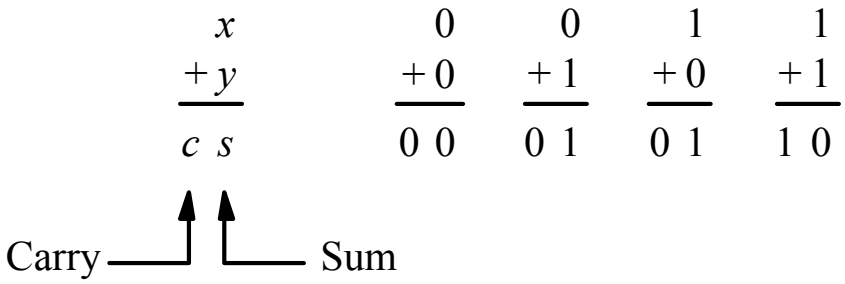
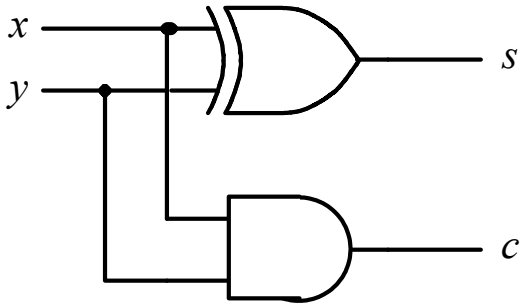


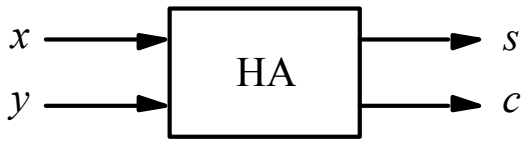
(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit

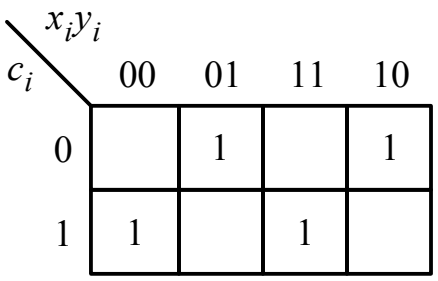


(d) Graphical symbol

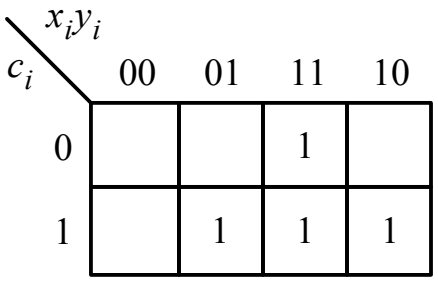
Figure 3.1. Half-adder.

