ECE 310
Project 2 Report
Dataflow Arithmetic Calculator

Mason Deal

March 25, 2026

Abstract

This project explored the hierarchy of hardware abstraction by utilizing a hybrid of structural and dataflow modeling to execute the function $Result = (A - B) + (C - D)$. The system architecture was partitioned into a command-decoding Control Unit and a logical Datapath to handle the arithmetic. Key hardware components were implemented without the use of high-level behavioral always blocks. Instead, continuous assign statements and discrete D-Flip-Flop (DFF) instantiations were used to maintain relative hardware control and accurate 2's complement sign-extension. Verification was conducted through a comprehensive testbench covering standard arithmetic, negative results, idle-cycle timing delays, and the max edge case. The project successfully demonstrated how dataflow modeling provides an appropriate middle ground for digital design, offering implementation efficiency while retaining the control of gate-level logic.

1 Introduction, Background, and Theory

In the field of Hardware Description Languages (HDL), Verilog provides three primary levels of abstraction to model digital systems: structural, dataflow, and behavioral. Each level represents a different trade-off between design complexity and hardware transparency.

- Structural modeling sits at the lowest level, where the designer manually instantiates and wires together primitive gates or existing modules. While this offers the highest degree of control, it becomes unmanageably complex as systems scale.
- Behavioral modeling is at the highest level, utilizing procedural blocks (such as always or initial) to describe what a circuit does rather than how it is built. While flexible and fast to implement, it hides the underlying hardware logic, often leading to inefficient synthesis if the designer is not careful.
- Dataflow modeling serves as the critical middle ground. By using continuous assignments (assign statements), dataflow modeling describes the flow of signals through the system using Boolean equations and operators. It retains the hardware-centric mindset of structural design—ensuring the designer understands the physical paths and timing while providing the efficiency of higher-level languages.

To demonstrate this "middle ground," this project implemented a 2's complement arithmetic calculator using a hybrid of dataflow and structural modeling. By avoiding behavioral procedural blocks (except for the D flip-flop), the design ensured that every operation mapped directly to predictable combinational and sequential hardware. This approach highlighted how dataflow modeling provides sufficient abstraction for complex arithmetic while maintaining the timing control required for synchronized capture and validity signaling.

1.1 Arithmetic Calculator Composition

The system was partitioned into two primary functional modules: the Control Unit and the Datapath. This separation ensured that the arithmetic logic remained independent of the command signals, a standard practice in digital system design.

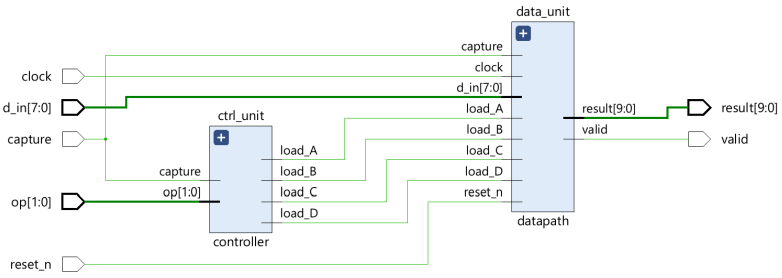

Figure 1: Vivado generated high-level design of datapath, controller units and their inputs, outputs, and nets.

1. The Controller (Command Decoding): The Controller was implemented using pure dataflow logic. It takes a 2-bit operation code (op) and a capture signal as inputs. Using a series of continuous assignments, it decodes these signals into four unique "load" enables, through the following assignments:

   ```
   assign load_A = (capture && (op == 2'b00));
   assign load_B = (capture && (op == 2'b01));
   assign load_C = (capture && (op == 2'b10));
   assign load_D = (capture && (op == 2'b11));
   ```

   This logic ensures that a register only updates its value when the capture signal is asserted and the specific address matches the operation code. This prevents accidental data overwrites during idle clock cycles.

2. 2. The Datapath (Storage, Arithmetic, and Valid Signal): The Datapath houses the physical storage registers and the arithmetic logic.

   - Structural Registers: Each of the four 8-bit registers is built structurally from eight D-Flip-Flops (dff). An internal 2:1 multiplexer is used for each bit to decide whether to "hold" the current value or "load" the new *d_in* value based on the enable signal from the Controller.

