//==================== BOOTH MULTIPLIER (Signed) =====================

module PARTIALPRODUCT (

input signed [7:0] INPUT1,

input [2:0] SEGMENT,

output reg signed [15:0] OUTPUT1

);

always @(\*) begin

case (SEGMENT)

3'b000, 3'b111: OUTPUT1 = 16'sd0;

3'b001, 3'b010: OUTPUT1 = INPUT1;

3'b011: OUTPUT1 = INPUT1 <<< 1;

3'b100: OUTPUT1 = -(INPUT1 <<< 1);

3'b101, 3'b110: OUTPUT1 = -INPUT1;

endcase

end

endmodule

module booth\_multiplier(

input signed [7:0] A,

input signed [7:0] B,

output signed [15:0] PRODUCT

);

wire signed [15:0] P[0:3];

wire [2:0] SEGS[0:3];

assign SEGS[0] = {B[1:0], 1'b0};

assign SEGS[1] = {B[3:2], B[1]};

assign SEGS[2] = {B[5:4], B[3]};

assign SEGS[3] = {B[7:6], B[5]};

genvar i;

generate

for (i = 0; i < 4; i = i + 1) begin: partial\_gen

PARTIALPRODUCT pp\_inst (

.INPUT1(A),

.SEGMENT(SEGS[i]),

.OUTPUT1(P[i])

);

end

endgenerate

wire signed [15:0] PP0 = P[0];

wire signed [15:0] PP1 = P[1] <<< 2;

wire signed [15:0] PP2 = P[2] <<< 4;

wire signed [15:0] PP3 = P[3] <<< 6;

wire signed [15:0] SUM1, SUM2;

hybrid\_adder\_16bit ADD1 (.a(PP0), .b(PP1), .cin(1'b0), .sum(SUM1), .cout());

hybrid\_adder\_16bit ADD2 (.a(SUM1), .b(PP2), .cin(1'b0), .sum(SUM2), .cout());

hybrid\_adder\_16bit ADD3 (.a(SUM2), .b(PP3), .cin(1'b0), .sum(PRODUCT), .cout());

endmodule

**Delay = 17.864 and luts 95 , adp = 1697.08**

**Modified booth implementations started:**

**One with original 16 bit hybrid adder and modified boooth multiplier**

**One with bk hc ling ks + modified booth - best code**

`timescale 1ns/1ps

//============================================================

// Module: Booth Encoder

//============================================================

module booth\_encoder (

input [2:0] bits,

output reg [2:0] op

);

always @(\*) begin

case(bits)

3'b000, 3'b111: op = 3'b000;

3'b001, 3'b010: op = 3'b001;

3'b101, 3'b110: op = 3'b010;

3'b011: op = 3'b011;

3'b100: op = 3'b100;

default: op = 3'b000;

endcase

end

endmodule

//============================================================

// Module: Booth Decoder

//============================================================

module booth\_decoder #(

parameter WIDTH = 8

)(

input [WIDTH-1:0] multiplicand,

input [2:0] op,

output reg [(2\*WIDTH)-1:0] pp

);

wire [WIDTH-1:0] twoA = multiplicand << 1;

always @(\*) begin

case(op)

3'b000: pp = 0;

3'b001: pp = {{WIDTH{1'b0}}, multiplicand};

3'b010: pp = {{WIDTH{1'b0}}, ~multiplicand + 1'b1};

3'b011: pp = {{WIDTH{1'b0}}, twoA};

3'b100: pp = {{WIDTH{1'b0}}, ~twoA + 1'b1};

default: pp = 0;

endcase

end

endmodule

//============================================================

// Module: Wallace Tree Reduction

//============================================================

module wallace\_tree\_reduction #(

parameter PP\_WIDTH = 16,

parameter NUM\_PP = 4

)(

input [NUM\_PP\*PP\_WIDTH-1:0] partial\_products,

output [PP\_WIDTH-1:0] sum\_out,

output [PP\_WIDTH-1:0] carry\_out

);

reg [PP\_WIDTH-1:0] temp;

integer i;

always @(\*) begin

temp = 0;

for (i = 0; i < NUM\_PP; i = i+1)

temp = temp + partial\_products[i\*PP\_WIDTH +: PP\_WIDTH];

end

assign {carry\_out, sum\_out} = temp;

endmodule

//============================================================

// 4-Bit Adder Modules

//============================================================

module brent\_kung\_4bit(

input [3:0] a,

input [3:0] b,

input cin,

output [3:0] sum,

output cout

);

wire [3:0] p, g;

wire [4:0] c;

assign c[0] = cin;

assign p = a ^ b;

