PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013) 100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

LOW POWER VLSI

UE21EC342BB3

REPORT

On

Topic – Multiplexer Based Error Efficient Fixed-Width Adder Tree Design for Signal Processing Applications

Submitted by

SAHANA	**- PES1UG21EC236**
SHREYAANSH D	**- PES1UG21EC270**
SHRIYA J K	**- PES1UG21EC278**
NISHANT S	**- PES1UG21EC915**

under the guidance of

Associate Professor. Rashmi Seethur

PES University

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING PROGRAM B.TECH

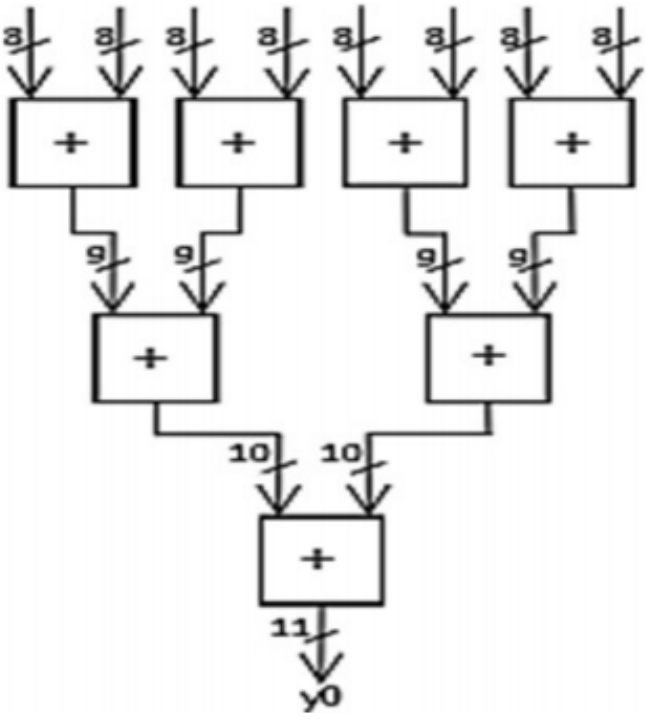
CONTENT

Full Width Adder

Methodology

A full-width adder tree is a hardware structure used to efficiently perform addition operations on multiple inputs, typically with a width equal to the sum of the individual input widths. The full-width adder tree uses multiple 8-bit inputs as its operands, which can be represented as arrays or individual signals.The full-width adder tree is typically implemented as a binary tree structure, where each level of the tree consists of adder blocks. At each level, the inputs are divided into pairs, and each pair is added together to produce partial sums. The adder blocks at each level operate in parallel, allowing for fast addition of multiple inputs.

Each adder block performs addition on two inputs and generates a partial sum and a carry-out. The carry-out from each adder block is propagated to the next level of the tree to ensure that carries are properly accounted for in the final result. At the top level of the tree, the partial sums from the lowest level adder blocks are combined to produce the final sum. Any carry-out generated at the highest level indicates an overflow. The full-width adder tree structure is highly efficient for adding multiple inputs simultaneously. By dividing the inputs into pairs and performing additions in parallel at each level, it minimizes the number of stages required to compute the final result.



