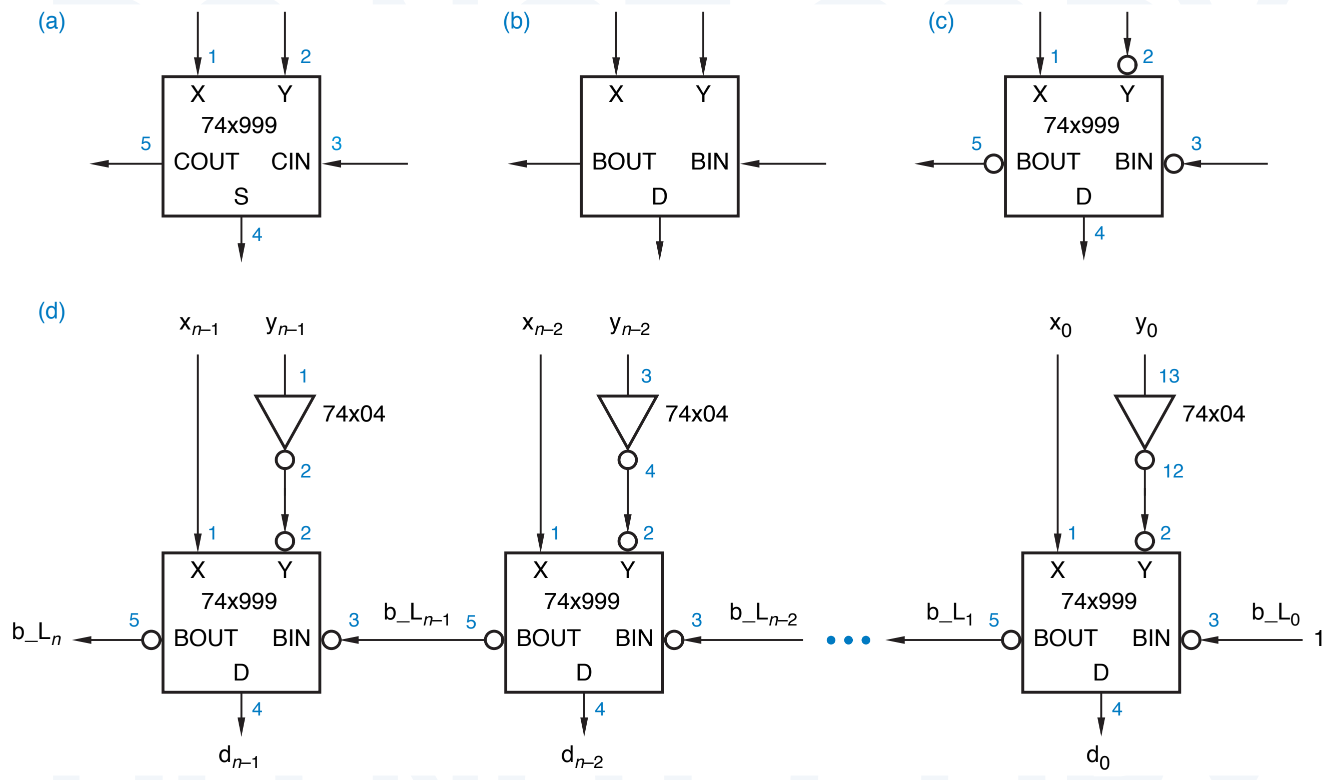


Figure 5-87 Designing subtractors using adders: (a) full adder; (b) full subtractor; (c) interpreting the device in (a) as a full subtractor; (d) ripple subtractor.

*5.10.4 Carry Lookahead Adders

The logic equation for sum bit i of a binary adder can actually be written quite simply:

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

More complexity is introduced when we expand c_i above in terms of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0, and we get a real mess expanding the XORs. However, if we're willing to forego the XOR expansion, we can at least streamline the design of c_i logic using ideas of *carry lookahead* discussed in this subsection.

carry lookahead

Figure 5-88 shows the basic idea. The block labeled "Carry Lookahead Logic" calculates c_i in a fixed, small number of logic levels for any reasonable value of i. Two definitions are the key to carry lookahead logic:

carry generate

- For a particular combination of inputs x_i and y_i, adder stage i is said to *generate* a carry if it produces a carry-out of 1 ($c_{i+1} = 1$) independent of the inputs on $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0.

carry propagate

- For a particular combination of inputs x_i and y_i, adder stage i is said to *propagate* carries if it produces a carry-out of 1 ($c_{i+1} = 1$) in the presence of an input combination of $x_0 - x_{i-1}$, $y_0 - y_{i-1}$, and c_0 that causes a carry-in of 1 ($c_i = 1$).

