**ECE 310**
**Project 2 Report**

Aryan Goyal

March 1, 2025

**Abstract**

This project presents the design and implementation of a digital system that computes the arithmetic expression $(A - B) + (C - D)$ using an 8-bit shared input interface. The system sequentially captures operands through a structured control mechanism using a 2-bit operand selector (`op`) and a capture signal (`capture`). Synchronization is managed by a free-running clock, with an active-low synchronous reset (`reset_n`) ensuring proper initialization. The computation is completed within 10 clock cycles after receiving the final operand, and the output is validated with a `valid` signal. The design employs a combination of dataflow modeling and behavioral modeling for the D flip-flop, ensuring a robust controller and datapath.

# 1 Introduction

In this project, we will design and implement a digital system that processes 8-bit numerical inputs to compute the arithmetic expression $(A-B)+(C-D)$ and provide the result as an output. The system incorporates a structured input interface, where operands $A, B, C$, and $D$ are sequentially provided through a shared 8-bit input port (`d_in`). The operand selection is managed by a control signal (`op`) that specifies which value is currently being captured. Data is registered only on the rising edge of the clock when the `capture` signal is asserted.

Additionally, the system includes an active-low synchronous reset (`reset_n`) to initialize registers and ensure a defined starting state. The output computation must be completed within 10 clock cycles following the receipt of the final operand, with the `valid` signal being asserted for one cycle when the result is ready. This project emphasizes efficient sequential design, synchronization, and state management while adhering to specified timing constraints.

## 1.1 DataFlow Modelling

Dataflow modeling in Verilog describes a circuit using equations and continuous assignments rather than explicitly defining its structural composition with logic gates. This approach focuses on how data moves through the design and how outputs depend on inputs through logical and arithmetic operations. It primarily utilizes the `assign` statement and operators like `+`, `-`, and `|` to define behavior at a higher abstraction level. While gate-level modeling closely mirrors the physical hardware, dataflow modeling allows for more concise and readable code, making it easier to design and optimize complex circuits like adders.

## 1.2 Behavioral Modelling

Behavioral modeling in Verilog describes a circuit's functionality using high-level constructs rather than specifying individual logic gates or connections. It allows designers to define digital circuits in a way that closely resembles algorithmic descriptions, making it easier to design complex systems efficiently.

# 2 DataPath Design

## 2.1 Top-Level Signals

Figure 1 defines the input and output signals of the system. The `reset_n` signal is an active-low synchronous reset that initializes the system. The `clock` is a free-running input clock that synchronizes the circuit operation. The `d_in` signal is an 8-bit unsigned input used for data processing, while the `op` signal is a 2-bit operand selector that determines the operation to be performed. The `capture` signal is a 1-bit input indicating when data should be captured. The system outputs include `result`, a 9-bit unsigned value representing the computed output, and `valid`, a 1-bit indicator signaling the validity of the output.

| Signal | Direction | Description |
|---|---|---|
| reset_n | input | Active-low synchronous reset |
| clock | input | Free-running clock |
| d_in | input | 8-bit unsigned input |
| op | input | 2-bit operand selector |
| capture | input | 1-bit capture indicator |
| result | output | 9-bit unsigned result as output |
| valid | output | 1-bit valid indicator |

Table 1: Top-Level Ports

Figure 1: Top-Level Ports Table

## 2.2 D Flip Flop

The D flip-flop (`dff`) module (Figure 3,5) is implemented using a behavioral description. It models the behavior of a D flip-flop with an active-low synchronous reset (`reset_n`) using an `always` block that is triggered on the positive edge of the clock (`posedge clk`). The `posedge clk` sensitivity ensures that updates occur only at the rising edge of the clock cycle, mimicking real-world synchronous flip-flops. Within the `always` block, an `if` statement (`if (!reset_n)`) ensures that when the reset signal is asserted low (0), the output (`q`) is reset to 0. Otherwise, `q` is updated with the input (`d`) on each rising edge of the clock. This behavioral modeling approach enables an abstract description of the flip-flop functionality, allowing for a simple and efficient implementation without specifying gate-level behavior.

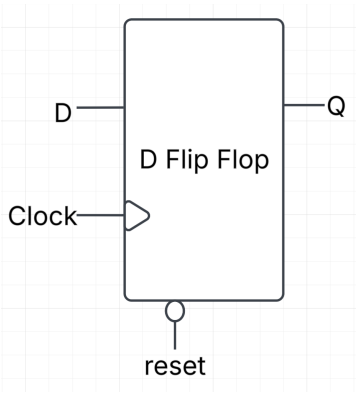


Figure 2: D Flip-Flop Diagram

## 2.3 8-bit Register

The 8-bit register (Figure 4, 5) consists of eight D flip-flops connected in parallel, sharing a common clock and reset signal. Each flip-flop stores one bit of the input data (`d_in`), and on the clock's rising edge, the register captures the input value if the capture signal is active. The stored data is then

available on the output (`Q`). This structure allows the register to function as a temporary storage element in digital systems, commonly used in data storage and synchronization applications.