DESIGN

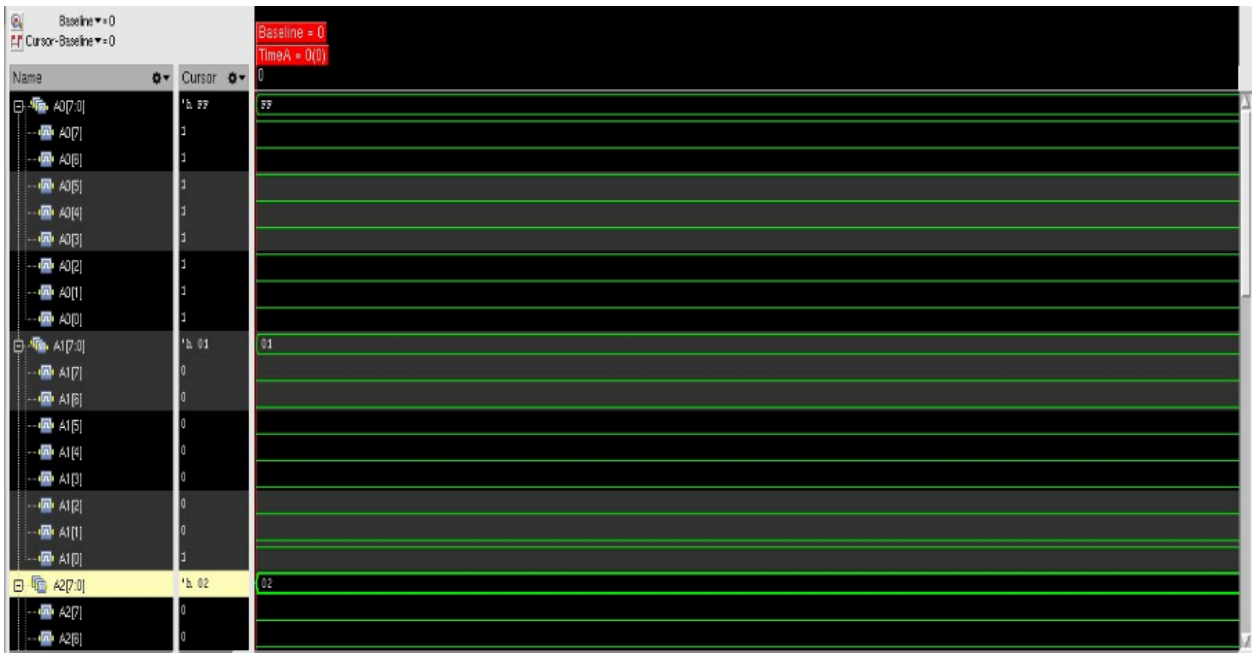
```
module full_width_adder_tree(
    input [7:0] A0, A1, A2, A3, A4, A5, A6, A7,
    output [10:0] sum  // Adjusted width to accommodate full sum without overflow
);
wire [8:0] sum1, sum2, sum3, sum4; // Intermediate sums, assuming potential overflow from 8-bit sum
wire [9:0] sum5, sum6; // Further adding may increase the bit-width
// First stage of addition
assign sum1 = A0 + A1;
assign sum2 = A2 + A3;
assign sum3 = A4 + A5;
assign sum4 = A6 + A7;
// Second stage of addition
assign sum5 = sum1 + sum2;
assign sum6 = sum3 + sum4;
// Final addition stage
assign sum = sum5 + sum6;
endmodule

module fixed_width_post_truncated_adder_tree(
    input [7:0] A0, A1, A2, A3, A4, A5, A6, A7,
    output [7:0] sum  // Truncated sum output
);
wire [10:0] full_sum; // Full width from a non-truncated adder tree
// Instantiate the full-width adder tree
full_width_adder_tree fwa(
    .A0(A0), .A1(A1), .A2(A2), .A3(A3),
    .A4(A4), .A5(A5), .A6(A6), .A7(A7),
    .sum(full_sum)
);
// Truncate the output to the required width
assign sum = full_sum[7:0]; // Taking the least significant bits only
endmodule
```

