32-Bit Central Processing Unit

Electrical Circuitry and Assembler

Built by Krishna

Based on:
Computer Organization and Design MIPS Edition (5th ed., 2014)
David Patterson and John Hennessey

Table of Contents

1 Overview

1.1 Summary

This project was inspired by a Computer Organization class, where we explored the low-level design and implementation of MIPS architecture. I chose to recreate a 32-bit CPU to deepen my understanding of low-level architecture. At times, I deviated from the MIPS design to simplify certain elements or due to gaps in available documentation. The completed CPU can execute many standard assembly instructions. I also developed a Python assembler that converts assembly code into machine code, which can be loaded into the CPU's memory and executed on the simulated hardware.

1.2 Tools

The electrical circuitry is built and simulated using Logisim-Evolution 3.9.0, and the assembler is implemented in Python. Although Logisim provides many pre-built CPU components, I aimed to avoid using them whenever possible, opting instead to construct components from basic logic gates (AND, OR, etc.) to achieve a more fundamental design. However, in some cases, I used built-in components to work around simulation errors that arose in Logisim's mode, which shouldn't have affected the circuit. The documentation for each component includes additional context on these design choices.

1.3 Documentation

The documentation will primarily focus on the design and explanation of the circuitry, with an additional section dedicated to the assembler at the end. Most of the content will delve into the details of the circuitry, including insights into specific design decisions and the rationale behind them where relevant. This structure aims to provide a comprehensive understanding of the CPU's construction, while also highlighting the reasoning behind certain choices made during the project.

2 CPU Instructions

2.1 Structure

The CPU uses 32-bit instructions, each containing a 6-bit opcode field that specifies the operation to be executed. The CPU interprets and parses the remaining 26 bits differently depending on which of the two instruction types is indicated. This approach allows the CPU to dynamically adapt its decoding process for different instruction formats, providing the flexibility to handle varied operations within the same instruction width.

2.2 R-Type

R-Type instructions operate exclusively with registers and are primarily used for arithmetic operations. They consist of several key fields within the 32-bit instruction:

- 6-bit Opcode: Specifies the high-level category of the operation.
- Three 5-bit Register Fields:
    - Source Register: Provide the first operand register for the operation.
    - Target Register: Provide the second operand register for the operation.
    - Destination Register: Holds the result of the operation.
- 5-bit Shift Amount: Specifies the number of bits to shift left or right in operations.
- 6-bit Function Field: Provides additional specification for the operation, allowing multiple functions to share the same opcode. For instance, both add and subtract can be represented by the same opcode but differentiated through function values.

This structure allows R-Type instructions to efficiently manage register-based operations, with flexibility for both arithmetic and bitwise manipulation based on the function and shift fields.

31-26 [6-Bits]	25-21 [5-Bits]	20-16 [5-Bits]	15-11 [5-Bits]	10-6 [5-Bits]	5-0 [6-Bits]
Opcode	Source Reg.	Target Reg.	Dest. Reg.	Shift Amount	Function

2.3 I-Type

I-Type instructions support operations that use an "immediate" value, a constant embedded directly within the instruction rather than stored in a register:

- 6-bit Opcode: Specifies the high-level category of the operation.
- Two 5-bit Register Fields.
- 16-bit Immediate Value: A constant value used as an operand, enabling operations with a direct value without needing an additional register.

This format allows I-Type instructions to efficiently handle operations that require immediate values, such as loading constants or performing arithmetic with a fixed value.

31-26 [6-Bits]	25-21 [5-Bits]	20-16 [5-Bits]	15-0 [16-Bits]
Opcode	Source Reg.	Target Reg.	Immediate

3 1-Bit Data Pathway Control

3.1 Multiplexor

A simple multiplexer receives two 1-bit inputs, labeled A and B, along with a 1-bit selector, S. When S is off (0), the output X takes the value of input A. When S is on (1), the output X takes the value of input B. This behavior allows the multiplexer to select between A and B based on the state of the selector S.