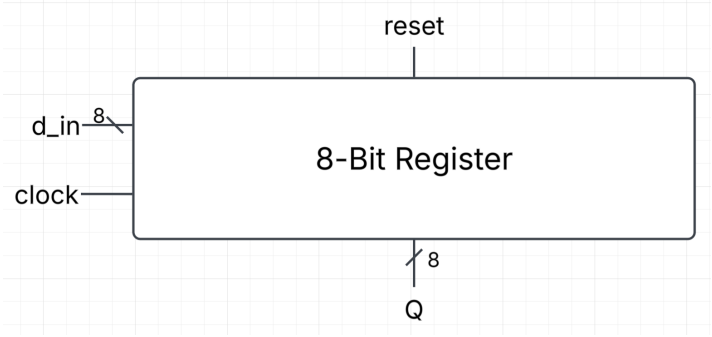


Figure 3: 8-bit Register Diagram

## 2.4 Full Adder

A Full Adder is a fundamental combinational circuit used in digital arithmetic to perform the addition of three input bits: `A`, `B`, and `Cin` (Carry-in). It produces two outputs: `Sum`, which represents the sum of the three inputs, and `Cout` (Carry-out), which indicates whether a carry is generated for the next stage in multi-bit addition.

The Full Adder is implemented using basic logic gates, including XOR, AND, and OR gates. The `Sum` output is derived using two XOR gates, where the first XOR computes the intermediate sum of `A` and `B`, and the second XOR combines this result with `Cin`, ensuring proper binary addition. The `Carry-out` is determined using a combination of AND and OR gates. This gate-level implementation of a Full Adder serves as the building block for constructing multi-bit adders such as the Ripple Carry Adder.

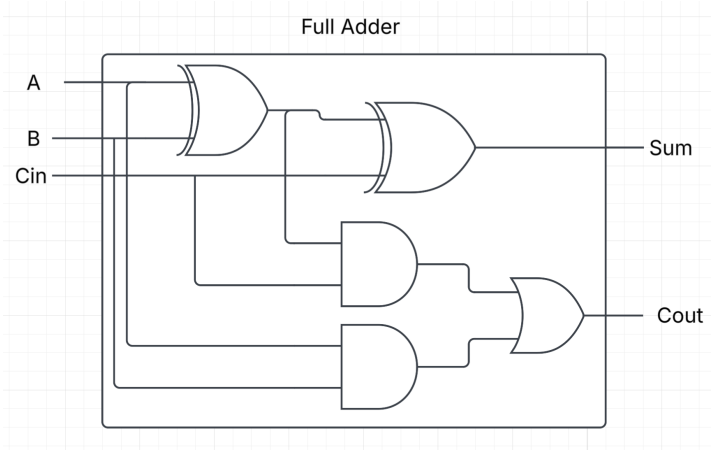


Figure 4: 8-bit Register Diagram

## 2.5 Ripple Carry Adder

A ripple carry adder is a digital circuit used to perform binary addition by cascading multiple full adder modules. The design shown consists of eight full adders connected in series to form an 8-bit ripple carry adder. It takes two 8-bit binary inputs, $A$ and $B$, along with an optional carry-in ($C_{in}$), and produces an 8-bit sum ($S$) along with a carry-out ($C_{out}$). The carry generated by each full adder propagates to the next stage, leading to a delay proportional to the number of bits. While simple and efficient in hardware implementation, its major limitation is the propagation delay caused by sequential carry computation.

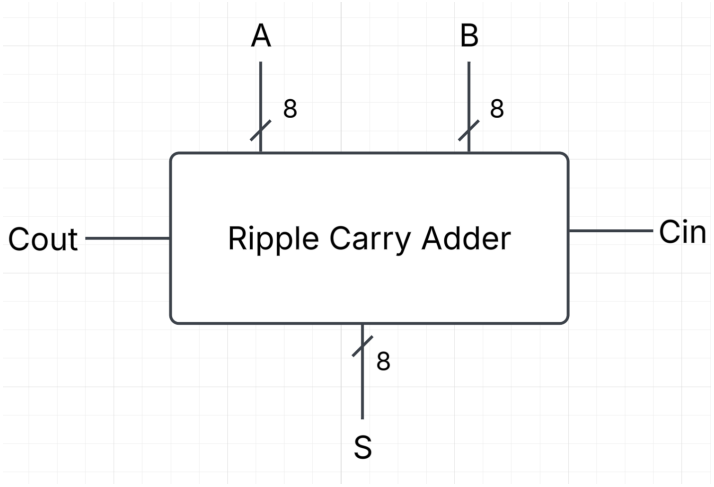


Figure 5: Ripple Carry Adder Diagram

## 2.6 Ripple Carry Subtractor

A ripple carry subtractor is a combinational circuit that performs binary subtraction using an 8-bit ripple carry adder with a two's complement approach. The circuit consists of eight full adders, where the second operand ($B$) is inverted using a bitwise negation, effectively converting subtraction into the addition of $A$ and $-B$ (two's complement). The circuit takes three inputs: two 8-bit operands ($A$ and $B$) and a carry-in ($C_{in}$), which acts as the borrow-in for subtraction. It produces an 8-bit difference ($S$) and a carry-out ($C_{out}$), which indicates whether a borrow occurred. This design is efficient for arithmetic operations but suffers from the same propagation delay as the ripple carry adder.

