N-Bit Unary Adder with Sorting Network for Metastability Handling and Comparison with Binary and Gray- Coded Adders

Abstract

This project presents the design and implementation of three digital adders: a Binary Adder, a Gray-Coded Adder, and a novel Unary Adder with an integrated sorting network. The primary focus is on the Unary Adder design, which aims to enhance metastability handling and address potential computation accuracy issues found in traditional Gray-Coded and Binary Adders. The Binary and Gray-Coded adders establish a baseline for comparison.

The Unary Adder leverages unary number representation, where numbers are represented by a sequence of consecutive '1's. It incorporates a sorting network to resolve potentially metastable input bits, forcing them into valid logic states. The adders' designs are implemented in Verilog.

Simulation and analysis (details to be included in the full documentation) will be conducted to evaluate the effectiveness of each adder in mitigating metastability-related issues. Performance metrics such as propagation delay, area utilization , metastability impact on computation accuracy will be used to facilitate a comprehensive comparison.

Introduction

Metastability is a persistent challenge in digital circuits, arising when a signal violates setup and hold times, leading to unpredictable behavior in flip-flops. Metastability can introduce errors, increase propagation delay, and compromise the overall reliability of digital systems. Adders, being fundamental components in digital arithmetic, are particularly susceptible to metastability-related issues.

Traditional adders, such as Binary Adders and Gray-Coded Adders, offer different trade-offs between speed, power consumption, and accuracy. Binary Adders are fast but can exhibit significant errors when metastable states propagate through the carry chain. Gray-Coded Adders reduce the probability of multi-bit errors due to their single-bit change property, but they are not immune to metastability and can still introduce computational inaccuracies.

This project introduces a novel Unary Adder design augmented with a sorting network. The primary goal is **Enhanced Metastability Handling:** The sorting network aims to resolve metastable input bits, minimizing their impact on the adder's output.

Problem Statement

Metastability handling is crucial in digital circuits, especially adders, because unresolved metastable states can lead to incorrect calculations, unpredictable output behavior, and system-level failures. The impact of metastability becomes more severe as clock frequencies increase and signal transitions become faster.

Project Scope

This project will cover the following:

- **Design and Implementation:** Design of a Unary Adder with sorting network, a Binary Adder, and a Gray-Coded Adder in Verilog/VHDL.
- **Simulation:** Thorough simulation environments to test adder designs under normal operation and intentionally induced metastable conditions.
- **Comparison:** Comprehensive analysis of the three adders with respect to metastability handling, computational accuracy, propagation delay, and resource utilization.

Design and Implementation

1. Unary Adder with Sorting Network

- **Theoretical Basis**
  - In unary representation, a number 'N' is represented by a sequence of '0's followed by a N '1's (e.g., 3 is "0111"). Addition is performed by simple concatenation of the unary input streams.
  - Sorting networks consist of interconnected compare-and-swap elements, for achieving valid output of ‘0’s followed by ‘1’s.
- **Circuit Design**
  presented is verilog code:

```
`timescale  1ns/1ps

module UnaryAdder #(
    parameter NOF_BITS = 32
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [NOF_BITS-1:0] data_a, data_b,
    output reg [2*NOF_BITS-1:0] data_out,
    output reg done
);
    localparam IDLE = 2'b01;
    localparam SORT = 2'b10;

    // Define state register
    reg [1:0] state, next_state;
    wire sort_done;
    wire[2*NOF_BITS-1:0] data_in;
    wire [2*NOF_BITS-1:0] sort_out;
    reg sort_start;

    assign data_in = {data_a, data_b};
    // FSM
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            state <= IDLE;
            // busy <= 0;
            done <= 0;
            data_out <=  0;
        end else begin
            state <= next_state;
```

```
            // busy <= busy;
            done <= done;
            data_out <= data_out;
        end
    end

    always @(*) begin
        case (state)
            IDLE:
                if (start) begin
                    next_state = SORT;
                end else begin
                    next_state = IDLE;
                end
            SORT:
                if (done) begin
                    next_state = IDLE;
                end else begin
                    next_state = SORT;
                end
        endcase
    end



    always @(posedge clk)  begin
        if(sort_done) begin
        data_out <= sort_out;
        done<=1;
        end
        else begin
        data_out <= 0;
        done<=0;
        end
    end

    always @(posedge clk) begin
        sort_start <= start;
    end

    BubbleSort #(2*NOF_BITS) sorter (
        .clk(clk),
        .start(sort_start), // each time this signal is high the run
starts again
        .done(sort_done), // this signal is high for only one cycle
```

```
        .in_data(data_in),
        .out_data(sort_out),
        .rst_n(rst_n)
    );

endmodule

module BubbleSort # (
    parameter NOF_BITS = 16
)(
    input wire clk,
    input wire start,
    input wire rst_n,
    input wire [NOF_BITS-1:0] in_data,
    output reg [NOF_BITS-1:0] out_data,
    output reg done
);

    // Define states
    localparam IDLE = 2'b01;
    localparam SORT = 2'b10;

    reg [NOF_BITS-1:0] temp;
    reg [1:0] state, next_state;
    reg [$clog2(2*NOF_BITS-3):0] i;

    integer j;

    always @(posedge start) begin
        temp <= in_data;
        i = 2;
    end

    always @(negedge rst_n) begin
        temp<=0;
        out_data<=0;
        done<=0;
        next_state<=IDLE;
    end

    always @(*) begin
        case (state)
            IDLE:
                if (start) begin
                    next_state = SORT;
```

```
                end else begin
                    next_state = IDLE;
                end
            SORT:
                if (done) begin
                    next_state = IDLE;
                end else begin
                    next_state = SORT;
                end
            default:
                next_state = IDLE;
        endcase
    end

    always @(posedge clk) begin
        state <= next_state;
    end

    always @(posedge clk) begin
        if (state == SORT) begin
            //COMPARATOR
            for (j = i[0]; j<NOF_BITS-1; j= j+2) begin
                temp[j] <= temp[j]|temp[j+1];
                temp[j+1] <= temp[j]&temp[j+1];
            end
            i <= i+1;
        end else begin
            i<= i;
            temp[j] <= temp[j];
            temp[j+1] <= temp[j+1];
        end
    end

    always @(posedge clk) begin
        if (i==2*NOF_BITS-4)
            done <= 1;
        else
            done<=0;

        out_data<=temp;
    end
endmodule
```

The core math concept in the Unary Adder itself is simple:

- **Concatenation:** The primary mathematical operation is concatenation of the outputs from the BubbleSort module. Essentially, the Unary Adder treats the sorted input streams as unary representations and joins them to produce the sum. Recall that the sorting network's job is to ensure the inputs have valid '1's and '0's to correctly form the unary numbers.

BubbleSort

The BubbleSort module implements a sorting algorithm. While it doesn't perform calculations in the traditional sense, it re-arranges the input bits, which is key for the Unary Adder's operation. Let's analyze the core comparison logic in the BubbleSort:

```
for (j = i[0]; j<NOF_BITS-1; j= j+2) begin
    temp[j] <= temp[j]|temp[j+1];
    temp[j+1] <= temp[j]&temp[j+1];
end
```

- **Bitwise OR and AND:** Each iteration compares adjacent bits (`temp[j]` and `temp[j+1]`). It uses bitwise OR and AND operations to enforce a specific order. The goal is to push the larger values (likely '1's in the unary context) towards the higher-order bits.
- **Why this matters for Unary:** In a correct unary representation, all '1's should be grouped together. The sorting network uses iterative comparisons like these, likely over multiple passes, to force metastable bits into a pattern that the Unary Adder can reliably understand as a unary number.

High-Level Overview:

1. **Bubble Sort:** Resolves potential metastability by "sorting" input bits ensuring a valid unary pattern of consecutive '1's.
2. **Unary Adder:** Performs a straightforward concatenation of the now-sorted "unary" streams.

