ECE 310
Project 2 Report: Arithmetic Datapath and Controller

Ohm Patel

October 21, 2025

Abstract

This project implemented a 9-bit arithmetic datapath and control system in Verilog to perform the operation $(A + B) - (C + D)$ on four unsigned 8-bit operands. The design emphasized modular and hierarchical modeling using structural and dataflow Verilog, following project constraints that prohibit behavioral multi-bit arithmetic operators. The implementation includes a datapath for operand capture and arithmetic computation, and a controller for sequencing and single-cycle valid signal generation. Simulation verified full functional correctness, confirming synchronization between datapath and control logic. This report details the design methodology, implementation hierarchy, verification process, and timing considerations associated with sequential datapath control.

1 Introduction and Background

Arithmetic datapaths form the foundation of digital processing systems, serving as essential components in processors, ALUs, and embedded control units. The goal of this project was to integrate multiple arithmetic elements: adders, subtractors, and registers, into a cohesive hardware system capable of computing $(A + B) - (C + D)$ without using high-level arithmetic operators.

The project was divided into two main subsystems:

- **Datapath:** Captures four 8-bit operands and performs structural addition and subtraction.
- **Controller:** Manages operand sequencing and asserts a one-cycle `valid` signal once computation is complete.

This design emphasized modularity, hierarchical structure, and precise timing synchronization—fundamental principles of digital arithmetic system design.

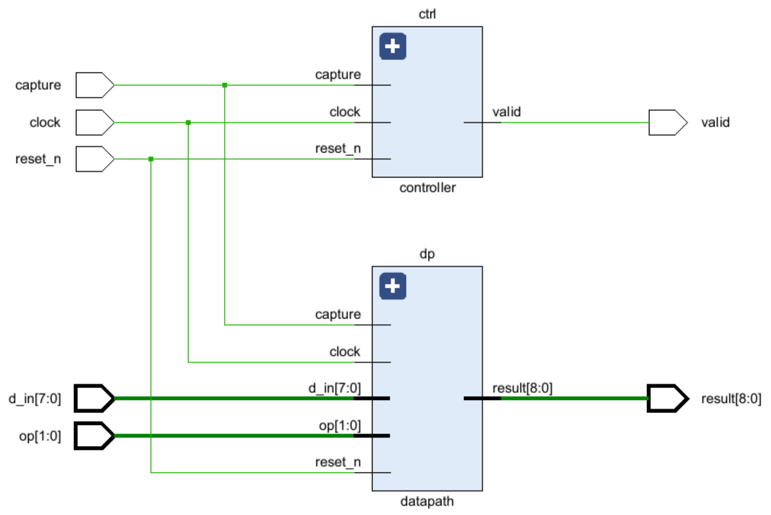

Figure 1: High-level overview of top-level module flow.

2 Design and Implementation

2.1 Overall Architecture

The top-level `Project2` module integrates the datapath and controller, shown conceptually in Figure 2. The controller governs operand loading and valid signal timing, while the datapath performs the arithmetic computation and stores the result.

