Chapter 3

Number Representation and Arithmetic Circuits

Binary numbers

Unsigned numbers

- all bits represent the magnitude of a positive integer

Signed numbers

- left-most bit represents the sign of a number

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

Table 3.1. Numbers in different systems.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.1. Half-adder.

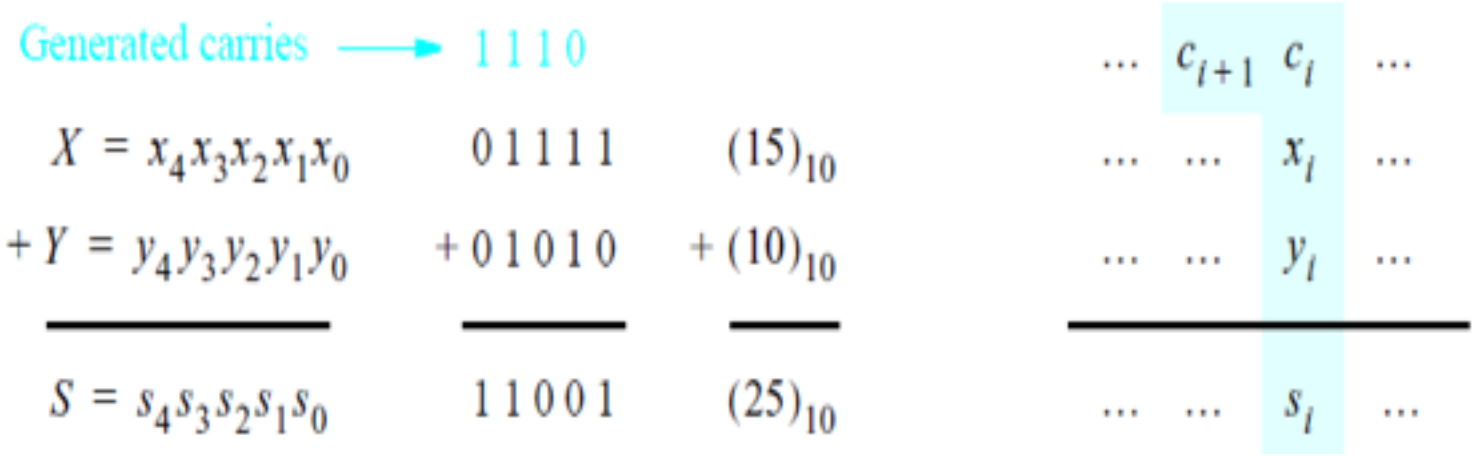


Figure 3.2. Addition of multibit numbers

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.3. Full-adder.

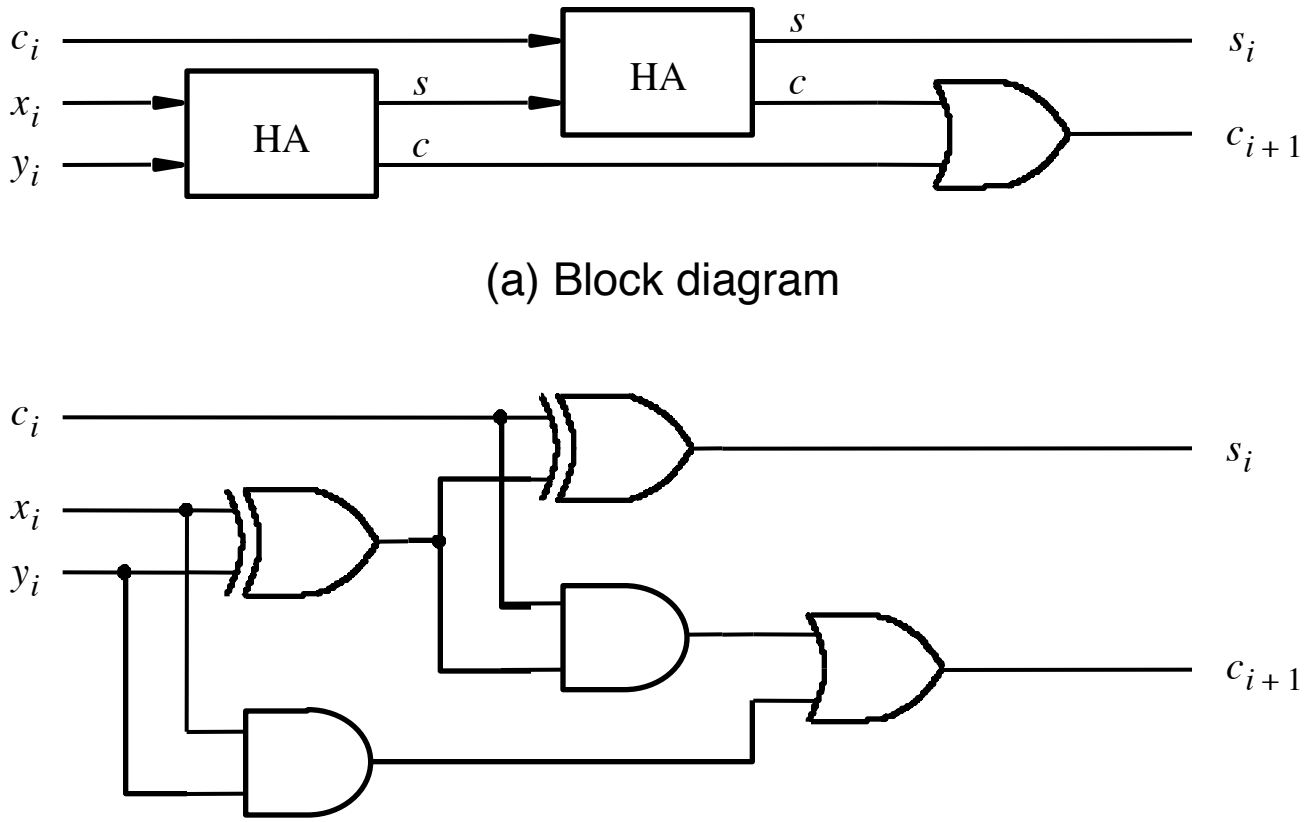


Figure 3.4. A decomposed implementation of the full-adder circuit.

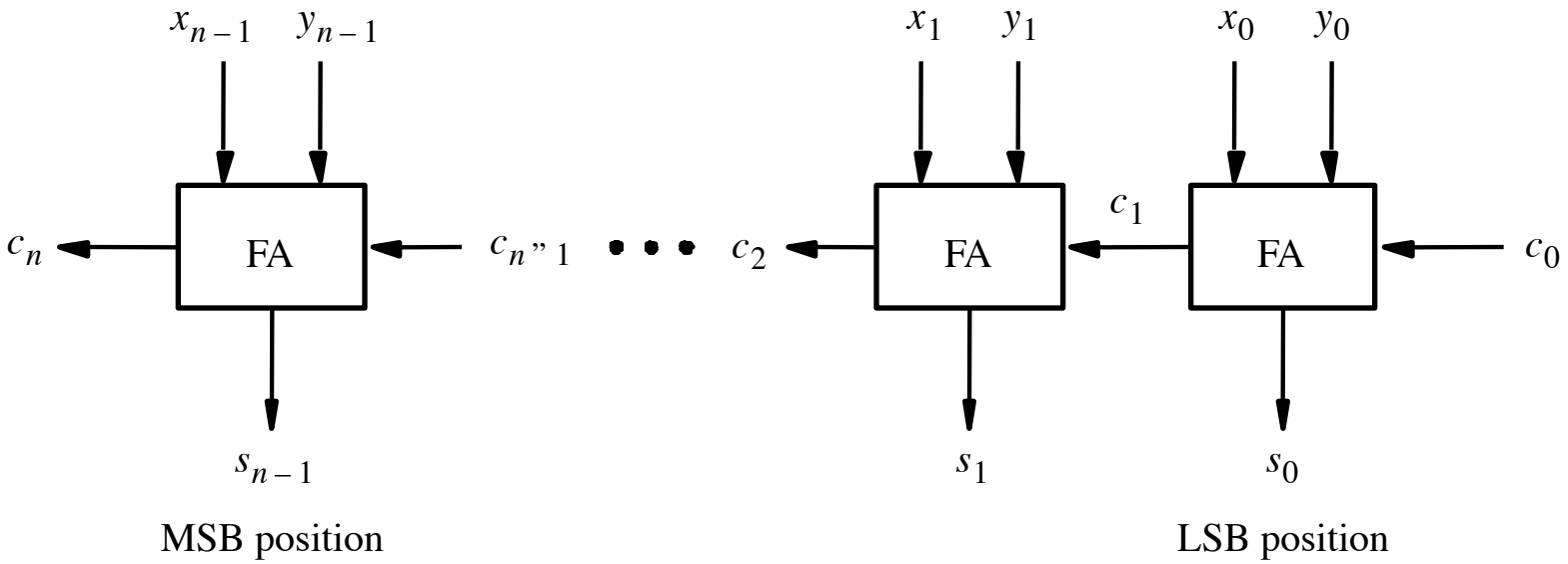


Figure 3.5. An *n*-bit ripple-carry adder.

Please see “portrait orientation” PowerPoint file for Chapter 3

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

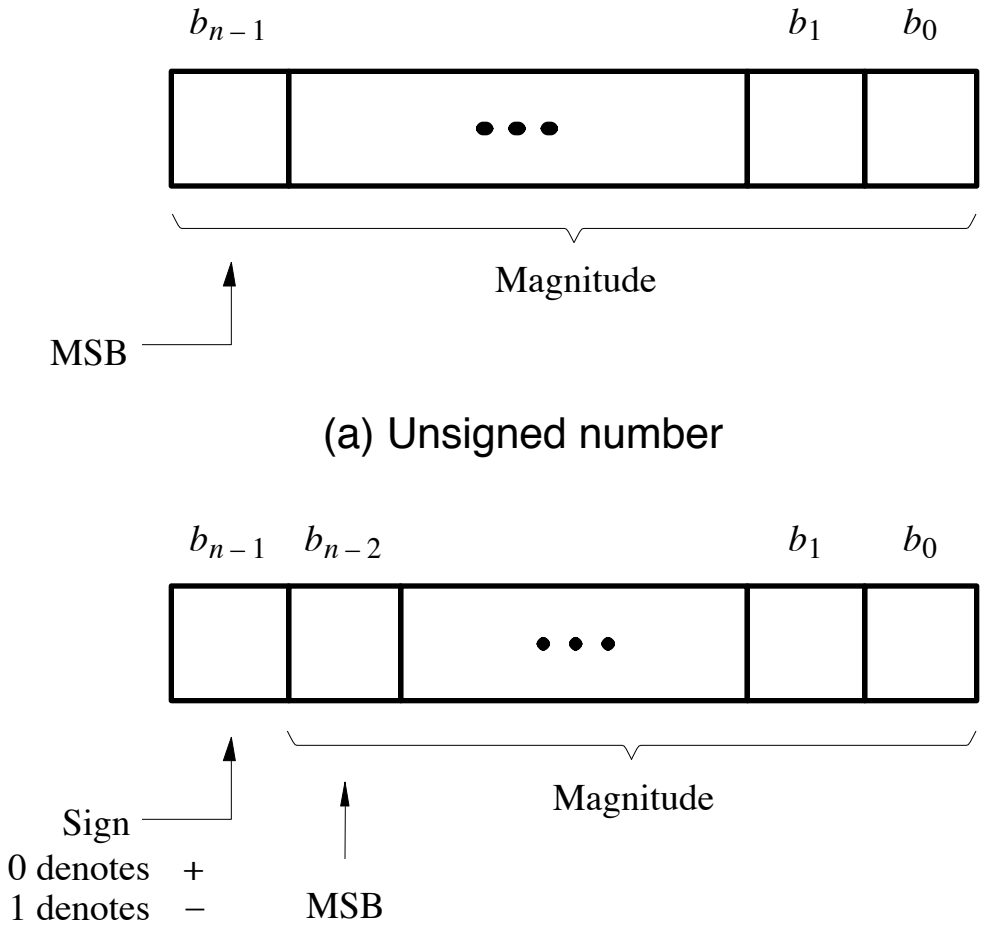


Figure 3.7. Formats for representation of integers.