2. Binary Adder

- **Circuit Design**
  presented verilog code:

```
`timescale 1ns / 1ps

module BinaryAdder #(
    parameter NOF_BITS = 8
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [NOF_BITS-1:0] data_a, data_b,
    output reg [NOF_BITS-1:0] data_out, // Reduced output size
    output reg done
);

   wire [NOF_BITS-1:0] sum;
   wire [NOF_BITS-1:0] cout;  // Carry out

    full_adder_d FA0(.carry(cout[0]), .sum(sum[0]), .a(data_a[0]),
.b(data_b[0]), .cin(1'b0));
    full_adder_d FA1(.carry(cout[1]), .sum(sum[1]), .a(data_a[1]),
.b(data_b[1]), .cin(cout[0]));
    full_adder_d FA2(.carry(cout[2]), .sum(sum[2]), .a(data_a[2]),
.b(data_b[2]), .cin(cout[1]));
    full_adder_d FA3(.carry(cout[3]), .sum(sum[3]), .a(data_a[3]),
.b(data_b[3]), .cin(cout[2]));
    full_adder_d FA4(.carry(cout[4]), .sum(sum[4]), .a(data_a[4]),
.b(data_b[4]), .cin(cout[3]));
    full_adder_d FA5(.carry(cout[5]), .sum(sum[5]), .a(data_a[5]),
.b(data_b[5]), .cin(cout[4]));
    full_adder_d FA6(.carry(cout[6]), .sum(sum[6]), .a(data_a[6]),
.b(data_b[6]), .cin(cout[5]));
    full_adder_d FA7(.carry(cout[7]), .sum(sum[7]), .a(data_a[7]),
.b(data_b[7]), .cin(cout[6]));

    always @(posedge clk)  begin
        if(start) begin // Or adapt triggering condition if needed
            data_out <= sum;
            done <= 1;
        end else begin
```

```
            data_out <= 0;
            done <= 0;
        end
    end

endmodule

module full_adder_d (
    input a,b,cin,
    output sum,carry
);

assign sum = a ^ b ^ cin;
assign carry = (a & b) | (b & cin)  | (cin & a) ;

endmodule
```

- **Metastability Considerations** Full Adders Binary Adders can be severely affected if metastability occurs in the carry chain. A metastable carry bit can propagate, causing multiple output bits to be incorrect.

3. Gray-Coded Adder

- In Gray code, for any two adjacent values, only one bit position will differ.
- In order to use gray code adder the input should be gray coded.

```
`timescale 1ns/1ps

module gray_adder #(
    parameter NOF_BITS = 8
)(
    input wire clk,
    input wire rst_n,
    input wire start,
    input wire [NOF_BITS-1:0] A, B,
    input wire PA, PB,
    output reg [NOF_BITS-1:0] S,
    output reg done
);

    // FSM States
    localparam IDLE  = 2'b01;
    localparam ADD   = 2'b10;

    // State and intermediate registers
    reg [1:0] state, next_state;
    reg [NOF_BITS-1:0] E, F;
    reg [$clog2(NOF_BITS):0] counter; // Introduce a counter

    // next state
    always @(*) begin
        case(state)
            IDLE: begin
                next_state = (start) ? ADD : IDLE;
            end
            ADD: begin
                if (counter == NOF_BITS - 1) begin
                    next_state = IDLE;
                end else begin
                    next_state = ADD;
                end
            end
            default: next_state = IDLE;
```

```
        endcase
    end

    // Sequential updates of state and outputs
    always @(posedge clk or negedge rst_n) begin
        if (~rst_n) begin
            state <= IDLE;
            done <= 0;
        end else begin
            state <= next_state;
            // ... update registers based on next_state if needed
            if (state == ADD && next_state == IDLE) begin
                done <= 1;
            end else begin
                done <= 0;
            end
        end
    end

    always @(posedge clk) begin
        if (state == IDLE) begin
            E <= {{NOF_BITS-1{1'b0}}, PA};  // Initialize E
            F <= {{NOF_BITS-1{1'b0}}, PB};  // Initialize F
            counter <= 0;                   // Reset the counter
        end else begin
            S[counter] <= (E[counter] & F[counter]) ^ A[counter] ^
B[counter];
            E[counter+1] <= (E[counter] & (~F[counter])) ^ A[counter];
            F[counter+1] <= ((~E[counter]) & F[counter]) ^ B[counter];
            counter <= counter + 1;
        end
    end

endmodule
```

Let's unpack the lines:

- `S[counter] <= (E[counter] & F[counter]) ^ A[counter] ^ B[counter];`
    - This line calculates a single output bit of the Gray-Coded sum (S) for the current position (`counter`).
    - The logic performs an XOR operation three times:
        - Between the current bits `E[counter]` and `F[counter]`
        - Between the first XOR's result and the Gray-coded input 'A' bit (`A[counter]`)

        - Between the second XOR's result and the Gray-coded input 'B' bit (`B[counter]`)
- `E[counter+1] <= (E[counter] & (~F[counter])) ^ A[counter];`
    - This updates the intermediate value 'E' for the next bit position in the addition.
    - It performs AND between `E[counter]` and the complement of `F[counter]` (~ means complement), followed by an XOR with the 'A' input bit.
- `F[counter+1] <= ((~E[counter]) & F[counter]) ^ B[counter];`
    - This updates the intermediate value 'F' for the next bit position.
    - It performs AND between the complement of `E[counter]` and `F[counter]`, followed by an XOR with the 'B' input bit.

Key Points:

- **Polarity Bits** The 'PA' and 'PB' inputs are assumed to carry the "polarity" of Gray-coded numbers. The internal logic likely uses them to correctly track and resolve potential metastability on these bits.
- **Iterative Calculation:** The code calculates the Gray-coded sum bit by bit using a counter to track positions. This is due to the sequential nature of carry propagation in addition.

- **Metastability Considerations** While Gray coding reduces the likelihood of errors due to multi-bit changes, a metastable input bit can still result in one incorrect output bit.

Simulation and Testing

1. Simulation Environment

- **Simulator Choice:** we have selected Xcelium as the simultaion envierment.
- **Testbench Development:** we have created test benches to simulate each adder beheviour on normal operation and metastable operation.

2. Test Scenarios

Accuracy:

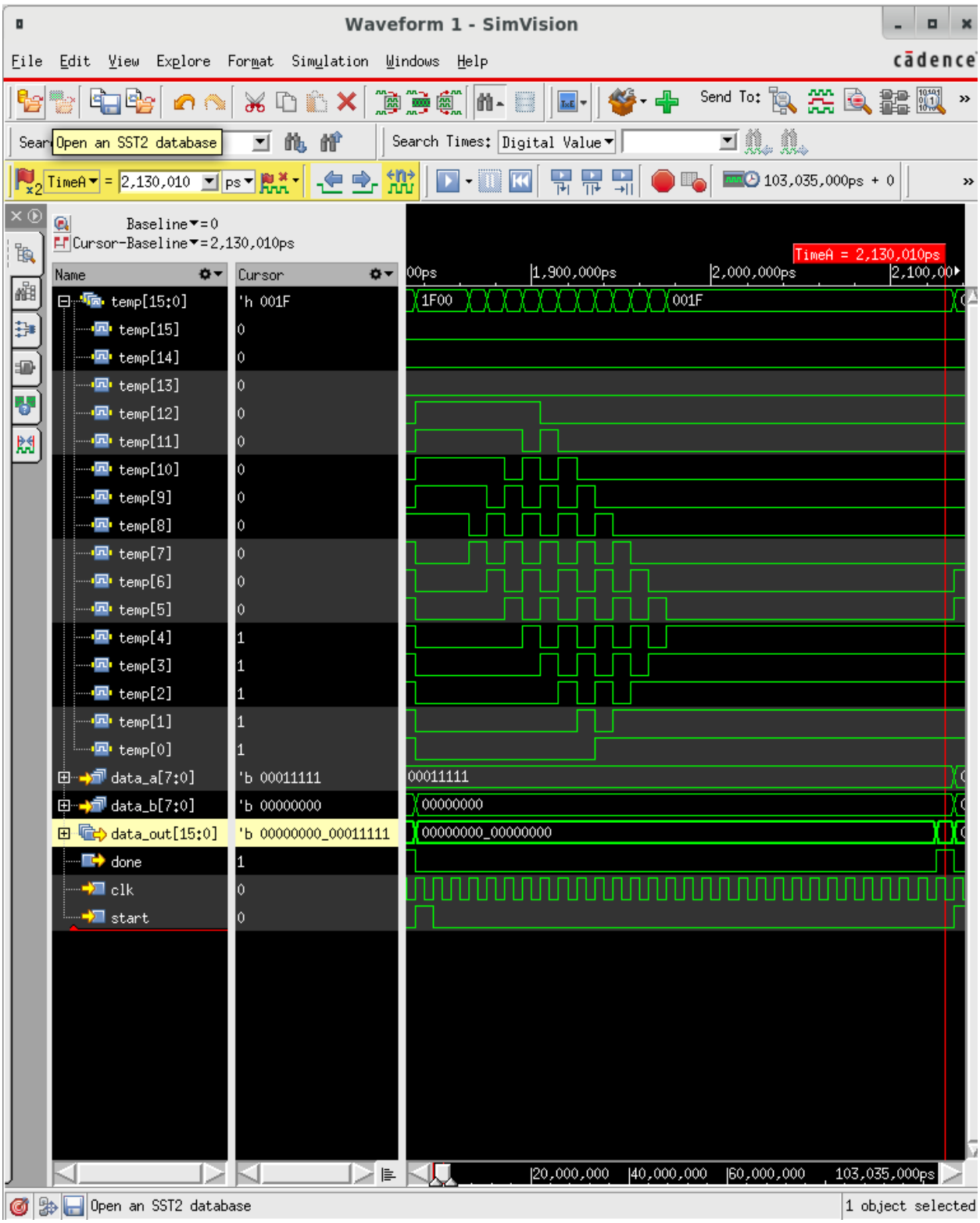
***code for each test becnh is attached.**

