"Design and Implementation of an 8-bit Booth's Multiplier Using Behavioural Verilog"

1.Abstract

The Booth's multiplier is an efficient algorithm for multiplying binary numbers, leveraging the principle of Booth's coding to reduce the number of required addition operations. This Verilog implementation of an 8-bit Booth's multiplier demonstrates a hardware design that performs signed multiplication by simulating Booth's algorithm in a sequential manner. The result is a 16-bit product of two 8-bit signed integers.

2. Introduction

Multiplication of binary numbers is a fundamental operation in digital systems, utilized in various applications ranging from arithmetic operations in processors to digital signal processing. Booth's multiplication algorithm optimizes this process by minimizing the number of addition and subtraction operations required. This report provides a detailed overview of the 8-bit Booth's multiplier implemented in Verilog, describing its design, functionality, and efficiency.

3.History of Booth’s Multiplier

Booth's algorithm, developed by Andrew Donald Booth in 1950, was initially designed to improve the speed of multiplication operations in computer arithmetic. The algorithm was particularly effective in handling binary numbers with large sequences of zeros and ones, thereby reducing the number of necessary arithmetic operations. Booth's approach to encoding the multiplicand and adjusting the partial products dynamically became a foundation for designing efficient digital multipliers.

4.Algorithm

```
1)Start
2)Accept Inputs
    In1 & in2.
3)Initialize Registers
    comp2s_In1 = {~In1 + 1'b1, 9'b000000000}.
    Accu0 = {In1, 9'b000000000}.
    Accu1 = {8'b00000000, In2, 1'b0}.
4)Loop (i = 0 to 7)
    Check Accu1[1:0]
    If Accu1[1:0] == 2'b00 or 2'b11:
        Perform Arithmetic Right Shift: Accu1 = {Accu1[16], Accu1[16:1]}.
    If Accu1[1:0] == 2'b01:
        Add Accu0 to Accu1: Accu1 = Accu1 + Accu0.
        Perform Arithmetic Right Shift: Accu1 = {Accu1[16], Accu1[16:1]}.
        If Accu1[1:0] == 2'b10:
            Add comp2s_In1 to Accu1: Accu1 = Accu1 + comp2s_In1.
            Perform Arithmetic Right Shift: Accu1 = {Accu1[16], Accu1[16:1]}.
            Default:
                Accu1=17'bxxxxxxxxxxxxxxxxx.
    End Loop
5)Set Product
    Product = Accu1[16:1].
6)End
```

5.Clk_divider module(Design source)

```
`timescale 1ns / 1ps

module Clk_divider(
   input clk_in,
   output reg clk_out
);
integer counter;
// Initial values for clk_out and counter
initial begin
   clk_out = 0;
   counter = 0;
end

always @(posedge clk_in )
begin
      counter <= counter + 1;
      if (counter == 6)   // Say counter==K. where K can be calculated as
                            [(Desired Time period of output clock)*(10^8)]
      begin
         clk_out <= ~clk_out;
         counter <= 0;
       end
end
endmodule
```

6.8bit Booths Multiplier module (Design Source)

```
`timescale 1ns / 1ps

module Booths_Multiplier_8bit(clk_out,clk_in,Product,In1,In2);
input clk_in;
input [7:0]In1;
input [7:0]In2;
output reg [15:0]Product;
output clk_out;
integer i;
reg [16:0]comp2s_In1,Accu0,Accu1;

Clk_divider C1(clk_in,clk_out);

always @ (posedge clk_out) // Positive edge triggred operations
begin
      comp2s_In1 = {~In1 + 1'b1,9'b000000000}; // 2s complement of In1
      Accu0 = {In1,9'b000000000};
      Accu1 = {8'b00000000,In2,1'b0};

      for(i=0; i<8; i=i+1)
      begin
      case(Accu1[1:0])
      2'b00: Accu1 = {Accu1[16], Accu1[16:1]}; // Performing Signed right shift
      2'b01: begin
               Accu1 = Accu1 + Accu0;
               Accu1 = {Accu1[16],Accu1[16:1]}; // Performing Signed right shift
            end
      2'b10: begin
               Accu1 = Accu1 + comp2s_In1;
               Accu1 = {Accu1[16],Accu1[16:1]}; // Performing Signed right shift
            end
      2'b11: Accu1 = {Accu1[16], Accu1[16:1]}; // Performing Signed right shift
      default : Accu1 = 17'bxxxxxxxxxxxxxxxxx;
      endcase
      end // end of the loop
Product = Accu1[16:1];
end // end of always block
endmodule
```