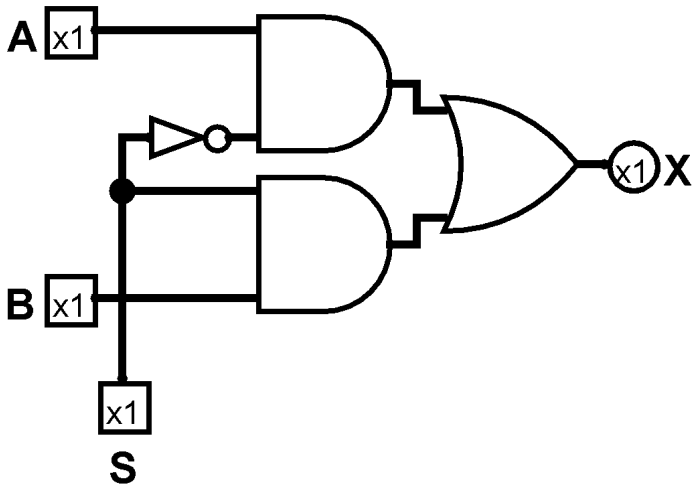

Fig 3.1: Simple Multiplexor

A	**B**	**S**	**X**
0	*X*	0	**0**
1	*X*	0	**1**
X	0	1	**0**
X	1	1	**1**

Fig 3.1a: Simple Multiplexor Truth Table

3.2 2-to-1 Multiplexor

A more versatile multiplexer receives four 1-bit inputs—A, B, C, and D—along with a 2-bit selector that can be represented by two 1-bit inputs, S1 and S2. The selector bits control which input is routed to the output X:

- When both S1 and S2 are off (00), input A is sent to output.
- When S1 is off and S2 is on (01), input B is sent to output.
- When S1 is on and S2 is off (10), input C is sent to output.
- When both S1 and S2 are on (11), input D is sent to output.

This arrangement enables the selection of one of four inputs based on the binary state of the selector bits S1 and S2.

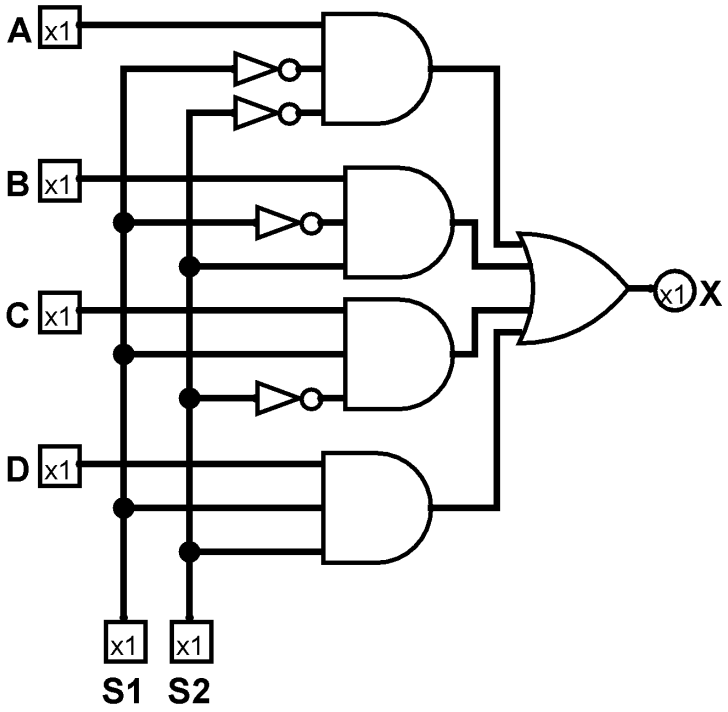

Fig 3.2: 2-to-1 Multiplexor

A	**B**	**C**	**D**	**S1**	**S2**	**X**
0	*X*	*X*	*X*	0	0	**0**
1	*X*	*X*	*X*	0	0	**1**
X	0	*X*	*X*	0	1	**0**
X	1	*X*	*X*	0	1	**1**
X	*X*	0	*X*	1	0	**0**
X	*X*	1	*X*	1	0	**1**
X	*X*	*X*	0	1	1	**0**
X	*X*	*X*	1	1	1	**1**