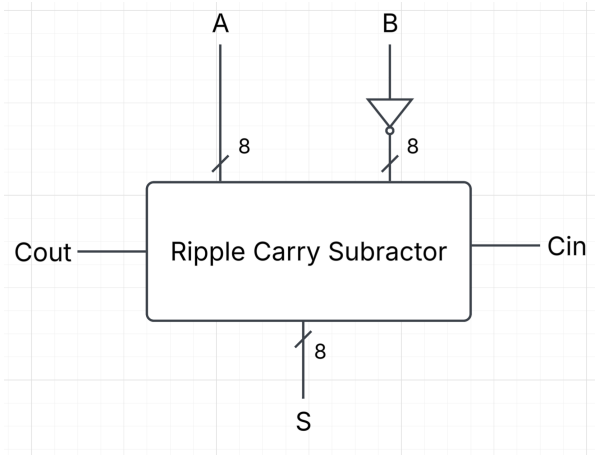


Figure 6: Ripple Carry Subtractor Diagram

## 2.7 De-multiplexer

The given diagram represents a 1-to-4 demultiplexer (Demux) designed using dataflow modeling in Verilog. The module takes an 8-bit input signal (`d_in`) and routes it to one of four output lines (`A, B, C`, or `D`), based on the 2-bit selection signal (`op[1:0]`). The demultiplexer functions by decoding the selection inputs and enabling the corresponding output while keeping the others at a logic low state. Using dataflow modeling, this design is implemented through continuous assignments with ternary (`?`) operators, ensuring efficient synthesis and optimization. Such a demultiplexer is fundamental in digital circuits for signal-routing applications.

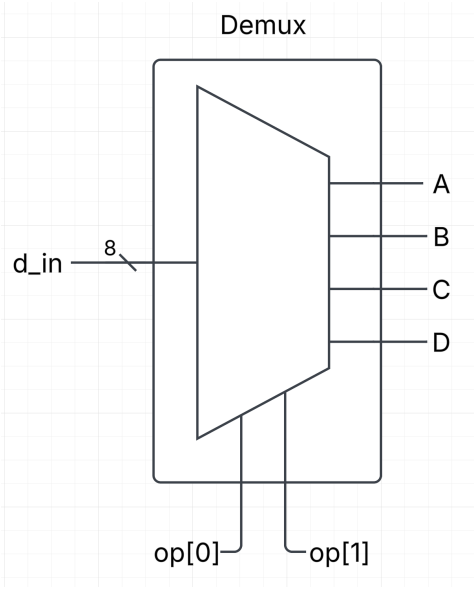


Figure 7: De-multiplexer Diagram

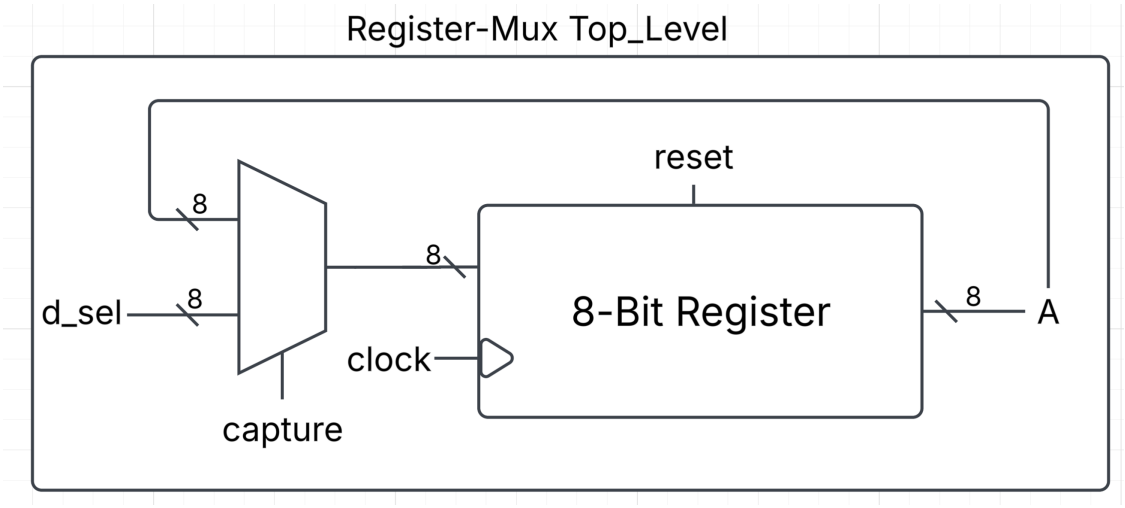


Figure 8: 8-bit Register and 2x1 Mux Module Diagram

## 2.8 Register-Mux Module

The given diagram represents a top-level design of a register-multiplexer system, incorporating an 8-bit register and a 2-to-1 multiplexer (Mux) for data selection. The multiplexer, labeled as `capture`, takes two 8-bit inputs—one from the external signal `d_sel` and the other from the feedback loop connected to the register output (`A`). Based on the selection control signal, the Mux determines which input is forwarded to the register. The register, clocked by the clock signal and resettable via the reset signal, stores the selected data and outputs it through `A`. This design enables conditional data capture and storage, making it useful for sequential logic applications where data needs to be retained or updated selectively. The use of a Mux allows dynamic switching between new input data and the stored value, ensuring flexibility in data handling.

## 2.9 Top-Level Design

Figure 9 represents a datapath with arithmetic operations, including addition and subtraction, controlled by a finite state machine. The system comprises four 8-bit registers that store input values

and a 9-bit register to hold the final computed result. Data is received through an 8-bit input (`d_in`) and distributed to the registers via a demultiplexer, which is controlled by a finite state machine. Multiplexers are used to select the appropriate inputs for arithmetic operations. The circuit includes two ripple carry subtractors and a ripple carry adder, responsible for performing arithmetic computations. The computed results are stored in a 9-bit register to account for possible carry-out or borrow conditions. A controller manages the operation flow based on control signals, ensuring correct data capture and computation execution.