Negative numbers can be represented in following ways:

- Sign and magnitude
- 1’s complement
- 2’s complement

1’s complement

Let K be the negative equivalent of an n-bit positive number P.

Then, in 1’s complement representation K is obtained by subtracting P from $2^n - 1$, namely

$$K = (2^n - 1) - P$$

This means that K can be obtained by inverting all bits of P.

2’s complement

Let K be the negative equivalent of an n-bit positive number P.

Then, in 2’s complement representation K is obtained by subtracting P from 2^n , namely

$$K = 2^n - P$$

Deriving 2's complement

For a positive n-bit number P, let K_1 and K_2 denote its 1's and 2's complements, respectively.

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

Since $K_2 = K_1 + 1$, it is evident that in a logic circuit the 2's complement can computed by inverting all bits of P and then adding 1 to the resulting 1's-complement number.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

Table 3.2. Interpretation of four-bit signed integers.

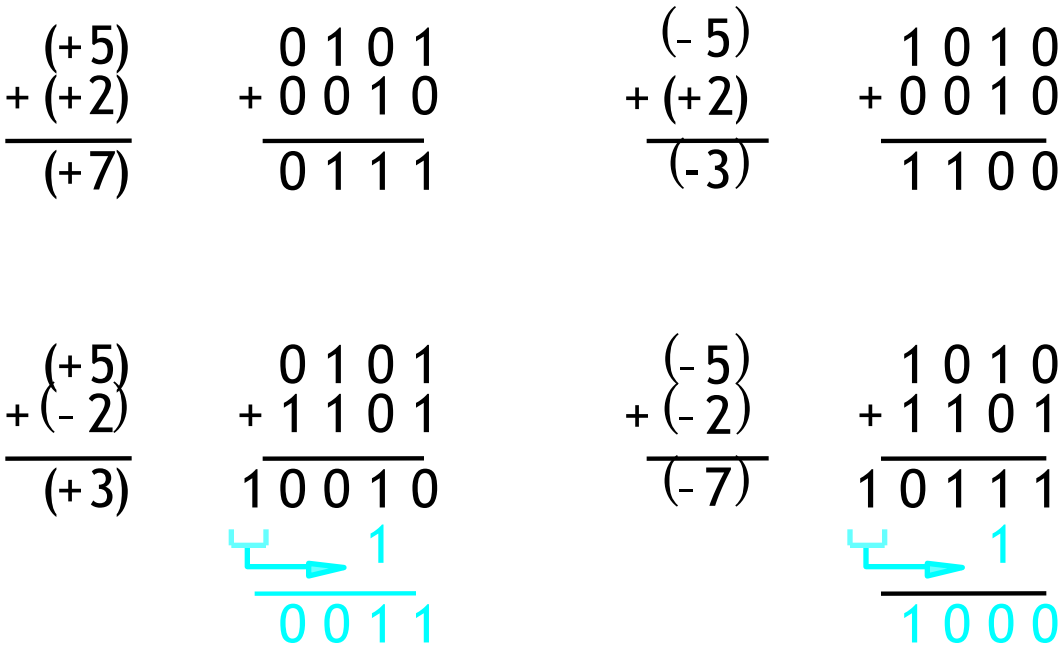


Figure 3.8. Examples of 1's complement addition.

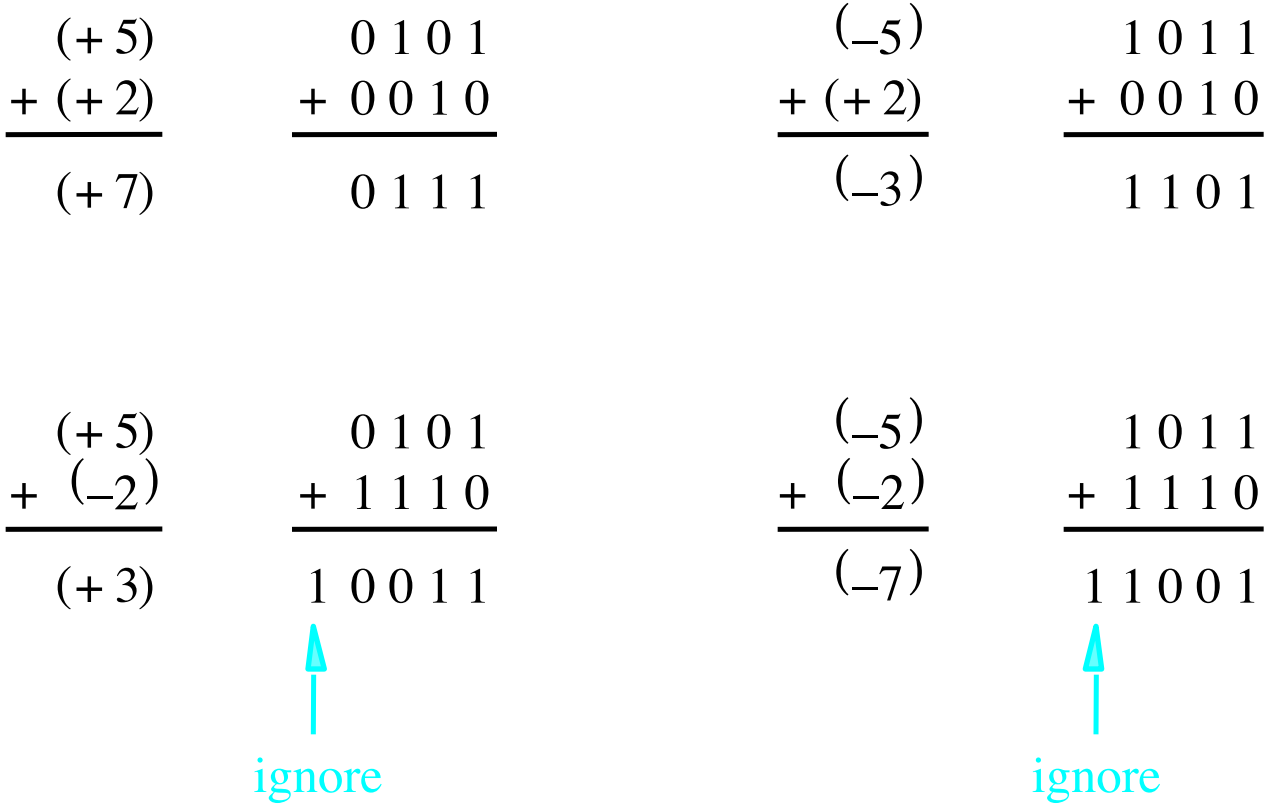


Figure 3.9. Examples of 2's complement addition.