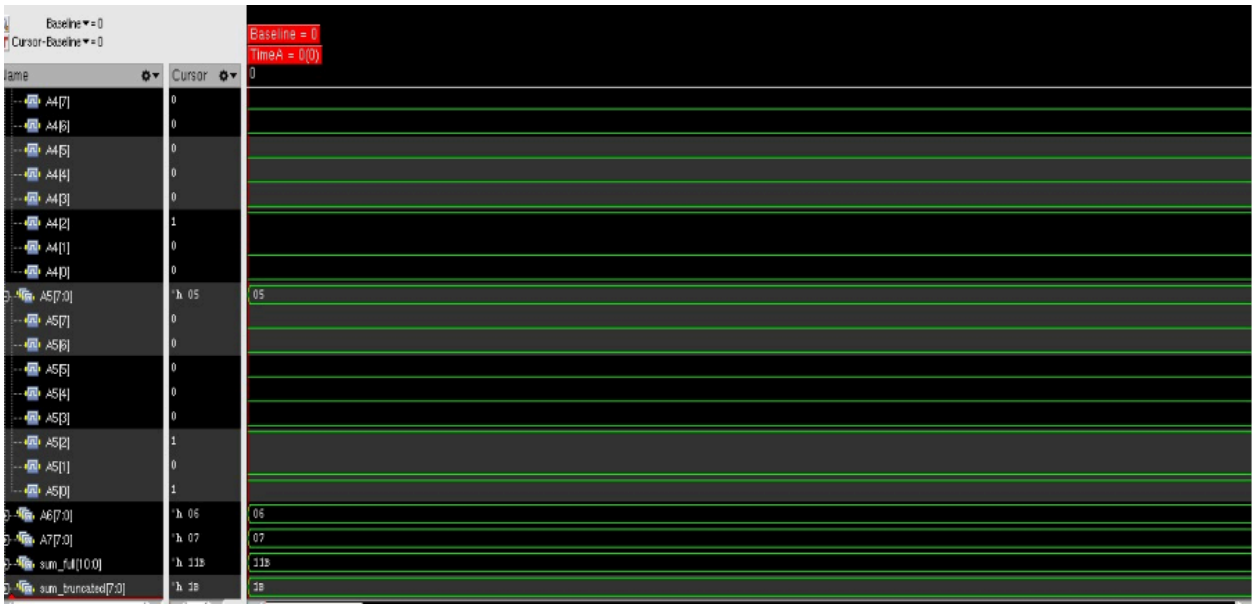
TESTBENCH

Open ▾ *lpp_tb.v ~/Desktop/SML

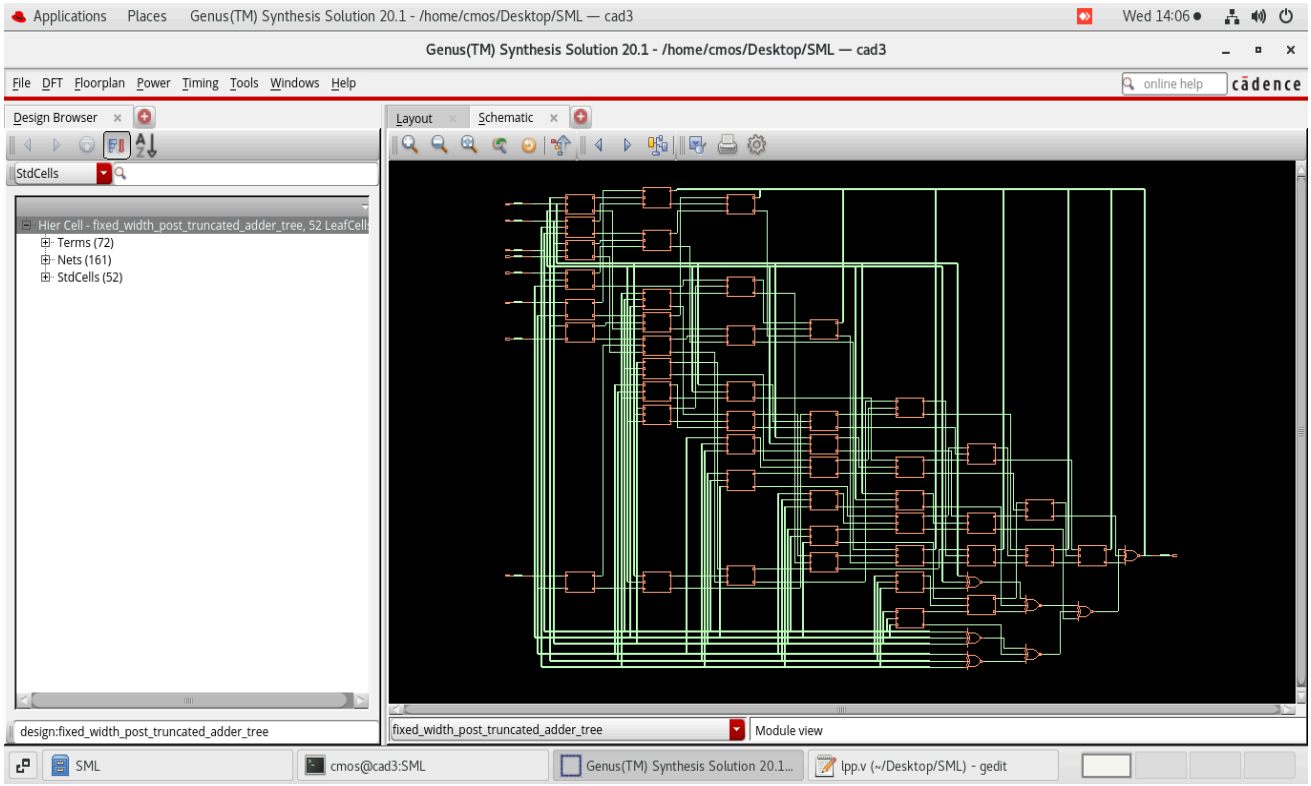
```
module tb_adder_tree;
// Test inputs
reg [7:0] A0, A1, A2, A3, A4, A5, A6, A7;
wire [7:0] sum_truncated;
wire [10:0] sum_full;

// Instantiate the Full Width and Fixed Width Adder Trees
full_width_adder_tree full_tree(
    .A0(A0), .A1(A1), .A2(A2), .A3(A3),
    .A4(A4), .A5(A5), .A6(A6), .A7(A7),
    .sum(sum_full)
);
fixed_width_post_truncated_adder_tree fixed_tree(
    .A0(A0), .A1(A1), .A2(A2), .A3(A3),
    .A4(A4), .A5(A5), .A6(A6), .A7(A7),
    .sum(sum_truncated)
);
// Initialize all inputs and simulate
initial begin
    // Initialize Inputs
    A0 = 8'hFF; A1 = 8'h01; A2 = 8'h02; A3 = 8'h03;
    A4 = 8'h04; A5 = 8'h05; A6 = 8'h06; A7 = 8'h07;

    // Wait for some time to observe the outputs
    #10;

    // Display the results
    $display("Full Sum: %d, Truncated Sum: %d", sum_full, sum_truncated);

    // Finish simulation
    $finish;
end

endmodule
```