The circuit contains a total of five registers (four 8-bit and one 9-bit), four 8-bit 2to1 multiplexers, and a demultiplexer. The number of D flip-flops utilized in the circuit amounts to $4 \times 8 + 1 \times 9 = 41$, accounting for the storage elements in all registers.

# 3 Controller Design

## 3.1 2-to-4 Decoder

A 2-to-4 decoder (Figure 10) with a capture signal is a combinational circuit that takes a 2-bit input and activates one of four output lines while all others remain low, depending on the binary value of the input. The inclusion of a capture signal acts as an enable control, ensuring that the decoder outputs are only active when `capture` is asserted; otherwise, all outputs remain at a default low state. Implemented using dataflow modeling in Verilog, the design utilizes the conditional (`? :`) operator to determine the output states dynamically. Specifically, the decoder checks the `capture` signal first, and if it is high, it decodes the 2-bit input using conditional assignments to drive the corresponding output. This type of decoder is fundamental in digital systems for control signal generation.

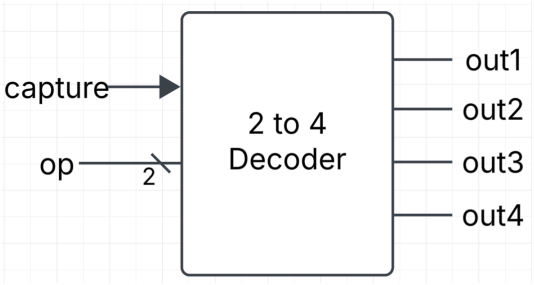


Figure 10: 2-to-4 Decoder Diagram

## 3.2 Design

The controller (Figure 11) is responsible for managing the sequential operation of the system, ensuring proper synchronization of data capture and processing. It consists of a state machine, a set of D flip-flops to store operand load flags, and a decoder to facilitate state transitions. The operand capture process is coordinated by a 2-to-4 decoder, which interprets a 2-bit control signal (`op`) to select the appropriate operand register (`dff A, dff B, dff C, dff D`). These D flip-flops are used to indicate whether each operand has been loaded, and their outputs are combined using a logic gate to generate the `all_loaded` signal, which signifies the completion of the operand-loading phase. The `capture` signal ensures that data is stored only when it's turned on, preventing accidental overwrites or data corruption.

The state machine (Figure 12) operates using a 2-bit state register, which determines the next operational phase of the controller. It updates its state based on the current inputs and clock cycles, ensuring that computation is ready for retrieval only when all operands are captured. The state transitions are governed by a combinational logic block, which outputs the next state based on the `capture` signal and the `all_loaded` flag. Another 2-to-4 decoder is used to generate control signals (`Cap1, Cap2, Cap3, Cap4`), which correspond to different stages of the operation. Additionally, the controller asserts the `Valid` signal once the computation is complete, indicating that the result is ready for retrieval. The design balances sequential and combinational logic to achieve an efficient and reliable control mechanism for arithmetic computation.