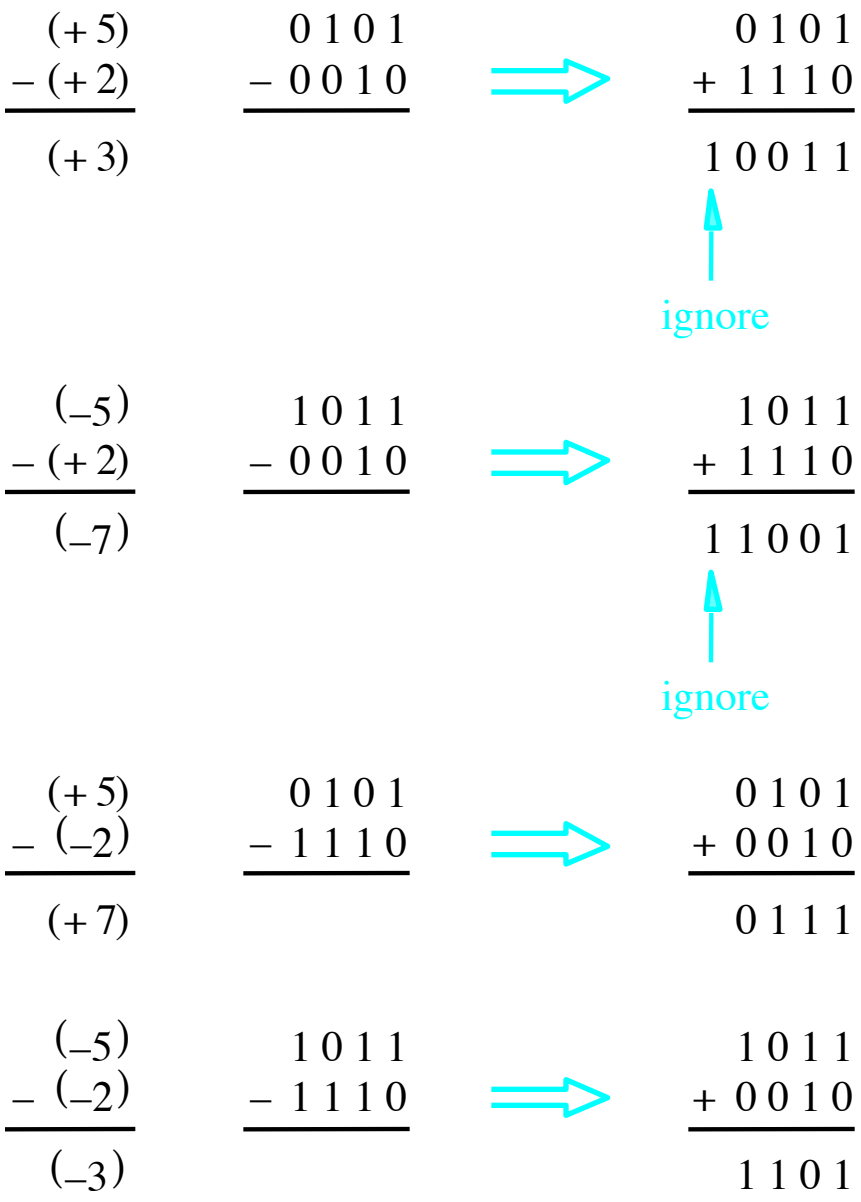
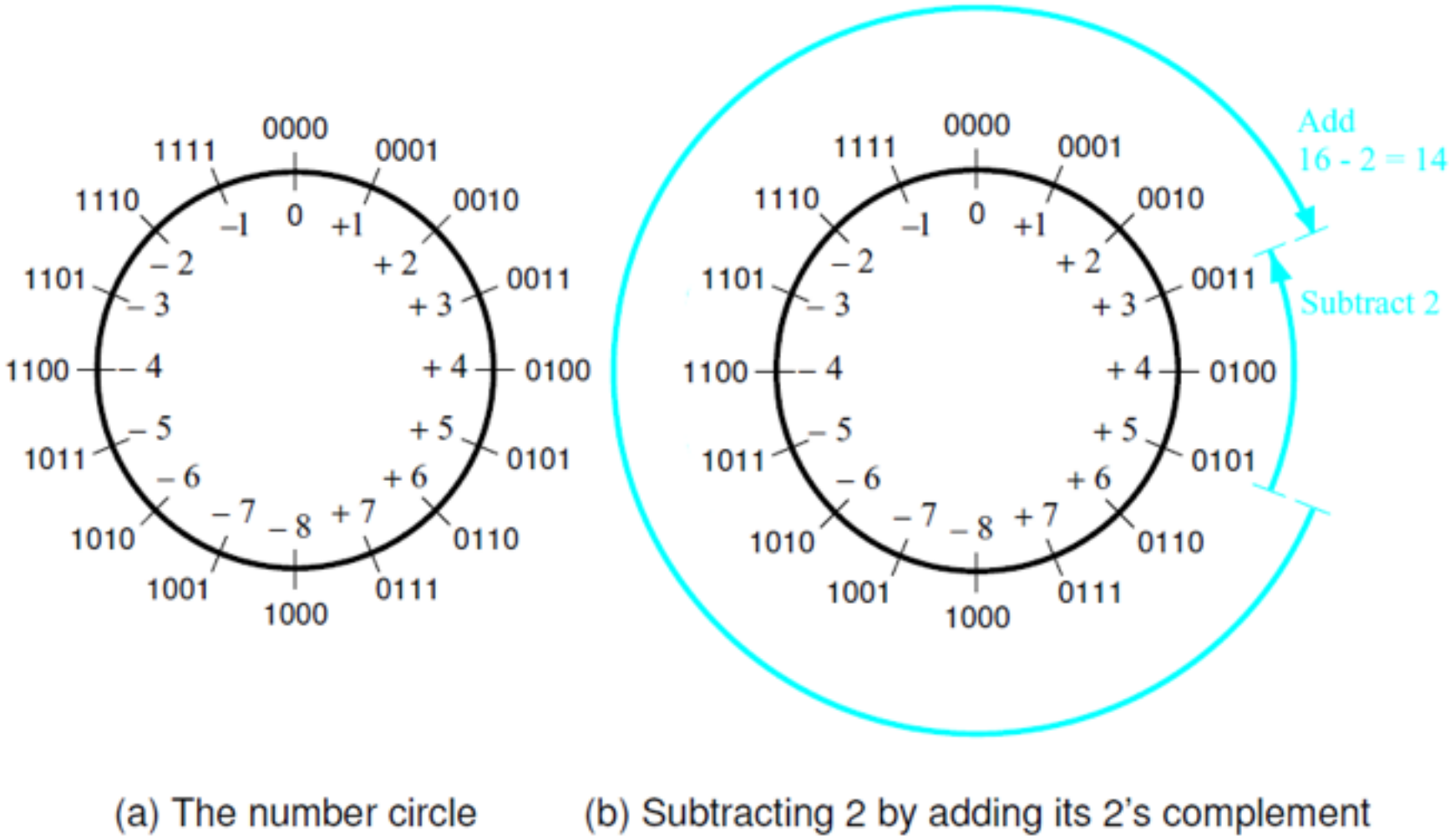


Figure 3.10. Examples of 2's complement subtraction.

Graphical interpretation of four-bit 2's complement numbers



(a) The number circle

(b) Subtracting 2 by adding its 2's complement

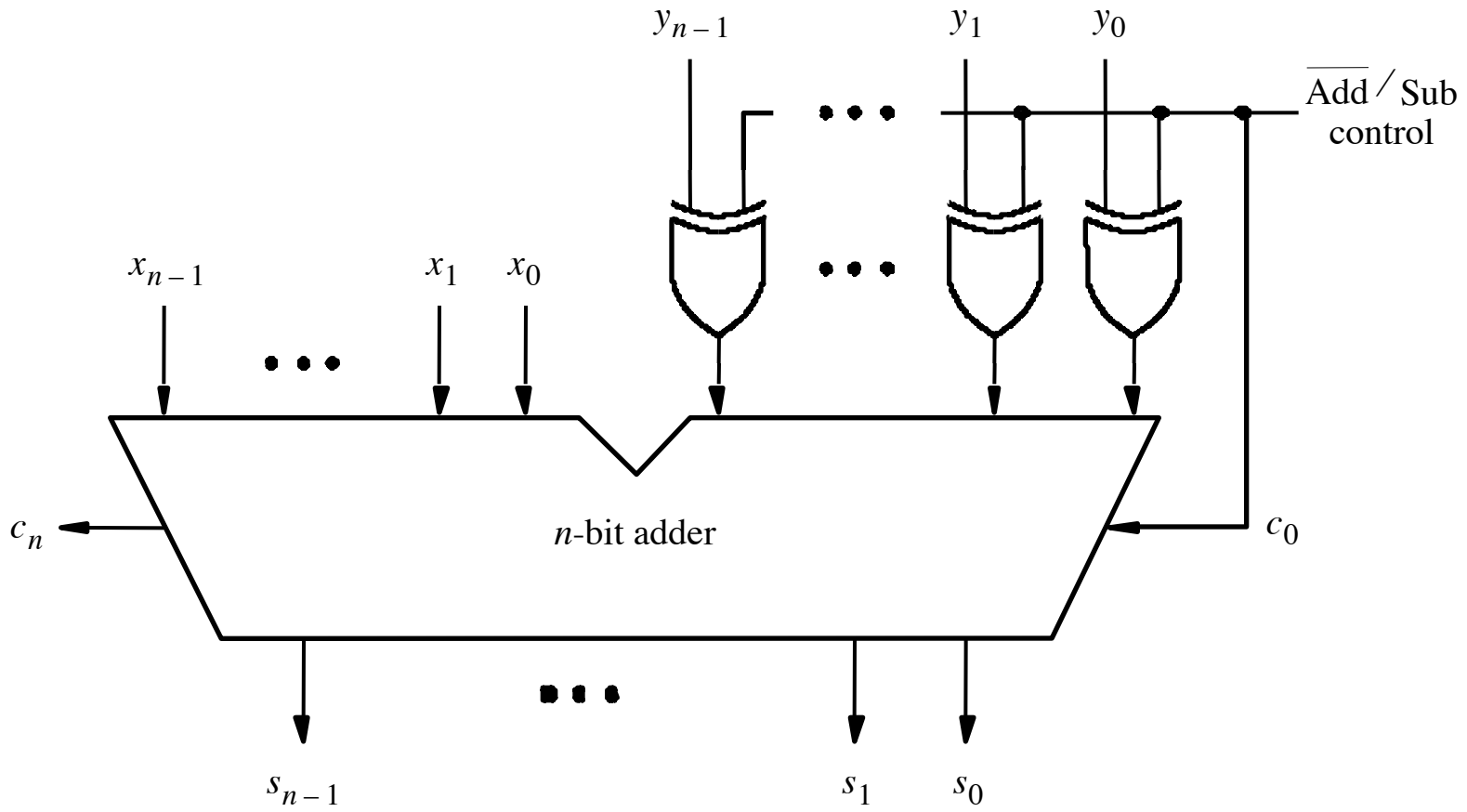


Figure 3.12. Adder/subtractor unit.

$(+7)$	$0\ 1\ 1\ 1$	(-7)	$1\ 0\ 0\ 1$
$+\ (+2)$	$+\ 0\ 0\ 1\ 0$	$+\ (+2)$	$+\ 0\ 0\ 1\ 0$
$(+9)$	$1\ 0\ 0\ 1$	(-5)	$1\ 0\ 1\ 1$
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
$(+7)$	$0\ 1\ 1\ 1$	(-7)	$1\ 0\ 0\ 1$
$+\ (-2)$	$+\ 1\ 1\ 1\ 0$	$+\ (-2)$	$+\ 1\ 1\ 1\ 0$
$(+5)$	$1\ 0\ 1\ 0\ 1$	(-9)	$1\ 0\ 1\ 1\ 1$
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figure 3.13. Examples of determination of overflow.