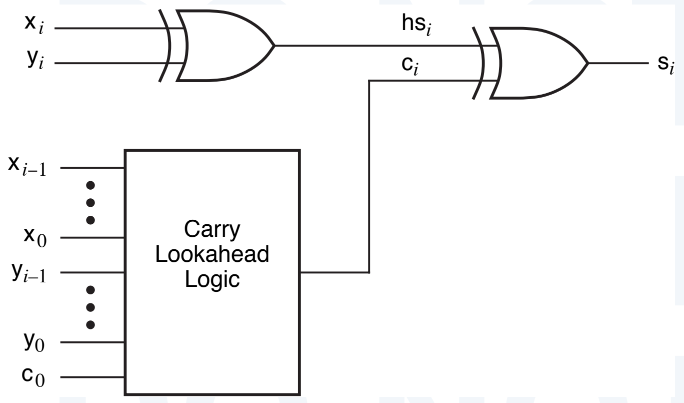


Figure 5-88 Structure of one stage of a carry lookahead adder.

Corresponding to these definitions, we can write logic equations for a carry-generate signal, g_i, and a carry-propagate signal, p_i, for each stage of a carry lookahead adder:

$$g_i = x_i \cdot y_i$$
$$p_i = x_i + y_i$$

That is, a stage unconditionally generates a carry if both of its addend bits are 1, and it propagates carries if at least one of its addend bits is 1. The carry output of a stage can now be written in terms of the generate and propagate signals:

$$c_{i+1} = g_i + p_i \cdot c_i$$

To eliminate carry ripple, we recursively expand the c_i term for each stage, and multiply out to obtain a 2-level AND-OR expression. Using this technique, we can obtain the following carry equations for the first four adder stages:

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \\
&= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}$$

Each equation corresponds to a circuit with just three levels of delay—one for the generate and propagate signals, and two for the sum-of-products shown. A *carry lookahead adder* uses three-level equations such as these in each adder stage for the block labeled "carry lookahead" in Figure 5-88. The sum output for

carry lookahead adder

a stage is produced by combining the carry bit with the two addend bits for the stage as we showed in the figure. In the next subsection, we'll study some commercial MSI adders and ALUs that use carry lookahead.

*5.10.5 MSI Adders

74x283

74x83

The *74x283* is a 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic, using the carry lookahead technique. Figure 5-89 is a logic symbol for the 74x283. The older *74x83* is identical except for its pinout, which has nonstandard locations for power and ground.

The logic diagram for the '283, shown in Figure 5-90, has just a few differences from the general carry-lookahead design that we described in the preceding subsection. First of all, its addends are named A and B instead of X and Y; no big deal. Second, it produces active-low versions of the carry-generate (g_i') and carry-propagate (p_i') signals, since inverting gates are generally faster than noninverting ones. Third, it takes advantage of the fact that we can algebraically manipulate the half-sum equation as follows:

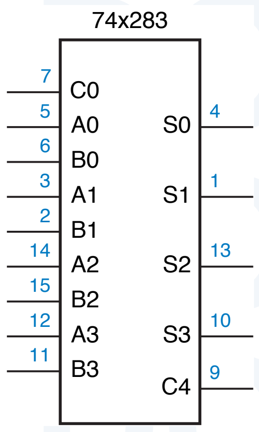


Figure 5-89 Traditional logic symbol for the 74x283 4-bit binary adder.

$$\begin{aligned}
hs_i &= x_i \oplus y_i \\
&= x_i \cdot y_i' + x_i' \cdot y_i \\
&= x_i \cdot y_i' + x_i \cdot x_i' + x_i' \cdot y_i + y_i \cdot y_i' \\
&= (x_i + y_i) \cdot (x_i' + y_i') \\
&= (x_i + y_i) \cdot (x_i \cdot y_i)' \\
&= p_i \cdot g_i'
\end{aligned}$$

Thus, an AND gate with an inverted input can be used instead of an XOR gate to create each half-sum bit.