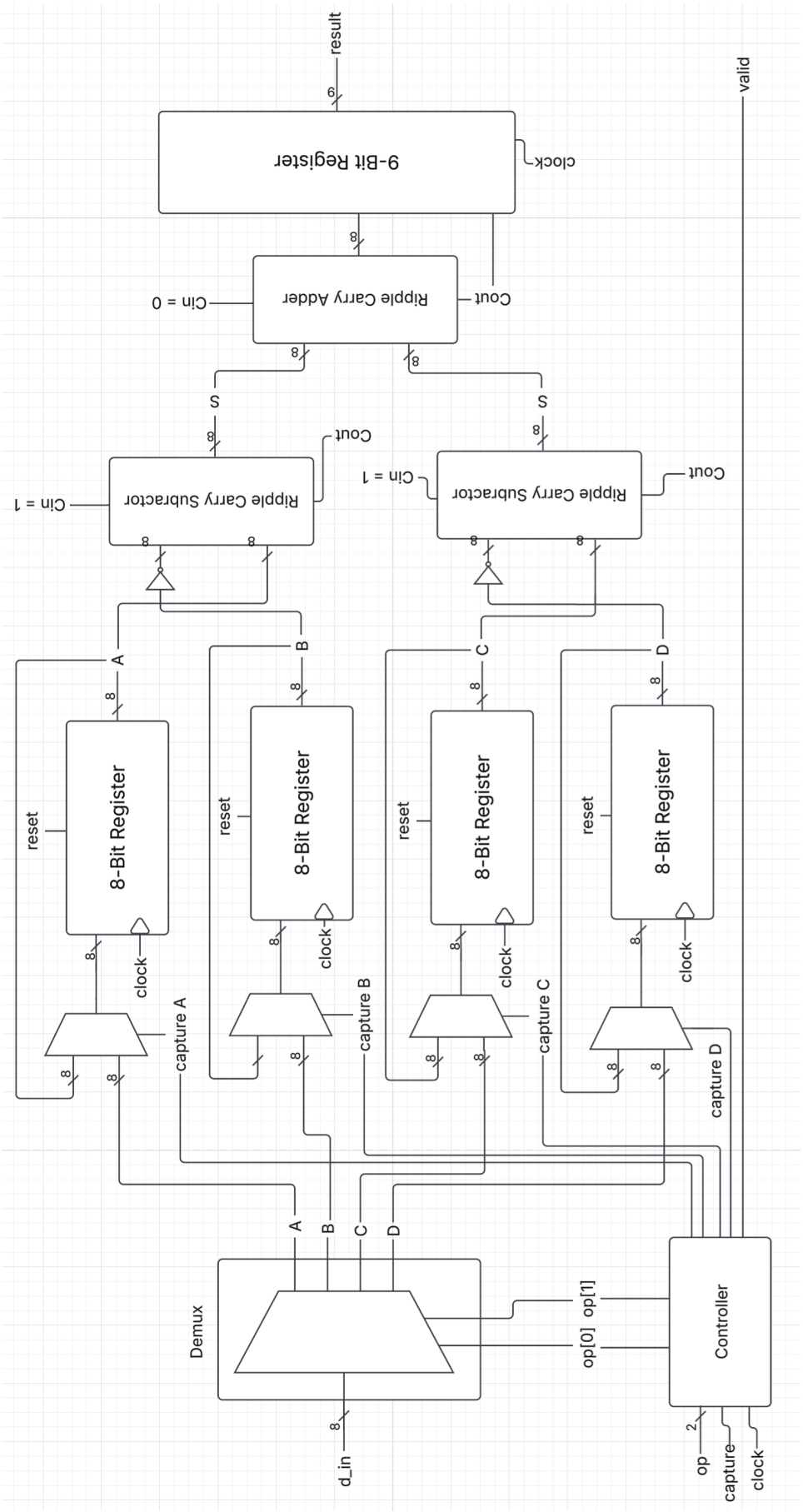


Figure 9: Complete Top-Level Diagram of the Datapath

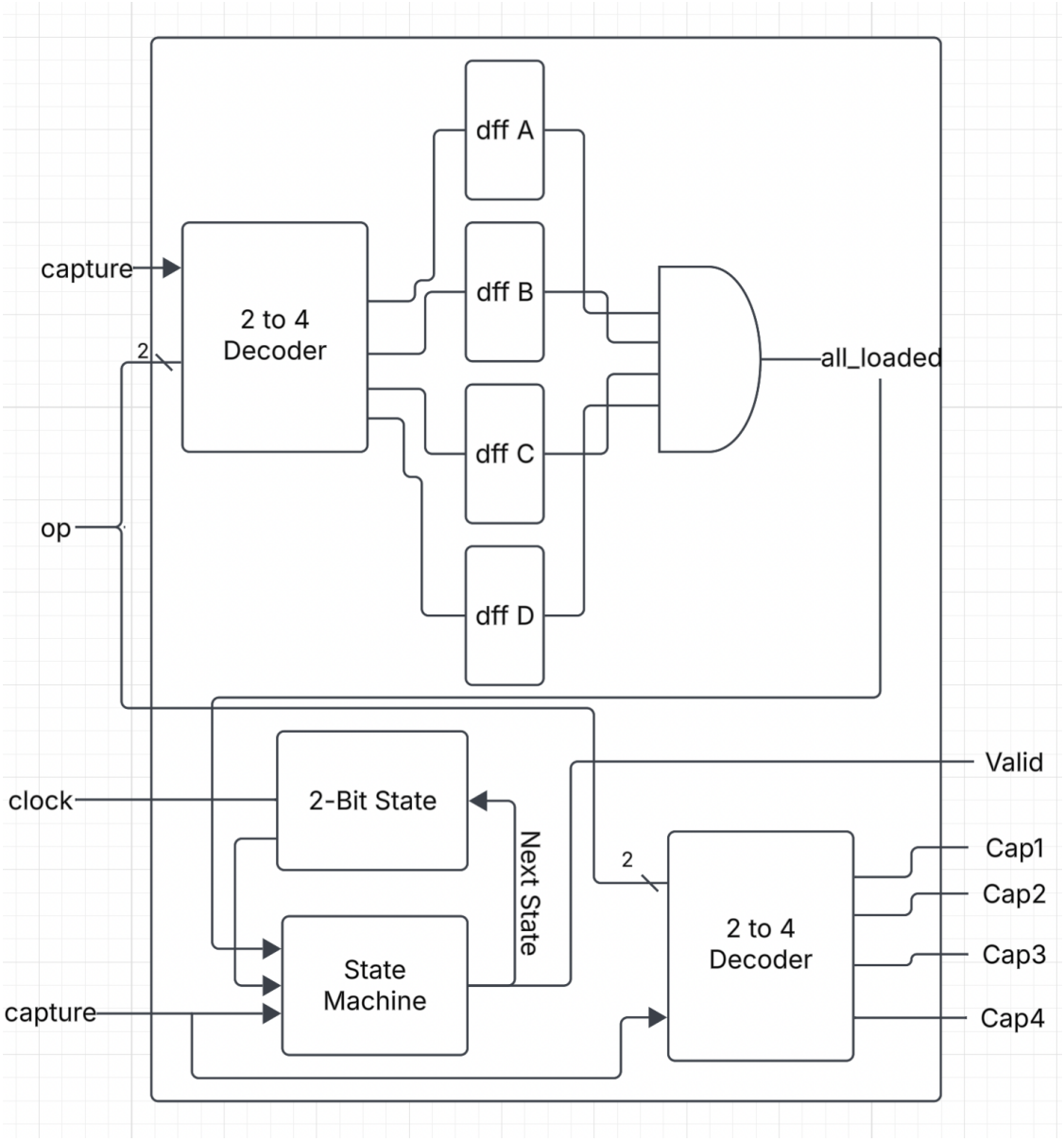


Figure 11: Controller Diagram

### 3.3 State Machine

The state machine depicted in Figure 12 consists of three states: **IDLE (00)**, **CAPTURE (01)**, and **VALID (10)**. It transitions between these states based on the control signals `capture` and `all_loaded`. The system starts in the **IDLE** state, where `capture = False`. When `capture` is asserted (`True`), the system moves to the **CAPTURE** state, indicating data acquisition. If `all_loaded` remains `False`, it stays in this state; otherwise, it transitions to the **VALID** state when `all_loaded = True`, signifying that all necessary data has been successfully captured. From the **VALID** state, the system returns to **IDLE** when the process resets or completes, ensuring readiness for the next cycle. This finite state machine efficiently controls a sequential process, managing data capture and validation based on predefined conditions.