WAVEFORM


Baseline = 0
Cursor-Baseline = 0
Name
Cursor
Baseline = 0
TimeA = 0(0)
A0[7:0]
A0[7]
A0[6]
A0[5]
A0[4]
A0[3]
A0[2]
A0[1]
A0[0]
A1[7:0]
A1[7]
A1[6]
A1[5]
A1[4]
A1[3]
A1[2]
A1[1]
A1[0]
A2[7:0]
A2[7]
A2[6]


Baseline = 0
Cursor-Baseline = 0
Cursor
Baseline = 0
TimeA = 0(0)
A4[7]
A4[6]
A4[5]
A4[4]
A4[3]
A4[2]
A4[1]
A4[0]
A5[7:0]
A5[7]
A5[6]
A5[5]
A5[4]
A5[3]
A5[2]
A5[1]
A5[0]
A6[7:0]
A7[7:0]
sum_full[10:0]
sum_truncated[7:0]

SYNTHESIS



POWER REPORT

Category	Leakage	Internal	Switching	Total	Row%
memory	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
register	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
latch	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
logic	7.23650e-06	6.05255e-05	2.58119e-05	9.35739e-05	100.00%
bbox	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
clock	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pad	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pm	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
Subtotal	7.23650e-06	6.05255e-05	2.58119e-05	9.35739e-05	100.00%
Percentage	7.73%	64.68%	27.58%	100.00%	100.00%

Applications

Microprocessors and digital signal processors frequently employ full-width adder trees for efficient arithmetic operations on multi-bit operands.

Full-width adder trees are commonly utilized in DSP applications like FIR filters and FFT algorithms, where multi-bit addition operations are frequently performed.

Advantages

Fast addition of multi-bit inputs.

Suitable for applications requiring high-speed arithmetic operations.

Efficient implementation of complex algorithms in domains like ALUs and DSP.

Disadvantages

High hardware complexity, leading to increased area and power consumption.