7.Test Bench (Simulation Source)

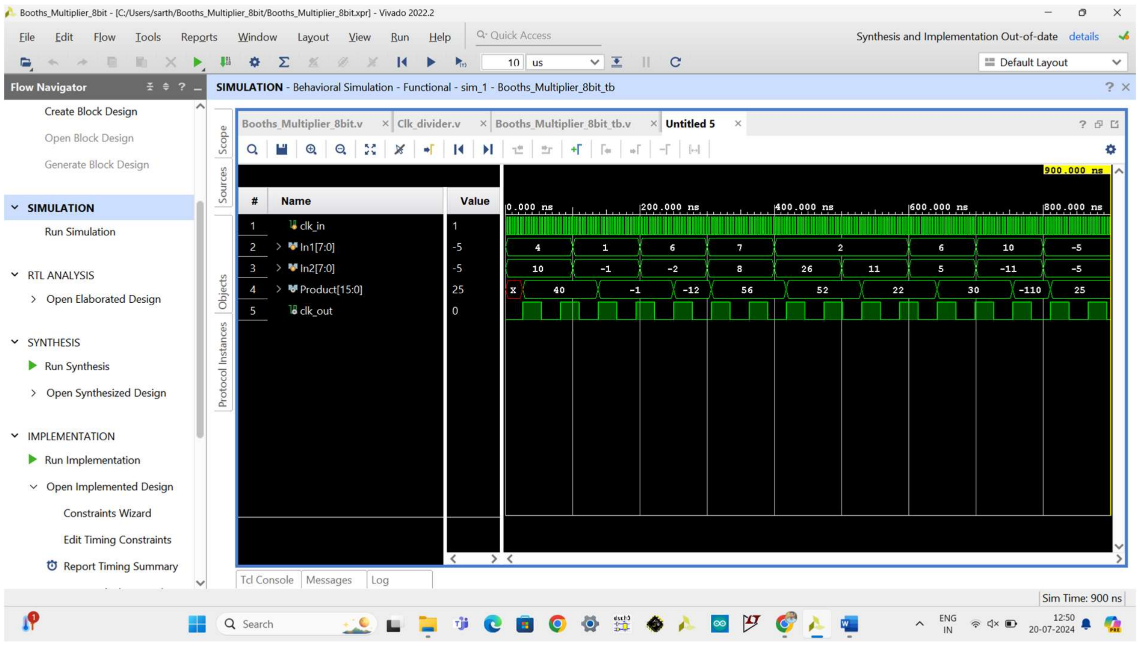
```
`timescale 1ns / 1ps

module Booths_Multiplier_8bit_tb();
// Inputs
reg clk-in;
reg [7:0] In1;
reg [7:0] In2;
// Outputs
wire [15:0] Product;
wire clk_out;

Booths_Multiplier_8bit dut (clk_in,clk_out,Product,In1,In2);
        initial begin
        // Initialize Inputs
        clk_in = 0;
        forever #2 clk_in =~clk_in;
        end

initial begin
        In1 = 8'h04; // In1=4
        In2 = 8'h0a; // In2=10
        #4
        In1 = 1; // In1=1
        In2 = -1; // In2=-1
        #4
        In1 = 8'h06; // In1=6
        In2 = 8'h02; // In2=2
        #4
        In1 = 8'b00000111; // In1=7
        In2 = 8'b1000; // In2=8
        #4
        In1 = 8'h02; // In1=2
        In2 = 8'h1a; // In2=26
        #4
        In1 = 8'h02; //In1=2
        In2 = 8'h0b; //In2=11
        #4
        In1 = 8'h06; //In1=6
        In2 = 8'h05; //In2=5
        #4
        In1 = 10;
        In2 = -11;
        #4
        In1 = -5;
        In2 = -5;
        #4 $finish;
end
endmodule
```

