HDL Design Homework: 4-Bit Adder Architectures

Yousef "Quil" Awad
EEL 4768 - Computer Architecture

February 11, 2026

Part 1: Design Specifications

Below is the synthesizable HDL code for the Ripple-Carry Adder (RCA), Carry-Lookahead Adder (CLA), and the associated testbench.

A. Ripple-Carry Adder (RCA)

File: `rca.sv`

```
`timescale 1ns / 10ps

module rca (
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output Cout,
    output [3:0] S
);
  genvar i;
  wire [4:0] carry_temp;

  // b0 is lsb, for context
  generate
    for (i = 0; i < 4; i = i + 1) begin : g_adder_loop
      full_adder adder (
          .a(A[i]),
          .b(B[i]),
          .c_in(carry_temp[i]),
          .c_out(carry_temp[i+1]),
          .sum(S[i])
      );
    end
  endgenerate

  assign Cout = carry_temp[4];
  assign carry_temp[0] = Cin;
endmodule
```

File: `full_adder.sv`

```
`timescale 1ns / 10ps

module full_adder (
    input  a,
    input  b,
```

```
    input  c_in,
    output c_out,
    output sum
);
  assign #1 sum   = a ^ b ^ c_in;
  assign #1 c_out = (a & b) | (b & c_in) | (a & c_in);
endmodule
```

B. Carry-Lookahead Adder (CLA)

File: `cla.sv`

```
`timescale 1ns / 10ps

module cla (
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output Cout,
    output [3:0] S
);
  // create the generate and propagate and buffers for carries
  wire [3:0] Generate;
  wire [3:0] Propogate;
  wire temp_carry[2:0];

  // define the logic for the generate and propagate
  assign Generate = A & B;
  assign Propogate = A ^ B;

  // define the logic for the buffer carries and output carry
  assign temp_carry[0] = Generate[0] | (Propogate[0] & Cin);

  assign temp_carry[1] = Generate[1] | (Propogate[1] & Generate[0]) |
                         (Propogate[1] & Propogate[0] & Cin);

  assign temp_carry[2] = Generate[2] | (Propogate[2] & Generate[1]) |
                         (Propogate[2] & Propogate[1] & Generate[0]) |
                         (Propogate[2] & Propogate[1] & Propogate[0] & Cin);

  assign Cout = Generate[3] | (Propogate[3] & Generate[2]) |
                (Propogate[3] & Propogate[2] & Generate[1]) |
                (Propogate[3] & Propogate[2] & Propogate[1] & Generate[0]) |
                (Propogate[3] & Propogate[2] & Propogate[1] & Propogate[0] & Cin);

  // defining the summation logic
  assign S[0] = Propogate[0] ^ Cin;
  assign S[1] = Propogate[1] ^ temp_carry[0];
  assign S[2] = Propogate[2] ^ temp_carry[1];
  assign S[3] = Propogate[3] ^ temp_carry[2];

endmodule
```

Part 2: Testbench & Simulation

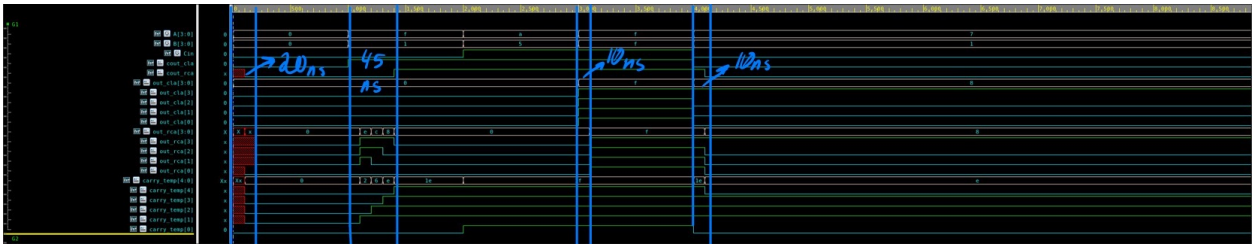

Figure 1: **Functional Comparison.** The waveforms confirm that both RCA and CLA produce identical Sum and Carry-out values for all test vectors.