     ```
     // Initialize registers
     reg8 regA(.clock(clock), .reset_n(reset_n), .en(load_A), .d(d_in), .q(reg_8A));
     reg8 regB(.clock(clock), .reset_n(reset_n), .en(load_B), .d(d_in), .q(reg_8B));
     reg8 regC(.clock(clock), .reset_n(reset_n), .en(load_C), .d(d_in), .q(reg_8C));
     reg8 regD(.clock(clock), .reset_n(reset_n), .en(load_D), .d(d_in), .q(reg_8D));
     ```

   - Arithmetic Chain: The arithmetic logic is wired as a dataflow path. As soon as the registers update, the signals "ripple" through the 8-bit and 9-bit adders. Because this is implemented with assign statements and structural adders, the mathematical result is calculated as a combinational function of the register outputs.

     The following equation was modeled in the arithmetic chain:

     $$(A + B) - (C - D) = Result \tag{1}$$

     ```
     // Arithmetic logic
     wire [7:0] a_minus_b, c_minus_d;
     wire [8:0] val1, val2;
     wire Cout_1, Cout_2;

     // A-B & C-D
     rca_8bit rca_a_minus_b(.A(reg_8A), .B(~reg_8B), .Cin(1'b1), .S(a_minus_b), .Cout(Cout_1));
     rca_8bit rca_c_minus_d(.A(reg_8C), .B(~reg_8D), .Cin(1'b1), .S(c_minus_d), .Cout(Cout_2));
     ```

```
// Calculate true sign extensions for the 9th bit using XOR logic
wire val1_sign = reg_8A[7] ^ ~reg_8B[7] ^ Cout_1;
wire val2_sign = reg_8C[7] ^ ~reg_8D[7] ^ Cout_2;

assign val1 = {val1_sign, a_minus_b};
assign val2 = {val2_sign, c_minus_d};

// Add the results
wire [8:0]result_9bit;
wire result_cout;
rca_9bit rca_result(.A(val1), .B(val2), .S(result_9bit), .Cout(result_cout));

// Calculate true sign extension for the 10th bit
wire result_sign = val1[8] ^ val2[8] ^ result_cout;

assign result = {result_sign, result_9bit};
```

3. Sequential Valid Logic (4-Cycle Counter): The most critical implementation detail is the generation of the valid pulse. Since the calculator requires four distinct inputs to produce a meaningful result, a 3-bit structural counter tracks the number of capture events. To ensure the output is only read when the data is stable, the counter is designed with a one-cycle latency. When the fourth capture signal is received (loading Register D), the counter increments to 4 ($3'b100$) on the following clock edge. This delay is intentional because it ensures that the final register has fully latched its data and the arithmetic signals have finished propagating through the Ripple Carry Adders before the valid signal is asserted (an issue that was first encountered during testing). Once valid goes high, a feedback Mux in the dataflow logic resets the counter to 0 ($3'b000$), clearing the state for the next set of operations.

```
wire [2:0] count_reg;
wire [2:0] rca_out;

// 1. 2:1 Mux (Adds 1 if capturing, 0 if not)
wire [2:0] mux_out;
assign mux_out = {2'b00, capture};

// 2. 3-Bit RCA
assign rca_out = count_reg + mux_out;

// 3. CHECK FOR 4 ON THE DELAYED REGISTER, NOT THE INSTANT RCA
wire is_four;
assign is_four = count_reg[2] & ~count_reg[1] & ~count_reg[0];

// 4. RESET MUX: If we hit 4, force the next state to be 0
wire [2:0] next_count;
assign next_count = is_four ? 3'b000 : rca_out;

// 5. 3-Bit Register (Structural DFFs catching the next_count)
dff count_bit0 (.clock(clock), .reset_n(reset_n), .d(next_count[0]), .q(count_reg[0]));
dff count_bit1 (.clock(clock), .reset_n(reset_n), .d(next_count[1]), .q(count_reg[1]));
dff count_bit2 (.clock(clock), .reset_n(reset_n), .d(next_count[2]), .q(count_reg[2]));

assign valid = is_four;
```