Finally, the '283 creates the carry signals using an INVERT-OR-AND structure (the DeMorgan equivalent of an AND-OR-INVERT), which has about the same delay as a single CMOS or TTL inverting gate. This requires some explaining, since the carry equations that we derived in the preceding subsection are used in a slightly modified form. In particular, the c_{i+1} equation uses the term $p_i \cdot g_i$ instead of g_i. This has no effect on the output, since p_i is always 1 when g_i is 1. However, it allows the equation to be factored as follows:

$$\begin{aligned}
c_{i+1} &= p_i \cdot g_i + p_i \cdot c_i \\
&= p_i \cdot (g_i + c_i)
\end{aligned}$$

This leads to the following carry equations, which are used by the circuit :

$$\begin{aligned}
c_1 &= p_0 \cdot (g_0 + c_0) \\
c_2 &= p_1 \cdot (g_1 + c_1) \\
&= p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0)) \\
&= p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)
\end{aligned}$$

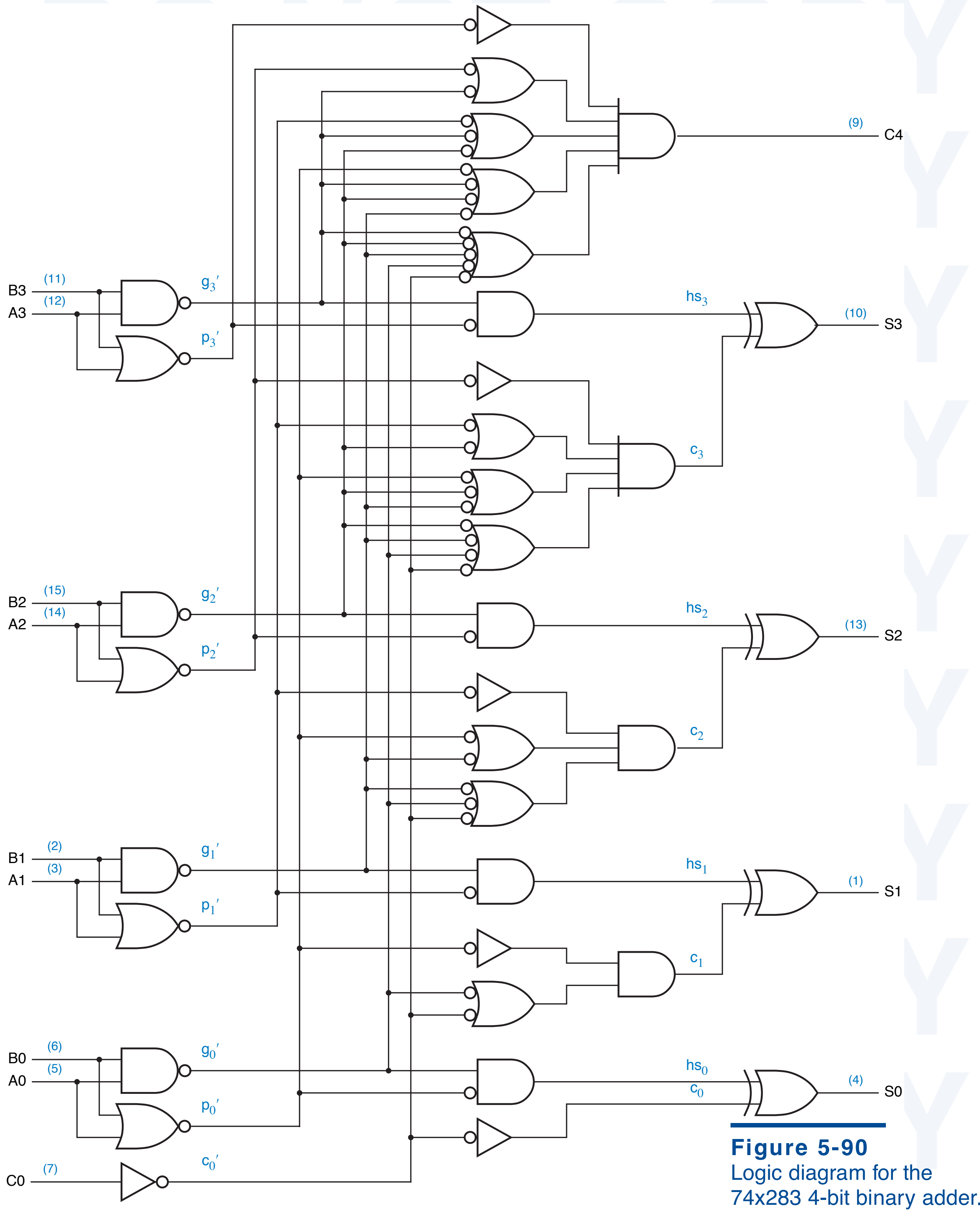


Figure 5-90
Logic diagram for the 74x283 4-bit binary adder.

$$
\begin{aligned}
c_3 &= p_2 \cdot (g_2 + c_2)\\
&= p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0))\\
&= p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)\\
c_4 &= p_3 \cdot (g_3 + c_3)\\
&= p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0))\\
&= p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \cdot (g_3 + g_2 + g_1 + g_0 + c_0)
\end{aligned}
$$

If you've followed the derivation of these equations and can obtain the same ones by reading the '283 logic diagram, then congratulations, you're up to speed on switching algebra! If not, you may want to review Sections 4.1 and 4.2.

group-ripple adder

The propagation delay from the C0 input to the C4 output of the '283 is very short, about the same as two inverting gates. As a result, fairly fast *group-ripple adders* with more than four bits can be made simply by cascading the carry outputs and inputs of '283s, as shown in Figure 5-91 for a 16-bit adder. The total propagation delay from C0 to C16 in this circuit is about the same as that of eight inverting gates.