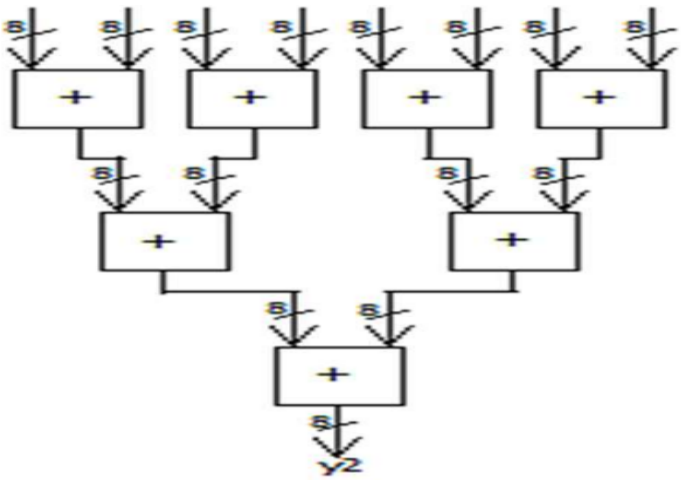
Limited scalability in terms of input width and hardware resources.

Fixed Width Direct Truncation Adder

Methodology

A Fixed Width adder tree maintains a constant width of input and output operands, while a 1-bit FX-AT-DT has each input and output being 1 bit wide. An adder tree is a hierarchical structure that efficiently performs addition operations on multiple inputs by combining partial sums from previous layers in multiple layers. The FX-AT-DT algorithm for 8-bit inputs consists of three stages or layers of adders. In the first stage, 8-bit inputs are divided into pairs and added using 1-bit adders to generate partial sums. The second stage combines the partial sums to create another set, and the third stage combines the partial sums to produce the final 8-bit sum.

The FX-AT-DT is a powerful tool that can perform multiple addition operations in parallel, reducing overall computation time significantly compared to sequential addition methods, as each stage of the adder tree operates independently. Direct Truncation is a technique where overflow bits are truncated instead of propagated, discarding significant bits beyond the output width. In a 1-bit DT adder, overflow occurs when two inputs produce a carry-out. In the FX-AT-DT, each adder operates on 1-bit inputs, eliminating the need for carry propagation and performing truncation implicitly. Output is limited to 1 bit. The FX-AT-DT is an efficient 8-bit input adder with parallel processing capabilities and fixed-width 1-bit adders, achieving faster computation times compared to traditional ripple-carry adders by dividing the addition operation into multiple stages