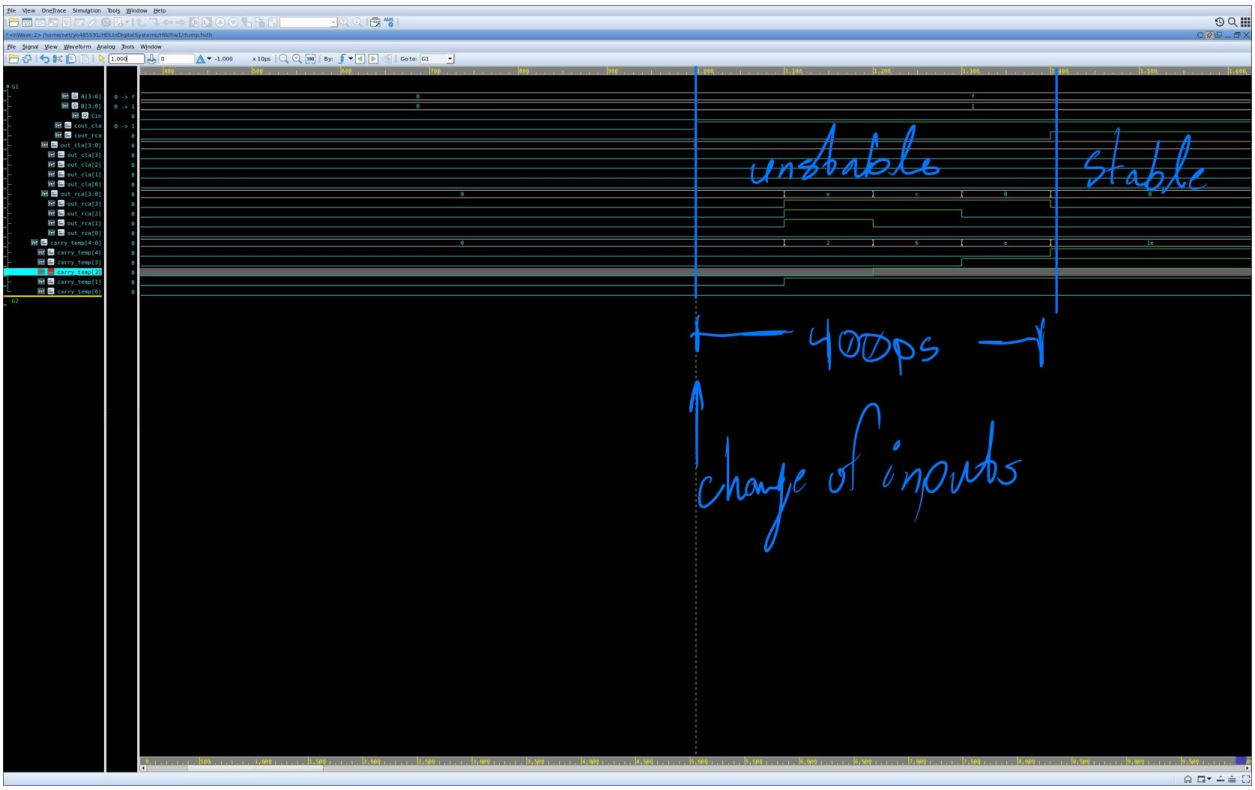

Figure 2: **Critical Path Zoom-In.** The timing detail reveals the "ripple" effect in the RCA carry chain ($C_1 \rightarrow C_2 \rightarrow C_3$), while the CLA output settles significantly faster.

Part 3: Analysis & Discussion

1. Timing Comparison

For test vector #5 (Critical Path Test), the propagation delays were measured from the input change to stable C_{out}.

- **RCA Delay:** $\approx$ 400 ps (approximate, see Fig 2)
- **CLA Delay:** Negligible / Instantaneous relative to RCA

Explanation: The Ripple-Carry Adder exhibits sequential propagation because the carry-out of bit i (C_{i+1}) depends directly on the carry-in from the previous bit (C_i). This creates a dependency chain where the signal must traverse every Full Adder physically. In contrast, the CLA calculates all carry bits simultaneously using flattened logic equations based purely on inputs A, B, and C_{in}, eliminating the "waiting" period.

2. Circuit Explanation

At the gate level, the RCA delay scales as $O(N)$ because the critical path flows through N Full Adder blocks in series. If a single Full Adder has a delay of t_{FA}, the total delay is $N \times t_{FA}$.

The CLA delay scales as $O(\log N)$ (or effectively $O(1)$ for fixed logic depths). This is because the carry logic is expanded into a Sum-of-Products form. For example, C_4 is calculated as:

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{in}$$

This equation is realized with a 2-level AND-OR logic depth regardless of the bit position (limited only by the fan-in capability of the gates), allowing simultaneous calculation.

3. Area-Speed Tradeoff

The CLA requires significantly more logic gates than the RCA.

- **RCA:** Uses $\approx$ 5 gates per bit. Total for 4-bit $\approx$ 20 gates. Linear area growth.
- **CLA:** Requires complex logic for the Generate/Propagate lookahead units. The gate count grows cubically/exponentially depending on implementation details.

Designers might still choose RCA for wide adders (e.g., >16 bits) if the application is **low-power** or **area-constrained** and the adder is not on the critical timing path. CLA is preferred strictly when minimizing delay is the highest priority.

4. Real-World Relevance

Modern FPGAs (like Xilinx 7-Series or Ultrascale) contain dedicated hardware resources known as **Carry Chains** (e.g., CARRY4 or CARRY8 primitives). These are hardened, high-speed silicon paths located adjacent to the LUTs. This hardware optimization allows Ripple-Carry structures to operate at extremely high speeds, often mitigating the theoretical delay disadvantage of RCA and negating the need for complex soft-logic CLAs for standard integer addition.