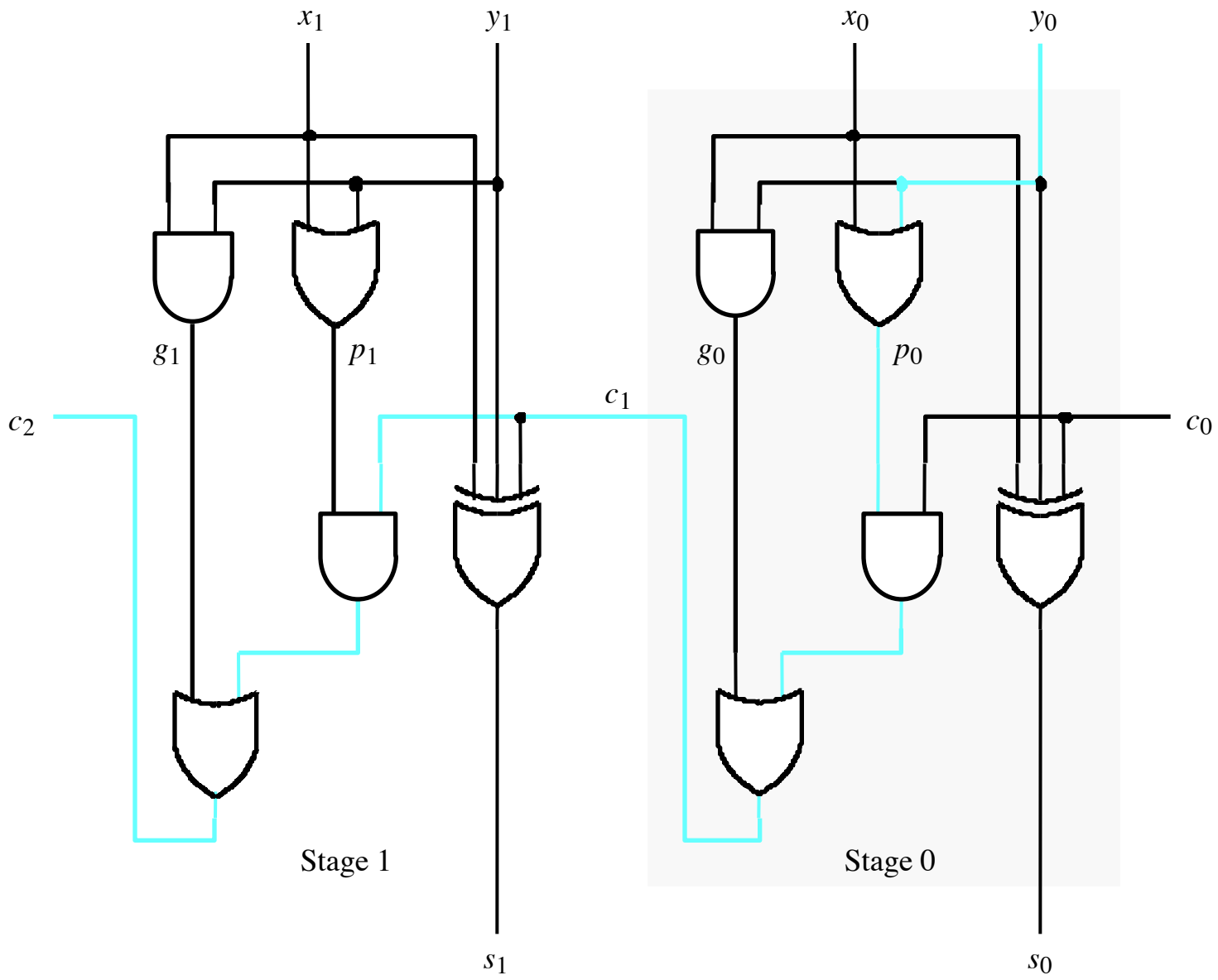


Figure 3.14. A ripple-carry adder based on Expression 3.3.

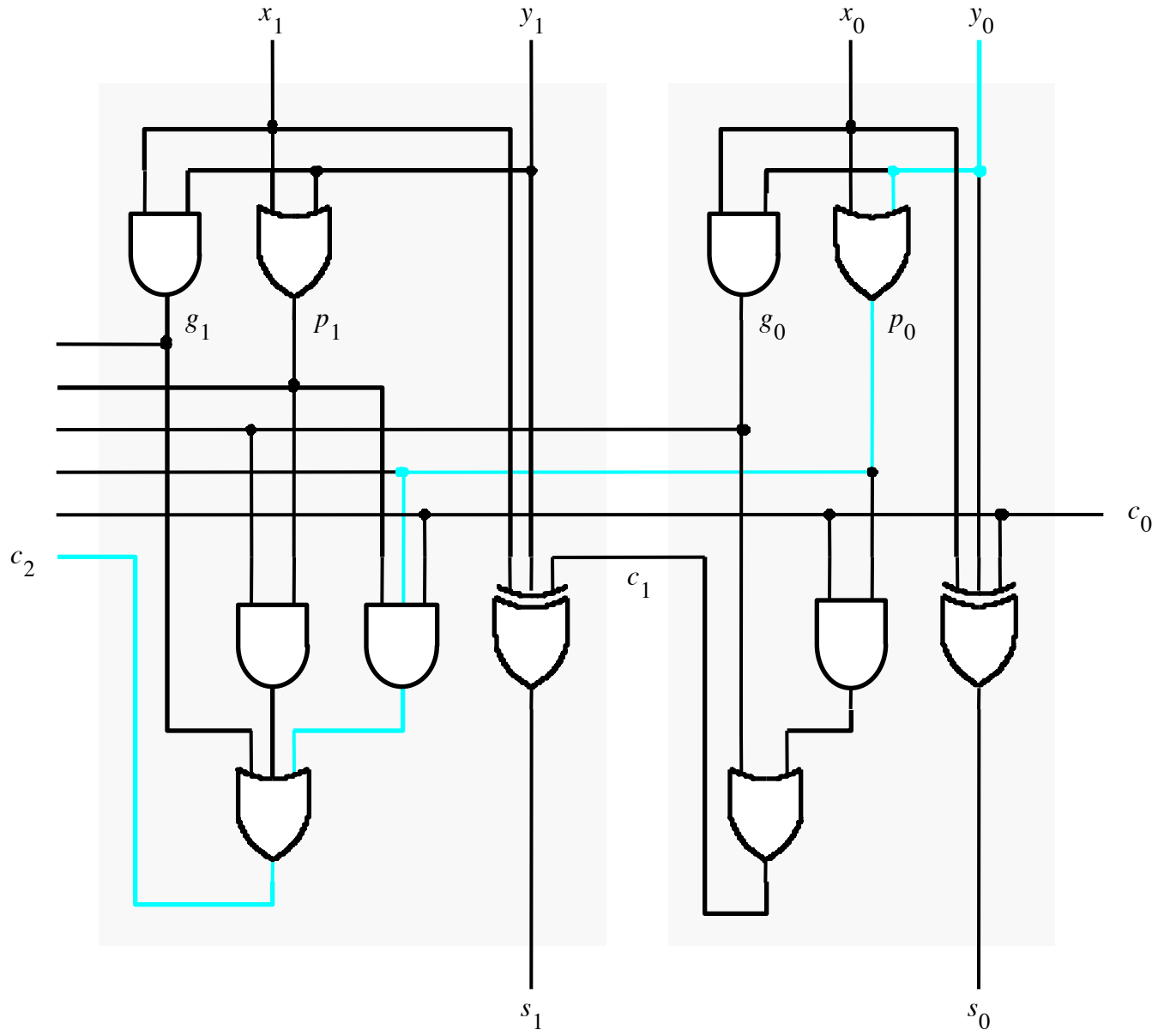


Figure 3.15. The first two stages of a carry-lookahead adder.

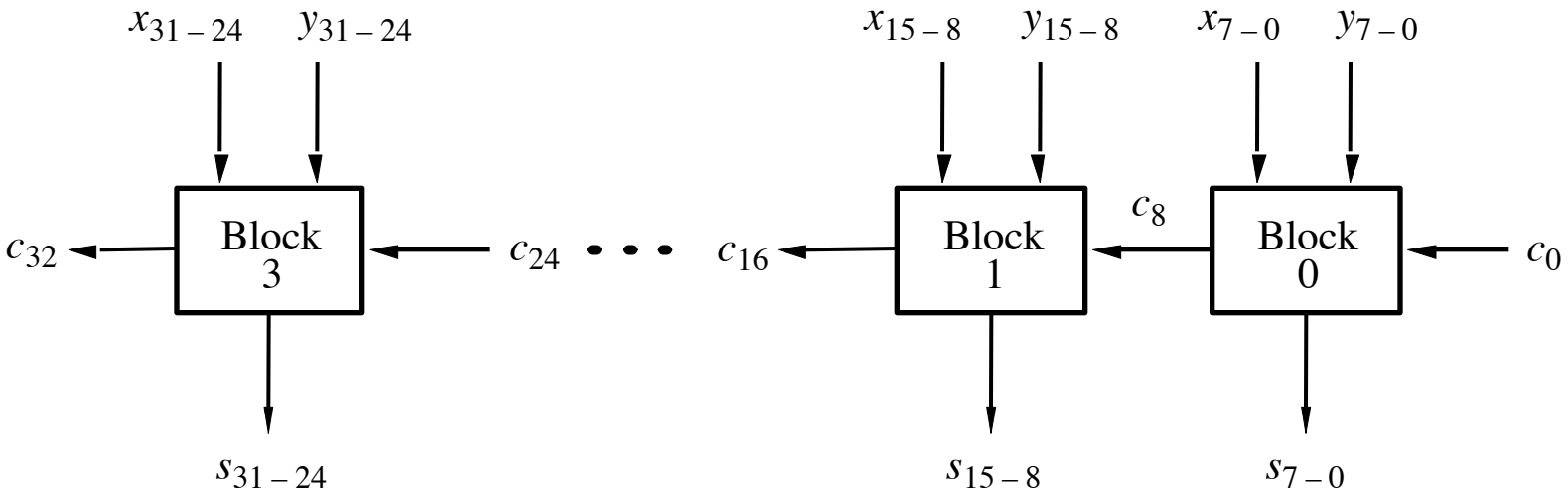


Figure 3.16. A hierarchical carry-lookahead adder with ripple-carry between blocks.