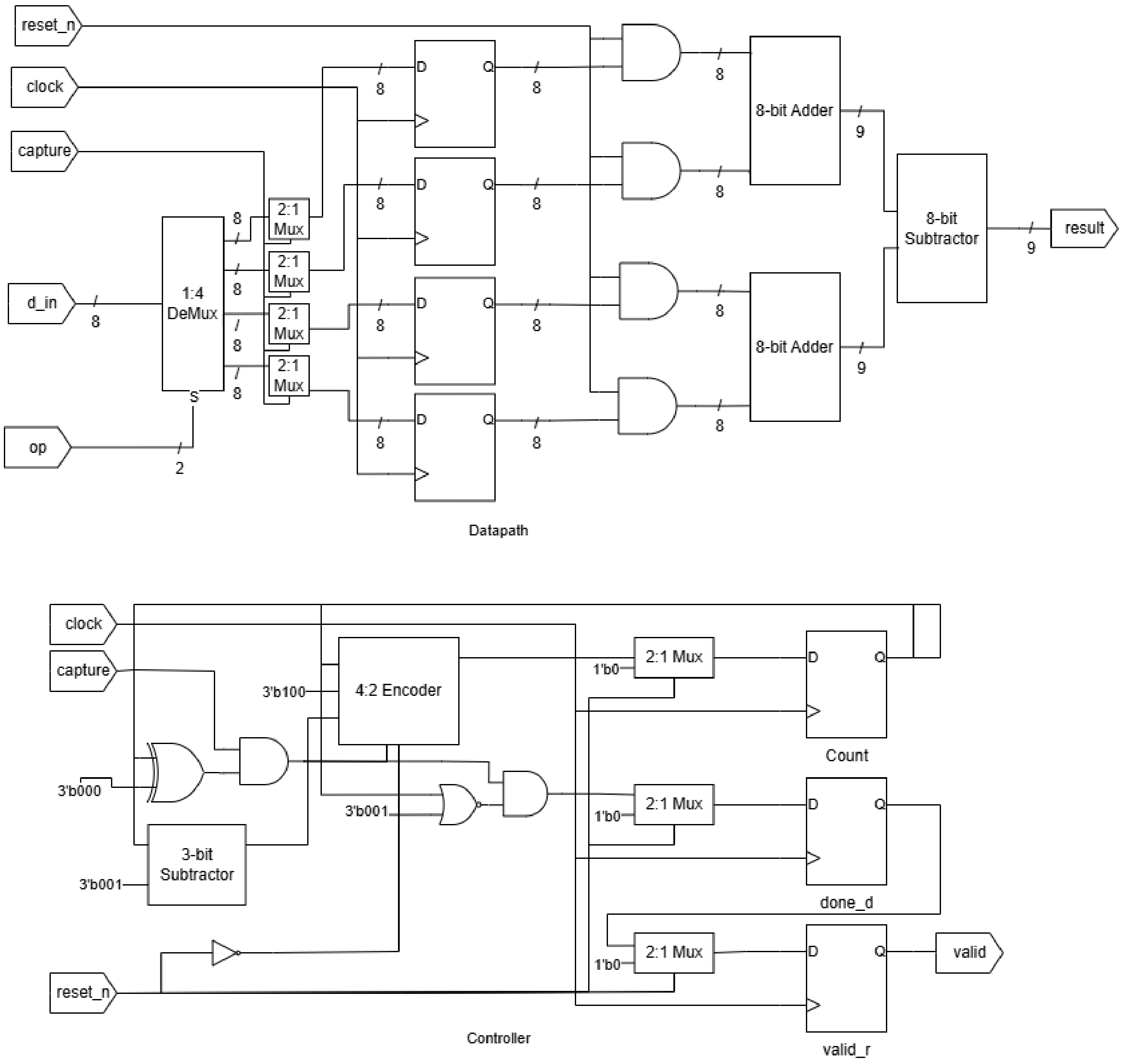

Figure 2: Block-level diagram of the Project 2 datapath and controller integration.

2.2 Datapath Design

The datapath captures four unsigned 8-bit operands (A, B, C, D) through conditional-load registers. Operands are selected via a 2-bit `op` code and stored when the `capture` signal is asserted. After capturing, the system computes $(A+B)$ and $(C+D)$ using ripple-carry adders, then performs subtraction through a 9-bit structural subtractor.

Listing 1: Datapath module performing (A+B) - (C+D)

```
module datapath(
    input clock,
    input reset_n,
    input [7:0] d_in,
    input [1:0] op,
    input capture,
    output [8:0] result
);
    reg [7:0] A, B, C, D;
    wire [7:0] demuxA, demuxB, demuxC, demuxD;

    assign demuxA = (op == 2'b00 && capture) ? d_in : A;
    assign demuxB = (op == 2'b01 && capture) ? d_in : B;
    assign demuxC = (op == 2'b10 && capture) ? d_in : C;
    assign demuxD = (op == 2'b11 && capture) ? d_in : D;

    always @(posedge clock) begin
        if (!reset_n) begin
            A <= 8'b0; B <= 8'b0; C <= 8'b0; D <= 8'b0;
        end else begin
            A <= demuxA; B <= demuxB; C <= demuxC; D <= demuxD;
        end end

    wire [8:0] sumAB, sumCD;
    rca_8bit AB (sumAB, A, B);
    rca_8bit CD (sumCD, C, D);
    subtractor_9bit sub (result, sumAB, sumCD);
endmodule
```

2.3 Controller Design

The controller manages operand capture sequencing and generates the one-cycle `valid` signal when all four operands have been received. A 3-bit countdown register is initialized to four and decrements on each capture using a structural 3-bit subtractor.

Listing 2: Controller module generating single-cycle valid signal

```
module controller(
    input clock, reset_n, capture,
    output valid
);
    reg [2:0] count; reg done_d, valid_r;
    wire [2:0] next_count, subtracted_count;
    wire decrement, done;
    assign decrement = capture & (count != 3'b000);
    subtractor_3bit sub3 (subtracted_count, count, 3'b001);

    assign next_count = (!reset_n)  ? 3'b100 :
                        (decrement) ? subtracted_count : count;
    assign done = decrement & (count == 3'b001);

    always @(posedge clock) begin
        if (!reset_n) begin
            count <= 3'b100; done_d <= 1'b0; valid_r <= 1'b0;
        end else begin
            count <= next_count; done_d <= done; valid_r <= done_d;
        end end

    assign valid = valid_r;
endmodule
```

2.4 Top-Level Integration

The `Project2` module instantiates both the datapath and controller, connecting arithmetic and control signals.