Normal Operation:
***we have randomized 10 inputs sets on each adder to verify its opreation.**

Unary adder:

```
xcelium> run
01111111 + 00000011 = 0000000111111111
00001111 + 00000011 = 0000000000111111
00001111 + 11111111 = 0000111111111111
00000000 + 01111111 = 0000000001111111
00000000 + 00000001 = 0000000000000001
00011111 + 00001111 = 0000000111111111
00011111 + 00000000 = 0000000000011111
00000000 + 01111111 = 0000000001111111
01111111 + 00000011 = 0000000111111111
00011111 + 01111111 = 0000111111111111
Simulation complete via $finish(1) at time 103035 NS + 0
../src/tb/unary_tb.sv:77        #100000 $finish; // End simulation
```

here we can see the sorting network operation and the compration proccess:



we can see that the sorting is operating well, the temp signal shows the bubble sort sorting netwrok as we saw in class.

Binary adder:

```
xcelium> run
00111101 + 00010100 = 01010001, correct: True
00001010 + 01100100 = 01101110, correct: True
00101000 + 00000001 = 00101001, correct: True
01110101 + 00110100 = 10101001, correct: True
01100101 + 00011111 = 10000100, correct: True
00001000 + 00100011 = 00101011, correct: True
00001010 + 00111110 = 01001000, correct: True
00101110 + 01010110 = 10000100, correct: True
00111100 + 00001111 = 01001011, correct: True
01100110 + 00001010 = 01110000, correct: True
Simulation complete via $finish(1) at time 1210 NS + 0
../src/tb/binary_adder_tb.sv:49         #1000 $finish; // End the sim
ulation
```

Gray adder:

```
xcelium> run
00111101 + 00010100 = 01100001
00001010 + 01100100 = 01111010
00101000 + 00000001 = 00101001
01110101 + 00110100 = 11000000
01100101 + 00011111 = 01110110
00001000 + 00100011 = 01101010
00001010 + 00111110 = 00101100
00101110 + 01010110 = 11010100
00111100 + 00001111 = 00101011
01100110 + 00001010 = 01111000
Simulation complete via $finish(1) at time 2035 NS + 0
../src/tb/grey_tb.sv:84         #1000 $finish; // End the simulation
xcelium> exit
```

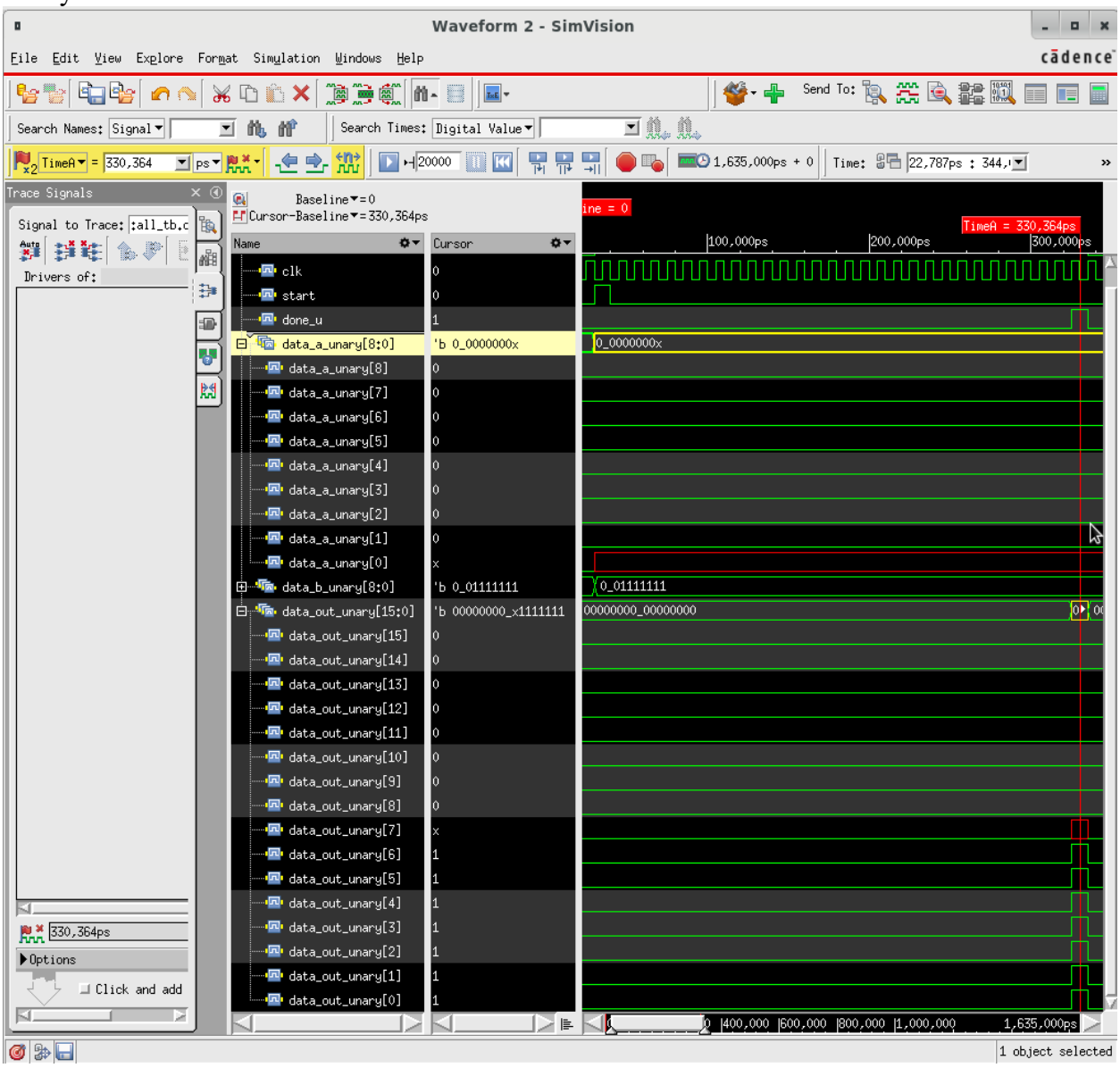
Metastability Inducement:
***we have tested the adders togother with the same inputs (each in its own encoding) , that included metastability bits to simulate different cases.**

Inputs A

```
data_a_binary = 8'b0000000x; // the input may be 0 or 1
data_a_unary = 16'b000000000000000x;
data_a_gray = 8'b0000000x;

data_b_binary = 8'b00000111;    // 7 in binary
data_b_unary = (1<<7)-1;        // 7 unary
data_b_gray = 8'b00000100;      //7 gray
```

