Multiplier Design using Booth Encoding

- Booth encoding techniques are used to reduce the number of terms that must be added.
- Various forms of Booth encoding techniques have been proposed.
- As an illustration, we will focus on one variation (usually called modified Booth encoding) that reduces the number of partial products that must be added by a factor of two.
  - The method is valid for signed operands and results.
- We assume that the number of bits, n, in the multiplier operand Y is an even number.
- First, consider those cases where Y > 0. Since the bit $y_{n-1} = 0$, we can write:

$$Y = \sum_{k=0}^{n-1} y_k 2^k$$

- Each term in the summation can be written as:

$$y_k 2^k = (2y_k - \frac{1}{2} 2y_k)2^k = y_k 2^{k+1} - 2y_k 2^{k-1}$$

- Use the above expansion for the odd-k terms in the summation for Y to obtain:

$$Y = (y_{n-1}2^n - 2y_{n-1}2^{n-2}) + y_{n-2}2^{n-2} + (y_{n-3}2^{n-2} - 2y_{n-3}2^{n-4}) + y_{n-4}2^{n-4} + \cdots$$
$$+ (y_3 2^4 - 2y_3 2^2) + y_2 2^2 + (y_1 2^2 - 2y_1 2^0) + y_0 2^0$$

Multiplier Design using Booth Encoding - Continued

- Defining $y_{-1} \equiv 0$ and collecting together terms with the same power of two gives:

$$Y = y_{n-1}2^{n} + (-2y_{n-1} + y_{n-2} + y_{n-3})2^{n-2} + (-2y_{n-3} + y_{n-4} + y_{n-5})2^{n-4} + \cdots$$
$$+ (-2y_{3} + y_{2} + y_{1})2^{2} + (-2y_{1} + y_{0} + y_{-1})2^{0}$$

- The first term drops out since $y_{n-1} = 0$. Define a new set of coefficients z_k for k = even #: $z_k \equiv -2y_{k+1} + y_k + y_{k-1} \quad , \quad \text{for } k = 0, 2, 4, \cdots, n-2$
- Then, we can write expressions for Y and the product, P = XY, in terms of z_k as:

$$Y = \sum_{\substack{k=0 \\ (k\text{ even})}}^{n-2} z_k 2^k \quad , \quad P = XY = \sum_{\substack{k=0 \\ (k\text{ even})}}^{n-2} (z_k X) 2^k$$

- Notice what this says: We have to generate and sum only n/2 partial products z_kX, which is approximately half of the original number of partial products y_kX (remember that we had to prepend 1 or 2 additional 0s to Y). This represents a considerable savings in the required hardware. The only mitigating factor is that each partial product is slightly more complex to generate. Since each z_k depends on the values of 3 adjacent multiplier bits (y_{k+1}, y_k and y_{k-1}), there are 8 possible cases to consider. The corresponding value of z_k in each of the 8 cases is shown on the following page.
- It can be shown that the above results also hold for the case where the Y operand is negative.

Multiplier Design using Booth Encoding - Continued

- Using the equation for z_k, we obtain the following table of the 8 possible cases:

y_{k+1}	y_k	y_{k-1}	$z_k = -2y_{k+1} + y_k + y_{k-1}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

8-bit by 8-bit Booth Multiplier Design

- For an 8x8 multiplier design, we will generate 4 partial products (call them a, b, c and d):
  - Negative partial products –X, -2X can be implemented as a one's complement plus 1 to the LSB position.
  - Multiplication by 2 (for 2X, -2X) can be implemented as a shift left by one bit position.
  - The 9-bit partial products must be sign-extended to 16 bits

k = 0: y_1 y_0 y_{-1} =>	a_8	a_8	a_8	a_8	a_8	a_8	a_8	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
k = 2: y_3 y_2 y_1 =>	b_8	b_8	b_8	b_8	b_8	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0		u_0
k = 4: y_5 y_4 y_3 =>	c_8	c_8	c_8	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0		u_1		
k = 6: y_7 y_6 y_5 =>	d_8	d_8	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0		u_2				
										u_3						
	p_{15}	p_{14}	p_{13}	p_{12}	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Ref: Marco Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, 1986.

Booth Multiplier Verilog Code: Part 1 of 3

```
// behavioral model for a Booth-encoded 8x8 signed multiplier
// 16-bit output => interpreted as a 16-bit signed number
module booth16f(x, y, p);

input  [7:0]  x, y;
output [15:0] p;

reg    [8:0]  a, b, c, d;
reg    u0, u1, u2, u3;
wire   [15:0] pp0, pp1, pp2, pp3, pp4;

// perform the booth encoding
always @(x or y) begin
 case (y[1:0])
   2'b00 : begin a = 9'b000000000;      u0 = 0; end //  0
   2'b01 : begin a = {x[7], x[7:0]};    u0 = 0; end //  1
   2'b10 : begin a = {~x[7:0], 1'b1};   u0 = 1; end // -2
   2'b11 : begin a = {~x[7], ~x[7:0]};  u0 = 1; end // -1
 endcase
```