Listing 3: Top-level integration of datapath and controller

```
module Project2(
    input reset_n,
    input clock,
    input [7:0] d_in,
    input [1:0] op,
    input capture,
    output [8:0] result,
    output valid
);
    datapath dp (clock, reset_n, d_in, op, capture, result);
    controller ctrl (clock, reset_n, capture, valid);
endmodule
```

2.5 Arithmetic Units

All arithmetic components were implemented using fully structural design. Multi-bit adders and subtractors were constructed from 1-bit `full_adder` modules.

2.5.1 Full Adder

$$S = A \oplus B \oplus C_{in}, \quad C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

Listing 4: 1-bit Full Adder

```
module full_adder(
    output S, Cout,
    input A, B, Cin
);

    assign S = A ^ B ^ Cin;
    assign Cout = (A & B) | (Cin & (A ^ B));

endmodule
```

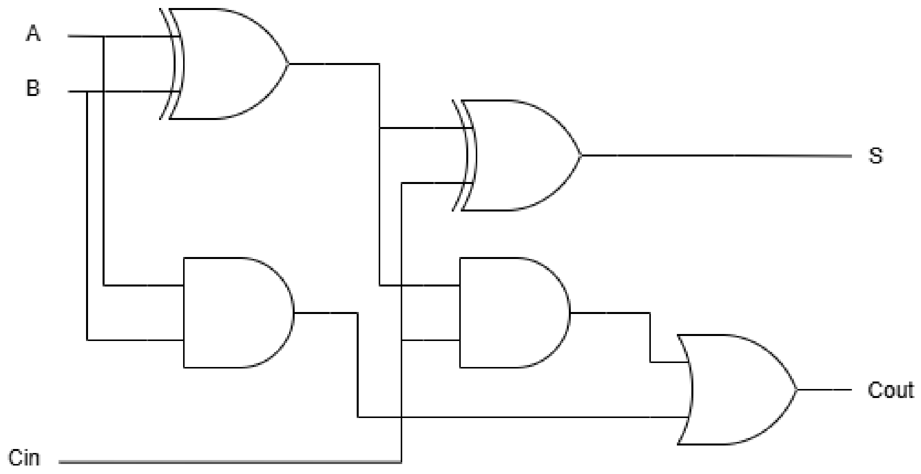

Figure 3: Gate-level diagram of the 1-bit full adder.

2.5.2 8-bit Ripple-Carry Adder (RCA)

Listing 5: 8-bit Ripple-Carry Adder

```
module rca_8bit(
    output [8:0] S,
    input [7:0] A, B
);
    wire [6:0] c;
    full_adder fa0 (S[0], c[0], A[0], B[0], 1'b0);
    full_adder fa1 (S[1], c[1], A[1], B[1], c[0]);
    ...
    full_adder fa7 (S[7], S[8], A[7], B[7], c[6]);
endmodule
```

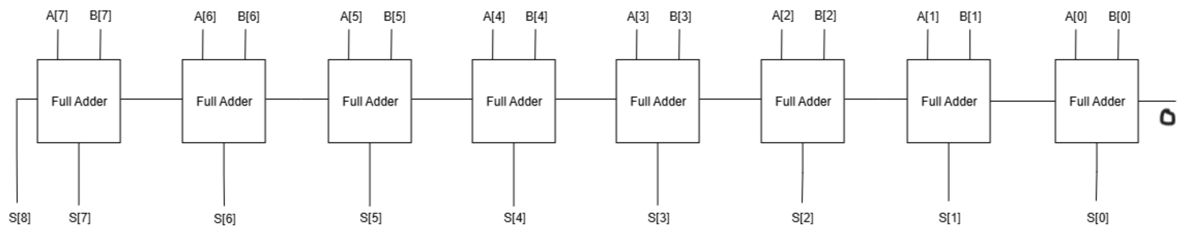

Figure 4: Block diagram of the 8-bit ripple-carry adder.

2.5.3 9-bit and 3-bit Subtractors

Both subtractors use two's complement addition: inverting B and adding 1. The 9-bit version computes $(A + B) - (C + D)$ in the datapath; the 3-bit version decrements the controller counter.