Fig 3.2a: 2-to-1 Multiplexor Truth Table

Note: Throughout the architecture, various multiplexers are used, all following the same selection principle. For an N-to-1 multiplexer (where N is the number of inputs), the selector bits must be wide enough to represent values from 0 to N-1. For example, a 32-to-1 multiplexer requires a 5-bit selector, as 5 bits can represent values from 0 to 31, allowing each of the 32 inputs to be selected.

4 Arithmetic

4.1 Adder

The adder accepts two 1-bit inputs, A and B, along with a carry-in bit, which enables multi-bit arithmetic by propagating carries between adder stages. It produces two 1-bit outputs: the sum and the carry-out.

- The carry-out is set to 1 when at least two of the three inputs (A, B, and carry-in) are on, indicating that the addition has overflowed this bit and requires a carry.
- The sum output is set to 1 when an odd number of inputs (either one or three) are on.

To achieve this, an XOR gate is used to determine the sum output, as it checks if an odd number of inputs is on, providing the correct sum value. This logic allows for both single-bit and multi-bit additions when the adder is chained.

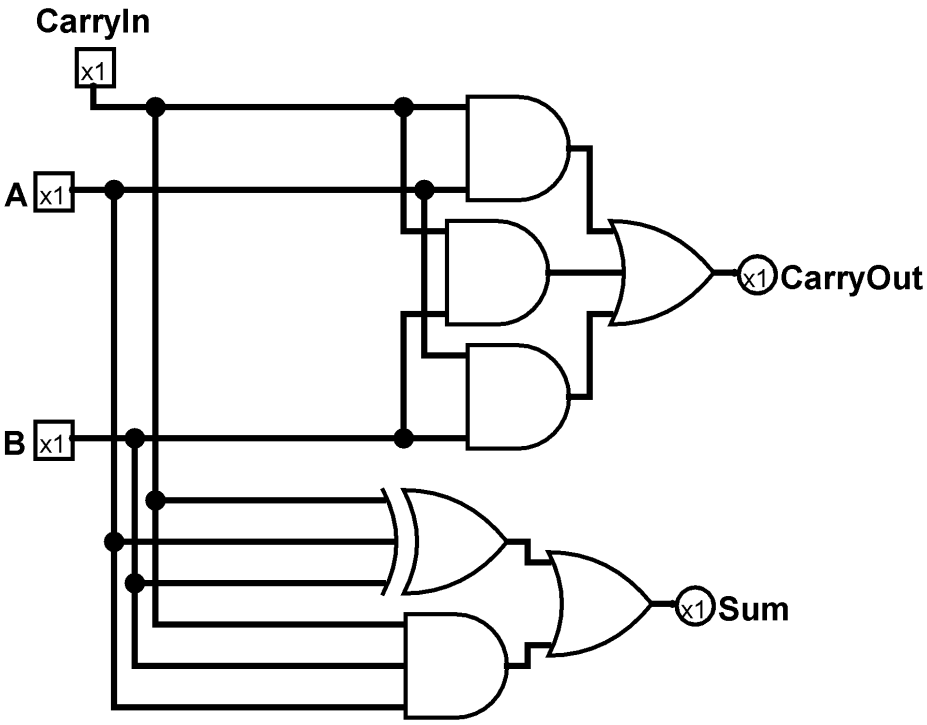

Fig 4.1: Adder

A	B	Cin	Cout	Sum
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Fig 4.1a: Adder Truth Table

4.2 1-Bit ALU

The 1-bit ALU receives two data inputs, A and B, along with several control inputs:

- Less Bit: A control bit used for operations in multi-bit ALUs, not relevant here.
- Operation Bits (O1 and O2): These 2-bit inputs are fed into a 2-to-1 multiplexer to select one of four operations the ALU can perform on the data inputs. The supported operations are:
    - ANDing A and B
    - ORing A and B
    - Adding A and B
    - Passing through the less bit directly as output
- Input Inversion Controls (A Invert and B Invert): Two basic multiplexers can invert the A and B inputs based on these control bits, allowing flexibility in the operations performed, such as creating a subtraction by inverting B.
- Addition with Carry: For the ADD operation, the ALU also takes a carry-in bit and produces a carry-out bit, essential for multi-bit arithmetic when chaining multiple 1-bit ALUs together. This carry-out output allows the ALU to pass along overflow.

Together, these inputs and control bits enable the 1-bit ALU to perform various logical and arithmetic functions, making it a versatile building block for constructing multi-bit ALUs.

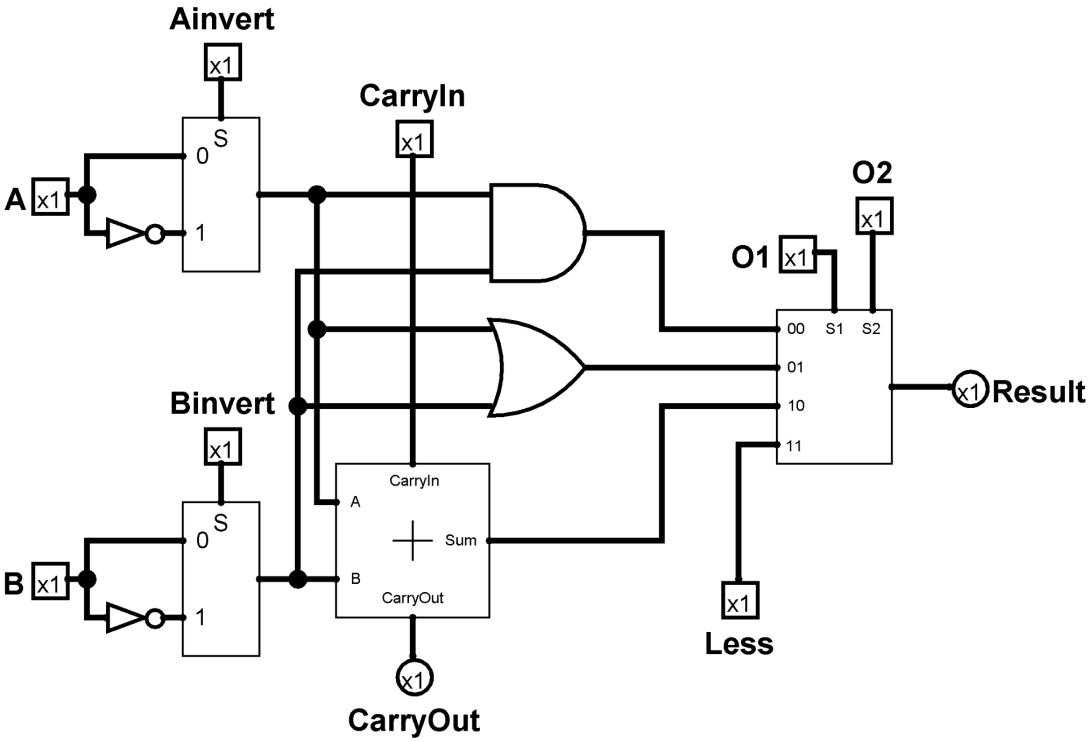

Fig 4.2: 1-Bit ALU

4.2 32-Bit ALU

The Main ALU processes two 32-bit data inputs and is controlled by a 4-bit operation signal. The four operation bits are for the following 1-bit ALU inputs:

- A Invert
- B Invert
- O1
- O2

The ALU is constructed by linking 32 individual 1-bit ALUs in parallel, each responsible for one bit of the 32-bit operation.