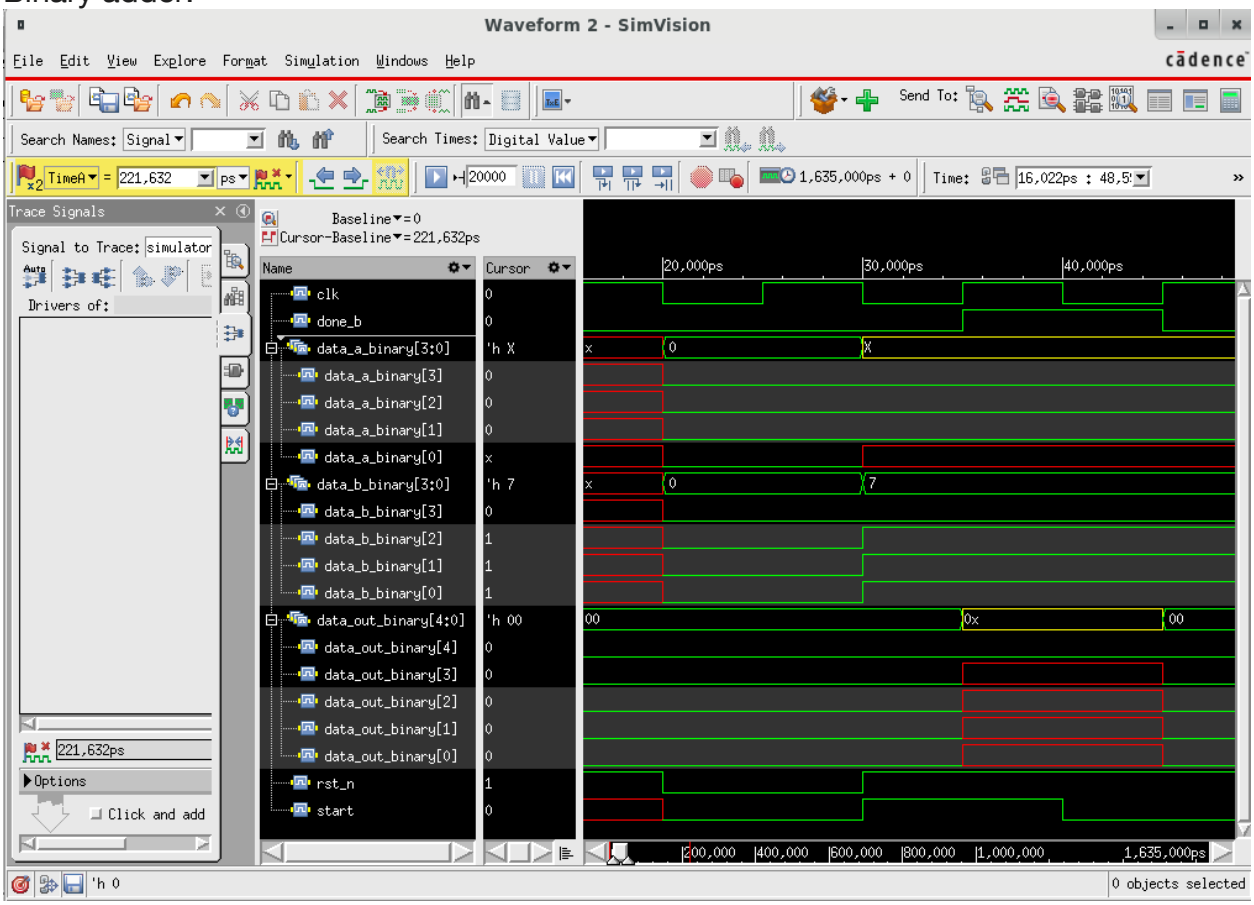
Outputs signals:

Unary adder:



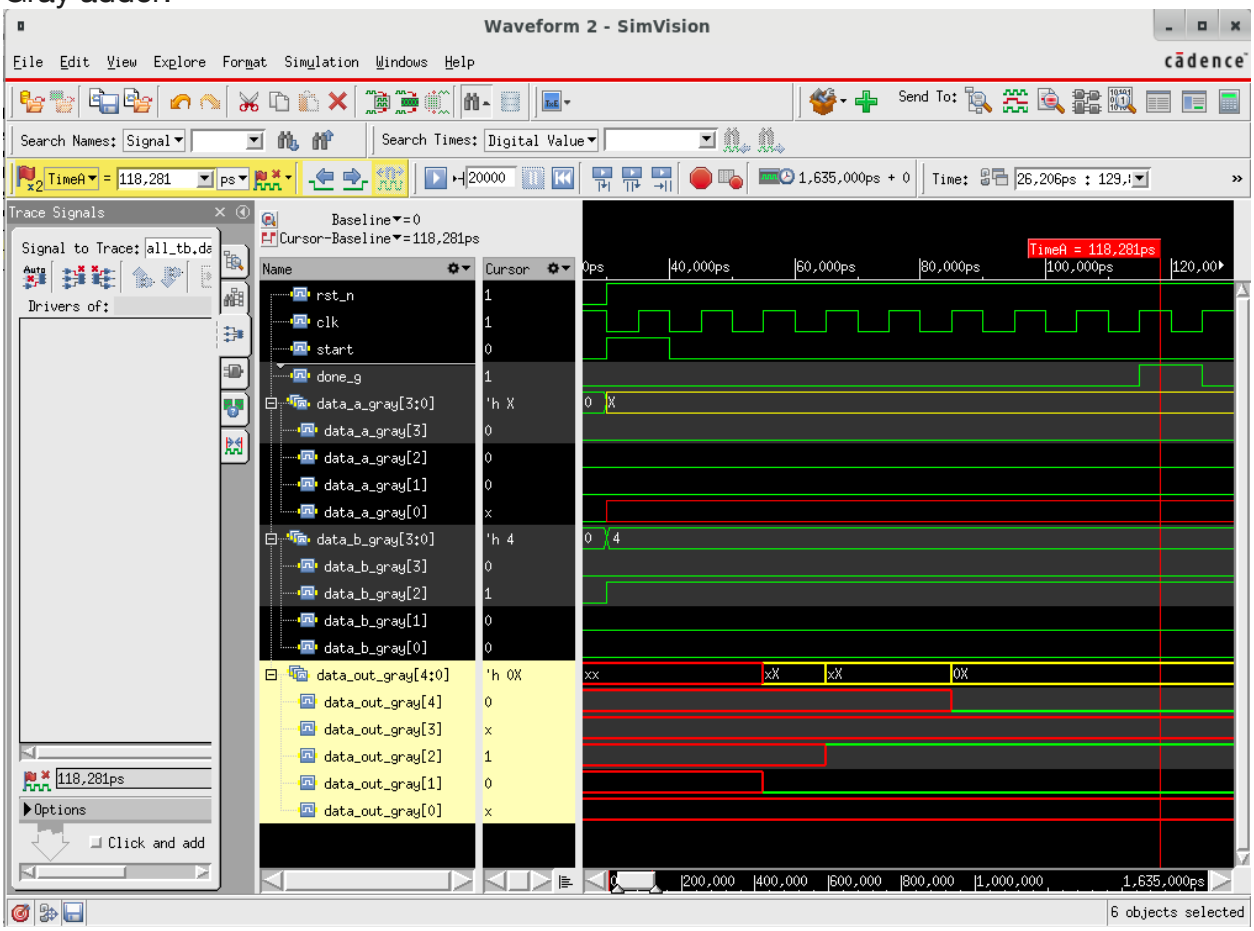
input data has one metastable bit and the output has only one metastable bit shown in reg [7], the input were 7 + [0,1] and the outputs is [7,8] which means we didn't loose accuracy.

Binary adder:



input data has one metastable bit and the outputs has 4 metastable bits, we can see that one metastable bit at the input propogated trough all output bits.
the input is 7 + [0,1] and the output is [0,15] which shows the full loss of our accuracy and information.