7.Simulation Results



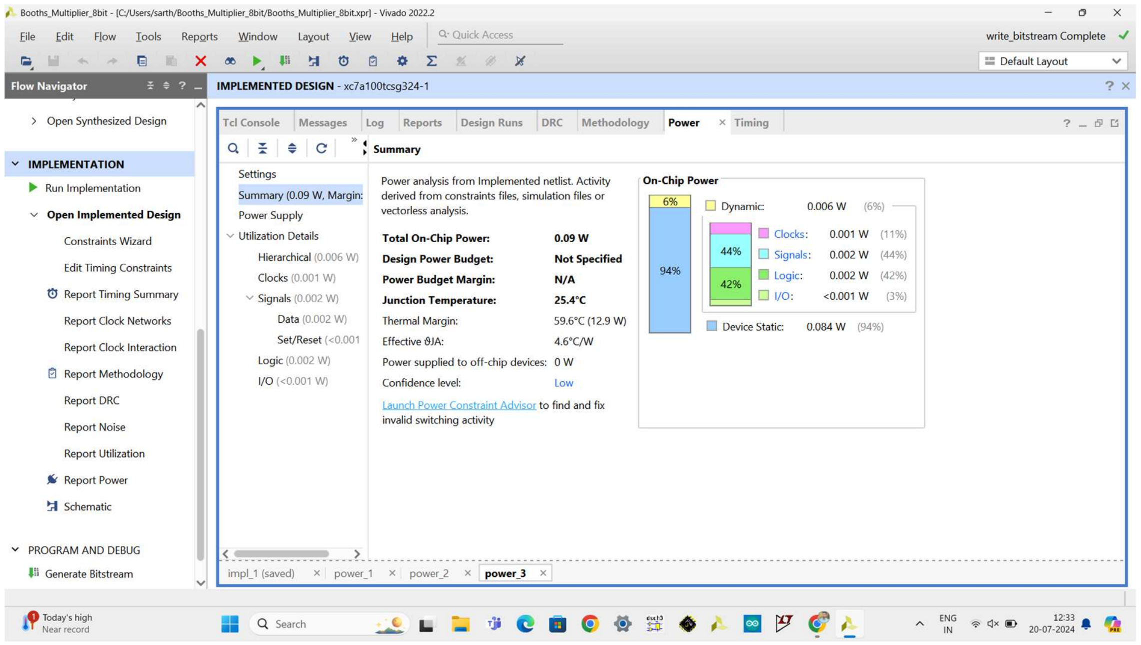
8.Constraint File (For Artix-7 Nexys 4 DDR)

```
#clock
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk_in }];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk_in}];

#Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { In1[0] }];
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { In1[1] }];
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 }[get_ports { In1[2] }];
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { In1[3] }];
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports { In1[4] }];
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { In1[5] }];
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { In1[6] }];
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports { In1[7] }];
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports { In2[0] }];
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {
In2[1] }];
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
In2[2] }];
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
In2[3] }];
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {
In2[4] }];
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
In2[5] }];
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {
In2[6] }];
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {
In2[7] }];

# LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {
Product[0] }];
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports {
Product[1] }];
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports {
Product[2] }];
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports {
Product[3] }];
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {
Product[4] }];
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {
Product[5] }];
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports {
Product[6] }];
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {
Product[7] }];
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {
Product[8] }];
set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 } [get_ports {
Product[9] }];
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {
Product[10] }];
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports {
Product[11] }];
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports {
Product[12] }];
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {
Product[13] }];
set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 } [get_ports {
Product[14] }];
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports {
Product[15] }];
```