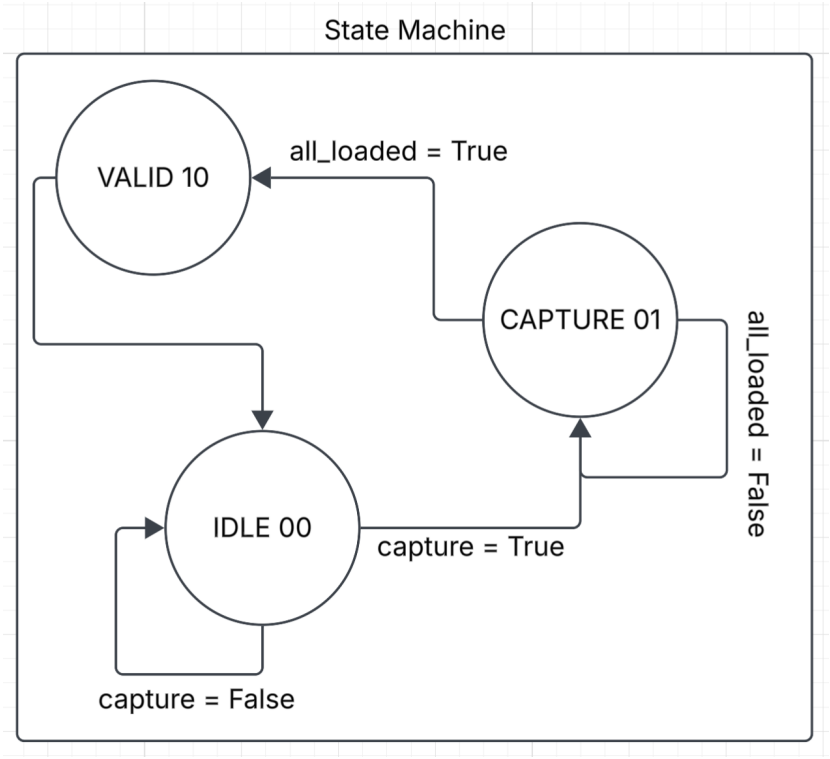


Figure 12: 3-State State Machine Diagram

# 4 Verilog Testbench

The three test cases shown in Figure 14 evaluate the functionality of a digital circuit by applying different combinations of inputs and verifying the expected output. Each test case consists of four primary inputs: $A_{in}$, $B_{in}$, $C_{in}$, and $D_{in}$, representing 8-bit hexadecimal values. The $OP_{in}$ input is an array that determines the operation order by specifying which inputs are involved in the arithmetic operations. The correct result for each test case is computed as:

$$(A_{in} - B_{in}) + (C_{in} - D_{in}) \tag{1}$$

with variations in the order of the operands set by $OP_{in}$. The `test_task` function applies the inputs to the design under test, and the obtained result is compared against the expected value. The simulation prints both the expected and received outputs for verification. The `pass` variable indicates whether the received output matches the expected result, and the total number of passed test cases is displayed at the end of the simulation. These test cases aim to verify the circuit's correctness under different input combinations and operand orders.

## 4.1 Test Case 6

Test Case 6 (Figure 13) evaluates the functionality of the arithmetic operation performed by the digital circuit using specific 8-bit hexadecimal input values. The inputs for this test case are:

- $A_{in} = 8'h88$ (136 in decimal)
- $B_{in} = 8'h77$ (119 in decimal)
- $C_{in} = 8'h66$ (102 in decimal)
- $D_{in} = 8'h55$ (85 in decimal)

The operation order is defined by the control input $OP_{in} = \{D, C, A, B\}$, meaning the circuit processes the inputs in the given sequence. The expected output is computed using the equation:

$$(A_{in} - B_{in}) + (C_{in} - D_{in})$$

Substituting the given values:

$$(136 - 119) + (102 - 85) = 17 + 17 = 34$$

Thus, the expected output of the circuit is $8'h22$ (34 in hexadecimal). The test case is considered successful if the received value matches the expected result, incrementing the pass counter accordingly.

### 4.1.1 Justification for Test Case 6

This test case is designed to verify the circuit's ability to correctly handle input subtraction and addition operations under different operand arrangements. Unlike standard cases, Test Case 6 introduces **modifications to the delay values**, where each input is applied with a different delay before processing:

- $de[0] = CLOCK\_PERIOD \times 5$
- $de[1] = CLOCK\_PERIOD \times 5$
- $de[2] = CLOCK\_PERIOD \times 5$
- $de[3] = CLOCK\_PERIOD \times 5$

By varying the delays, this test ensures that the system can handle asynchronous input arrivals while still computing the correct result. This simulates real-world conditions where data may not always arrive synchronously, making it a crucial functional test. Overall, Test Case 6 verifies both the arithmetic correctness of the circuit and its ability to function reliably under different timing constraints.