Booth Multiplier Verilog Code: Part 2 of 3

```
case (y[3:1])
   3'b000 : begin b = 9'b000000000;     u1 = 0; end //  0
   3'b001 : begin b = {x[7], x[7:0]};   u1 = 0; end //  1
   3'b010 : begin b = {x[7], x[7:0]};   u1 = 0; end //  1
   3'b011 : begin b = {x[7:0], 1'b0};   u1 = 0; end //  2
   3'b100 : begin b = {~x[7:0], 1'b1};  u1 = 1; end // -2
   3'b101 : begin b = {~x[7], ~x[7:0]}; u1 = 1; end // -1
   3'b110 : begin b = {~x[7], ~x[7:0]}; u1 = 1; end // -1
   3'b111 : begin b = 9'b000000000;     u1 = 0; end //  0
endcase

case (y[5:3])
   3'b000 : begin c = 9'b000000000;     u2 = 0; end //  0
   3'b001 : begin c = {x[7], x[7:0]};   u2 = 0; end //  1
   3'b010 : begin c = {x[7], x[7:0]};   u2 = 0; end //  1
   3'b011 : begin c = {x[7:0], 1'b0};   u2 = 0; end //  2
   3'b100 : begin c = {~x[7:0], 1'b1};  u2 = 1; end // -2
   3'b101 : begin c = {~x[7], ~x[7:0]}; u2 = 1; end // -1
   3'b110 : begin c = {~x[7], ~x[7:0]}; u2 = 1; end // -1
   3'b111 : begin c = 9'b000000000;     u2 = 0; end //  0
endcase
```

Booth Multiplier Verilog Code: Part 3 of 3

```
 case (y[7:5])
   3'b000 : begin d = 9'b000000000;     u3 = 0; end //  0
   3'b001 : begin d = {x[7], x[7:0]};   u3 = 0; end //  1
   3'b010 : begin d = {x[7], x[7:0]};   u3 = 0; end //  1
   3'b011 : begin d = {x[7:0], 1'b0};   u3 = 0; end //  2
   3'b100 : begin d = {~x[7:0], 1'b1};  u3 = 1; end // -2
   3'b101 : begin d = {~x[7], ~x[7:0]}; u3 = 1; end // -1
   3'b110 : begin d = {~x[7], ~x[7:0]}; u3 = 1; end // -1
   3'b111 : begin d = 9'b000000000;     u3 = 0; end //  0
 endcase
end

// form the partial product terms
assign pp0 = {a[8], a[8], a[8], a[8], a[8], a[8], a[8], a[8:0]};
assign pp1 = {b[8], b[8], b[8], b[8], b[8], b[8:0], 2'b00};
assign pp2 = {c[8], c[8], c[8], c[8:0], 4'b0000};
assign pp3 = {d[8], d[8:0], 6'b000000};
assign pp4 = {9'b000000000, u3, 1'b0, u2, 1'b0, u1, 1'b0, u0};

// add up the partial product terms
assign p = pp0 + pp1 + pp2 + pp3 + pp4;

endmodule
```

Booth Multiplier – Exhaustive Testbench: Part 1 of 2

```
module tb16; // testbench for the 8-bit by 8-bit Booth signed multiplier
             // exhaustive checking of all 256*256 possible cases

reg  [7:0]  x, y;        // 8-bit inputs
integer     xval, yval;  // numerical values of inputs x and y
wire [15:0] p;           // 16-bit output of the multiplier circuit
integer     pval;        // numerical value of the product
integer     check;       // value used to check correctness
integer     i, j;        // loop variables
integer     num_correct; // counter to keep track of the number correct
integer     num_wrong;   // counter to keep track of the number wrong

// instantiate the 8-bit by 8-bit radix-4 Booth-encoded signed multiplier
booth16f mult_instance(x, y, p);

// exhaustive simulation of all 256*256 = 65,536 possible cases
initial begin
    // initialize the counter variables
    num_correct = 0; num_wrong = 0;
```

Booth Multiplier – Exhaustive Testbench: Part 2 of 2

```
 // loop through all possible cases and record the results
    for (i = 0; i < 256; i = i + 1) begin
    x = i;
    xval = -x[7]*128 + x[6:0];
    for (j = 0; j < 256; j = j + 1) begin
            y = j;
        yval = -y[7]*128 + y[6:0];
            check = xval * yval;

        // compute and check the product
            #10 pval = -p[15]*32768 + p[14:0];
            if (pval == check)
            num_correct = num_correct + 1;
            else
            num_wrong = num_wrong + 1;

            // following line is commented out, but is useful for debugging
            // $display($time, "  %d * %d = %d (%d)", xval, yval, pval, check);
        end
    end

    // print the final counter values
    $display("num_correct = %d, num_wrong = %d", num_correct, num_wrong);
end

endmodule
```

Booth Multiplier – Exhaustive Testbench Results

```
num_correct =       65536, num_wrong =          0
```

some results if // is removed

```
650170          -3 *        -8 =        24 (        24)
650180          -3 *        -7 =        21 (        21)
650190          -3 *        -6 =        18 (        18)
650200          -3 *        -5 =        15 (        15)
650210          -3 *        -4 =        12 (        12)
650220          -3 *        -3 =         9 (         9)
650230          -3 *        -2 =         6 (         6)
650240          -3 *        -1 =         3 (         3)
650250          -2 *         0 =         0 (         0)
650260          -2 *         1 =        -2 (        -2)
650270          -2 *         2 =        -4 (        -4)
```