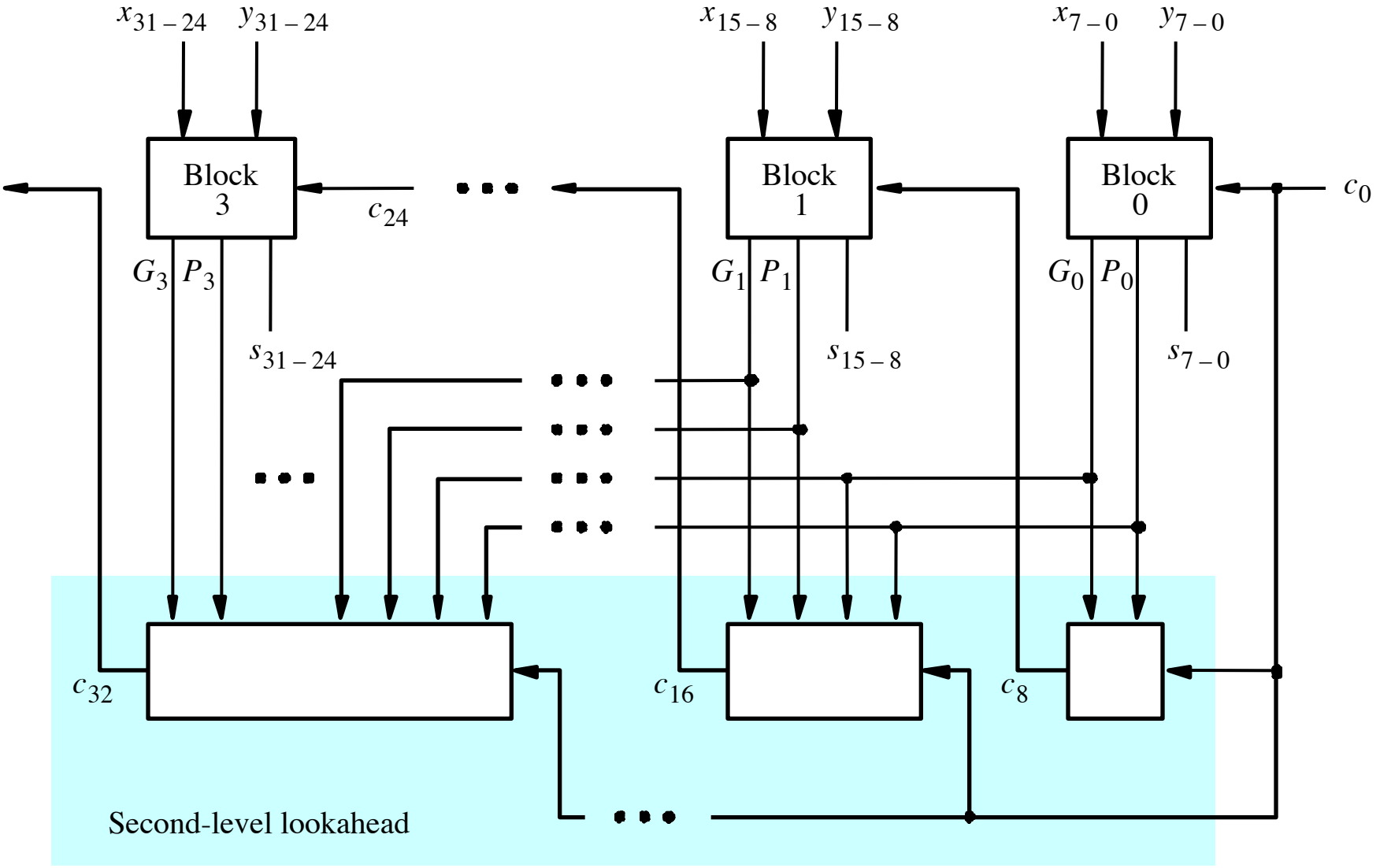


Figure 3.17. A hierarchical carry-lookahead adder.

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

     xor (s, x, y, Cin);
     and (z1, x, y);
     and (z2, x, Cin);
     and (z3, y, Cin);
     or (Cout, z1, z2, z3);

endmodule
```

Figure 3.18. Verilog code for the full-adder using gate level primitives.

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    xor (s, x, y, Cin);
    and (z1, x, y),
            (z2, x, Cin),
            (z3, y, Cin);
    or (Cout, z1, z2, z3);

endmodule
```

Figure 3.19. Another version of Verilog code from Figure 3.18.

```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.20. Verilog code for the full-adder using continuous assignment.

```
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 3.21. Another version of Verilog code from Figure 3.20.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;

    fulladd stage0 (carryin, x0, y0, s0, c1);
    fulladd stage1 (c1, x1, y1, s1, c2);
    fulladd stage2 (c2, x2, y2, s2, c3);
    fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin,
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 3.22. Verilog code for a four-bit adder.

```
module adder4 (carryin, X, Y, S, carryout);
    input carryin;
    input [3:0] X, Y;
    output [3:0] S;
    output carryout;
    wire [3:1] C;

    fulladd stage0 (carryin, 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], carryout);

endmodule
```

Figure 3.23. A four-bit adder using vectors.

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

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      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
      carryout = C[n];
   end

endmodule
```

Figure 3.24. A generic specification of a ripple-carry adder.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.25. A ripple-carry adder specified by using the **generate** statement.

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

    always @(X, Y, carryin)
            S = X + Y + carryin;

endmodule
```

Figure 3.26. Specification of an *n*-bit adder using arithmetic assignment.

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

    always @(X, Y, carryin)
    begin
        S = X + Y + carryin;
        carryout = (X[n-1] & Y[n-1]) | (X[n-1] & ~S[n-1]) | (Y[n-1] & ~S[n-1]);
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
    end

endmodule
```

Figure 3.27. An *n*-bit adder with carry-out and overflow signals.

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

    always @(X, Y, carryin)
    begin
        Sum = {1'b0,X} + {1'b0,Y} + carryin;
        S = Sum[n-1:0];
        carryout = Sum[n];
        overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] & S[n-1]);
    end

endmodule
```

Figure 3.28. An alternative specification of an *n*-bit adder with carry-out and overflow signals.

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

    always @(X, Y, carryin)
    begin
        {carryout, S} = X + Y + carryin;
         overflow = (X[n-1] & Y[n-1] & ~S[n-1]) | (~X[n-1] & ~Y[n-1] &
S[n-1]);
    end

endmodule
```

Figure 3.29. Simplified complete specification of an *n*-bit adder.

```
module fulladd (Cin, x, y, s, Cout);
      input Cin, x, y;
      output reg s, Cout;

      always @(x, y, Cin)
        {Cout, s} = x + y + Cin;

endmodule
```

Figure 3.30. Behavioral specification of a full-adder.

```
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 U1 (1'b0, A, B, S[15:0], S[16], o1);
    defparam U1.n = 16;
    addern U2 (1'b0, C, D, T[7:0], T[8], o2);
    defparam U2.n = 8;

    assign overflow = o1 | o2;

endmodule
```

Figure 3.31. An example of setting parameter values in Verilog code.

```
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  #(16) U1 (1'b0, A, B, S[15:0], S[16], o1);
    addern  #(8) U2 (1'b0, C, D, T[7:0], T[8], o2);

    assign overflow = o1 | o2;