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

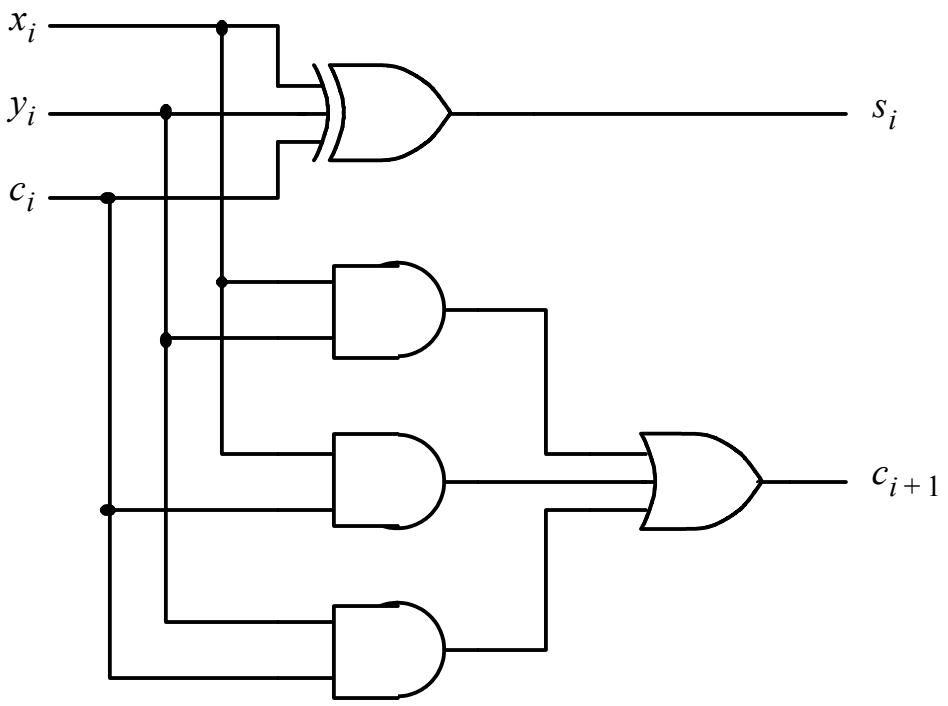


$$s_i = x_i \oplus y_i \oplus c_i$$



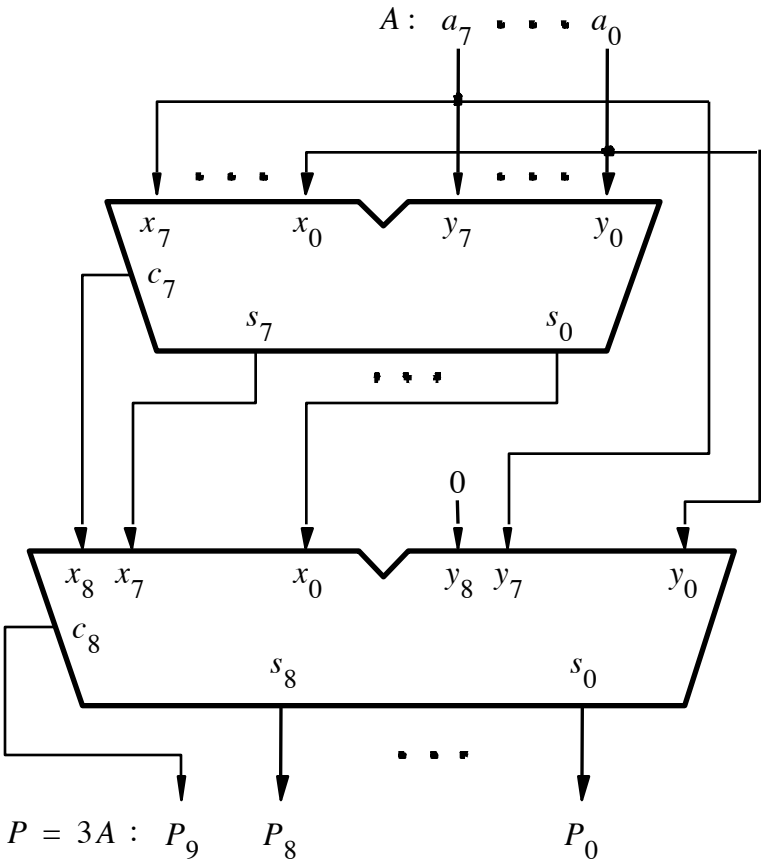
$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