Gray adder:

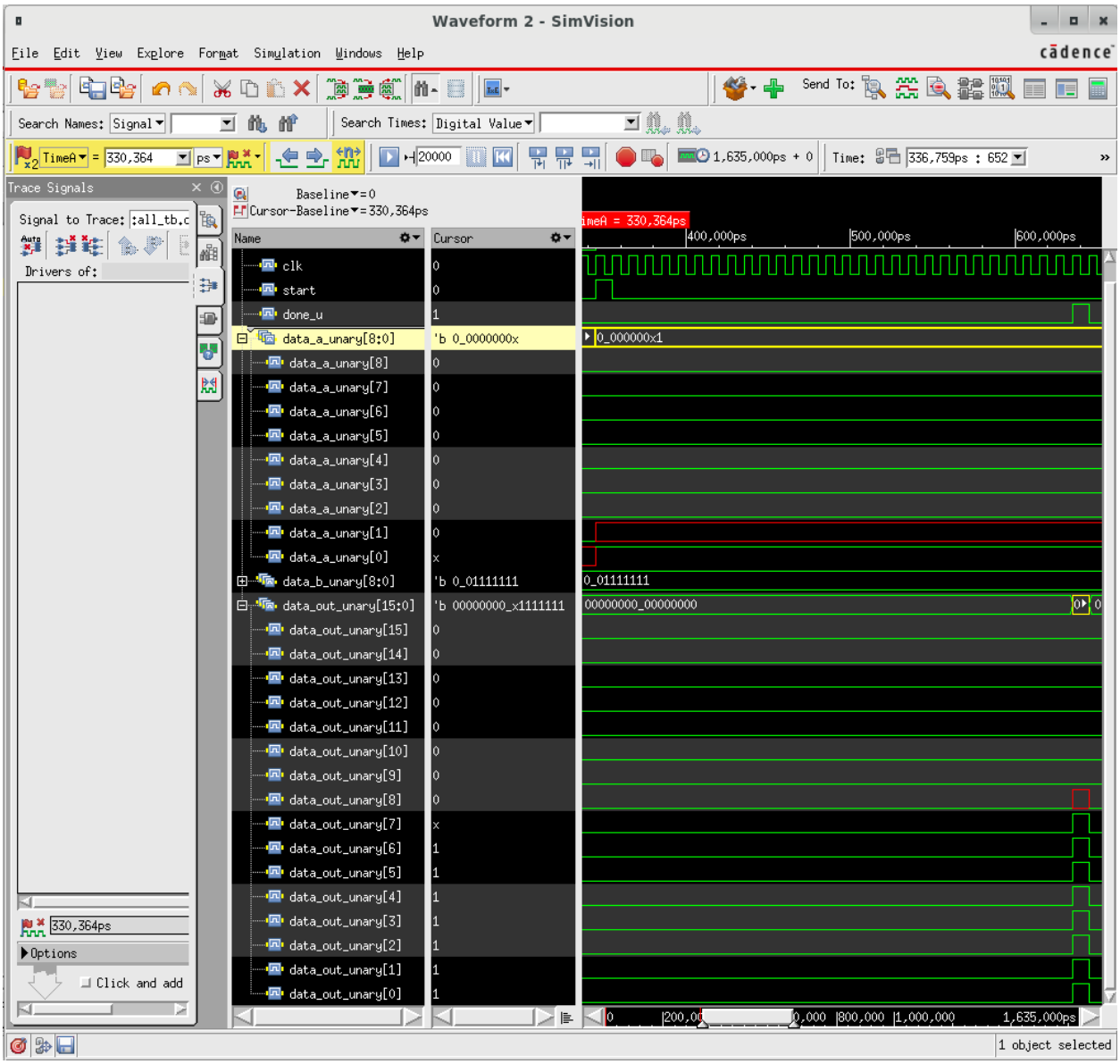


input data has one metastable bit and the outputs has two metastable bits,
the input is 7 + [0,1] and the output is [6,9]. We can see it is better then binary but still inaccuracy in output is greater than inaccuracy in input.

Inputs B

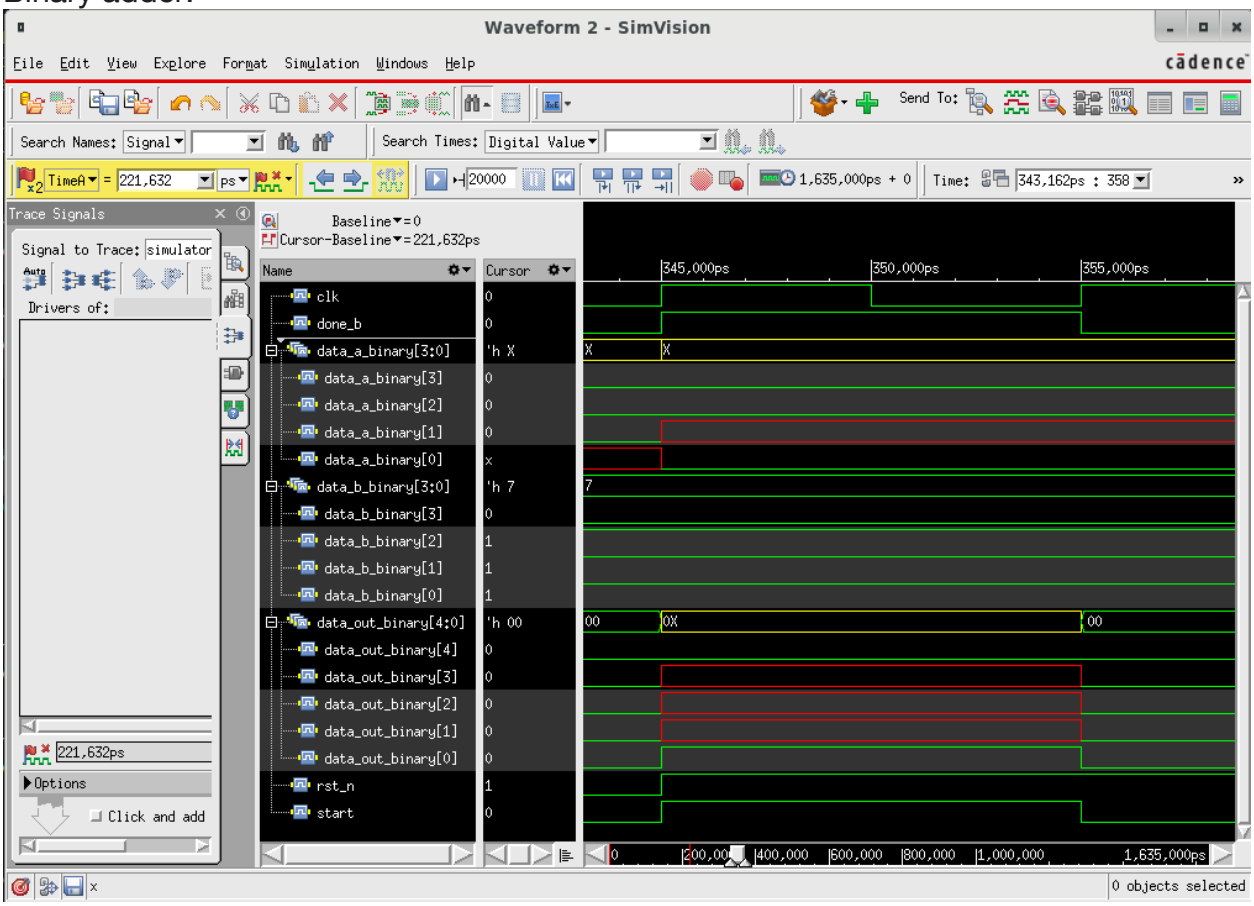
```
        // simulate x metastable bit
        data_a_binary = 8'b000000x0; // the input may be 0 or 2
        data_a_unary = 16'b00000000000000x1; // the input may be 1 or 2
        data_a_gray = 8'b000000x1; // the input may be 1 or 2

        data_b_binary = 8'b00000111;    // 7 in binary
        data_b_unary = (1<<7)-1;        // 7 unary
        data_b_gray = 8'b00000100;      //7 gray
```

Outputs signals:
Unary adder:



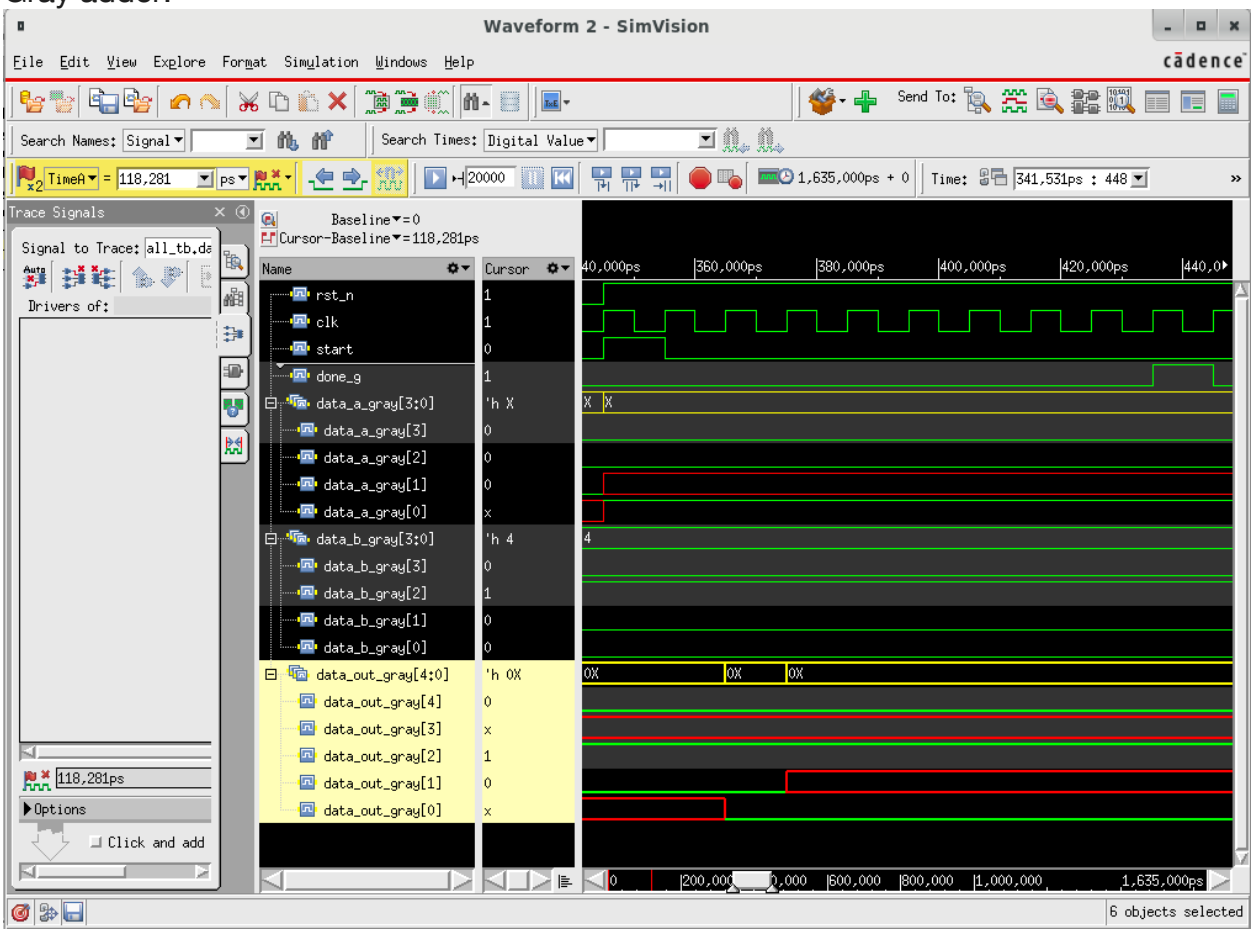
input data has one metastable bit and the output has only one metastable bit shown in reg [8], the input were 7 + [1,2] and the outputs is [8,9] which means we didn't loose accuracy.

Binary adder:



input data has one metastable bit and the outputs has 3 metastable bits, we can see that one metastable bit at the input propogated trough all output bits but the first.
the input is 7 + [1,2] and the output is [1,15] which shows that we lost accuracy on all the bits that were added with x bits (the x bits propogated to the carries).

Gray adder:

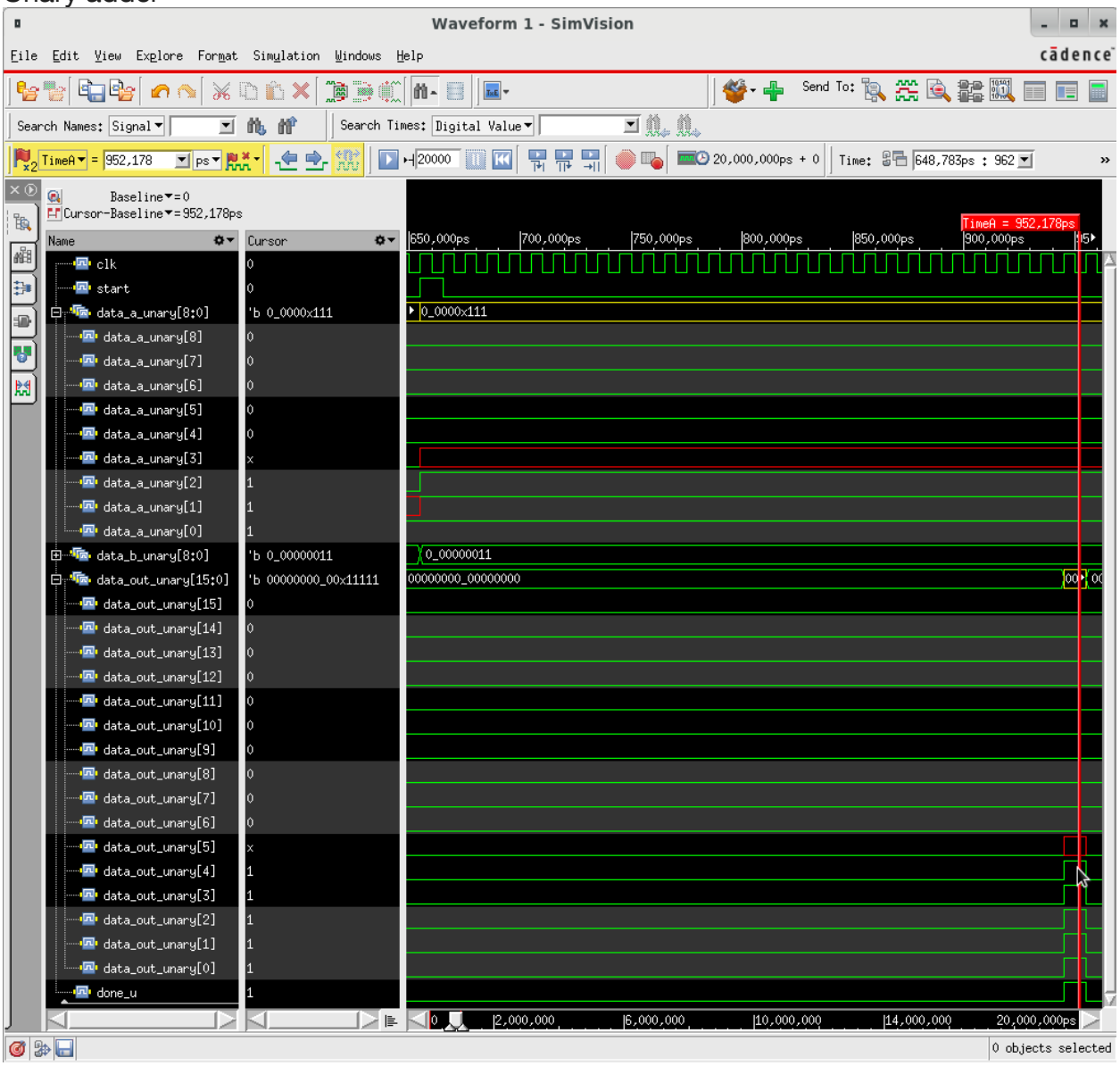


input data has one metastable bit and the outputs has two metastable bits,
the input is 7 + [1,2] and the output is [6,9] . We can see it is better then binary but still inaccuracy in output is greater than inaccuracy in input.

