```
module full_adder(a, b, cin, s, cout);

input  a, b, cin;
output s, cout;

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

endmodule

module RCA(p, q, ci, r);

input  [3:0] p, q;
input  ci;
output [4:0] r;

wire   [2:0] carry;

full_adder fa0(p[0], q[0], ci, r[0], carry[0]);
full_adder fa1(p[1], q[1], carry[0], r[1], carry[1]);
full_adder fa2(p[2], q[2], carry[1], r[2], carry[2]);
full_adder fa3(p[3], q[3], carry[2], r[3], r[4]);

endmodule
```

file ripple.v

```
verilog ripple.v tb3.v

        10    4 +  1 + 1 =  6 ( 6)
        20    3 + 13 + 1 = 17 (17)
        30    5 +  2 + 1 =  8 ( 8)
        40   13 +  6 + 1 = 20 (20)
        50   13 + 12 + 1 = 26 (26)
        60    6 +  5 + 0 = 11 (11)
        70    5 +  7 + 0 = 12 (12)
        80   15 +  2 + 0 = 17 (17)
        90    8 +  5 + 0 = 13 (13)
       100   13 + 13 + 1 = 27 (27)
       110    3 + 10 + 0 = 13 (13)
       120    0 + 10 + 1 = 11 (11)
       130    6 +  3 + 1 = 10 (10)
       140    3 + 11 + 1 = 15 (15)
       150    2 + 14 + 1 = 17 (17)
       160   15 +  3 + 0 = 18 (18)
       170   10 + 12 + 0 = 22 (22)
       180   10 +  1 + 0 = 11 (11)
       190    8 +  9 + 1 = 18 (18)
       200    6 +  6 + 0 = 12 (12)
```

```
module tb1; // specific input sequence, binary output

reg  [3:0] a, b;
reg  c;
wire [4:0] s;

// instantiate the 4-bit ripple-carry adder
RCA rca1(a, b, c, s);

// apply a specific set of input vectors every 10 time units
initial begin
    a = 4'b0010; b = 4'b1010; c = 1'b0;
#10 a = 4'b1111; b = 4'b0000; c = 1'b0;
#10 a = 4'b0001; b = 4'b0001; c = 1'b1;
end

// print the input and output values after they change
initial
$monitor($time, "  a = %b", a, " b = %b", b, " c = %b", c, " s = %b", s);

endmodule
```

file tb1.v

```
verilog ripple.v tb1.v

 0  a = 0010 b = 1010 c = 0 s = 01100
10  a = 1111 b = 0000 c = 0 s = 01111
20  a = 0001 b = 0001 c = 1 s = 00011
```

```
module tb3; // random inputs, decimal values including a check

reg  [3:0] a, b;  // 4-bit inputs (to be chosen randomly)
reg  c;           // carry input  (to be chosen randomly)
wire [4:0] s;     // 5-bit output of the RCA circuit
reg  [4:0] check; // 5-bit sum value used to check correctess

// instantiate the 4-bit ripple-carry adder
RCA rca1(a, b, c, s);

// simulation of 20 random addition operations
initial repeat (20) begin
    // get new operand values and compute a check value
    a = $random; b = $random; c = $random;
    check = a + b + c;

    // compute and display the sum every 10 time units
    #10 $display($time, "  %d + %d + %d = %d (%d)", a, b, c, s, check);
end

endmodule
```

file tb3.v

```
module tb5; // testbench for signed addition and subtraction

reg  [3:0] a, b;        // 4-bit inputs (to be chosen randomly)
integer    aval, bval;  // numerical values of inputs a and b
reg  c;                 // carry input  (to be used for subtraction)
wire [4:0] s;           // 5-bit output of the RCA circuit
integer    sval, dval;  // numerical values of the sum and difference
integer    sum_check;   // value used to check correctness of an addition
integer    dif_check;   // value used to check correctness of a subtraction

// instantiate the 4-bit ripple-carry adder
RCA rca1(a, b, c, s);

// simulation of 10 additions and 10 subtractions using random operands
initial repeat (10) begin
    // get new operand values and compute the two check values
    a = $random; b = $random; c = 0;
    aval = -a[3]*8 + a[2:0];
    bval = -b[3]*8 + b[2:0];
    sum_check = aval + bval;
    dif_check = aval - bval;

    // compute and display the sum with its check value
    #10 sval = -s[3]*8 + s[2:0];
    $display($time, "  %d + %d = %d (%d)", aval, bval, sval, sum_check);

    // compute and display the difference with its check value
    b = ~b; c = 1; // one's complement of b plus 1 into the LSB
    #10 dval = -s[3]*8 + s[2:0];
    $display($time, "  %d - %d = %d (%d)", aval, bval, dval, dif_check);
end

endmodule
```