(b) Karnaugh maps

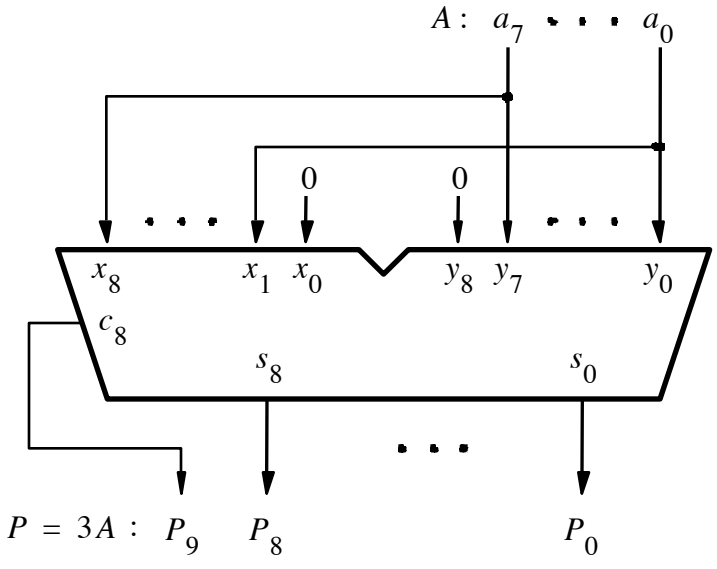


(c) Circuit

Figure 3.3. Full-adder.



(a) Naive approach



(b) Efficient design

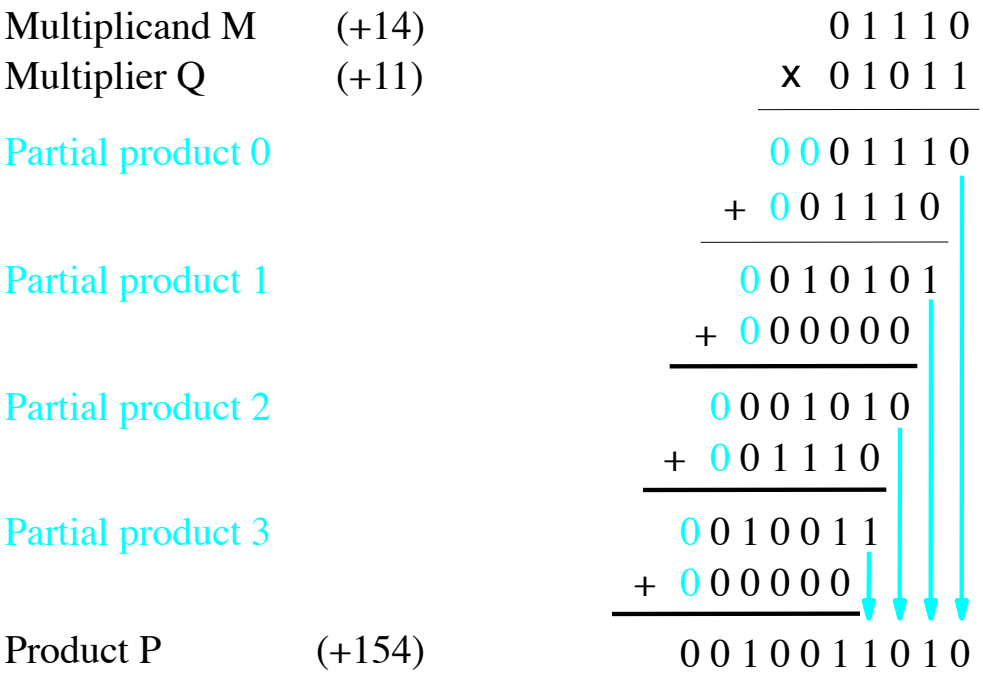
Figure 3.6. Circuit that multiplies an eight-bit unsigned number by 3.

```
module addern (carryin, X, Y, S, carryout);
  parameter n = 32;
  input carryin;
  input [n-1:0] X, Y;
  output [n-1:0] S;
  output carryout;
  wire [n:0] C;

  genvar i;
  assign C[0] = carryin;
  assign carryout = C[n];
  generate
    for (i = 0; i <= n-1; i = i+1)
    begin:addbit
      fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
    end
  endgenerate
endmodule


module fulladd (Cin, x, y, s, Cout);
      input Cin, x, y;
      output s, Cout;
      assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);
endmodule
```