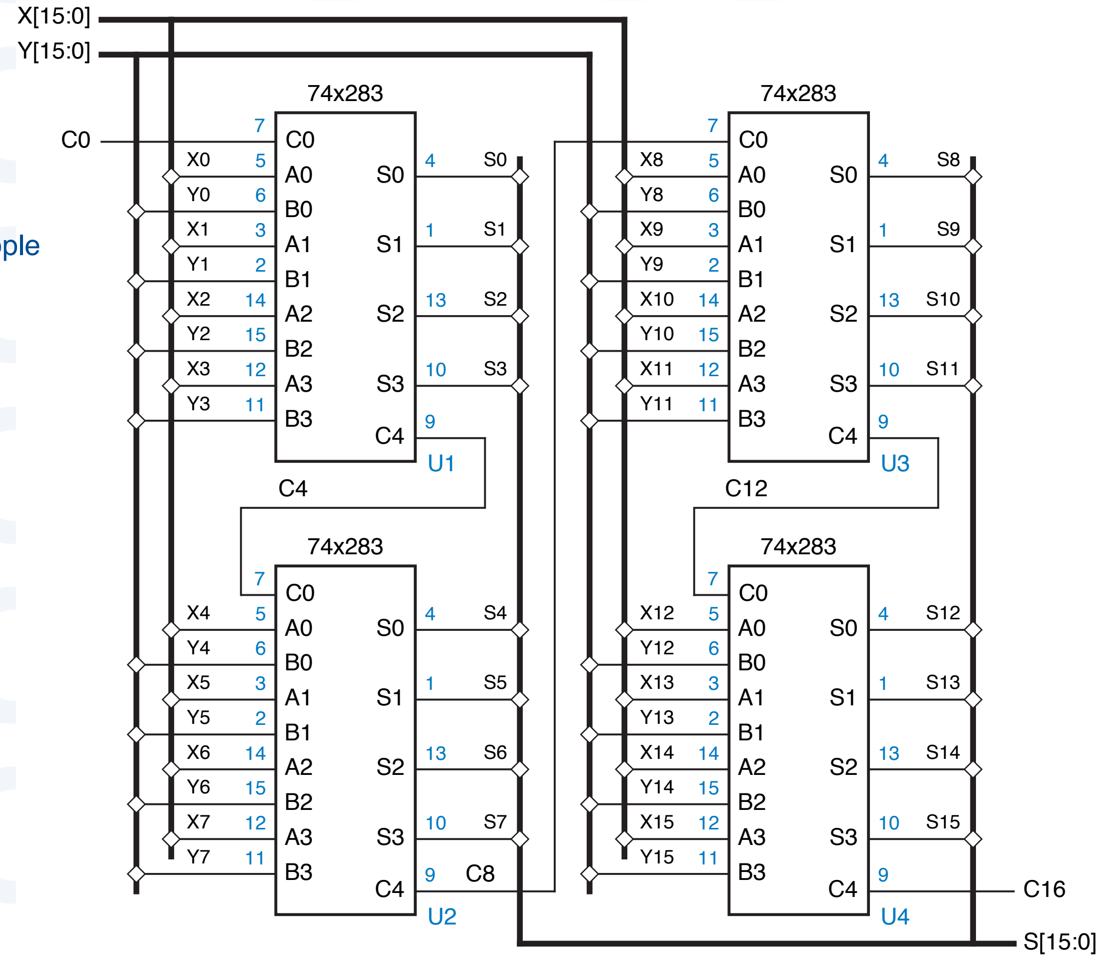


Figure 5-91 A 16-bit group-ripple adder.

*5.10.6 MSI Arithmetic and Logic Units

arithmetic and logic unit (ALU)

An *arithmetic and logic unit (ALU)* is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of *b*-bit operands. The operation to be performed is specified by a set of function-select inputs. Typical MSI ALUs have 4-bit operands and three to five function select inputs, allowing up to 32 different functions to be performed.

74x181

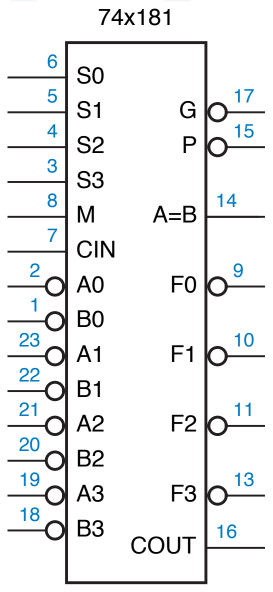
Figure 5-92 is a logic symbol for the *74x181* 4-bit ALU. The operation performed by the '181 is selected by the M and S3–S0 inputs, as detailed in Table 5-51. Note that the identifiers A, B, and F in the table refer to the 4-bit words A3–A0, B3–B0, and F3–F0; and the symbols · and + refer to logical AND and OR operations.

The 181's M input selects between arithmetic and logical operations. When M = 1, logical operations are selected, and each output Fi is a function only of the corresponding data inputs, Ai and Bi. No carries