file tb5.v

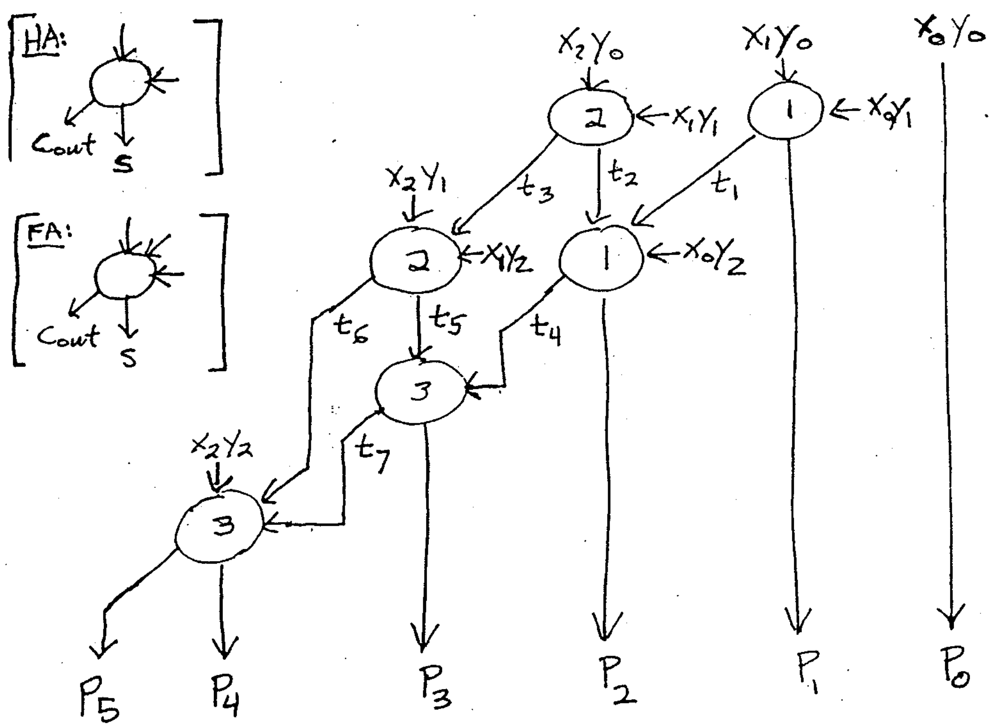
```
verilog ripple.v tb5.v

          10           4 +           1 =           5 (           5)
          20           4 -           1 =           3 (           3)
          30          -7 +           3 =          -4 (          -4)
          40          -7 -           3 =           6 (         -10)
          50          -3 +          -3 =          -6 (          -6)
          60          -3 -          -3 =           0 (           0)
          70           5 +           2 =           7 (           7)
          80           5 -           2 =           3 (           3)
          90           1 +          -3 =          -2 (          -2)
         100           1 -          -3 =           4 (           4)
         110           6 +          -3 =           3 (           3)
         120           6 -          -3 =          -7 (           9)
         130          -3 +          -4 =          -7 (          -7)
         140          -3 -          -4 =           1 (           1)
         150          -7 +           6 =          -1 (          -1)
         160          -7 -           6 =           3 (         -13)
         170           5 +          -6 =          -1 (          -1)
         180           5 -          -6 =          -5 (          11)
         190           5 +           7 =          -4 (          12)
         200           5 -           7 =          -2 (          -2)
```

```
// half adder component used in the multiplier
module half_adder(a, b, s, cout);

input  a, b;
output s, cout;

assign s = a^b;
assign cout = a&b;

endmodule

// full adder component used in the multiplier
module full_adder(a, b, cin, s, cout);

input  a, b, cin;
output s, cout;

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

endmodule

// 3-bit by 3-bit unsigned multiplier
module mult3(x, y, p);

input  [2:0] x, y;
output [5:0] p;

// internal nodes within the multiplier circuit
wire   t1, t2, t3, t4, t5, t6, t7;

// structural description of the multiplier circuit
assign p[0] = x[0]&y[0];
half_adder ha1(x[1]&y[0], x[0]&y[1], p[1], t1);
half_adder ha2(x[2]&y[0], x[1]&y[1], t2, t3);
full_adder fa1(t2, t1, x[0]&y[2], p[2], t4);
full_adder fa2(x[2]&y[1], t3, x[1]&y[2], t5, t6);
half_adder ha3(t5, t4, p[3], t7);
full_adder fa3(x[2]&y[2], t6, t7, p[4], p[5]);

endmodule
```