endmodule
```

Figure 3.32. Using the Verilog # operator to set the values of parameters.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.33. A ripple-carry adder specified by using the **generate** statement.

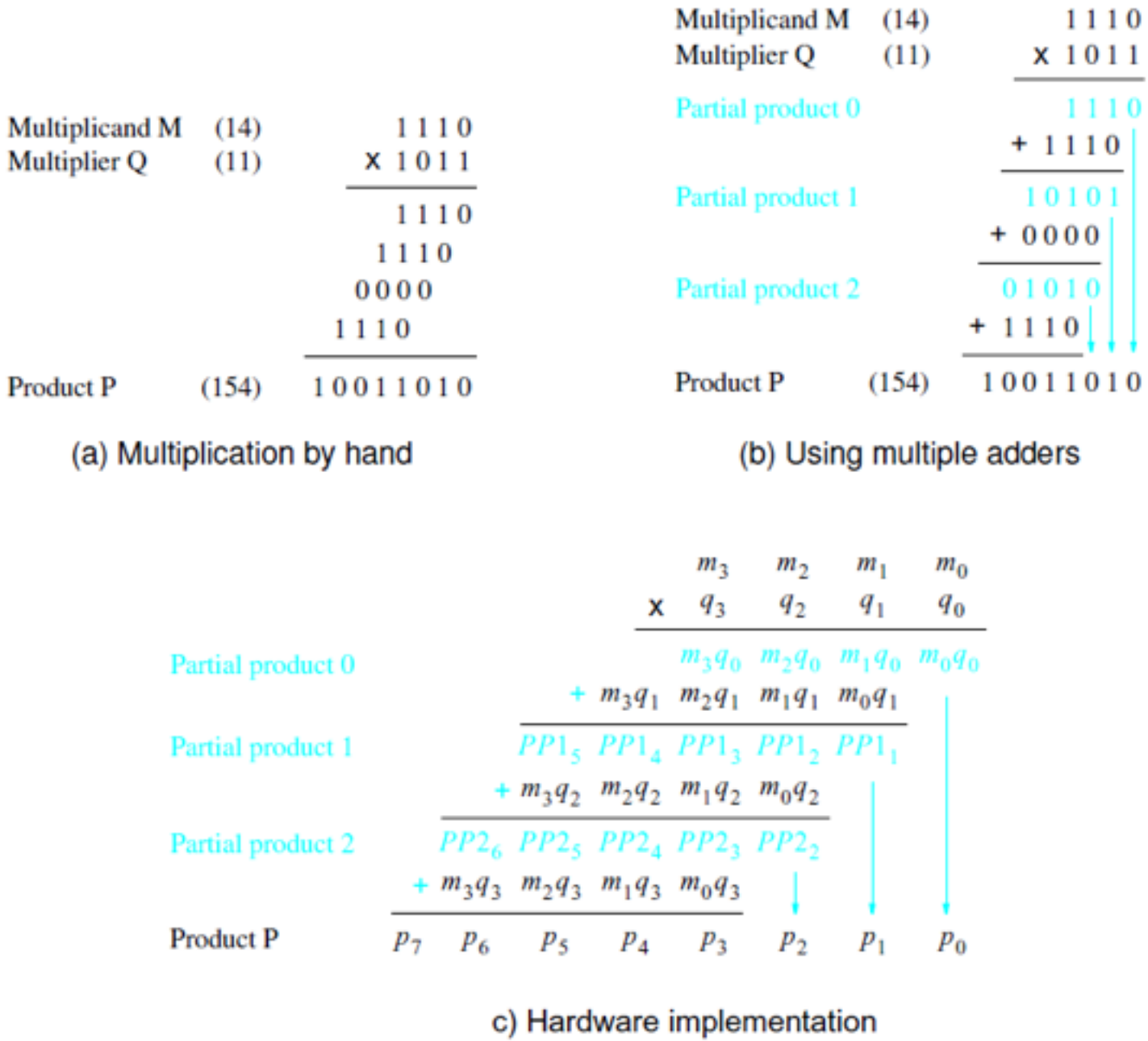


Figure 3.34. Multiplication of unsigned

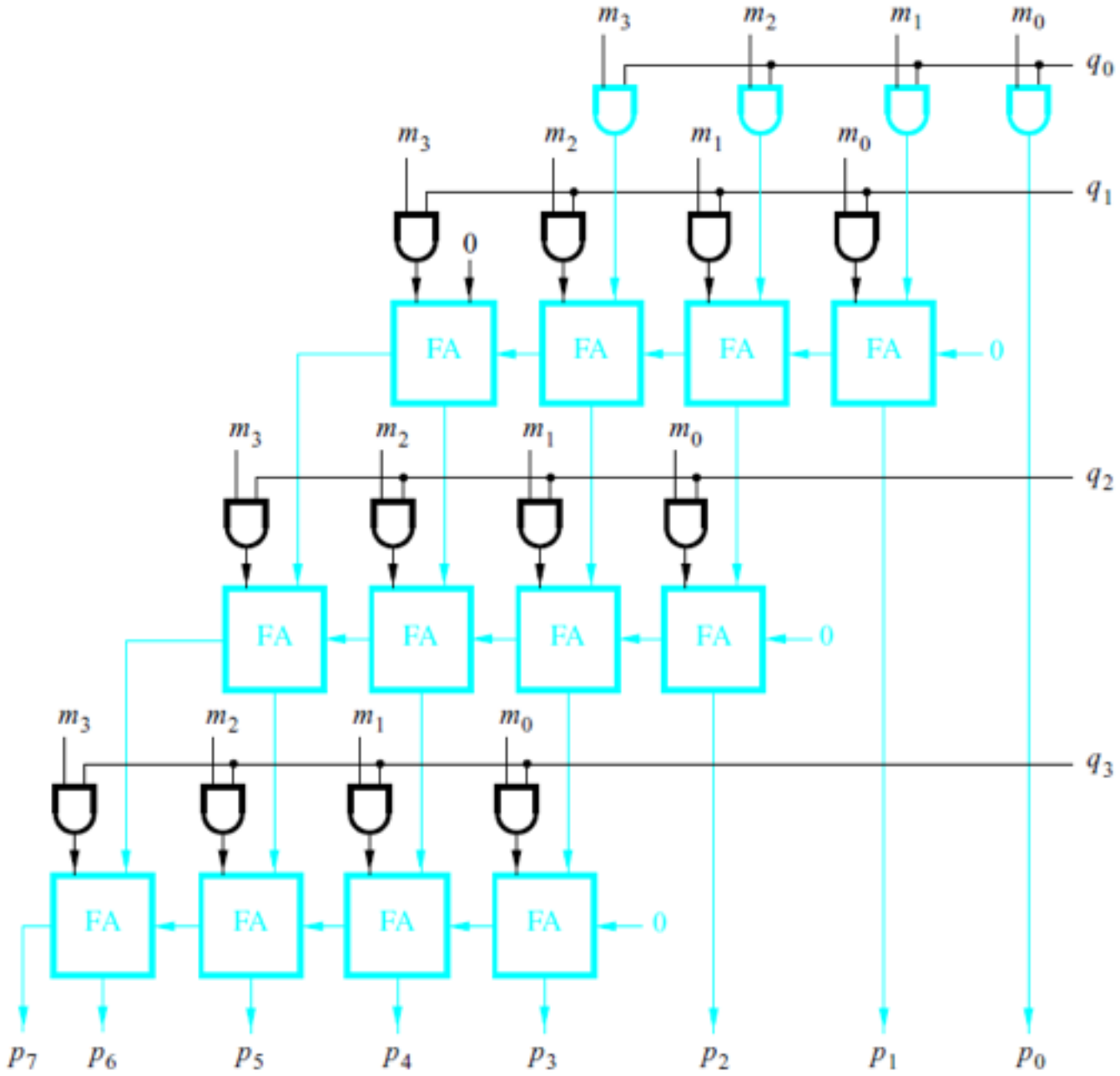


Figure 3.35. A 4x4 multiplier circuit.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.36. Multiplication of signed numbers.

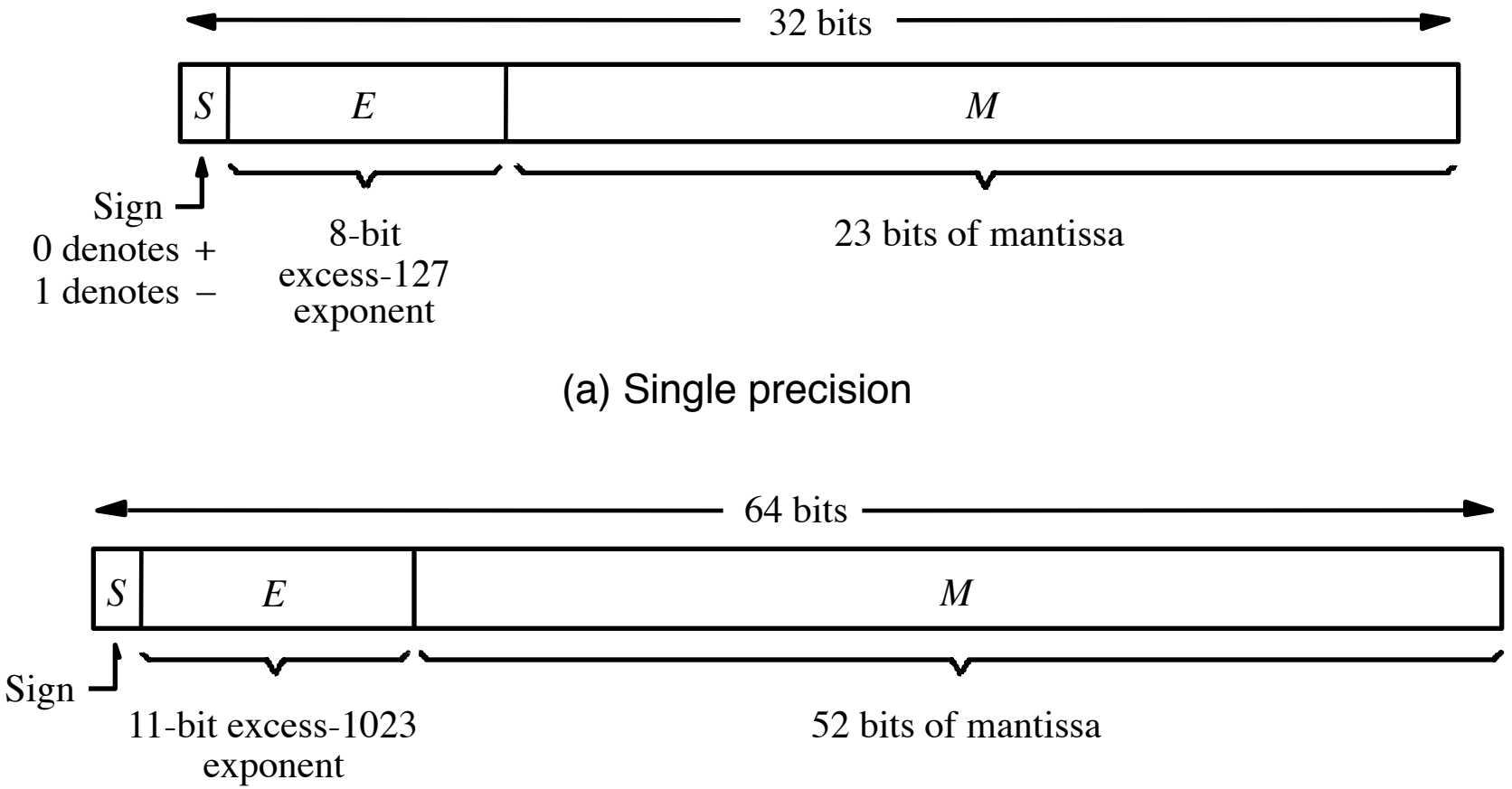


Figure 3.37. IEEE Standard floating-point formats.

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3.3. Binary-coded decimal digits.

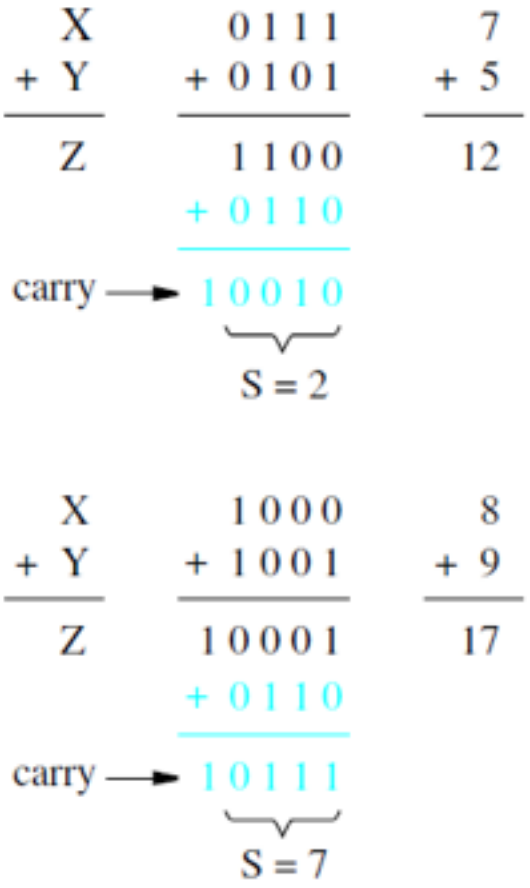


Figure 3.38. Addition of BCD digits.

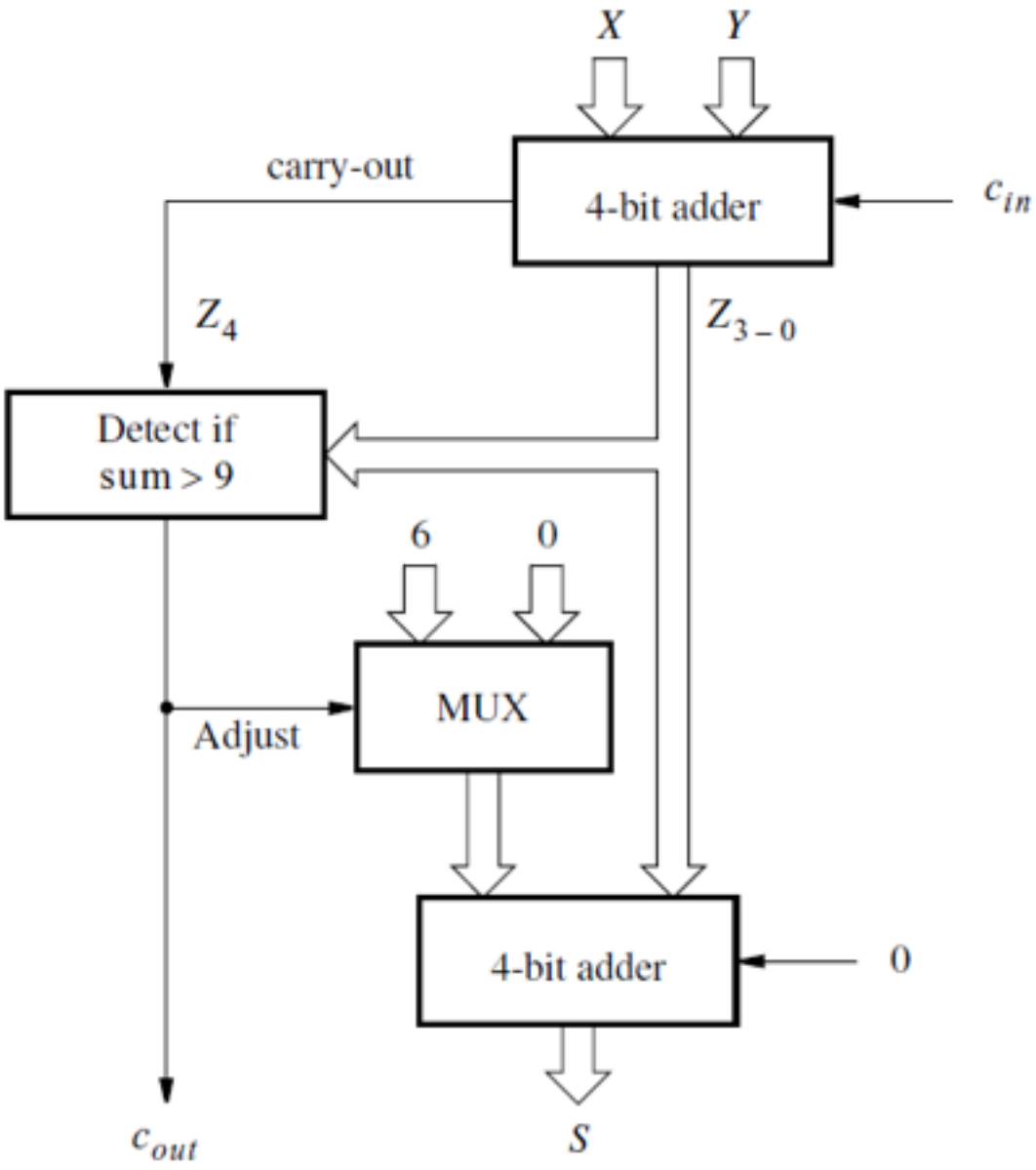


Figure 3.39. Block diagram for a one-digit BCD adder.