assign g = a & b;

assign c[1] = g[0] | (p[0] & c[0]);

assign c[2] = g[1] | (p[1] & c[1]);

assign c[3] = g[2] | (p[2] & c[2]);

assign c[4] = g[3] | (p[3] & c[3]);

assign sum = p ^ c[3:0];

assign cout = c[4];

endmodule

module han\_carlson\_4bit(

input [3:0] a,

input [3:0] b,

input cin,

output [3:0] sum,

output cout

);

wire [3:0] p, g;

wire [4:0] c;

assign c[0] = cin;

assign p = a ^ b;

assign g = a & b;

assign c[1] = g[0] | (p[0] & c[0]);

assign c[2] = g[1] | (p[1] & (g[0] | (p[0] & c[0])));

assign c[3] = g[2] | (p[2] & c[2]);

assign c[4] = g[3] | (p[3] & c[3]);

assign sum = p ^ c[3:0];

assign cout = c[4];

endmodule

module ling\_4bit(

input [3:0] a,

input [3:0] b,

input cin,

output [3:0] sum,

output cout

);

wire [3:0] p, g;

wire [4:0] h, c;

assign c[0] = cin;

assign p = a ^ b;

assign g = a & b;

assign h[0] = g[0] | (p[0] & c[0]);

assign h[1] = g[1] | (p[1] & h[0]);

assign h[2] = g[2] | (p[2] & h[1]);

assign h[3] = g[3] | (p[3] & h[2]);

assign c[1] = h[0];

assign c[2] = h[1];

assign c[3] = h[2];

assign c[4] = h[3];

assign sum = p ^ {c[3], c[2], c[1], c[0]};

assign cout = c[4];

endmodule

module kogge\_stone\_4bit(

input [3:0] a,

input [3:0] b,

input cin,

output [3:0] sum,

output cout

);

wire [3:0] p, g;

wire [4:0] c;

assign c[0] = cin;

assign p = a ^ b;

assign g = a & b;

wire [3:0] p1, g1;

wire [3:0] p2, g2;

assign g1[0] = g[0];

assign p1[0] = p[0];

assign g1[1] = g[1] | (p[1] & g[0]);

assign p1[1] = p[1] & p[0];

assign g1[2] = g[2] | (p[2] & g[1]);

assign p1[2] = p[2] & p[1];

assign g1[3] = g[3] | (p[3] & g[2]);

assign p1[3] = p[3] & p[2];

assign g2[1] = g1[1];

assign p2[1] = p1[1];

assign g2[2] = g1[2] | (p1[2] & g1[0]);

assign p2[2] = p1[2] & p1[0];

assign g2[3] = g1[3] | (p1[3] & g1[1]);

assign p2[3] = p1[3] & p1[1];

assign c[1] = g1[0] | (p1[0] & c[0]);

assign c[2] = g2[1] | (p2[1] & c[0]);

assign c[3] = g2[2] | (p2[2] & c[0]);

assign c[4] = g2[3] | (p2[3] & c[0]);

assign sum = p ^ c[3:0];

assign cout = c[4];

endmodule

//============================================================

// Enhanced 16-bit Hybrid Adder (BK + HC + Ling + KS)

//============================================================

module hybrid\_adder\_16bit(

input [15:0] a,

input [15:0] b,

input cin,

output [15:0] sum,

output cout

);

wire c3, c7, c11;

brent\_kung\_4bit bk0 (

.a(a[3:0]), .b(b[3:0]), .cin(cin),

.sum(sum[3:0]), .cout(c3)

);

han\_carlson\_4bit hc4 (

.a(a[7:4]), .b(b[7:4]), .cin(c3),

.sum(sum[7:4]), .cout(c7)

);

ling\_4bit ling8 (

.a(a[11:8]), .b(b[11:8]), .cin(c7),

.sum(sum[11:8]), .cout(c11)

);

kogge\_stone\_4bit ks12 (

.a(a[15:12]), .b(b[15:12]), .cin(c11),

.sum(sum[15:12]), .cout(cout)