- The carry-out from the final 1-bit ALU is output as an overflow bit, which signals when an arithmetic operation exceeds the 32-bit boundary.
- If all 1-bit ALU outputs result in 0, a "zero" output is activated. This signal is useful for conditional branching, detecting cases where an operation produces a zero.

This parallel setup allows the Main ALU to handle 32-bit arithmetic and logical operations effectively, while overflow and zero detection enhance its capability to support complex CPU operations like conditional branching.

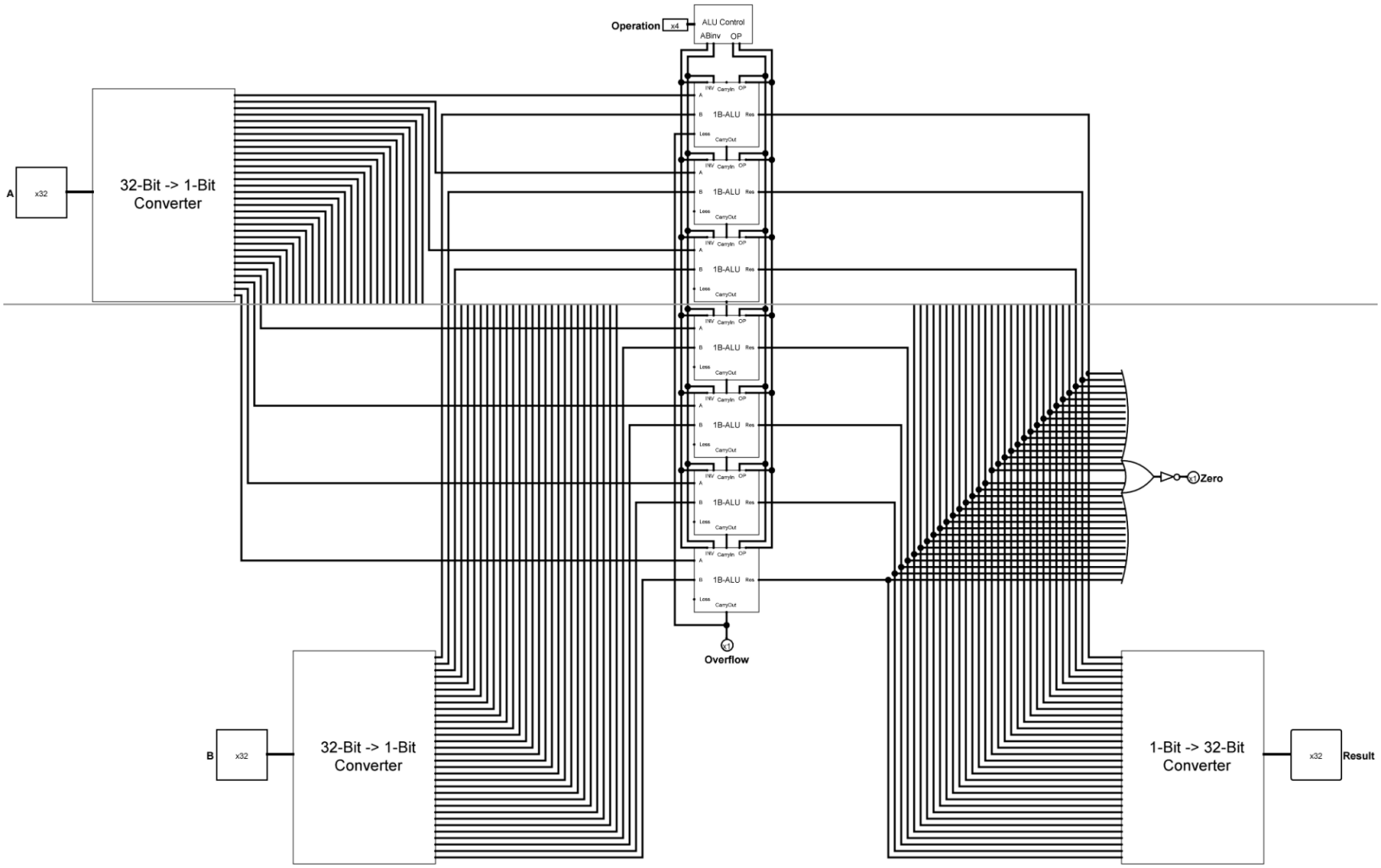

Fig 4.3: 32-Bit ALU (truncated along horizontal gray line)