propagate between stages, and the CIN input is ignored. The S3–S0 inputs select a particular logical operation; any of the 16 different combinational logic functions on two variables may be selected.

Table 5-51 Functions performed by the 74x181 4-bit ALU.

Inputs				Function	
S3	S2	S1	S0	***M = 0 (arithmetic)***	***M = 1 (logic)***
0	0	0	0	F = A minus 1 plus CIN	F = A′
0	0	0	1	F = A · B minus 1 plus CIN	F = A′ + B′
0	0	1	0	F = A · B′ minus 1 plus CIN	F = A′ + B
0	0	1	1	F = 1111 plus CIN	F = 1111
0	1	0	0	F = A plus (A + B′) plus CIN	F = A′ · B′
0	1	0	1	F = A · B plus (A + B′) plus CIN	F = B′
0	1	1	0	F = A minus B minus 1 plus CIN	F = A ⊕ B′
0	1	1	1	F = A + B′ plus CIN	F = A + B′
1	0	0	0	F = A plus (A + B) plus CIN	F = A′ · B
1	0	0	1	F = A plus B plus CIN	F = A ⊕ B
1	0	1	0	F = A · B′ plus (A + B) plus CIN	F = B
1	0	1	1	F = A + B plus CIN	F = A + B
1	1	0	0	F = A plus A plus CIN	F = 0000
1	1	0	1	F = A · B plus A plus CIN	F = A · B′
1	1	1	0	F = A · B′ plus A plus CIN	F = A · B
1	1	1	1	F = A plus CIN	F = A

Figure 5-92 Logic symbol for the 74x181 4-bit ALU.