);

endmodule

//============================================================

// Top-Level Modified Booth Multiplier

//============================================================

module modified\_booth\_multiplier (

input [7:0] multiplicand,

input [7:0] multiplier,

output [15:0] product

);

wire [8:0] mult\_pad = {multiplier, 1'b0};

wire [2:0] booth\_bits0 = {mult\_pad[1:0], 1'b0};

wire [2:0] booth\_bits1 = mult\_pad[3:1];

wire [2:0] booth\_bits2 = mult\_pad[5:3];

wire [2:0] booth\_bits3 = mult\_pad[7:5];

wire [2:0] op0, op1, op2, op3;

booth\_encoder be0(.bits(booth\_bits0), .op(op0));

booth\_encoder be1(.bits(booth\_bits1), .op(op1));

booth\_encoder be2(.bits(booth\_bits2), .op(op2));

booth\_encoder be3(.bits(booth\_bits3), .op(op3));

wire [15:0] pp0, pp1, pp2, pp3;

booth\_decoder #(8) bd0(.multiplicand(multiplicand), .op(op0), .pp(pp0));

booth\_decoder #(8) bd1(.multiplicand(multiplicand), .op(op1), .pp(pp1));

booth\_decoder #(8) bd2(.multiplicand(multiplicand), .op(op2), .pp(pp2));

booth\_decoder #(8) bd3(.multiplicand(multiplicand), .op(op3), .pp(pp3));

wire [15:0] pp0\_shifted = pp0;

wire [15:0] pp1\_shifted = pp1 << 2;

wire [15:0] pp2\_shifted = pp2 << 4;

wire [15:0] pp3\_shifted = pp3 << 6;

wire [63:0] pp\_bus = {pp3\_shifted, pp2\_shifted, pp1\_shifted, pp0\_shifted};

wire [15:0] wallace\_sum, wallace\_carry;

wallace\_tree\_reduction #(16,4) wt(

.partial\_products(pp\_bus),

.sum\_out(wallace\_sum),

.carry\_out(wallace\_carry)

);

hybrid\_adder\_16bit ha(

.a(wallace\_sum),

.b(wallace\_carry),

.cin(1'b0),

.sum(product),

.cout()

);

Endmodule

**Luts 49 delay 12.495 adp = 612 - Code for correction**

**Modified booth multiplier with our hybrid and rectified wallace tree working fine:**

`timescale 1ns/1ps

//============================================================

// Module: Booth Encoder (Unchanged)

//============================================================

module booth\_encoder (

input [2:0] bits,

output reg [2:0] op

);

always @(\*) begin

case(bits)

// Operation codes based on your original encoder logic:

3'b000, 3'b111: op = 3'b000; // 0

3'b001, 3'b010: op = 3'b001; // +A

3'b101, 3'b110: op = 3'b010; // -A

3'b011: op = 3'b011; // +2A

3'b100: op = 3'b100; // -2A

default: op = 3'b000; // Should not occur

endcase

end

endmodule

//============================================================

// Module: Booth Decoder - CORRECTED for 16-bit Signed Output

//============================================================

module booth\_decoder #(

parameter WIDTH = 8

)(

// Input multiplicand - declare as signed for correctness with Booth operations

input signed [WIDTH-1:0] multiplicand,

input [2:0] op,

// Output partial product - must be signed and 16 bits wide

output reg signed [(2\*WIDTH)-1:0] pp

);

localparam PP\_WIDTH = 2\*WIDTH; // 16

// Calculate potential partial product values based on multiplicand.

// Ensure all intermediate values are 16-bit signed for correct arithmetic.

wire signed [PP\_WIDTH-1:0] val\_A, val\_negA, val\_2A, val\_neg2A;

// Calculate +A: Sign extend multiplicand to 16 bits

assign val\_A = {{ (PP\_WIDTH-WIDTH){multiplicand[WIDTH-1]} }, multiplicand};

// Calculate +2A: Arithmetically shift the 16-bit +A value left by 1

// Use signed shift or logical shift on already sign-extended value

assign val\_2A = val\_A <<< 1;

// Calculate -A: Negate the 16-bit +A value (2's complement)

assign val\_negA = -val\_A;

// Calculate -2A: Negate the 16-bit +2A value OR shift -A left by 1

assign val\_neg2A = -val\_2A; // Or assign val\_neg2A = val\_negA <<< 1;

// Select the correct 16-bit partial product based on the operation code

always @(\*) begin

case(op)

// Using the op codes defined in the user's booth\_encoder:

3'b000: pp = 16'sd0; // Zero

3'b001: pp = val\_A; // +A (Correctly sized)

3'b010: pp = val\_negA; // -A (Correctly sized)

3'b011: pp = val\_2A; // +2A (Correctly sized)