```
module bcdadd(Cin, X, Y, S, Cout);
    input Cin;
    input [3:0] X,Y;
    output reg [3:0] S;
    output reg Cout;
    reg [4:0] Z;

    always @ (X, Y, Cin)
    begin
        Z = X + Y + Cin;
        if (Z < 10)
            {Cout,S} = Z;
        else
            {Cout, S} = Z + 6;
    end

endmodule
```

Figure 3.40. Verilog code for a one-digit BCD adder.

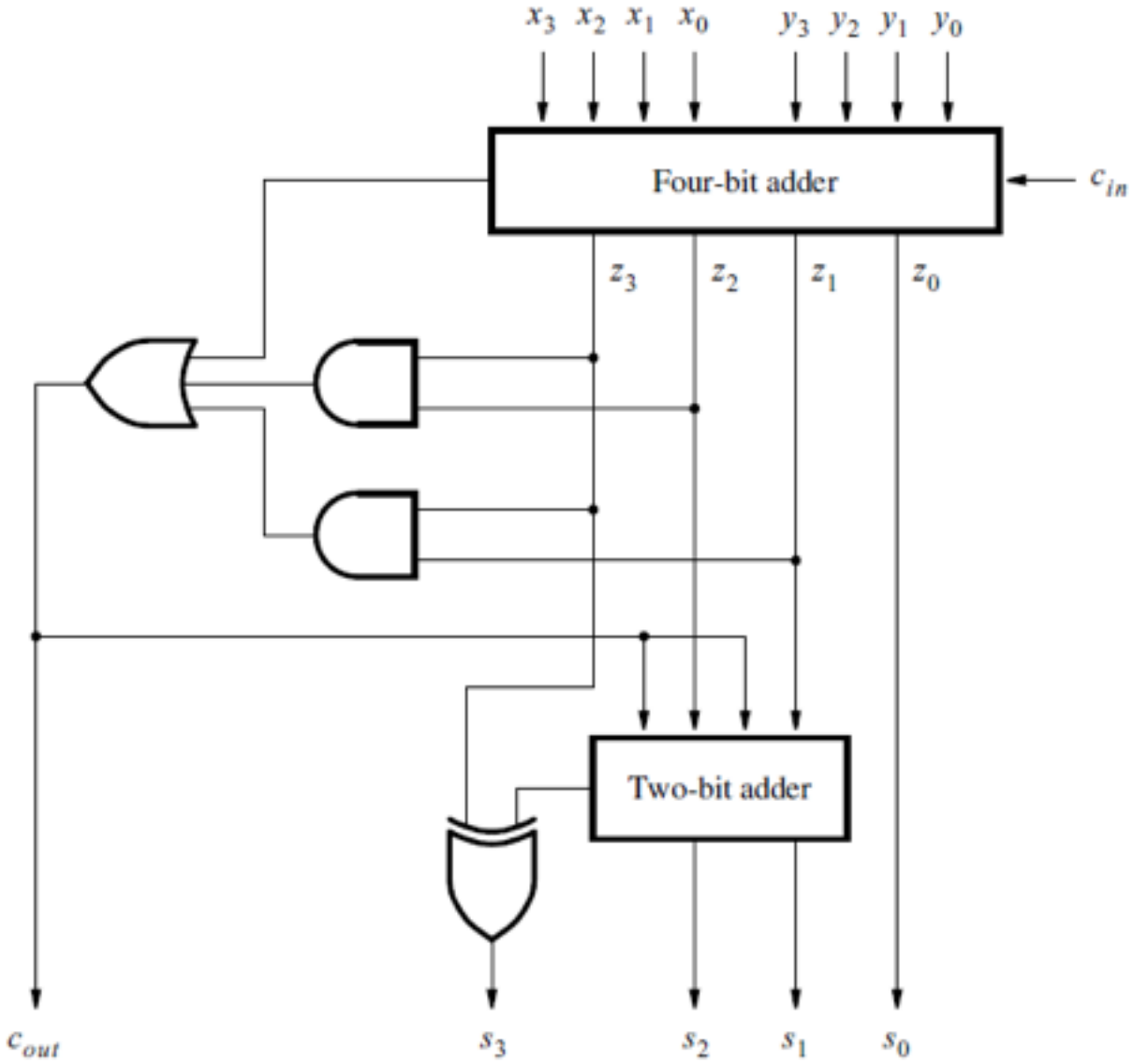


Figure 3.41. Circuit for a one-digit BCD adder.

Convert $(14959)_{10}$

		Remainder	Hex digit		
$14959 \div 16 =$	934		15	F	LSB
$934 \div 16 =$	58	6	6		
$58 \div 16 =$	3	10	A		
$3 \div 16 =$	0	3	3		MSB

Result is $(3A6F)_{16}$

Figure 3.42. Conversion from decimal to hexadecimal.

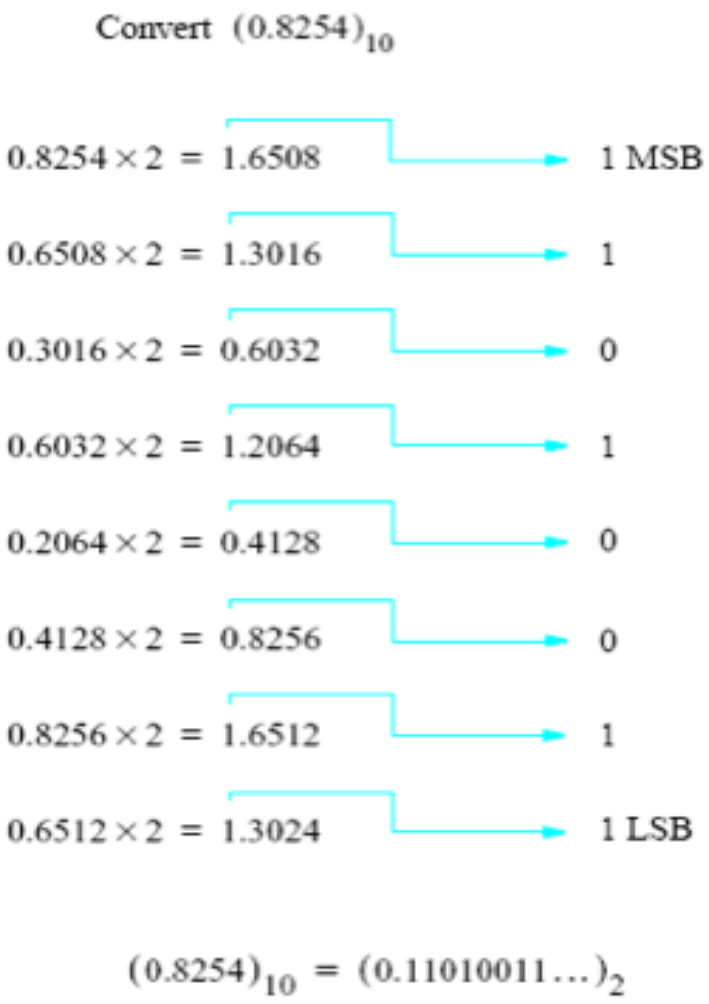


Figure 3.43. Conversion of fractions from decimal to binary.

Please see “portrait orientation” PowerPoint file for Chapter 3

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

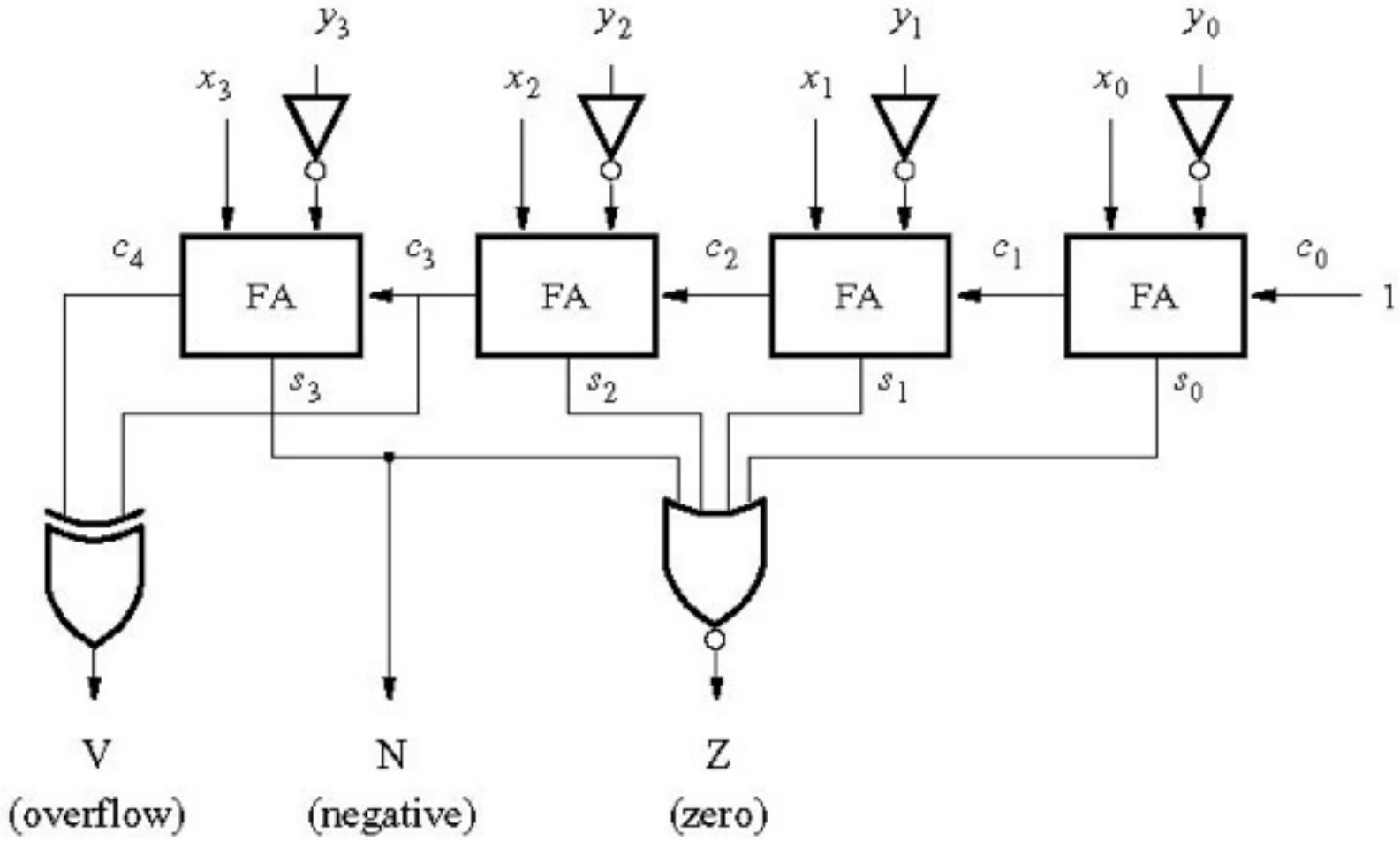


Figure 3.45. A comparator circuit.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.46. Structural Verilog code for the comparator circuit.