$\boxed{\text{file mult3.v}}$

			X_2	X_1	X_0
			Y_2	Y_1	Y_0
			X_2Y_0	X_1Y_0	X_0Y_0
		X_2Y_1	X_1Y_1	X_0Y_1	
	X_2Y_2	X_1Y_2	X_0Y_2		
P_5	P_4	P_3	P_2	P_1	P_0

3-bit by 3-bit unsigned multiplier



file tb7.v

```
module tb7; // testbench for the 3-bit by 3-bit multiplier
            // random inputs, decimal values including a check

reg  [2:0] x, y;  // 3-bit inputs (to be chosen randomly)
wire [5:0] p;     // 6-bit output of the multiplier circuit
reg  [5:0] check; // 6-bit product value used to check correctess

// instantiate the 3-bit by 3-bit multiplier
mult3 mult_instance(x, y, p);

// simulation of 20 random multiplication operations
initial repeat (20) begin
    // get new operand values and compute a check value
    x = $random; y = $random;
    check = x * y;

    // compute and display the product every 10 time units
    #10 $display($time, "  %d * %d = %d (%d)", x, y, p, check);
end

endmodule
```

```
verilog mult3.v tb7.v

 10  4 * 1 =  4 ( 4)
 20  1 * 3 =  3 ( 3)
 30  5 * 5 = 25 (25)
 40  5 * 2 = 10 (10)
 50  1 * 5 =  5 ( 5)
 60  6 * 5 = 30 (30)
 70  5 * 4 = 20 (20)
 80  1 * 6 =  6 ( 6)
 90  5 * 2 = 10 (10)
100  5 * 7 = 35 (35)
110  2 * 7 = 14 (14)
120  2 * 6 = 12 (12)
130  0 * 5 =  0 ( 0)
140  4 * 5 = 20 (20)
150  5 * 5 = 25 (25)
160  3 * 2 =  6 ( 6)
170  0 * 0 =  0 ( 0)
180  2 * 5 = 10 (10)
190  6 * 3 = 18 (18)
200  5 * 3 = 15 (15)
```

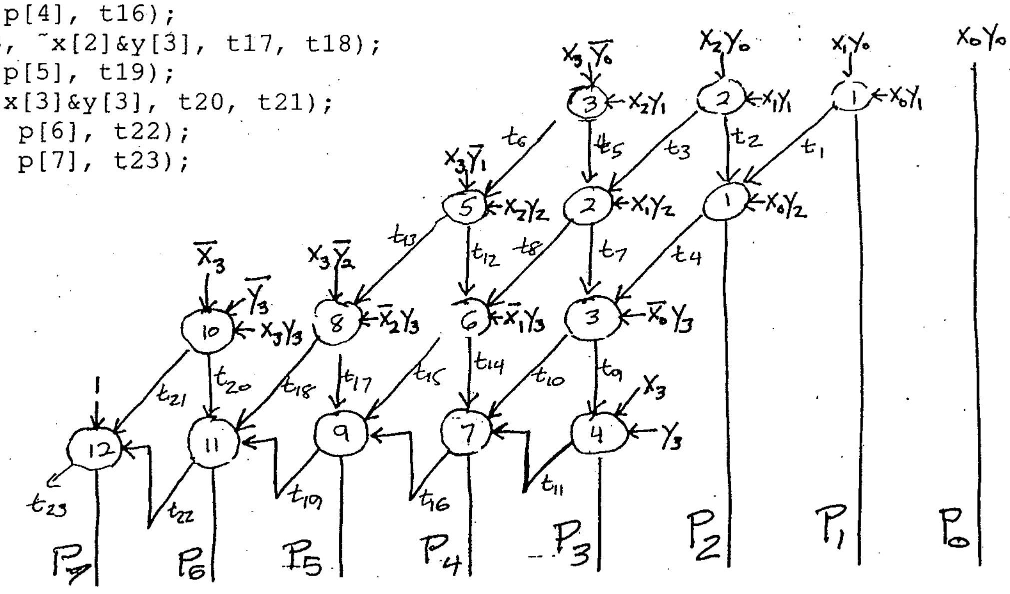
file tbe.v

```
module tbe; // testbench for the 3-bit by 3-bit unsigned multiplier
            // exhaustive checking of all 64 possible cases

reg  [2:0] x, y;  // 3-bit inputs
wire [5:0] p;     // 6-bit output of the multiplier circuit
reg  [5:0] check; // 6-bit product value used to check correctess
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 3-bit by 3-bit multiplier
mult3 mult_instance(x, y, p);

// exhaustive checking of all 64 possible cases
initial begin
    // initialize the counter variables
    num_correct = 0; num_wrong = 0;

    // loop through all possible cases and record the results
    for (i = 0; i < 8; i = i + 1) begin
        x = i;
        for (j = 0; j < 8; j = j + 1) begin
            y = j;
            check = x * y;

            // compute and check the product
            #10 if (p == 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)", x, y, p, check);
        end
    end

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

end

endmodule
```

```
verilog mult3.v tbe.v

                num_correct =          64, num_wrong =           0
```