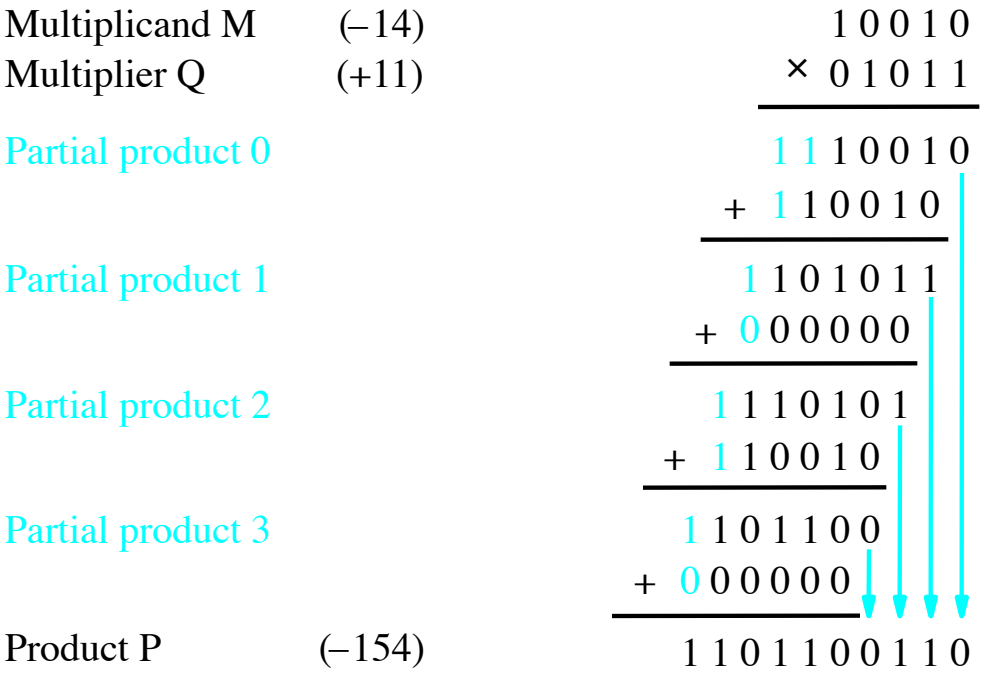
Figure 3.29. A ripple-carry adder specified using the **generate** statement.

```
module adder_hier (A, B, C, D, S, T, overflow);
    input [15:0] A, B;
    input [7:0] C, D;
    output [16:0] S;
    output [8:0] T;
    output overflow;

    wire o1, o2; // used for the overflow signals

    addern #(.n(16)) U1
    (
        .carryin (1'b0),
        .X (A),
        .Y (B),
        .S (S[15:0]),
        .carryout (S[16]),
        .overflow (o1)
    );
    addern #(.n(8)) U2
    (
        .carryin (1'b0),
        .X (C),
        .Y (D),
        .S (T[7:0]),
        .carryout (T[8]),
        .overflow (o2)
    );

    assign overflow = o1 | o2;

endmodule
```

Figure 3.33. An alternative version of the code in Figure 3.32.



(a) Positive multiplicand



(b) Negative multiplicand

Figure 3.36. Multiplication of signed numbers.