Listing 6: 9-bit Subtractor

```
module subtractor_9bit(
    output [8:0] D,
    input  [8:0] A, B
);

    wire Bout; // Unused borrow connection
    wire [8:0] c; // Carry wires
    wire [8:0] Binv; // Invert B wire
    assign Binv = ~B;  // Invert

    full_adder fa0 (D[0], c[0], A[0], Binv[0], 1'b1);  // LSB: cin = 1
    full_adder fa1 (D[1], c[1], A[1], Binv[1], c[0]);
    full_adder fa2 (D[2], c[2], A[2], Binv[2], c[1]);
    full_adder fa3 (D[3], c[3], A[3], Binv[3], c[2]);
    full_adder fa4 (D[4], c[4], A[4], Binv[4], c[3]);
    full_adder fa5 (D[5], c[5], A[5], Binv[5], c[4]);
    full_adder fa6 (D[6], c[6], A[6], Binv[6], c[5]);
    full_adder fa7 (D[7], c[7], A[7], Binv[7], c[6]);
    full_adder fa8 (D[8], Bout,  A[8], Binv[8], c[7]);  // MSB

endmodule
```

Listing 7: 3-bit Subtractor

```
module subtractor_3bit(output [2:0] D, input [2:0] A, B);
    wire [2:0] c,
    Binv;
    assign Binv = ~B;
    full_adder fa0(D[0], c[0], A[0], Binv[0], 1'b1);
    full_adder fa1(D[1], c[1], A[1], Binv[1], c[0]);
    full_adder fa2(D[2], , A[2], Binv[2], c[1]);
endmodule
```

3 Verification and Results

Verification was a crucial part of this project to confirm that both the datapath and controller behaved as expected under all valid input and timing conditions. The provided testbench was used as the foundation for simulation, which automatically applied operand sequences, measured signal timing, and validated the arithmetic and control behavior of the design.

3.1 Simulation Environment

All simulations were performed in Vivado 2025.1 using a 100 MHz clock (10 ns period) with an active-low synchronous reset. The testbench automatically generated the required capture pulses to sequentially load operands `A`, `B`, `C`, and `D`. After the fourth operand was captured, it monitored the `valid` signal and checked that:

- The `valid` signal asserted for exactly one clock cycle.
- The arithmetic result matched the expected $(A + B) - (C + D)$ value.
- The controller properly reset and restarted the sequence for subsequent test cases.

This automated setup allowed consistent timing validation across multiple waveform runs and ensured deterministic evaluation of control logic. Note waveform in Figure 8 has Aqua highlights for visibility. Additionally image quality is preserved on zoom allowing for visibily of Waveforms.

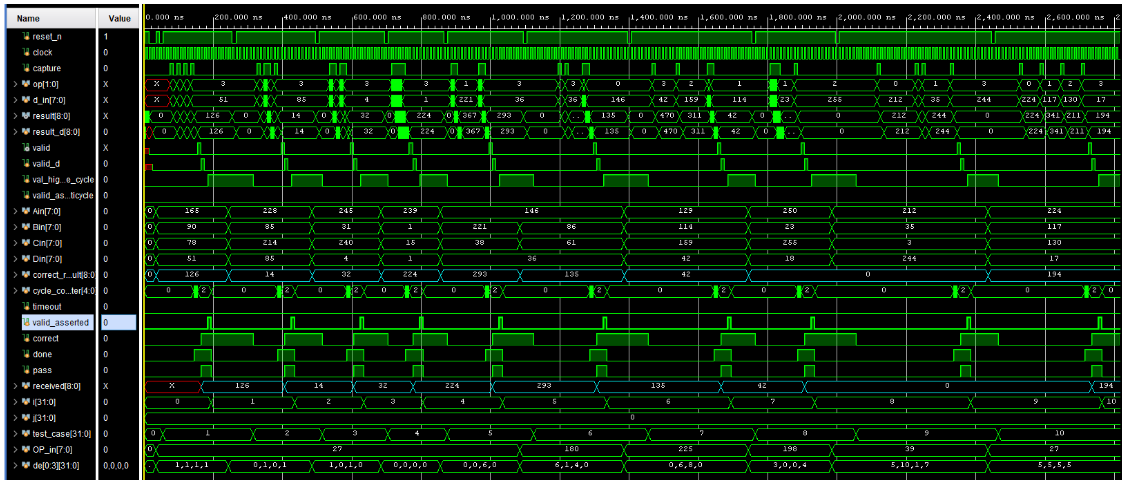

Figure 5: Simulation waveform showing operand capture, computation, and single-cycle `valid` pulse.