5 Memory and Registers

5.1 D-Latch

The D-Latch serves as the fundamental memory element for registers and other storage components in this architecture. It has two inputs:

- Clock (C): Controls when data is allowed to pass through to the output.
- Data (D): Holds the value to be stored.

When the clock input is on (1), the D-Latch allows the Data input to pass through to the output (Q). When the clock is off (0), the output (Q) retains its last value, and any changes to the Data input have no effect on the output.

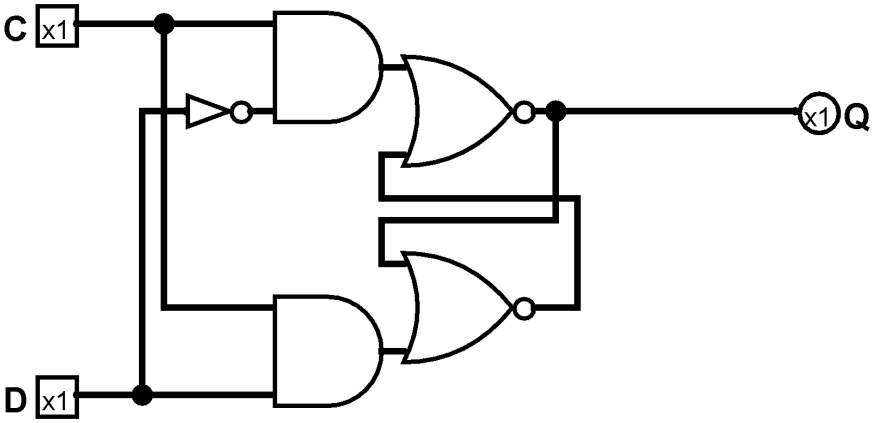

Fig 5.1: D-Latch

In most cases, the register only needs to update its value on the rising edge of the clock signal—when the clock transitions from low (0) to high (1)—rather than when the clock remains high or transitions from high to low. This behavior is known as edge-triggered operation, specifically rising edge-triggered. To achieve this, the register can be configured with a setup known as a master-slave D flip-flop, which combines two D-latches. The first D-latch receives an inverted clock signal, while the second D-latch receives a normal clock.

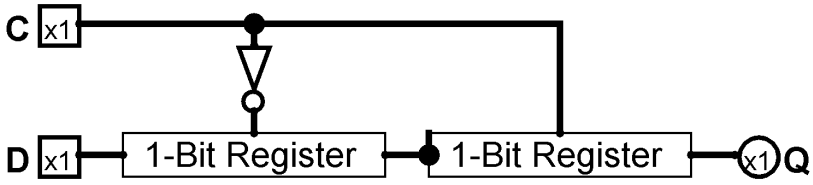

Fig 5.2: Rising Edge Triggered D-Latch

Note: In Logisim's simulation mode, initializing a D-Latch can lead to an "E" or Error state at the output, which disrupts the rest of the circuit. To address this, the built-in D-Latch component will be used instead, as it reliably initializes and avoids these issues.

5.2 32-Bit Register

The 32-bit register is built using the same design principle as the Main ALU, by placing 32 rising-edge-triggered D-latches in parallel. Each D-latch receives one bit of the 32-bit data input, allowing the entire input to be stored across the latches. The outputs of the individual D-latches are then combined to form a unified 32-bit output. A single 1-bit clock signal is distributed to all latches, ensuring that the register updates synchronously on the rising edge of the clock.