## 4.2 Test Case 10

Test Case 10 (Figure 14) evaluates the functionality of the arithmetic operation performed by the digital circuit using specific 8-bit hexadecimal input values. The inputs for this test case are:

- $A_{in} = 8'hFF$ (255 in decimal)
- $B_{in} = 8'h00$ (0 in decimal)
- $C_{in} = 8'hFF$ (255 in decimal)
- $D_{in} = 8'h01$ (1 in decimal)

The operation order is defined by the control input $OP_{in} = \{C, D, A, B\}$, which means that the circuit processes the inputs in the provided order. Once all the values are received, the system computes an output value using the equation:order:

$$(A_{in} - B_{in}) + (C_{in} - D_{in})$$

Substituting the given values:

$$(255 - 0) + (255 - 1) = 255 + 254 = 509$$

The expected output of the circuit is therefore $9'h1FD$ (509 in hexadecimal). The test case is considered passed if the received value matches the expected output. If successful, the test case counter is incremented, contributing to the total count of passed cases. This case helps validate whether the circuit correctly performs subtraction and addition operations with the specified input order.

### 4.2.1 Justification for Test Case 10

Test Case 10 is designed to evaluate the robustness of the digital circuit by pushing it to its operational limits. In this test, the inputs $A_{in}$ and $C_{in}$ are set to the maximum 8-bit value of $8'hFF$ (255 in decimal), while $B_{in}$ and $D_{in}$ are low single-digit values, specifically $8'h00$ (0 in decimal) and $8'h01$ (1 in decimal), respectively. This configuration ensures that the circuit processes both extreme and minimal values in a single operation.

Furthermore, the operation order is randomized using $OP_{in} = \{C, D, A, B\}$, altering the typical sequence of operations. This variation helps assess the circuit's ability to correctly handle different input arrangements and maintain accurate computations. By testing both large and small values in combination, this test effectively examines the circuit's ability to handle arithmetic operations near the upper limit of an 8-bit representation without overflow or computational errors.

Overall, this test case serves as a stress test for the system, ensuring that it can reliably process edge-case scenarios, making it a valuable addition to the verification process.

```
// TEST CASE-4
Ain = 8'h92; Bin = 8'h56; Cin = 8'h3D; Din = 8'h24;
OP_in = {C, D, B, A}; // Change order for variation
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 4: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Cin, Din, Bin, Ain}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;

// TEST CASE-5
Ain = 8'hFF; Bin = 8'h01; Cin = 8'hF0; Din = 8'h01;
OP_in = {D, C, A, B}; // Different operation order
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 5: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Din, Cin, Ain, Bin}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;

//Test Case 6
Ain = 8'h88; Bin = 8'h77; Cin = 8'h66; Din = 8'h55;
OP_in = {D, C, A, B};
de[0] = (CLOCK_PERIOD*5); de[1] = (CLOCK_PERIOD*5); de[2] = (CLOCK_PERIOD*5); de[3] = (CLOCK_PERIOD*5)
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 6: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Din, Cin, Ain, Bin}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;
```

Figure 13: 3 Test Cases from Testbench

```
//TestCase 8
Ain = 8'h5A; Bin = 8'h17; Cin = 8'h3F; Din = 8'h11;
OP_in = {D, C, A, B}; // Different operation order
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 8: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Din, Cin, Ain, Bin}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;

// TEST CASE-9
Ain = 8'h92; Bin = 8'h00; Cin = 8'h3D; Din = 8'h3D;
OP_in = {C, D, B, A}; // Change order for variation
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 9: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Cin, Din, Bin, Ain}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;

 // TEST CASE-10
Ain = 8'hFF; Bin = 8'h01; Cin = 8'hFF; Din = 8'h01;
OP_in = {C, D, A, B}; // Change order for variation
correct_result = (Ain - Bin) + (Cin - Din);
assert_reset();
$display("TEST CASE 10: A=%h B=%h C=%h D=%h OP(order of inputs)=%b",Ain,Bin,Cin,Din,OP_in);
test_task(OP_in, {Cin, Din, Ain, Bin}, correct_result, de[0], de[1], de[2], de[3]);
$display("EXPECTED: %h",correct_result);
$display("RECEIVED: %h",received);
if (pass) i = i + 1;
pass = 0;

$display("SCORE: %d/10 TEST CASES PASSED", i);
$finish;
```

Figure 14: 3 More Test Cases from Testbench

# 5 Results

The results from the ten test cases (Figures 15 and 16) confirm the correctness and reliability of our system. Each test case was carefully designed to evaluate different operational scenarios, including zero inputs, maximum and minimum values, random arithmetic computations, and varied input orders. The expected output values matched precisely with the received values in all ten cases, demonstrating that our system functions as intended. Furthermore, the diverse range of test inputs ensures that the circuit is robust and capable of handling different arithmetic operations under varying conditions. The consistent success across all cases verifies the accuracy of our implementation and validates the correctness of the underlying logic. These results provide strong evidence that our design meets the expected functional requirements and can reliably perform as per specifications.