Inputs C

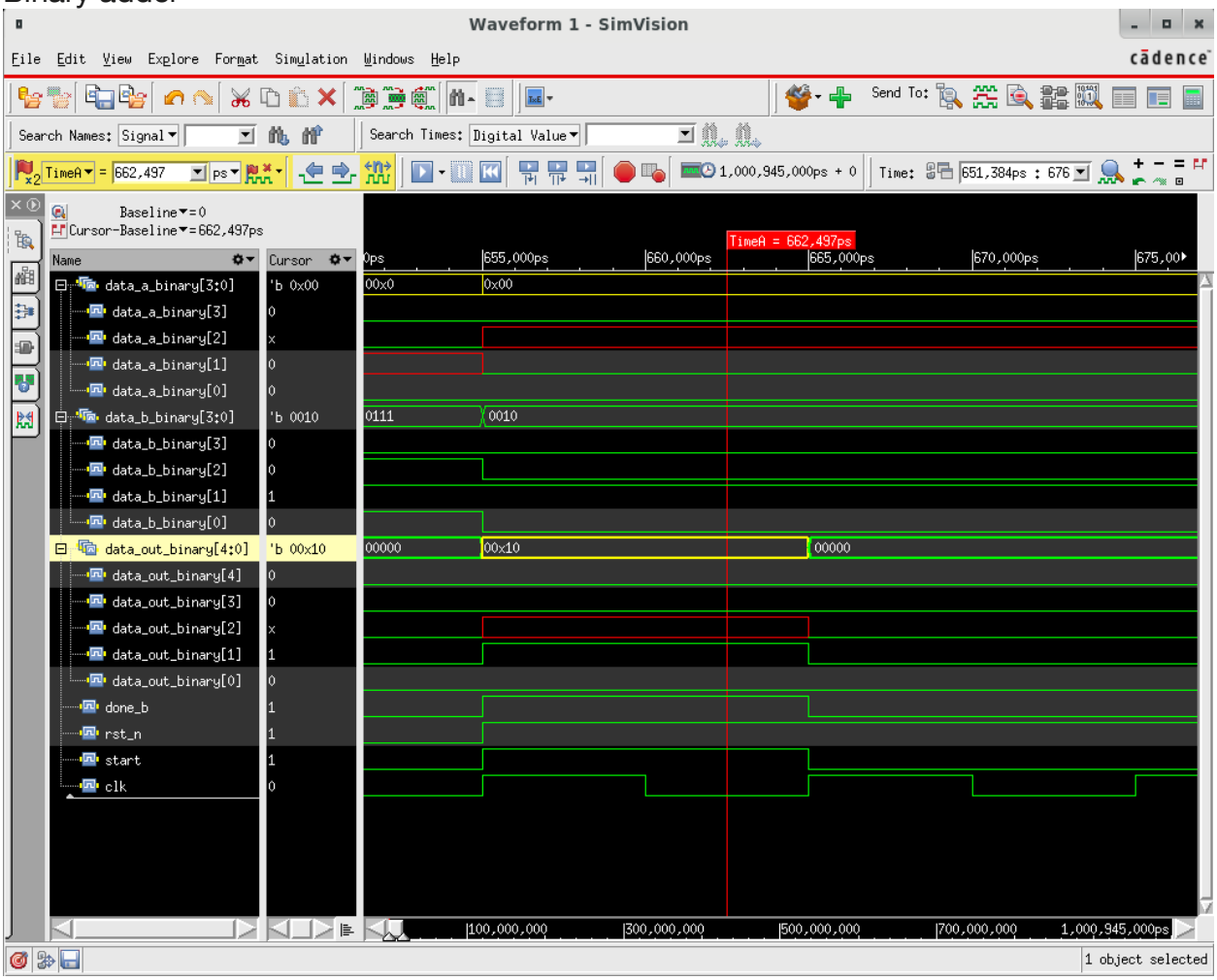
```
        // simulate x metastable bit
        data_a_binary = 8'b00000x00; // the input may be 0 or 4
        data_a_unary = 16'b0000000000000x111; // the input may be 3 or 4
        data_a_gray = 8'b00000x10; // the input may be 3 or 4

        data_b_binary = 8'b00000010;     // 2 in binary
        data_b_unary = (1<<2)-1;         // 2 unary
        data_b_gray = 8'b00000011;       // 2 gray
```

Unary adder



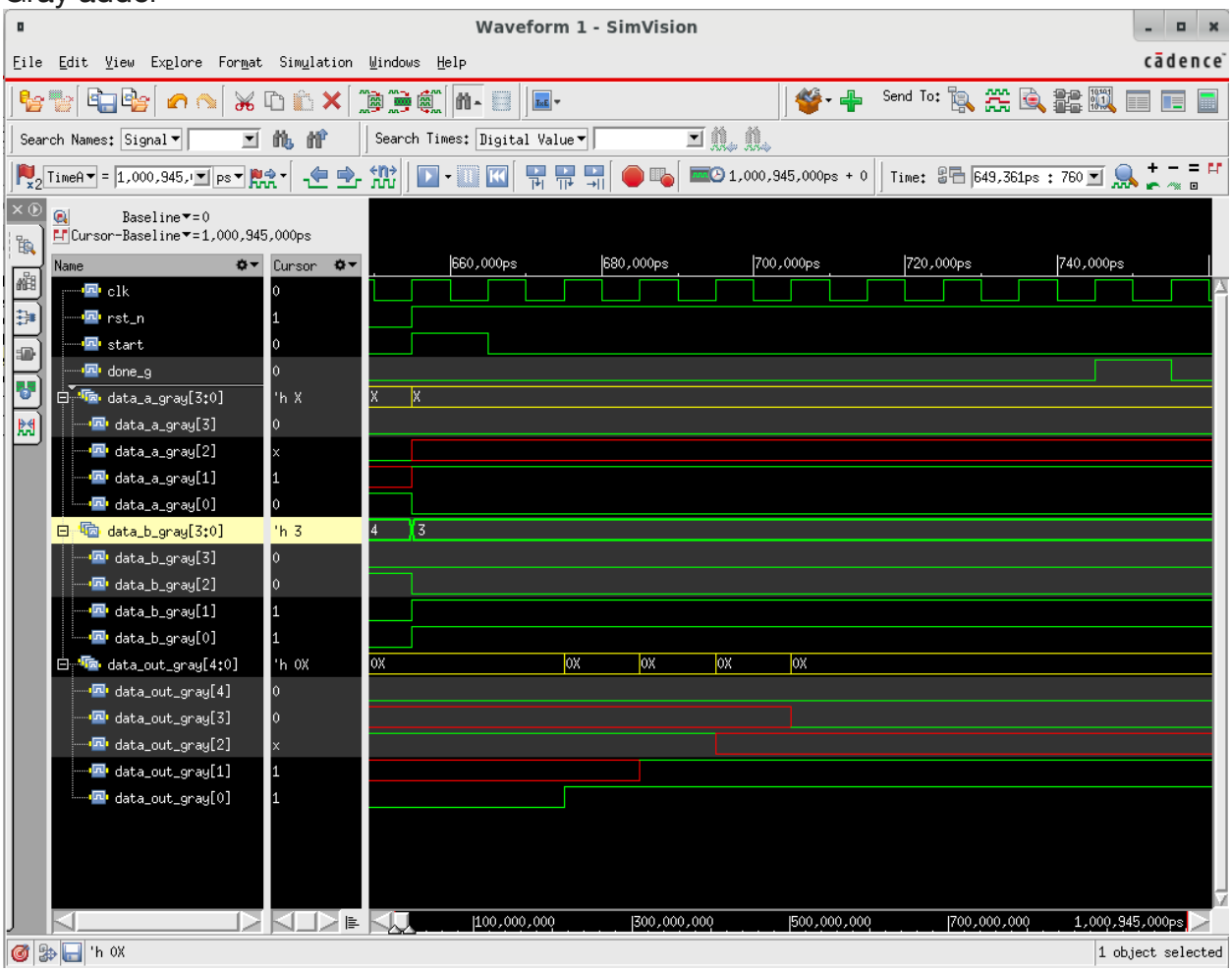
input data has one metastable bit and the output has only one metastable bit shown in reg [5], the input were 2 + [3,4] and the outputs is [5,6] which means we didn't loose accuracy.

