 15: Instruction Memory operation summary. This table lists the cycle-by-cycle behavior of the Instruction Memory module, showing the fetched instructions for corresponding `pc` values.

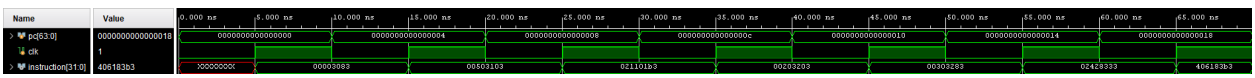

Figure 13: Waveform generated during Instruction Memory testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

Program Counter

The Program Counter (PC) module is an essential component of the 64-bit RISC-V single-cycle CPU. It determines the address of the next instruction to be executed, updating based on input signals for reset, enable, and the next PC value. The module supports synchronous updates on the rising edge of the clock, with reset functionality to initialize the PC to zero and enable functionality to selectively update the PC value.

Inputs and Outputs The Program Counter module accepts several inputs to determine its behavior. The `pc_in` input is a 64-bit value specifying the next PC value. The `reset` signal, when asserted, resets the PC to zero on the rising edge of the clock. The `enable` signal allows updates to the PC with the value of `pc_in` when asserted, while a de-asserted `enable` signal causes the PC to hold its current value. The `clk` input synchronizes these operations. The module outputs the current PC value as `pc`. Table 16 summarizes the operations based on different input conditions.

`reset`	`enable`	`clk`	**Operation**
1	X	`posedge`	The PC is reset to 0.
0	1	`posedge`	The PC updates with the value of `pc_in`.
0	0	`posedge`	The PC holds its current value.

Table 16: Program Counter operations under different input conditions.

Testbench Description The testbench validated the functionality of the Program Counter by simulating various scenarios, including reset, enable, and hold operations. Each test ensured the Program Counter behaved as expected under different input conditions. For instance, the reset operation was verified to initialize the Program Counter to zero. Similarly, the enable signal was tested to confirm selective updates of the Program Counter with the value of `pc_in`. These tests were documented in a structured format for clarity and ease of analysis.

Results The testbench results confirm the correct behavior of the Program Counter module across all tested scenarios. For instance, when `reset = 1`, the PC was successfully reset to zero. Subsequent tests verified that the PC updated correctly when `enable = 1` and held its value when `enable = 0`. Table 17 summarizes the results, detailing the input conditions and corresponding PC behavior.

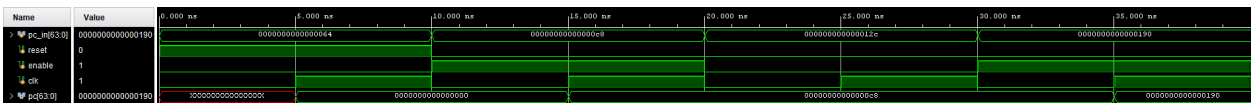

Figure 14: Waveform generated during Program Counter testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality across different scenarios.

clk	reset	enable	pc_in	pc
1	1	0	0100	0000
1	0	1	0200	0200
1	0	0	0300	0200
1	0	1	0400	0400

Table 17: Program Counter operation summary. This table shows the behavior of the PC module under various input conditions, including reset and enable operations.

Immediate Generator

The Immediate Generator is a crucial component of the 64-bit RISC-V single-cycle CPU. It generates a 64-bit sign-extended immediate value from the 32-bit instruction input. This immediate value is essential for supporting various instruction types, including I-Type, S-Type, B-Type, U-Type, and J-Type. The generator extracts and sign-extends specific fields of the instruction, adhering to the RISC-V ISA specifications.