Please see “portrait orientation” PowerPoint file for Chapter 3

Figure 3.47. Generic Verilog code for the comparator circuit.

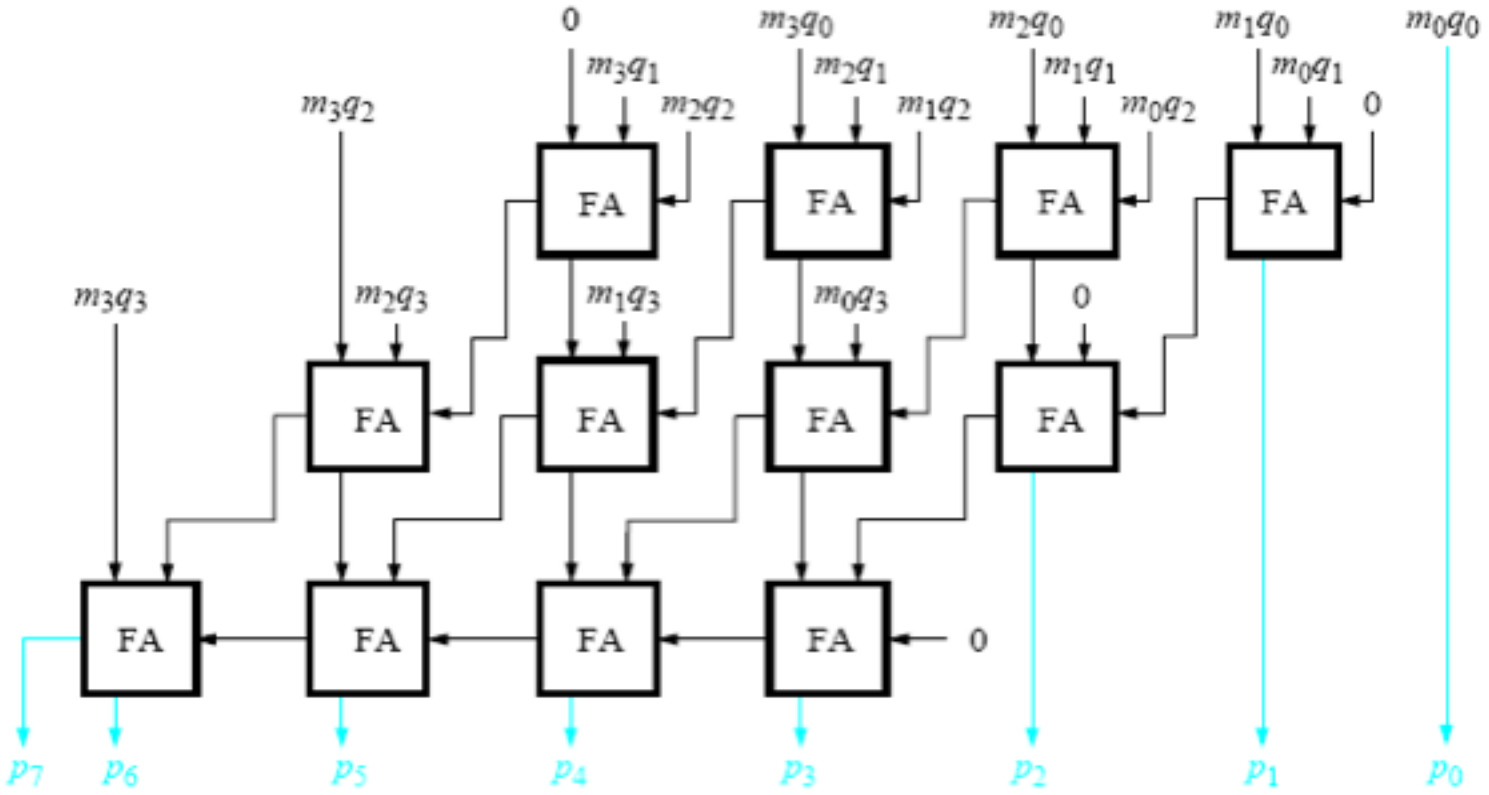


Figure 3.48. Multiplier carry-save array.

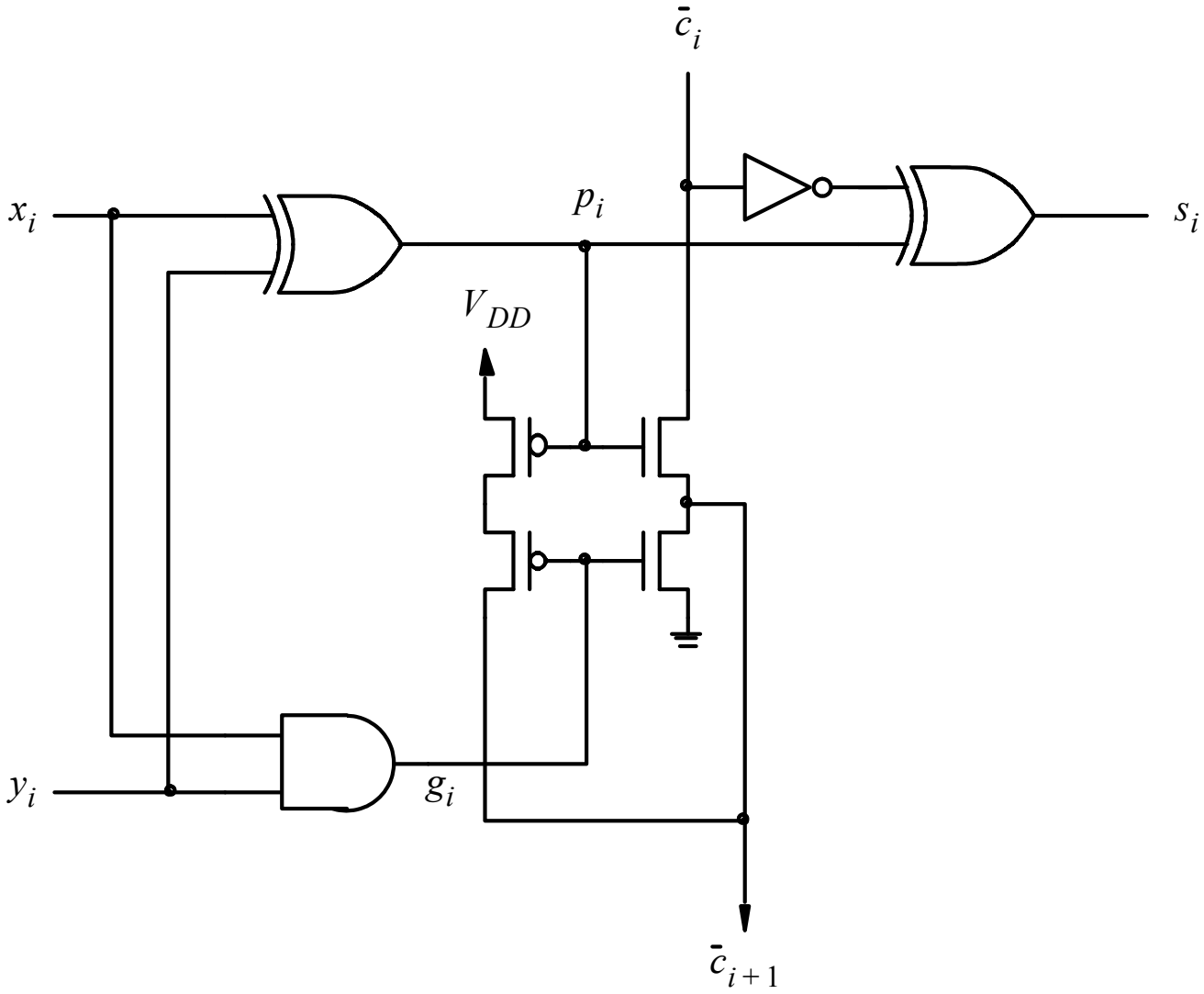


Figure P3.1. Circuit for Problem 3.11.

```
module problem5_17 (IN, OUT);
    input [3:0] IN;
    output reg [3:0] OUT;

    always @(IN)
        if (IN == 4'b0101)  OUT = 4'b0001;
        else if (IN == 4'b0110)  OUT = 4'b0010;
        else if (IN == 4'b0111)  OUT = 4'b0011;
        else if (IN == 4'b1001)  OUT = 4'b0010;
        else if (IN == 4'b1010)  OUT = 4'b0100;
        else if (IN == 4'b1011)  OUT = 4'b0110;
        else if (IN == 4'b1101)  OUT = 4'b0011;
        else if (IN == 4'b1110)  OUT = 4'b0110;
        else if (IN == 4'b1111)  OUT = 4'b1001;
        else  OUT = 4'b0000;

endmodule
```

Figure P3.2. The code for Problem 3.17.

$A\ B$	$Carry$	Sum
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figure P3.3. Ternary half-adder.