9.Power Report



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

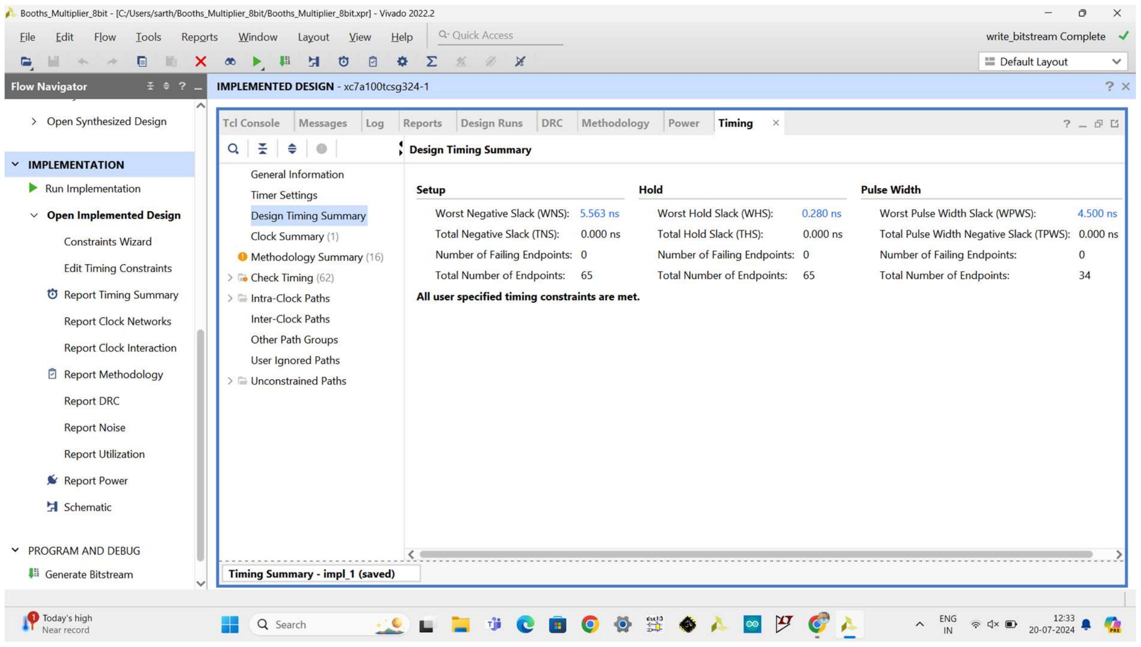
Total On-Chip Power:	**0.09 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	59.6°C (12.9 W)
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.006 W	(6%)
Clocks:	0.001 W	(11%)
Signals:	0.002 W	(44%)
Logic:	0.002 W	(42%)
I/O:	<0.001 W	(3%)
Device Static:	0.084 W	(94%)

10.Timing Report



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.563 ns	Worst Hold Slack (WHS):	0.280 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	65	Total Number of Endpoints:	65	Total Number of Endpoints:	34

All user specified timing constraints are met.

11.Conclusion

This report presents the design and implementation of an 8-bit Booth's multiplier in Verilog. By leveraging Booth's algorithm, the multiplier efficiently handles binary multiplication with fewer addition and subtraction operations. The design is suitable for digital systems requiring efficient arithmetic operations, demonstrating the practical application of historical algorithms in modern hardware design.

12.References

1)Booth, A. D. "A signed binary multiplication technique." Quarterly Journal of Mechanics and Applied Mathematics 4.2 (1951): 236-240.

2)Mano, M. Morris, and Michael D. Ciletti. "Digital Design." Pearson, 2013.

3)Wakerly, John F. "Digital Design: Principles and Practices." Pearson, 2018