Inputs and Outputs The Immediate Generator accepts a 32-bit instruction (`instr`) as input and produces a 64-bit immediate value (`imm`) as output. The extraction and sign-extension depend on the instruction's opcode, which determines the immediate format. For I-Type instructions, bits [31:20] are extracted and sign-extended to 64 bits. S-Type instructions combine bits [31:25] and [11:7] before sign-extending them to 64 bits. B-Type instructions combine bits [31], [7], [30:25], and [11:8], append a 0, and then sign-extend to 64 bits. U-Type instructions extract bits [31:12], append 12 zeros, and sign-extend to 64 bits. J-Type instructions combine bits [31], [19:12], [20], and [30:21], append a 0, and then sign-extend to 64 bits.

Testbench Description The testbench verified the functionality of the Immediate Generator by applying instructions corresponding to different immediate formats. Each instruction was decoded, and the generated immediate value was compared with the expected result. The correctness of the sign-extension for all tested formats was validated, and the results were displayed in a structured format for verification.

Results The testbench confirmed the correct operation of the Immediate Generator across all tested formats. For instance, an I-Type instruction with `instr = 0000000000110000 1000010100010011` produced the expected immediate value of `0000000000000003`. Similarly, a U-Type instruction with `instr = 00000000010100001000000010110111` correctly generated an immediate value of `0000000000508000`. Table 18 summarizes the observed behavior.

instr	imm
00000000001100001000010100010011	0000000000000003
00000000110000001010000000100011	0000000000000000
00000000101000001000000001100011	0000000000000000
00000000101000001000000010110111	0000000000508000
00000000101000000000111101101111	000000000000000A

Table 18: Immediate Generator operation summary. This table lists the 32-bit instructions and the corresponding 64-bit immediate values generated by the module.

Figure 15: Waveform generated during Immediate Generator testing. This figure illustrates the decoding of instructions and the generation of immediate values, validating the module's functionality.

2-to-1 Multiplexer

The 2-to-1 Multiplexer is a combinational circuit that selects one of two 64-bit inputs, based on a single-bit select signal (`sel`). It serves as a fundamental building block in the single-cycle RISC-V CPU, routing data to the desired path depending on the control signal.

Inputs and Outputs The multiplexer accepts three inputs: two 64-bit data inputs (`in0` and `in1`) and a 1-bit select signal (`sel`). The output (`out`) is a 64-bit value that matches either `in0` or `in1`, depending on the value of `sel`. The behavior of the multiplexer is summarized in Table 19.

sel	in0	in1	out
0	Selected	Ignored	Matches `in0`
1	Ignored	Selected	Matches `in1`

Table 19: 2-to-1 Multiplexer operation summary.

Testbench Description The testbench evaluated the functionality of the multiplexer by testing all combinations of the select signal and input values. Different scenarios were simulated, including selecting zero, selecting all ones, and choosing between mixed values. The results were recorded to ensure that the multiplexer behaved as expected for each test case.

Results The testbench results confirmed the correct operation of the 2-to-1 Multiplexer across all tested scenarios. For instance, when `sel = 0` and `in0 = 0000000000000005`, the output matched `in0`. Similarly, when `sel = 1` and `in1 = AAAAAAAAAAAAAAAA`, the output correctly matched `in1`. Table 20 summarizes the results.

sel	in0	in1	out
0	0000000000000005	AAAAAAAAAAAAAAAA	0000000000000005
1	0000000000000005	AAAAAAAAAAAAAAAA	AAAAAAAAAAAAAAAA
0	0000000000000000	0000000000000000	0000000000000000
1	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
0	123456789ABCDEF0	FEDCBA9876543210	123456789ABCDEF0
1	123456789ABCDEF0	FEDCBA9876543210	FEDCBA9876543210

Table 20: 2-to-1 Multiplexer operation summary. This table shows the behavior of the multiplexer for different values of `sel`, `in0`, and `in1`.

Figure 16: Waveform generated during 2-to-1 Multiplexer testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

Shift Left by 1

The Shift Left by 1 module is a fundamental component in the RISC-V single-cycle CPU. It takes a 64-bit input value and shifts all bits left by one position, with the least significant bit padded with 0. This operation is crucial for address calculations and other arithmetic tasks.