3'b100: pp = val\_neg2A; // -2A (Correctly sized)

default: pp = 16'sd0; // Default case

endcase

end

endmodule

module full\_adder (input a, input b, input cin, output sum, output cout);

assign sum = a ^ b ^ cin;

assign cout = (a & b) | (a & cin) | (b & cin);

endmodule

module half\_adder (input a, input b, output sum, output cout);

assign sum = a ^ b;

assign cout = a & b;

endmodule

module wallace\_tree\_16bit (

input signed [15:0] p0,

input signed [15:0] p1,

input signed [15:0] p2,

input signed [15:0] p3,

output signed [15:0] wt\_sum,

output signed [15:0] wt\_carry

);

wire [15:0] sum\_fa, carry\_fa, sum\_ha, carry\_ha;

wire [16:0] shifted\_carry\_fa, shifted\_carry\_ha;

genvar i;

generate

for (i = 0; i < 16; i = i + 1) begin : level1

full\_adder fa (.a(p0[i]), .b(p1[i]), .cin(p2[i]), .sum(sum\_fa[i]), .cout(carry\_fa[i]));

end

endgenerate

generate

for (i = 0; i < 16; i = i + 1) begin : level2

half\_adder ha (.a(sum\_fa[i]), .b(p3[i]), .sum(sum\_ha[i]), .cout(carry\_ha[i]));

end

endgenerate

assign wt\_sum = sum\_ha;

assign shifted\_carry\_fa = {carry\_fa, 1'b0};

assign shifted\_carry\_ha = {carry\_ha, 1'b0};

assign wt\_carry = shifted\_carry\_fa[15:0] + shifted\_carry\_ha[15:0];

endmodule

//============================================================

// 4-Bit Adder Modules (Assumed Correct - Unchanged)

//============================================================

module brent\_kung\_4bit(

input [3:0] a, input [3:0] b, input cin, output [3:0] sum, output cout

);

wire [3:0] p = a ^ b; wire [3:0] g = a & b; wire [4:0] c;

assign c[0] = cin;

assign c[1] = g[0] | (p[0] & c[0]); assign c[2] = g[1] | (p[1] & c[1]);

assign c[3] = g[2] | (p[2] & c[2]); assign c[4] = g[3] | (p[3] & c[3]);

assign sum = p ^ c[3:0]; assign cout = c[4];

endmodule

module han\_carlson\_4bit(

input [3:0] a, input [3:0] b, input cin, output [3:0] sum, output cout

);

wire [3:0] p = a ^ b; wire [3:0] g = a & b; wire [4:0] c;

assign c[0] = cin;

assign c[1] = g[0] | (p[0] & c[0]);

assign c[2] = g[1] | (p[1] & (g[0] | (p[0] & c[0]))); // Specific Han-Carlson term

assign c[3] = g[2] | (p[2] & c[2]); assign c[4] = g[3] | (p[3] & c[3]);

assign sum = p ^ c[3:0]; assign cout = c[4];

endmodule

module ling\_4bit(

input [3:0] a, input [3:0] b, input cin, output [3:0] sum, output cout

);

wire [3:0] p = a ^ b; wire [3:0] g = a & b; wire [4:0] h; wire [4:0] c;

assign c[0] = cin;

assign h[0] = g[0] | (p[0] & c[0]); assign h[1] = g[1] | (p[1] & h[0]);

assign h[2] = g[2] | (p[2] & h[1]); assign h[3] = g[3] | (p[3] & h[2]);

assign c[1] = h[0]; assign c[2] = h[1]; assign c[3] = h[2]; assign c[4] = h[3];

assign sum = p ^ c[3:0]; assign cout = c[4]; // Corrected sum based on p and calculated carries

endmodule

module kogge\_stone\_4bit(

input [3:0] a, input [3:0] b, input cin, output [3:0] sum, output cout

);

wire [3:0] p = a ^ b; wire [3:0] g = a & b; wire [4:0] c;

wire [3:0] p1, g1, p2, g2;

assign c[0] = cin;

assign g1[0] = g[0]; assign p1[0] = p[0];

assign g1[1] = g[1] | (p[1] & g[0]); assign p1[1] = p[1] & p[0];

assign g1[2] = g[2] | (p[2] & g[1]); assign p1[2] = p[2] & p[1];

assign g1[3] = g[3] | (p[3] & g[2]); assign p1[3] = p[3] & p[2];

assign g2[1] = g1[1]; assign p2[1] = p1[1]; // Note: Kogge-Stone level 2 starts at index 1

assign g2[2] = g1[2] | (p1[2] & g1[0]); assign p2[2] = p1[2] & p1[0];

assign g2[3] = g1[3] | (p1[3] & g1[1]); assign p2[3] = p1[3] & p1[1];

assign c[1] = g1[0] | (p1[0] & c[0]); // c[1] uses level 1 results

assign c[2] = g2[1] | (p2[1] & c[0]); // c[2] uses level 2 results (index 1)

assign c[3] = g2[2] | (p2[2] & c[0]); // c[3] uses level 2 results (index 2)

assign c[4] = g2[3] | (p2[3] & c[0]); // c[4] uses level 2 results (index 3)

assign sum = p ^ c[3:0]; assign cout = c[4];

endmodule

//============================================================

// Enhanced 16-bit Hybrid Adder (Unchanged)