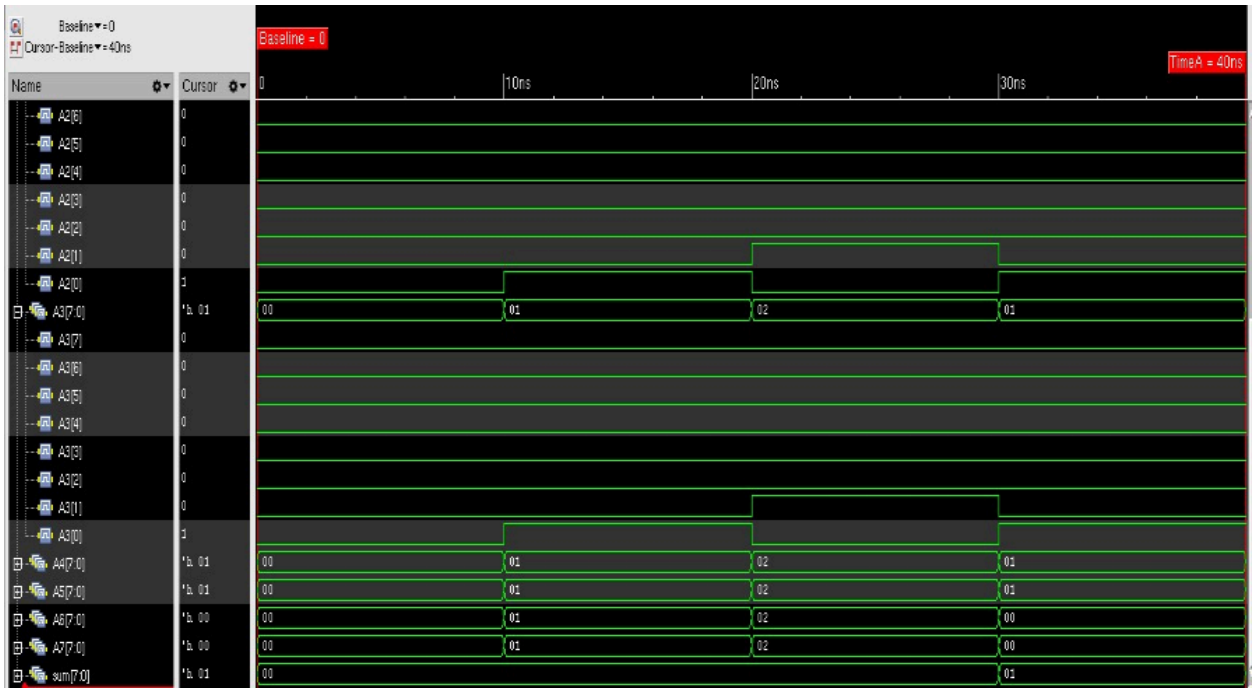
DESIGN

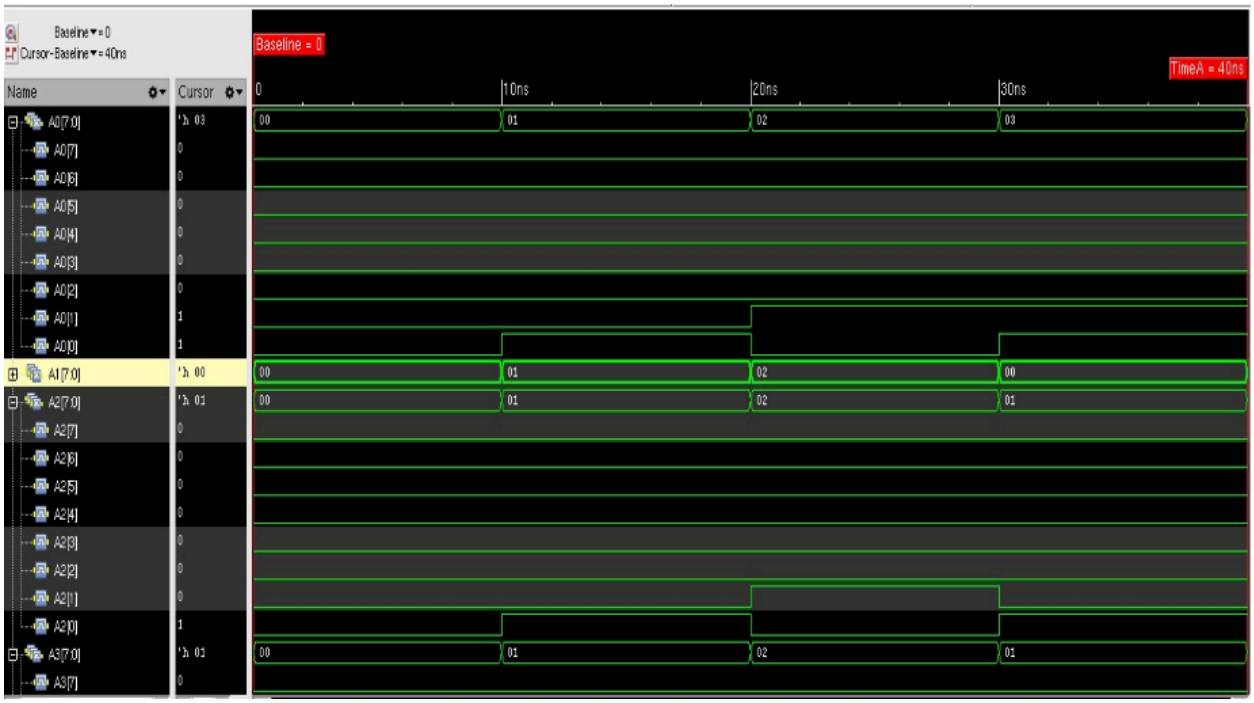
```
module fixed_width_1bit_direct_truncation_adder_tree (
    input wire [7:0] A0, A1, A2, A3, A4, A5, A6, A7,
    output reg [7:0] sum
);

// Stage 1: Perform 1-bit addition for each pair of inputs
wire [3:0] sum_stage1;
assign sum_stage1[0] = A0[0] + A1[0];
assign sum_stage1[1] = A2[0] + A3[0];
assign sum_stage1[2] = A4[0] + A5[0];
assign sum_stage1[3] = A6[0] + A7[0];

// Stage 2: Further add the intermediate sums
wire [1:0] sum_stage2;
assign sum_stage2[0] = sum_stage1[0] + sum_stage1[1];
assign sum_stage2[1] = sum_stage1[2] + sum_stage1[3];

// Stage 3: Final addition with 1-bit truncation
wire [0:0] sum_stage3;
assign sum_stage3[0] = sum_stage2[0] + sum_stage2[1];

// Assign the truncated sum to the output
always @* begin
    sum = sum_stage3[0]; // Take only the least significant bit
end

endmodule
```