Inputs and Outputs The module accepts a 64-bit input (`in`) and produces a 64-bit output (`out`), where the input value has been shifted left by one bit. This behavior ensures compatibility with the CPU's addressing and computational needs.

Testbench Description The testbench validated the module's functionality by simulating various 64-bit input values and verifying the corresponding outputs. The `in` value was set to diverse test cases, ranging from edge cases (e.g., all zeros or all ones) to mixed bit patterns, ensuring the module's correctness in all scenarios. The results were presented in a cycle-by-cycle format to confirm accurate functionality.

Results The testbench confirmed the module's correct operation for all test cases. For example, an input of `0000000000000001` produced an output of `0000000000000002`, and an input of `123456789ABCDEF0` resulted in an output of `2468ACF13579BDE0`. Table 21 summarizes the observed behavior.

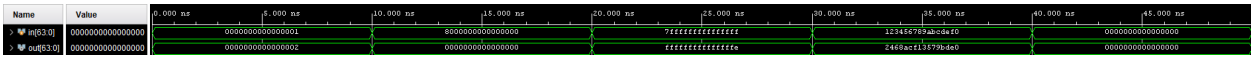

Figure 17: Waveform generated during Shift Left by 1 testing. This figure illustrates the transitions of inputs and outputs, validating the module's functionality.

in	out
0000000000000001	0000000000000002
8000000000000000	0000000000000000
7FFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFE
123456789ABCDEF0	2468ACF13579BDE0
0000000000000000	0000000000000000

Table 21: Shift Left by 1 operation summary. This table shows the input-output relationship for the testbench cases.

Processor Integration and Instruction Execution

This section provides a comprehensive explanation of how the processor modules are integrated and describes the step-by-step execution of instructions loaded in the instruction memory. Each instruction's execution is detailed with reference to the cycle-by-cycle flow of control and data through the processor. The simulation results, **presented as printed statements due to the waveform's length**, correspond to the instruction memory and the functionality of the implemented processor.

Instruction Memory and Corresponding Cycles

The instruction memory is initialized with the following program:

1. `LD R1, 0x00`: Load the value at address `0x00` (`a`) into `R1`.
2. `LD R2, 0x05`: Load the value at address `0x05` (`f`) into `R2`.
3. `MUL R3, R1, R2`: Multiply `R1` and `R2` (`a * f`) and store the result in `R3`.
4. `LD R4, 0x02`: Load the value at address `0x02` (`c`) into `R4`.
5. `LD R5, 0x03`: Load the value at address `0x03` (`d`) into `R5`.
6. `MUL R6, R4, R5`: Multiply `R4` and `R5` (`c * d`) and store the result in `R6`.
7. `SUB R7, R3, R6`: Subtract `R6` from `R3` (`a * f - c * d`) and store the result in `R7`.
8. `LD R8, 0x04`: Load the value at address `0x04` (`e`) into `R8`.
9. `ADD R9, R7, R8`: Add `R7` and `R8` ((`a * f - c * d`) `+ e`) and store the result in `R9`.
10. `LD R10, 0x01`: Load the value at address `0x01` (`b`) into `R10`.
11. `DIV R11, R9, R10`: Divide `R9` by `R10` (((`a * f - c * d`) `+ e`) `/ b`) and store the result in `R11`.
12. `SD R11, 0x06`: Store the value in `R11` (`y`) at address `0x06`.