# 6 Conclusion

The successful execution of all ten test cases (Figures 15 and 16) confirms the correctness and reliability of our digital system for computing the arithmetic expression $(A-B)+(C-D)$. The system accurately processes sequentially captured 8-bit inputs, adheres to the specified control mechanism, and produces the expected results within the defined timing constraints. The consistency between expected and

received outputs across diverse test scenarios validates the effectiveness of both the controller and datapath design. Additionally, the proper synchronization of inputs and the correct assertion of the `valid` signal demonstrate the robustness of the implementation. These results reinforce that the design meets functional requirements and performs as intended, making it a reliable solution for arithmetic computations in digital systems.

```
TEST CASE 1: A=00 B=00 C=00 D=00 OP(order of inputs)=00011011
PASS: Test case passed!
EXPECTED: 000
RECEIVED: 000
TEST CASE 2: A=55 B=55 C=55 D=55 OP(order of inputs)=00011011
PASS: Test case passed!
EXPECTED: 000
RECEIVED: 000
TEST CASE 3: A=f0 B=0f C=f0 D=f0 OP(order of inputs)=00011011
PASS: Test case passed!
EXPECTED: 0e1
RECEIVED: 0e1
TEST CASE 4: A=92 B=56 C=3d D=24 OP(order of inputs)=10110100
PASS: Test case passed!
EXPECTED: 055
RECEIVED: 055
TEST CASE 5: A=ff B=01 C=f0 D=01 OP(order of inputs)=11100001
PASS: Test case passed!
EXPECTED: 1ed
RECEIVED: 1ed
TEST CASE 6: A=88 B=77 C=66 D=55 OP(order of inputs)=11100001
PASS: Test case passed!
EXPECTED: 022
RECEIVED: 022
TEST CASE 7: A=81 B=72 C=9f D=2a OP(order of inputs)=00011011
PASS: Test case passed!
EXPECTED: 084
RECEIVED: 084
TEST CASE 8: A=5a B=17 C=3f D=11 OP(order of inputs)=11100001
PASS: Test case passed!
EXPECTED: 071
RECEIVED: 071
TEST CASE 9: A=92 B=00 C=3d D=3d OP(order of inputs)=10110100
PASS: Test case passed!
EXPECTED: 092
RECEIVED: 092
TEST CASE 10: A=ff B=01 C=ff D=01 OP(order of inputs)=10110001
PASS: Test case passed!
EXPECTED: 1fc
RECEIVED: 1fc
SCORE:          10/10 TEST CASES PASSED
```

Figure 15: Testbench results of 10 test cases

This report was compiled using LaTeX .

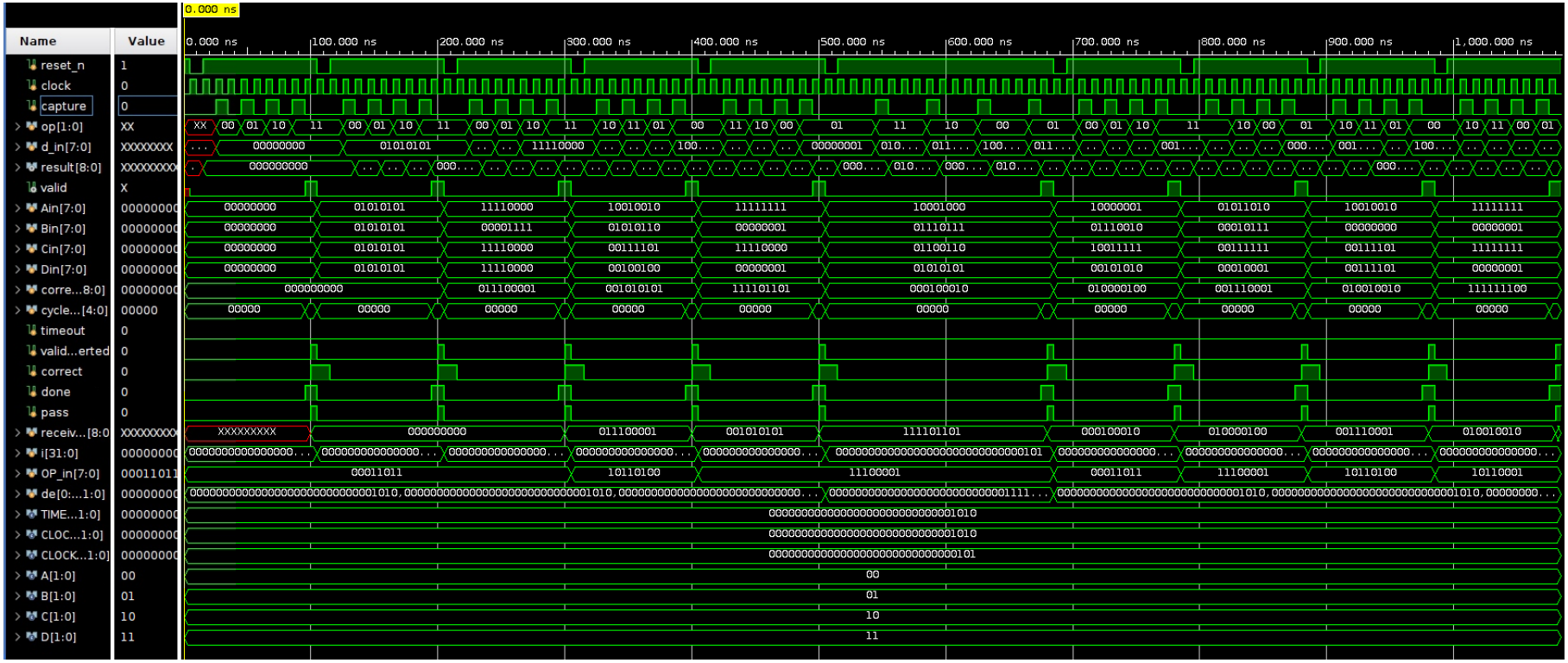


Figure 16: Waveform result of 10 test cases