TESTBENCH

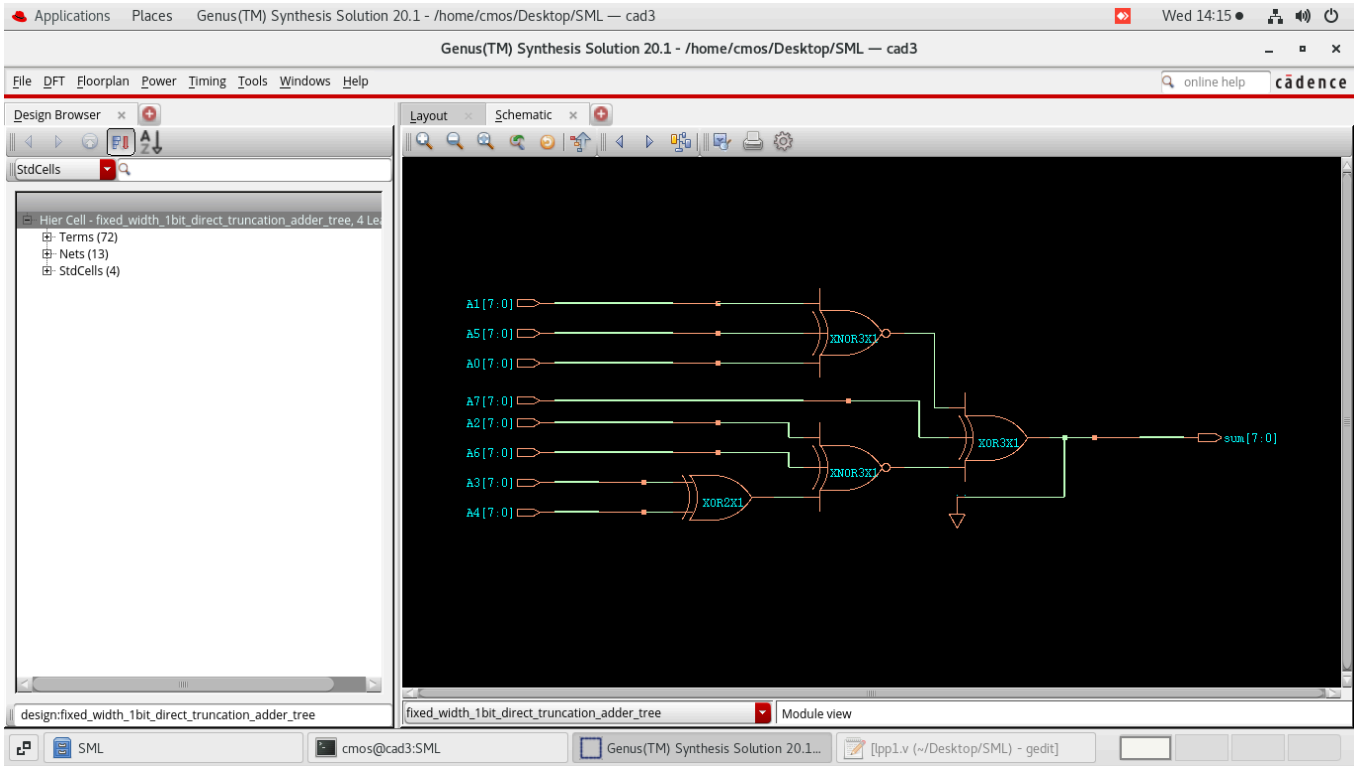
```
module tb_fixed_width_1bit_direct_truncation_adder_tree();

    // Inputs to the module
    reg [7:0] A0, A1, A2, A3, A4, A5, A6, A7;
    // Output from the module
    wire [7:0] sum;
    // Instantiate the Device Under Test (DUT)
    fixed_width_1bit_direct_truncation_adder_tree DUT (
        .A0(A0), .A1(A1), .A2(A2), .A3(A3),
        .A4(A4), .A5(A5), .A6(A6), .A7(A7),
        .sum(sum)
    );
    // Testbench procedure
    initial begin
        // Initialize all inputs
        A0 = 0; A1 = 0; A2 = 0; A3 = 0;
        A4 = 0; A5 = 0; A6 = 0; A7 = 0;

        // Display the output
        $monitor("Time = %d : A0 = %b, A1 = %b, A2 = %b, A3 = %b, A4 = %b, A5 = %b, A6 = %b, A7 = %b -> sum = %b",
                 $time, A0, A1, A2, A3, A4, A5, A6, A7, sum);

        // Apply some test vectors
        #10 A0 = 8'b00000001; A1 = 8'b00000001; A2 = 8'b00000001; A3 = 8'b00000001;
            A4 = 8'b00000001; A5 = 8'b00000001; A6 = 8'b00000001; A7 = 8'b00000001;

        #10 A0 = 8'b00000010; A1 = 8'b00000010; A2 = 8'b00000010; A3 = 8'b00000010;
            A4 = 8'b00000010; A5 = 8'b00000010; A6 = 8'b00000010; A7 = 8'b00000010;

        #10 A0 = 8'b00000011; A1 = 8'b00000000; A2 = 8'b00000001; A3 = 8'b00000001;
            A4 = 8'b00000001; A5 = 8'b00000001; A6 = 8'b00000000; A7 = 8'b00000000;
        #10 $finish;
    end
endmodule
```