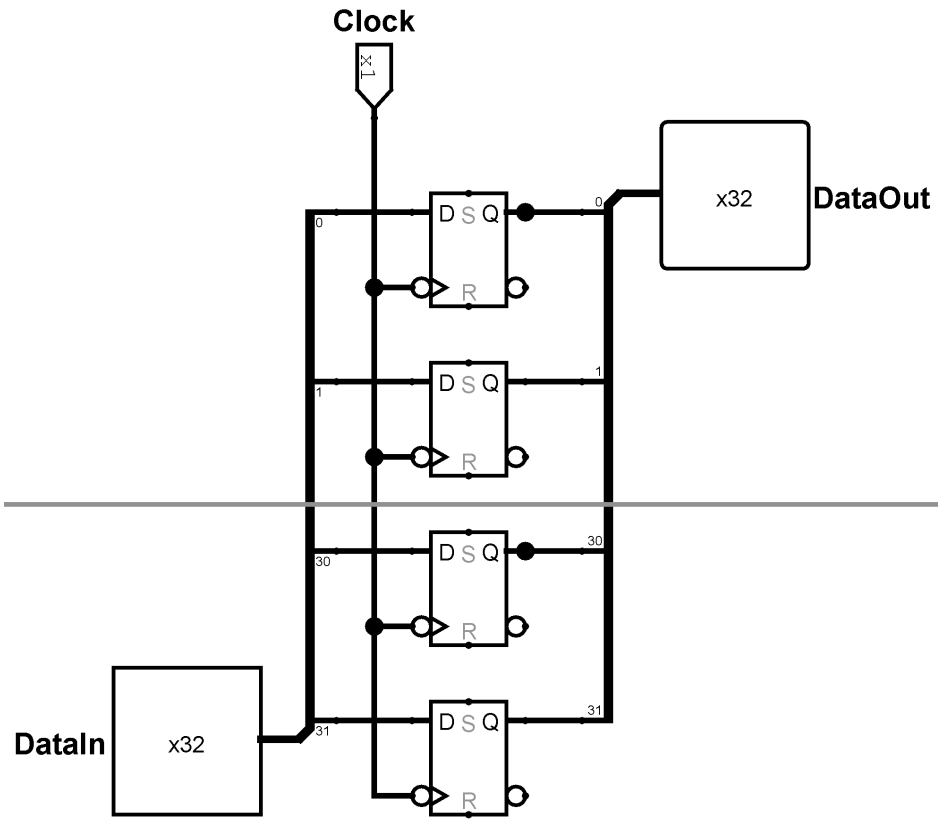

Fig 5.3: A Single 32-Bit Register (truncated along horizontal gray line)

Note: The built-in Logisim D-latch includes additional pins that provide extra functionality: Inverse of Q outputs the complement of Q, set (S) forces Q to 1 when activated, reset (R) forces Q to 0 when activated. However, these additional inputs are not required in the architecture. As a result, they are left unconnected.

5.3 Register File

The register file consists of multiple 32-bit registers arranged in parallel. Each register receives a 32-bit data input and a 1-bit clock signal. Here's how the components interact:

- Write Address Decoding: A 5-bit write address is decoded into 32 individual signals, each enabling one of the registers.
- Data Outputs: Each register's 32-bit output is split into two sets, feeding two 32-to-1 multiplexers. These multiplexers allow selective reading from a register.
- Read Address Selection: Each multiplexer has a 5-bit selector to choose one of the 32 registers based on a read address.

This structure enables simultaneous reading from two registers while controlling which register is written to, making it efficient for parallel data access.