//============================================================

module hybrid\_adder\_16bit(

input [15:0] a,

input [15:0] b,

input cin,

output [15:0] sum,

output cout

);

wire c3, c7, c11;

// Instantiate the 4-bit adder blocks

brent\_kung\_4bit bk0 ( .a(a[3:0]), .b(b[3:0]), .cin(cin), .sum(sum[3:0]), .cout(c3) );

han\_carlson\_4bit hc4 ( .a(a[7:4]), .b(b[7:4]), .cin(c3), .sum(sum[7:4]), .cout(c7) );

ling\_4bit ling8 ( .a(a[11:8]), .b(b[11:8]), .cin(c7), .sum(sum[11:8]), .cout(c11) );

kogge\_stone\_4bit ks12 ( .a(a[15:12]), .b(b[15:12]), .cin(c11), .sum(sum[15:12]), .cout(cout) );

endmodule

//============================================================

// Top-Level Modified Booth Multiplier - WITH WALLACE TREE

//============================================================

module modified\_booth\_multiplier (

// Use signed types for inputs/output for Booth compatibility

input signed [7:0] multiplicand,

input signed [7:0] multiplier,

output signed [15:0] product

);

// Pad multiplier for Booth encoding groups - handle sign extension

// We need groups based on (y\_{2i+1}, y\_{2i}, y\_{2i-1})

// Pad with multiplier sign bit and an implicit 0 at LSB end

wire signed [8:0] mult\_pad = {multiplier[7], multiplier}; // Sign extend to 9 bits

// Define Booth triplets correctly using padded multiplier

wire [2:0] booth\_bits0 = {mult\_pad[1], mult\_pad[0], 1'b0}; // {mult[1], mult[0], y-1=0}

wire [2:0] booth\_bits1 = mult\_pad[3:1]; // {mult[3], mult[2], mult[1]}

wire [2:0] booth\_bits2 = mult\_pad[5:3]; // {mult[5], mult[4], mult[3]}

wire [2:0] booth\_bits3 = mult\_pad[7:5]; // {mult[7], mult[6], mult[5]}

// Encode Booth triplets using the encoder module

wire [2:0] op0, op1, op2, op3;

booth\_encoder be0(.bits(booth\_bits0), .op(op0));

booth\_encoder be1(.bits(booth\_bits1), .op(op1));

booth\_encoder be2(.bits(booth\_bits2), .op(op2));

booth\_encoder be3(.bits(booth\_bits3), .op(op3));

// Generate 16-bit signed partial products using the corrected decoder

wire signed [15:0] pp0, pp1, pp2, pp3;

booth\_decoder #(8) bd0(.multiplicand(multiplicand), .op(op0), .pp(pp0));

booth\_decoder #(8) bd1(.multiplicand(multiplicand), .op(op1), .pp(pp1));

booth\_decoder #(8) bd2(.multiplicand(multiplicand), .op(op2), .pp(pp2));

booth\_decoder #(8) bd3(.multiplicand(multiplicand), .op(op3), .pp(pp3));

// Shift partial products according to their position (Radix-4)

// Logical left shift (<<<) is okay here since ppX are already 16-bit signed values.

wire signed [15:0] pp0\_shifted = pp0; // Shift by 0

wire signed [15:0] pp1\_shifted = pp1 <<< 2; // Shift by 2

wire signed [15:0] pp2\_shifted = pp2 <<< 4; // Shift by 4

wire signed [15:0] pp3\_shifted = pp3 <<< 6; // Shift by 6

// \*\*\* Partial Product Reduction Stage (Wallace Tree) \*\*\*

wire signed [15:0] wt\_sum, wt\_carry;

wallace\_tree\_16bit wt (

.p0(pp0\_shifted),

.p1(pp1\_shifted),

.p2(pp2\_shifted),

.p3(pp3\_shifted),

.wt\_sum(wt\_sum),

.wt\_carry(wt\_carry)

);

// \*\*\* Final Addition Stage \*\*\*

wire signed [15:0] product\_internal;

wire carry\_final;

hybrid\_adder\_16bit final\_adder (

.a(wt\_sum),

.b(wt\_carry),

.cin(1'b0),

.sum(product\_internal),

.cout(carry\_final)

);

assign product = product\_internal;

Endmodule

**Adp = 1687.932 = 15.629 x 108**