When M = 0, arithmetic operations are selected, carries propagate between the stages, and CIN is used as a carry input to the least significant stage. For operations larger than four bits, multiple ’181 ALUs may be cascaded like the group-ripple adder in the Figure 5-91, with the carry-out (COUT) of each ALU connected to the carry-in (CIN) of the next most significant stage. The same function-select signals (M, S3–S0) are applied to all the ’181s in the cascade.

To perform two’s-complement addition, we use S3–S0 to select the operation “A plus B plus CIN.” The CIN input of the least-significant ALU is normally set to 0 during addition operations. To perform two’s-complement subtraction, we use S3–S0 to select the operation A minus B minus plus CIN. In this case, the CIN input of the least significant ALU is normally set to 1, since CIN acts as the complement of the borrow during subtraction.

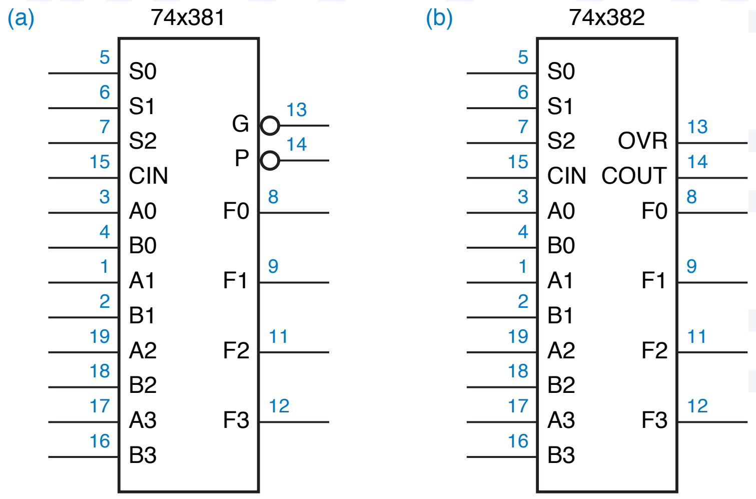
The ’181 provides other arithmetic operations, such as “A minus 1 plus CIN,” that are useful in some applications (e.g., decrement by 1). It also provides a bunch of weird arithmetic operations, such as “A · B′ plus (A + B) plus CIN,” that are almost never used in practice, but that “fall out” of the circuit for free.

Notice that the operand inputs A3_L–A0_L and B3_L–B0_L and the function outputs F3_L–F0_L of the ’181 are active low. The ’181 can also be used with active-high operand inputs and function outputs. In this case, a different version of the function table must be constructed. When M = 1, logical operations are still performed, but for a given input combination on S3–S0, the function obtained is precisely the dual of the one listed in Table 5-51. When M = 0, arithmetic operations are performed, but the function table is once again different. Refer to a ’181 data sheet for more details.

74x381
74x382

Two other MSI ALUs, the *74x381* and *74x382* shown in Figure 5-93, encode their select inputs more compactly, and provide only eight different but useful functions, as detailed in Table 5-52. The only difference between the ’381 and ’382 is that one provides group-carry lookahead outputs (which we explain next), while the other provides ripple carry and overflow outputs.

Figure 5-93
Logic symbols for 4-bit ALUs: (a) 74x381; (b) 74x382.



Inputs			
S2	S1	S0	***Function***
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111

Table 5-52 Functions performed by the 74x381 and 74x382 4-bit ALUs.