mult4bw.v

```
// half adder component used in the multiplier
module half_adder(a, b, s, cout);

input  a, b;
output s, cout;

assign s = a^b;
assign cout = a&b;

endmodule

// full adder component used in the multiplier
module full_adder(a, b, cin, s, cout);

input  a, b, cin;
output s, cout;

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

endmodule

// 4-bit by 4-bit Baugh-Wooley signed multiplier
module mult4bw(x, y, p);

input  [3:0] x, y;
output [7:0] p;

// constant logic-one value
supply1 one;

// internal nodes within the multiplier circuit
wire   t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12,
       t13, t14, t15, t16, t17, t18, t19, t20, t21, t22, t23;

// structural description of the multiplier circuit
assign p[0] = x[0]&y[0];
half_adder ha1(x[1]&y[0], x[0]&y[1], p[1], t1);
half_adder ha2(x[2]&y[0], x[1]&y[1], t2, t3);
full_adder fa1(t2, t1, x[0]&y[2], p[2], t4);
half_adder ha3(x[3]&~y[0], x[2]&y[1], t5, t6);
full_adder fa2(t5, t3, x[1]&y[2], t7, t8);
full_adder fa3(t7, t4, ~x[0]&y[3], t9, t10);
full_adder fa4(t9, x[3], y[3], p[3], t11);
full_adder fa5(x[3]&~y[1], t6, x[2]&y[2], t12, t13);
full_adder fa6(t12, t8, ~x[1]&y[3], t14, t15);
full_adder fa7(t14, t10, t11, p[4], t16);
full_adder fa8(x[3]&~y[2], t13, ~x[2]&y[3], t17, t18);
full_adder fa9(t17, t15, t16, p[5], t19);
full_adder fa10(~x[3], ~y[3], x[3]&y[3], t20, t21);
full_adder fa11(t20, t18, t19, p[6], t22);
full_adder fa12(one, t21, t22, p[7], t23);

endmodule
```



tb9bw.v

```
module tb9; // testbench for the 4-bit by 4-bit Baugh-Wooley signed multiplier
            // exhaustive checking of all 256 possible cases

reg  [3:0] x, y;        // 4-bit inputs (to be chosen randomly)
integer    xval, yval;  // numerical values of inputs x and y
wire [7:0] p;           // 8-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 4-bit by 4-bit Baugh-Wooley signed multiplier
mult4bw mult_instance(x, y, p);

// exhaustive simulation of all 256 possible cases
initial begin
    // initialize the counter variables
    num_correct = 0; num_wrong = 0;

    // loop through all possible cases and record the results
    for (i = 0; i < 16; i = i + 1) begin
        x = i;
        xval = -x[3]*8 + x[2:0];
        for (j = 0; j < 16; j = j + 1) begin
            y = j;
            yval = -y[3]*8 + y[2:0];
            check = xval * yval;

            // compute and check the product
            #10 pval = -p[7]*128 + p[6:0];
            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
```

```
Compiling source file "tb9bw.v"
Compiling source file "mult4bw.v"
Highest level modules:
tb9

num_correct =          256, num_wrong =           0
```

Some results if // is removed

```
2200      -3 *      -5 =      15 (       15)
2210      -3 *      -4 =      12 (       12)
2220      -3 *      -3 =       9 (        9)
2230      -3 *      -2 =       6 (        6)
2240      -3 *      -1 =       3 (        3)
2250      -2 *       0 =       0 (        0)
2260      -2 *       1 =      -2 (       -2)
2270      -2 *       2 =      -4 (       -4)
2280      -2 *       3 =      -6 (       -6)
2290      -2 *       4 =      -8 (       -8)
2300      -2 *       5 =     -10 (      -10)
2310      -2 *       6 =     -12 (      -12)
```

booth16f.v

```
// 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'b00000000;          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

    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

    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
```

tb16.v

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

reg  [7:0]   x, y;         // 4-bit inputs (to be chosen randomly)
integer      xval, yval;   // numerical values of inputs x and y
wire [15:0]  p;            // 8-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;

    // 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
```

```
Compiling source file "tb16.v"
Compiling source file "booth16f.v"
Highest level modules:
tb16

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)
```