Binary adder



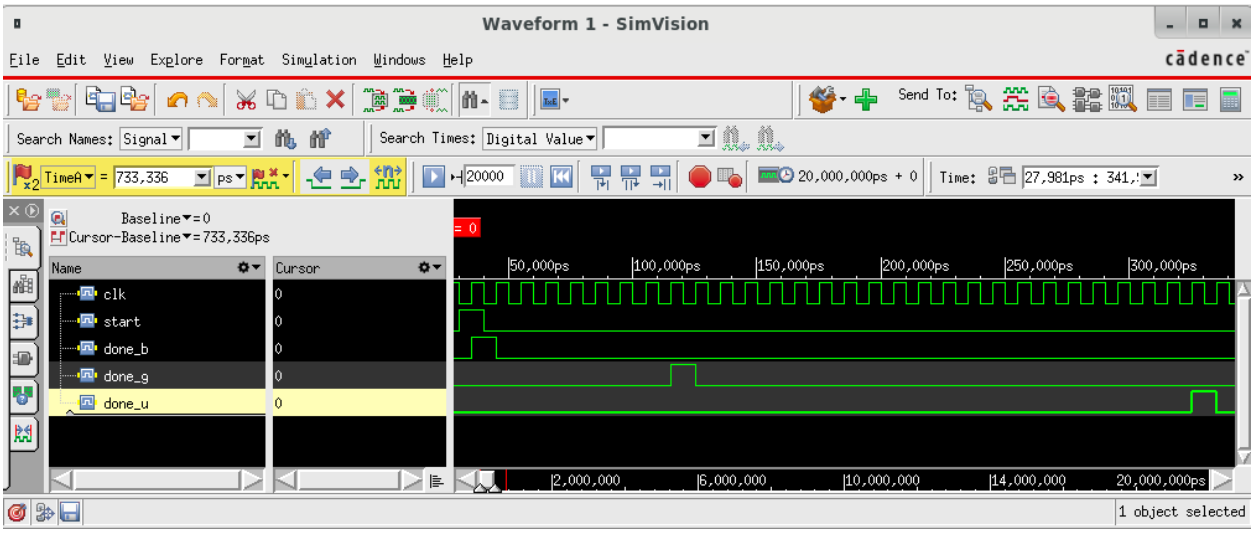
input data has one metastable bit and the outputs has also 1 metastable bits.
the input is 2 + [0,4] and the output is [6] or [2] . now inacuracy is 1 m bit but the range is large.

Gray adder



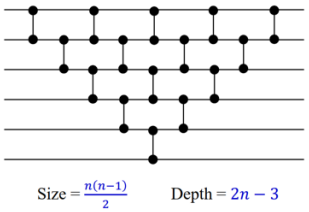
input data has one metastable bit and the outputs has two metastable bits,
the input is 2 + [3,4] and the output is [2] or [5] . We can see that the the metastable bit completely ruined the result.

Propogation delay



- **Binary Adder:** Fastest propagation delay (1 clock cycle), likely due to simultaneous sum calculation suitable for lower clock frequencies.
- **Gray-Coded Adder:** Moderate propagation delay (8 clock cycles), scaling linearly with the number of input bits as expected.
- **Unary Adder with Sorting Network:** Longest propagation delay (29 clock cycles, or 4N - 3 for input size N). The delay is primarily caused by the bubble sorting operation as we saw in class.

Selection/bubble Sort



Size = $\frac{n(n-1)}{2}$ Depth = $2n - 3$

Factors Influencing Differences

1. **Parallelism vs. Sequential Logic:**
    - The Binary Adder's speed advantage comes from its highly parallel computation. It likely calculates all bits of the output simultaneously, making it suitable for lower clock frequencies where all bits have time to stabilize.
    - Gray-Coded and Unary adders likely use a more sequential logic, calculating the result bit by bit, which inevitably takes more clock cycles.
2. **Sorting Network Overhead**

    - The Unary Adder with sorting network incurs the highest delay due to the bubble sort algorithm. Each comparison and potential swap within the sorting network contributes to the total propagation delay.

Important Considerations

- **Clock Frequency:** The suitability of the binary adder for low clock frequencies is because, under high frequencies, the simultaneous calculation may not have enough time for all output bits to settle correctly.
- **Trade-off:** The faster Binary Adder may be more susceptible to errors caused by metastability, while the slower Gray-Coded and Unary adders offer different degrees of metastability handling.

Gates nubmer utilization

Unary adder:

$$G_{count} = 2N * \frac{2N-1}{2} * 2 = 2N(2N-1)$$

$\frac{2N(2N-1)}{2}$ – number of compartors in bubblesort netwrok, and each comparator has 2 gates (AND, OR)

Binary adder:

$$G_{count} = 5N$$

N - number of input bits, and each FULL ADDER consists 2 XOR, 2 AND, 1 OR gates.

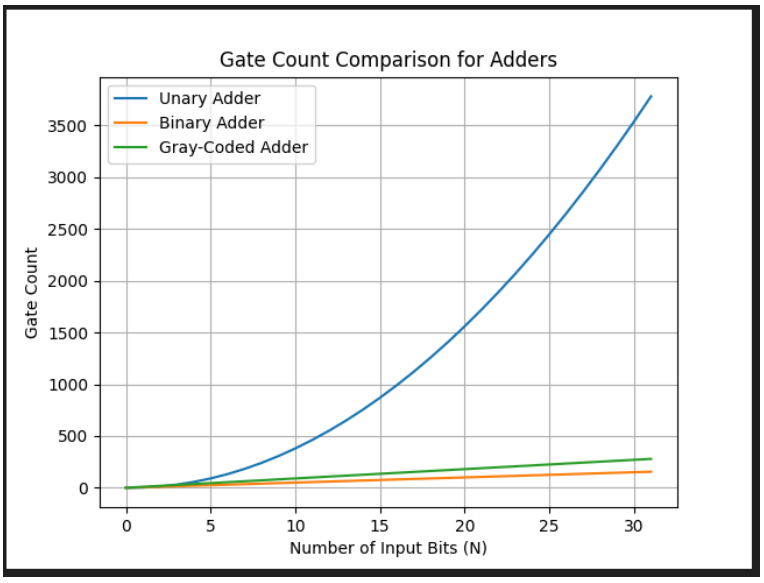
Gray adder:

$$G_{count} = 9N$$

N -number of input bits, and each bitwise adding consists 4 XOR, 3 AND, 2 NOT.

*please note that N for unary adder is different than N for binary and gray adders due to unary encoding bits utiliztion.
for example for adding two numbers in range [0,15]:

$$G_{count_{unary}}(N = 16) = 992_{gates}$$
$$G_{count_{binary}}(N = 4) = 20_{gates}$$
$$G_{count_{gray}}(N = 4) = 36_{gates}$$



We can see in the graph that gate number for unary adder raises exponantally and binary and gray adders raises logarithmically as the number of bits (N) input raises.

onclusions

- **Gate Count and Adder Complexity:** The gate count estimations illustrate the varying complexity of the adder designs:
    - **Unary Adder:** With its bubble sorting network, the Unary Adder has the highest gate count, scaling quadratically with the number of input bits (O(N^2)). This highlights the overhead of ensuring metastability handling at the input stage.
    - **Binary Adder:** The Binary Adder exhibits a linear relationship between gate count and input bits (O(N)). Its simplicity makes it efficient for standard addition.
    - **Gray-Coded Adder:** The Gray-Coded Adder also shows linear scaling (O(N)), with a slightly higher gate count than the Binary Adder due to additional NOT gates in its bitwise computation.
- **Trade-off: Speed vs. Metastability Handling:** The analysis underscores the classic trade-off between speed and resilience to metastability:
    - The Binary Adder is likely the fastest due to its parallel calculation structure. However, it's the most susceptible to metastability-induced errors.
    - The Gray-Coded Adder sacrifices some speed compared to the Binary Adder but offers reduced susceptibility to multi-bit errors caused by metastability.
    - The Unary Adder with its sorting network prioritizes metastability handling but incurs the greatest computational latency and gate count overhead.
- **Unary Adder Note:** Importantly, due to the unary encoding, 'N' in the Unary Adder represents the number of bits needed to encode a single number in unary representation. For the example of adding numbers in the range [0, 15], this difference in 'N' explains the significantly higher gate count in the Unary Adder compared to the Binary and Gray-Coded adders.

Key Takeaways

This analysis demonstrates that there is no single "best" adder design. The appropriate choice depends heavily on:

1. **Metastability Sensitivity:** How critical is reliable operation in the presence of potential metastability?
2. **Speed Requirements:** What are the maximum acceptable propagation delay and clock frequency targets?
3. **Resource Constraints (ASIC):** In the context of hardware implementation, how important are gate counts and area utilization?