*5.10.7 Group-Carry Lookahead

group-carry lookahead

The ’181 and ’381 provide *group-carry lookahead* outputs that allow multiple ALUs to be cascaded without rippling carries between 4-bit groups. Like the 74x283, the ALUs use carry lookahead to produce carries internally. However, they also provide G_L and P_L outputs that are carry lookahead signals for the entire 4-bit group. The G_L output is asserted if the ALU generates a carry, that is, if it will produce a carry-out (COUT = 1) whether or not there is a carry-in (CIN = 1):

$$\text{G\_L} = (g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0)'$$

The P_L output is asserted if the ALU propagates a carry, that is, if it will produce a carry-out if there is a carry-in:

$$\text{P\_L} = (p_3 \cdot p_2 \cdot p_1 \cdot p_0)'$$

lookahead carry circuit
74x182

When ALUs are cascaded, the group-carry lookahead outputs may be combined in just two levels of logic to produce the carry input to each ALU. A *lookahead carry circuit*, the *74x182* shown in Figure 5-94, performs this operation. The ’182 inputs are C0, the carry input to the least significant ALU (“ALU 0”), and G0–G3 and P0–P3, the generate and propagate outputs of ALUs 0–3. Using these inputs, the ’182 produces carry inputs C1–C3 for ALUs 1–3. Figure 5-95 shows the connections for a 16-bit ALU using four ’381s and a ’182.

The 182’s carry equations are obtained by “adding out” the basic carry lookahead equation of Section 5.10.4:

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ &= (g_i + p_i) \cdot (g_i + c_i) \end{aligned}$$

Expanding for the first three values of i, we obtain the following equations:

$$\begin{aligned} C1 &= (G0+P0) \cdot (G0+C0) \\ C2 &= (G1+P1) \cdot (G1+G0+P0) \cdot (G1+G0+C0) \\ C3 &= (G2+P2) \cdot (G2+G1+P1) \cdot (G2+G1+G0+P0) \cdot (G2+G1+G0+C0) \end{aligned}$$

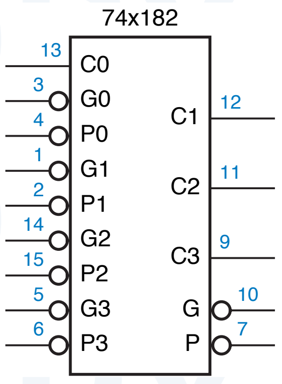
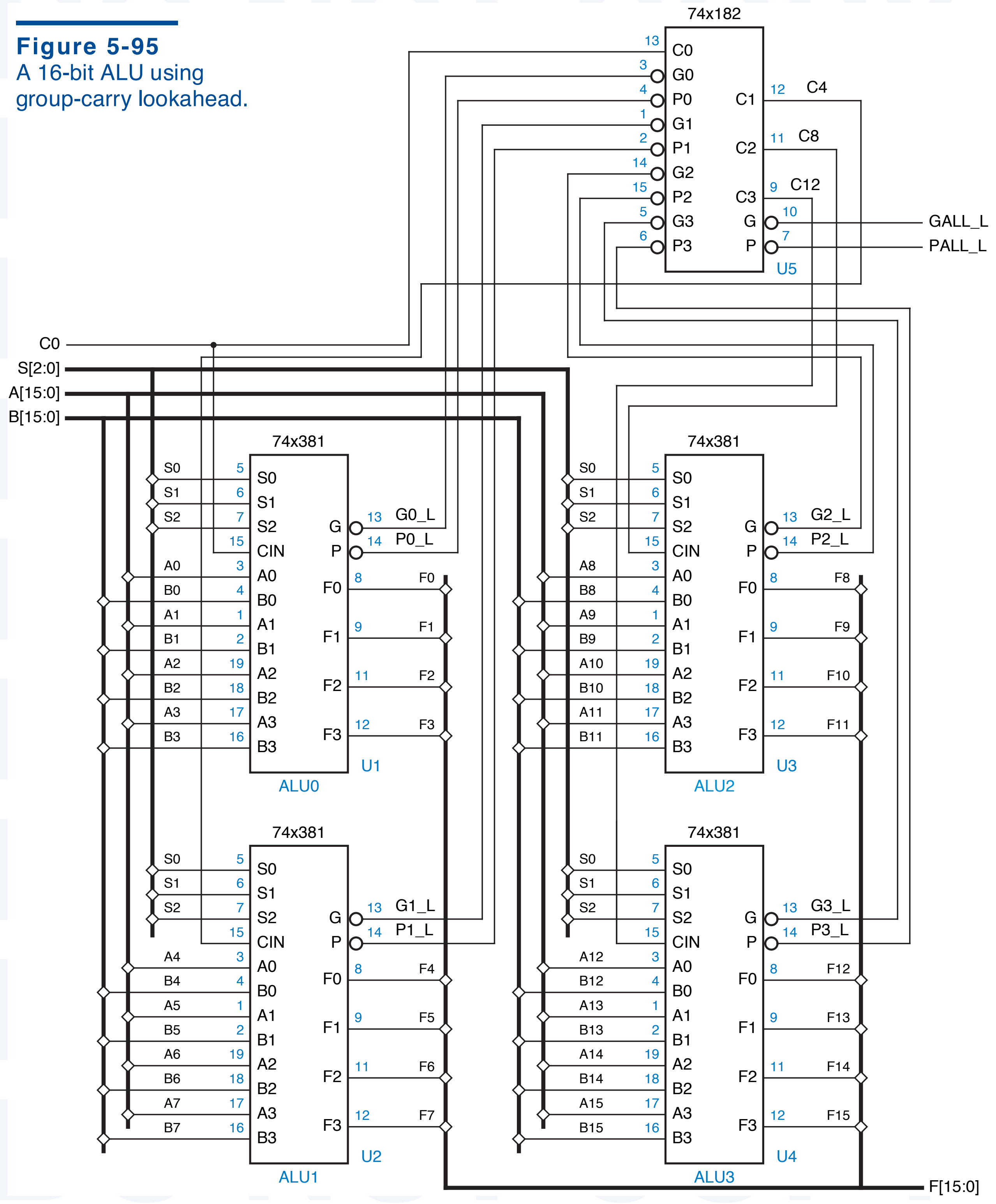