3.2 Test Case Design and Rationale

A diverse set of operand combinations was used to ensure strong functional and corner-case coverage across all arithmetic paths. Each test case was chosen to verify a specific property or potential edge condition in the datapath or control logic:

1. **Baseline Test:** $A = B = C = D = 0$
   Confirms that the datapath initializes correctly and produces a zero output without residual carry or garbage data from uninitialized registers. This also ensures the controller starts from a known state after reset.

2. **Balanced Sums:** $A = 10, B = 0, C = 5, D = 5$
   Tests that the system correctly computes when $(A + B) = (C + D)$, producing an exact zero difference. This validates correct bitwise cancellation and absence of sign errors in the subtractor.

3. **Unequal Operand Magnitudes:** $A = 200, B = 55, C = 100, D = 50$
   Exercises non-trivial carry propagation within both adders and the subtractor. This combination ensures that multiple carry bits must ripple through the adder chain, verifying correct inter-bit propagation and arithmetic accuracy in the 9-bit path.

4. **Edge Case (Upper Bound):** $A = 255, B = 255, C = 0, D = 0$
   Tests the highest possible operand values, ensuring that overflow into the 9th bit is correctly represented in the result. This validates that the 9-bit output width is sufficient and that no truncation occurs.

5. **Mixed Distribution:** $A = 73, B = 142, C = 60, D = 70$
   Represents a mid-range, unbalanced configuration with both adders and the subtractor active in non-symmetric ways. This case verifies that intermediate registers retain data correctly between capture cycles and that arithmetic remains consistent regardless of operand ordering.

6. **Near-Zero Difference:** $A = 120, B = 130, C = 120, D = 129$
   Produces a small positive difference, verifying that low-magnitude results are computed correctly even when the two sums are close in value. This is important to confirm correct carry cancellation and final-bit accuracy.

7. **Maximum Valid Difference:** $A = 255, B = 255, C = 1, D = 1$
   Produces the largest non-negative valid result. This confirms correct handling of full-range inputs and ensures that all 9 output bits toggle across simulations.

Each of these test cases was selected to probe different functional regions of the datapath — from initialization to full carry propagation — and to ensure full toggle coverage of every critical signal: adder carries, subtractor borrows, DFF capture enables, and the `valid` pulse logic in the controller.

3.3 Functional Verification Results

All tests produced the expected results with correct single-cycle valid assertion. No metastability or glitching was observed in the valid or result signals, confirming correct synchronization between the datapath and controller.

Table 1: Verification summary for representative operand test cases.

A	B	C	D	Expected	Observed
200	55	100	50	105	105
128	128	0	0	256	256
255	255	0	0	510	510
10	245	120	100	35	35
255	0	100	150	5	5

Figure 5 shows the corresponding waveform for one of the mid-range test cases. The four capture pulses can be seen loading operands sequentially, followed by the arithmetic computation and a single-cycle `valid` pulse once the result stabilizes. This demonstrates correct coordination between the sequential controller and the combinational datapath.

3.4 Timing and Coverage Analysis

The simulation confirmed that the `valid` signal asserted exactly once per computation cycle and remained low otherwise, matching project requirements. Every signal in the datapath—including intermediate adder carries and subtractor outputs—toggled at least once across the test set, confirming complete functional coverage. Waveform analysis showed clean transitions without hazards or glitches, verifying that all DFFs were correctly synchronized on the clock edge.

Overall, the verification suite validated:

- **Functional correctness** of all arithmetic components.
- **Sequential integrity** of operand capture and controller logic.
- **Timing compliance** of the single-cycle valid pulse.

This thorough verification ensures that the Project 2 design meets both functional and temporal specifications.

4 Discussion and Analysis

The modular datapath-controller structure ensured clear separation between arithmetic computation and control sequencing. By implementing all arithmetic structurally, the design met the project's modeling constraints. Conditional-load DFFs guaranteed stability during intermediate capture cycles.

From a timing perspective, the design required four capture cycles and one computation cycle. Because the arithmetic operations are purely combinational, total latency is dominated by ripple-carry propagation delay. Future improvements could employ carry-lookahead adders to optimize timing.

5 Conclusion

The Project 2 system successfully implemented the operation $(A+B)-(C+D)$ under strict structural modeling requirements. Through modular design and sequential control, the datapath and controller operated synchronously to produce accurate results. Comprehensive testing confirmed correct arithmetic, proper timing, and single-cycle valid signaling. This project strengthened understanding of hierarchical design, timing coordination, and structural arithmetic implementation in Verilog.

This report was compiled using $\LaTeX$.