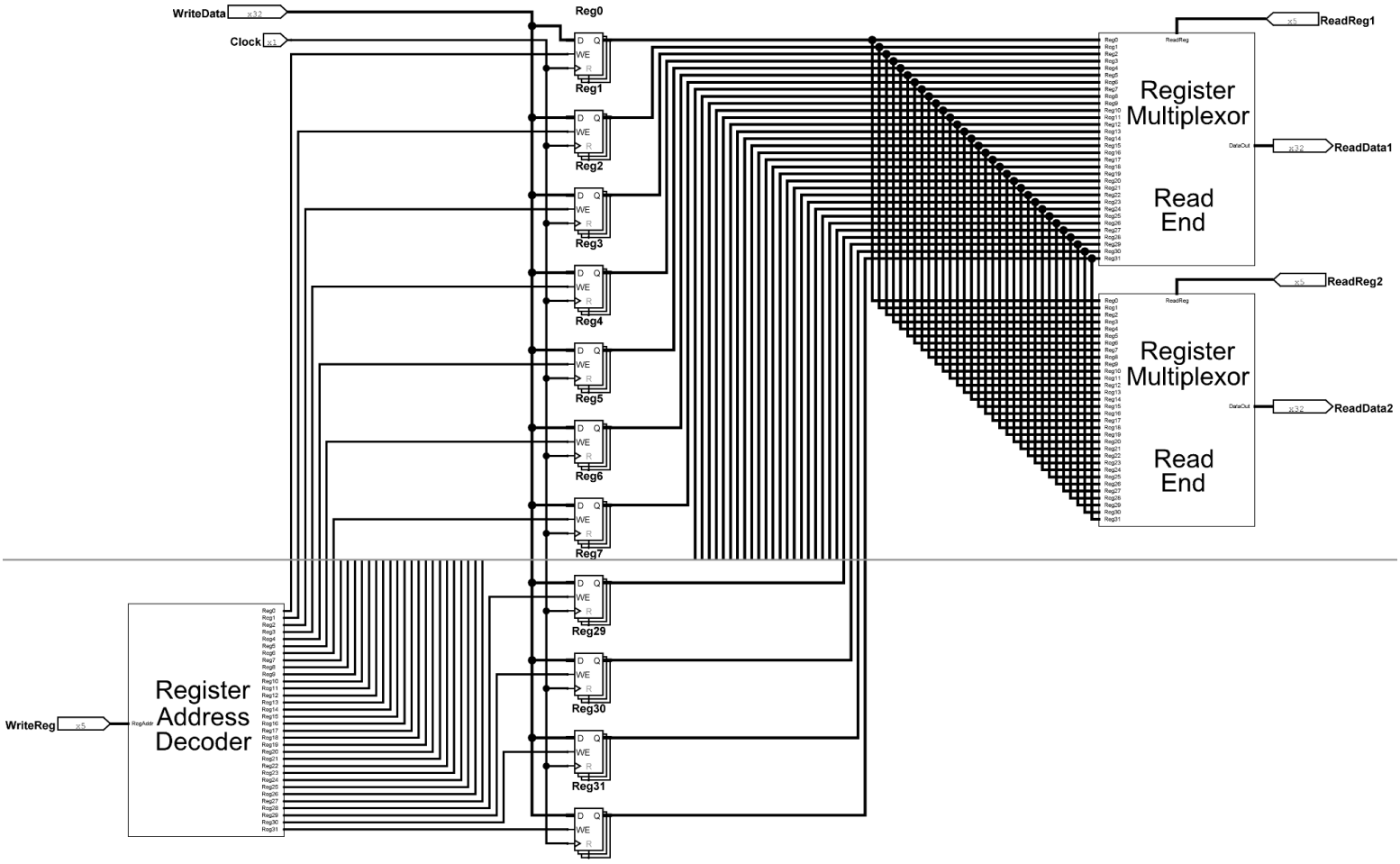

Fig 5.4: Register File (truncated along horizontal gray line)

Note: To enhance performance in Logisim's simulation mode, the built-in 32-bit register components are used. These registers have separate inputs for the clock signal and a decoder value. In earlier designs, only a single clock input was used. In those cases, an AND gate combines the clock signal and the decoder value.