Figure 5-94 Logic symbol for the 74x182 lookahead carry circuit.

Figure 5-95
A 16-bit ALU using group-carry lookahead.



The ’182 realizes each of these equations with just one level of delay—an INVERT-OR-AND gate.

When more than four ALUs are cascaded, they may be partitioned into “supergroups,” each with its own ’182. For example, a 64-bit adder would have four supergroups, each containing four ALUs and a ’182. The G_L and P_L outputs of each ’182 can be combined in a next-level ’182, since they indicate whether the supergroup generates or propagates carries:

$$\mathrm{G\_L} = ((\mathrm{G3}+\mathrm{P3})\cdot(\mathrm{G3}+\mathrm{G2}+\mathrm{P2})\cdot(\mathrm{G3}+\mathrm{G2}+\mathrm{G1}+\mathrm{P1})\cdot(\mathrm{G3}+\mathrm{G2}+\mathrm{G1}+\mathrm{G0}))'$$

$$\mathrm{P\_L} = (\mathrm{P0}\cdot\mathrm{P1}\cdot\mathrm{P2}\cdot\mathrm{P3})'$$

*5.10.8 Adders in ABEL and PLDs

ABEL supports addition (+) and subtraction (-) operators which can be applied to sets. Sets are interpreted as unsigned integers; for example, a set with n bits represents an integer in the range of 0 to 2^n-1. Subtraction is performed by negating the subtrahend and adding. Negation is performed in two’s complement; that is, the operand is complemented bit-by-bit and then 1 is added.

Table 5-53 shows an example of addition in ABEL. The set definition for SUM was made one bit wider than the addends to accommodate the carry out of the MSB; otherwise this carry would be discarded. The set definitions for the addends were extended on the left with a 0 bit to match the size of SUM.

Even though the adder program is extremely small, it takes a long time to compile and it generates a huge number of terms in the minimal two level sum of products. While SUM0 has only two product terms, subsequent terms SUM*i* have $5\cdot2^i-4$ terms, or 636 terms for SUM7! And the carry out (SUM8) has $2^8-1=255$ product terms. Obviously, adders with more than a few bits cannot be practically realized using two levels of logic.

```
module add
title 'Adder Exercise'

" Input and output pins
A7..A0, B7..B0          pin;
SUM8..SUM0              pin istype 'com';

" Set definitions
A = [0, A7..A0];
B = [0, B7..B0];
SUM = [SUM8..SUM0];

equations
SUM = A + B;

end add
```

Table 5-53
ABEL program for an 8-bit adder.