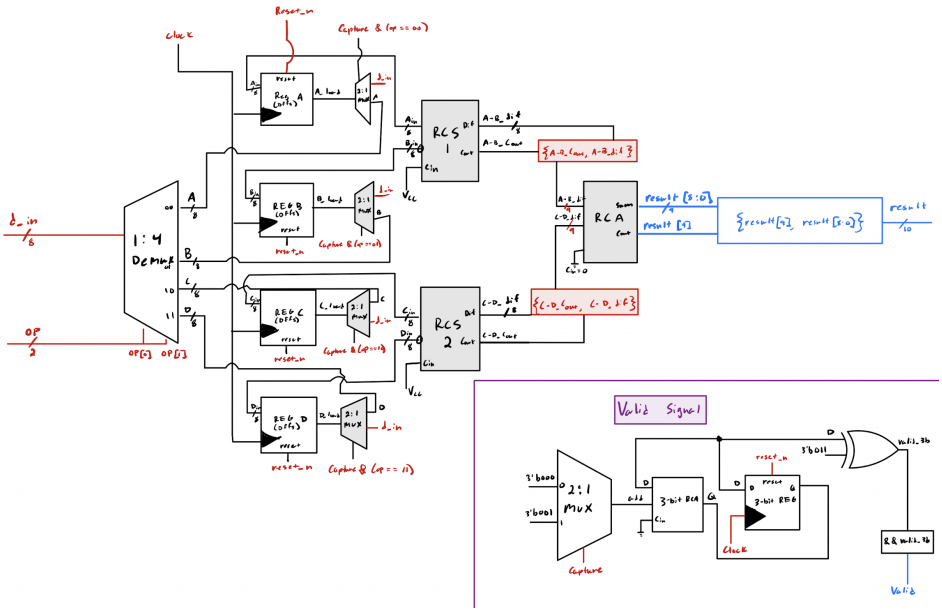

Figure 2: Pen/paper design for Controller, Datapath, and Valid units for the arithmetic calculator.

2 Results & Analysis

The functional correctness of the arithmetic calculator was verified through a comprehensive simulation testbench. The testbench provided a stable 10ns clock and cycled through various input vectors to observe the internal register states and the resulting 10-bit output. While the capture signal was primarily asserted in back-to-back cycles, a specific "idle-cycle" test case was implemented to ensure the structural registers maintained data integrity during periods of inactivity. Additional test cases were executed to stress test 2's complement edge cases, including maximum positive and negative values, to verify that the sign-extension logic prevented bit overflow. Waveform analysis confirmed that the 10-bit result consistently matched the expected theoretical values. Crucially, closer inspection of the timing waveforms for the final capture event verified that the valid pulse was correctly delayed by one clock cycle, ensuring the data was stable and the Ripple Carry Adder propagation was complete before the output was flagged as ready.

A	B	C	D	Result	Case
10	5	20	10	15	Traditional Test
5	15	10	20	-20	Negative Test
42	-42	-5	5	0	Zero Test
100	50	10	5	55	Delay Test
127	-128	127	-128	510	Edge Test

Table 1: Testbench test values, expected results, and case description.

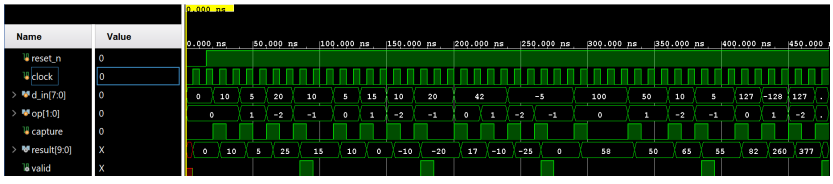

Figure 3: Waveform results from the testbench for testing all five test cases. Note: op[1:0] is incorrectly set to signed decimal instead of unsigned, so -2 and -1 are actually 2 and 3, respectively.

3 Conclusions

The implementation of the structural and dataflow arithmetic calculator successfully demonstrated a balance between hardware granularity and design efficiency. By partitioning the system into a distinct Control Unit and Datapath, the architecture maintained a clean separation between command decoding and arithmetic processing, mirroring real-world CPU design principles. The deliberate use of dataflow modeling through assign statements provided a significant advantage over pure structural modeling, since it allowed for the concise implementation of multiplexers, concatenations, and the 3-bit counter logic without the overhead of manual gate-level instantiation. Furthermore, simulation results verified that the system remained resilient under varying timing conditions, such as capture delays, while the 10-bit sign-extension logic ensured mathematical accuracy across the full range of 2's complement values. Ultimately, this project highlights dataflow modeling as the optimal middle ground in digital design. Executed by offering the abstraction necessary to bypass repetitive structural definitions while retaining the deterministic hardware control required for synchronized high-speed arithmetic.

This report was compiled using LaTeX .