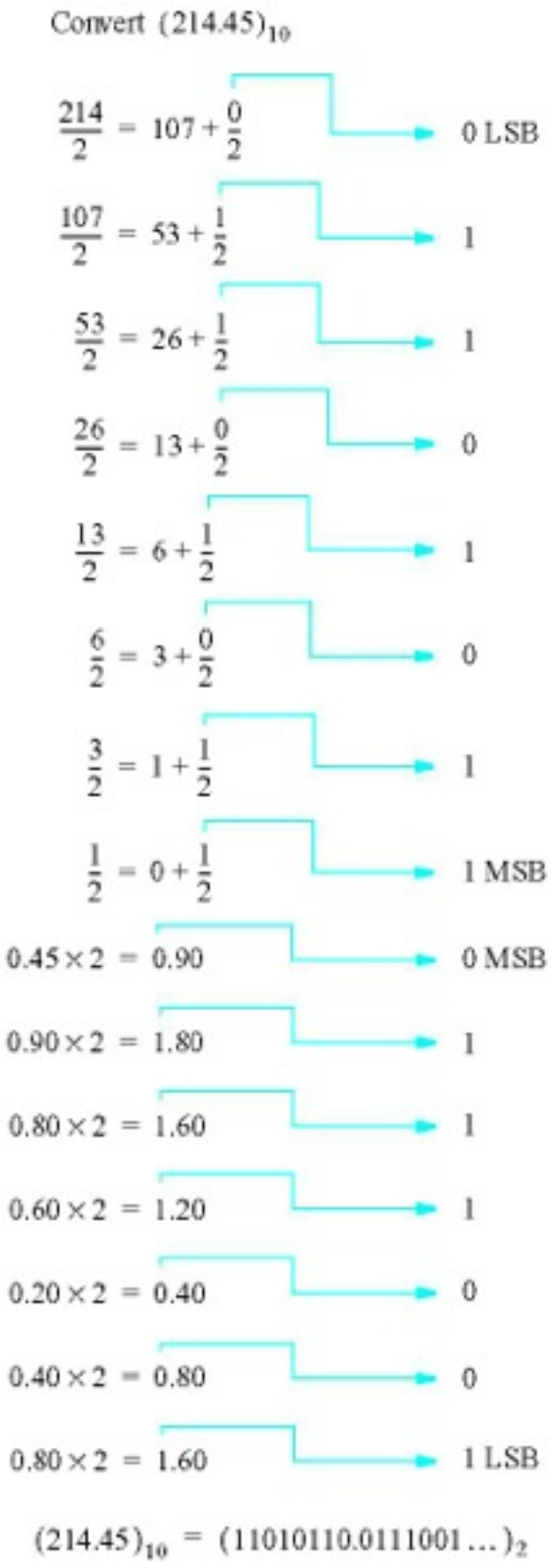


Figure 3.44. Conversion of fixed point numbers from decimal to binary.

```
module comparator (X, Y, V, N, Z);
  input [3:0] X, Y;
  output V, N, Z;
  wire [3:0] S;
  wire [4:1] C;

  fulladd stage0 (1'b1, X[0], ~Y[0], S[0], C[1]);
  fulladd stage1 (C[1], X[1], ~Y[1], S[1], C[2]);
  fulladd stage2 (C[2], X[2], ~Y[2], S[2], C[3]);
  fulladd stage3 (C[3], X[3], ~Y[3], S[3], C[4]);
  assign V  = C[4] ^ C[3];
  assign N = S[3];
  assign Z = !S;

  endmodule



module fulladd (Cin, x, y, s, Cout);
      input Cin, x, y;
      output s, Cout;
      assign s = x ^ y ^ Cin,
          Cout = (x & y) | (x & Cin) | (y & Cin);
endmodule
```

Figure 3.46. Structural Verilog code for the comparator circuit.

```
module comparator (X, Y, V, N, Z);
  parameter n = 32;
  input [n-1:0] X, Y;
  output reg V, N, Z;
  reg [n-1:0] S;
  reg [n:0] C;
  integer k;

  always @(X, Y)
  begin
    C[0] = 1'b1;
    for (k = 0; k < n;  k = k + 1)
    begin
      S[k] = X[k] ^ ~Y[k] ^ C[k];
      C[k+1] = (X[k] & ~Y[k]) | (X[k] & C[k]) | (~Y[k] & C[k]);
    end
    V  = C[n] ^ C[n-1];
    N = S[n-1];
    Z = !S;
  end

  endmodule
```

Figure 3.47. Generic Verilog code for the comparator circuit.