The processor executes these instructions cycle by cycle, with the following outputs generated:

Cycle	PC	Opcode	Branch	MemRead	MemtoReg	ALUOp	MemWrite	ALUSrc	RegWrite	ALU Result	DivBy0
0	0000000000000004	00000011	0	1	1	00	0	1	1	0000000000000000	0
1	0000000000000008	00000011	0	1	1	00	0	1	1	0000000000000005	0
2	000000000000000c	00110011	0	0	0	10	0	0	1	0000000000000bb8	0
3	0000000000000010	00000011	0	1	1	00	0	1	1	0000000000000002	0
4	0000000000000014	00000011	0	1	1	00	0	1	1	0000000000000003	0
5	0000000000000018	00110011	0	0	0	10	0	0	1	000000000001d4c0	0
6	000000000000001c	00110011	0	0	0	10	0	0	1	fffffffffffe36f8	0
7	0000000000000020	00000011	0	1	1	00	0	1	1	0000000000000004	0
8	0000000000000024	00110011	0	0	0	10	0	0	1	fffffffffffe3702	0
9	0000000000000028	00000011	0	1	1	00	0	1	1	0000000000000001	0
10	000000000000002c	00110011	0	0	0	10	0	0	1	ffffffffffffdb8	0
11	0000000000000030	00100011	0	0	0	00	1	1	0	0000000000000006	0
12	0000000000000034	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
13	0000000000000038	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
14	000000000000003c	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
15	0000000000000040	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
16	0000000000000044	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
17	0000000000000048	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
18	000000000000004c	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0
19	0000000000000050	0xxxxxxx	0	0	0	00	0	0	0	xxxxxxxxxxxxxxxx	0

```
Data Memory Contents (Actual Byte Addresses):
Format: Address (decimal) | Value (decimal, signed).
Address:        0 | Value:                      1000.
Address:        8 | Value:                       200.
Address:       16 | Value:                       300.
Address:       24 | Value:                       400.
Address:       32 | Value:                        10.
Address:       40 | Value:                         3.
Address:       48 | Value:                      -584.

Register File Contents:
Format: Register Number (decimal) | Value (decimal, signed).
Register:  0 | Value:                         0.
Register:  1 | Value:                      1000.
Register:  2 | Value:                         3.
Register:  3 | Value:                      3000.
Register:  4 | Value:                       300.
Register:  5 | Value:                       400.
Register:  6 | Value:                    120000.
Register:  7 | Value:                   -117000.
Register:  8 | Value:                        10.
Register:  9 | Value:                   -116990.
Register: 10 | Value:                       200.
Register: 11 | Value:                      -584.
```

Figure 18: Combined output showing execution cycles, data memory contents, and register file state after program execution.

Cycle-by-Cycle Execution Flow

1. **Cycle 0:** The Program Counter (`PC`) starts at `0x0004`, fetching the instruction `LD R1, 0x00`. The Control Unit decodes the opcode `00000011` (Load operation). This activates the `MemRead` and `RegWrite` control signals, while `ALUSrc` is set to 1 to enable the immediate value `0x00` as an operand for the ALU. The ALU computes the effective memory address, which is `0x00`, and the data memory returns the value of `a`. The result (`a`) is written to register `R1`. No flags (`ZeroFlag`, `DivideByZeroFlag`) are active in this cycle.

2. **Cycle 1:** The next instruction, `LD R2, 0x05`, is fetched at `PC = 0x0008`. The Control Unit decodes the same opcode `00000011`. The `ALUSrc` signal is again set to 1, allowing the immediate value `0x05` to be sent to the ALU. The ALU computes the effective memory address, `0x05`, and the data memory outputs the value of `f`. This value is

written to register R2, and the datapath remains active for memory read operations. No flags are triggered.

3. **Cycle 2:** At PC = 0x000C, the instruction MUL R3, R1, R2 is executed. The Control Unit decodes the opcode 00110011, enabling RegWrite. The Register File outputs R1 (containing a) and R2 (containing f) as operands to the ALU. The ALU is configured for multiplication (ALUOp = 10), and it computes a * f. The result is stored in R3. The datapath enables Register File write operations. No flags are active during this cycle.

4. **Cycle 3:** The instruction LD R4, 0x02 is fetched at PC = 0x0010. The Control Unit decodes the opcode 00000011, activating MemRead and RegWrite. The immediate value 0x02 is passed as an operand to the ALU via the ALUSrc signal. The ALU computes the effective memory address, 0x02, and the data memory outputs the value c. The result is written to R4. No flags are activated in this cycle.