WAVEFORM


Baseline = 0
TimeA = 40ns
0
10ns
20ns
30ns
Name
Cursor
A2[6]
A2[5]
A2[4]
A2[3]
A2[2]
A2[1]
A2[0]
A3[7:0]
A3[7]
A3[6]
A3[5]
A3[4]
A3[3]
A3[2]
A3[1]
A3[0]
A4[7:0]
A5[7:0]
A6[7:0]
A7[7:0]
sum[7:0]


Baseline = 0
TimeA = 40ns
0
10ns
20ns
30ns
Name
Cursor
A0[7:0]
A0[7]
A0[6]
A0[5]
A0[4]
A0[3]
A0[2]
A0[1]
A0[0]
A1[7:0]
A2[7:0]
A2[7]
A2[6]
A2[5]
A2[4]
A2[3]
A2[2]
A2[1]
A2[0]
A3[7:0]
A3[7]

SYNTHESIS



POWER REPORT

Category	Leakage	Internal	Switching	Total	Row%
memory	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
register	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
latch	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
logic	4.53503e-07	3.51728e-06	5.57820e-07	4.52860e-06	100.00%
bbox	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
clock	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pad	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pm	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
Subtotal	4.53503e-07	3.51728e-06	5.57820e-07	4.52860e-06	100.00%
Percentage	10.01%	77.67%	12.32%	100.00%	100.00%

Applications

FX-AT-DTs are utilized in data compression algorithms like Huffman coding or run-length encoding, where fixed-width addition operations are necessary for efficient data processing.

FX-AT-DTs are used in error detection and correction codes like Hamming codes or cyclic redundancy checks (CRCs) to compute checksums or parity bits.

Advantages

Flexibility in specifying the output width, enabling optimization of resource usage and efficiency.

Suitable for applications requiring fixed-width addition operations with configurable output sizes.

Disadvantages

Potential loss of precision or efficiency due to output truncation.

Complexity in determining the optimal output width for different applications and input data characteristics.

INFERENCE

Parameters	Full_width-Adder_tree	Fixed_width_Truncation_direct_Adder_Tree
Power	9.35739e-05 W	0.452860e-05 W

CONCLUSION

The development of a multiplexer-based error-efficient fixed-width adder tree design represents a significant advancement for signal processing applications. This innovative approach leverages multiplexer technology to optimize the truncation process in adder trees, effectively minimizing computational errors while maintaining a fixed-width output. This design is particularly valuable in the context of signal processing where precision and efficiency are crucial. By reducing the propagation of truncation errors and enhancing overall computational accuracy, this method improves the performance and reliability of signal processing systems. Additionally, the multiplexer-based design offers a scalable and flexible solution that can be tailored to meet specific requirements of various signal processing tasks, making it a versatile tool in both academic research and practical applications.