5. **Cycle 4:** The instruction LD R5, 0x03 is fetched at PC = 0x0014. Following the decoding of opcode 00000011, the immediate value 0x03 is passed to the ALU. The ALU computes the memory address, 0x03, and retrieves the value d from data memory. This value is stored in R5. The Control Unit ensures that the memory read and register write signals are appropriately enabled.

6. **Cycle 5:** At PC = 0x0018, the instruction MUL R6, R4, R5 is executed. The Control Unit configures the ALU for multiplication (ALUOp = 10). The Register File provides the values of R4 (c) and R5 (d) as operands. The ALU computes the product, c * d, and stores the result in R6. No flags are triggered in this cycle.

7. **Cycle 6:** The instruction SUB R7, R3, R6 is fetched at PC = 0x001C. The Control Unit configures the ALU for subtraction (ALUOp = 10). The Register File outputs R3 (containing a * f) and R6 (containing c * d) as operands. The ALU computes the result, a * f - c * d, and stores it in R7. The datapath ensures that the result is written back to the Register File.

8. **Cycle 7:** At PC = 0x0020, the instruction LD R8, 0x04 is fetched. The immediate value 0x04 is passed to the ALU, which computes the memory address. The data memory outputs the value e, which is written to R8.

9. **Cycle 8:** The instruction ADD R9, R7, R8 is executed at PC = 0x0024. The Register File provides R7 and R8 as operands, corresponding to (a * f - c * d) and e. The ALU computes the sum, (a * f - c * d) + e, and stores it in R9. The datapath writes the result back to the Register File.

10. **Cycle 9:** The instruction LD R10, 0x01 is fetched at PC = 0x0028. The immediate value 0x01 is passed to the ALU to compute the effective memory address. The data memory outputs the value b, which is written to R10.

11. **Cycle 10:** At PC = 0x002C, the instruction DIV R11, R9, R10 is executed. The Register File provides R9 and R10 as operands, corresponding to ((a * f - c * d)

`+ e)` and `b`, respectively. The ALU performs the division operation and stores the quotient in `R11`. The `DivideByZeroFlag` remains inactive because `R10` is non-zero.

12. **Cycle 11:** The instruction `SD R11, 0x06` is fetched at `PC = 0x0030`. The Register File provides `R11` (containing `y`), which is written to the data memory at address `0x06`.

Conclusion

This project demonstrates the successful design and implementation of a 64-bit single-cycle processor capable of executing a subset of RISC-V instructions. By meticulously integrating core modules—such as the Program Counter, Instruction Memory, ALU, Register File, Control Unit, and Data Memory—into a cohesive architecture, the processor efficiently executes arithmetic, logical, memory access, and control flow operations.

The instruction set was carefully chosen to validate the processor's capabilities, encompassing operations like multiplication, division, addition, subtraction, and load/store instructions. The functional correctness of the processor was verified through a simulation of a specific program, which calculates the expression $((a \cdot f) - (c \cdot d) + e)/b$ and stores the result `y` in memory. The results printed during the simulation illustrate step-by-step execution, showcasing how each instruction propagates through the processor's datapath, from fetching and decoding to memory access and write-back.

Special attention was given to the modularity of the design, enabling clear datapath flows and efficient debugging. The Control Unit played a pivotal role in decoding instructions and generating control signals for each module. The ALU's functionality was extended to handle signed and unsigned arithmetic operations, including division and multiplication, with a `DivideByZeroFlag` to handle edge cases. Additionally, the processor's performance was validated using printed simulation outputs instead of waveforms due to the extensive nature of the results, making them more accessible for analysis.

This implementation highlights the significance of modular design in processor development, ensuring scalability and flexibility for future enhancements. The processor could be further extended to support additional instructions, pipelining for improved performance, or multi-cycle execution for reduced complexity. The insights gained from this project form a solid foundation for advanced processor designs and architecture studies.