ECE 27000 S22 P 14

Chapter 8. Combinational Arithmetic Elements

8.1. Adding and Subtracting

An *adder* combines two arithmetic operands using the addition rules we have seen before for the various representations of numbers.
The rules are the same for unsigned and two's-complement numbers.

Subtraction can be implemented by adding the minuend with the two's-complement of the subtrahend.
It is also possible to design a *subtractor* circuit.

The simplest adder is called a *half adder*.
It accepts two 1-bit operands X and Y.
It produces a 2-bit sum.
Two bits are required to represent the possible sums 0, 1 and 2 of X and Y.
The least-significant bit is called the sum and denoted by S.
The most-significant bit is called the carry-out and denoted by CO.

X	Y	CO	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = X \oplus Y = X \cdot Y' + X' \cdot Y$
$CO = X \cdot Y$

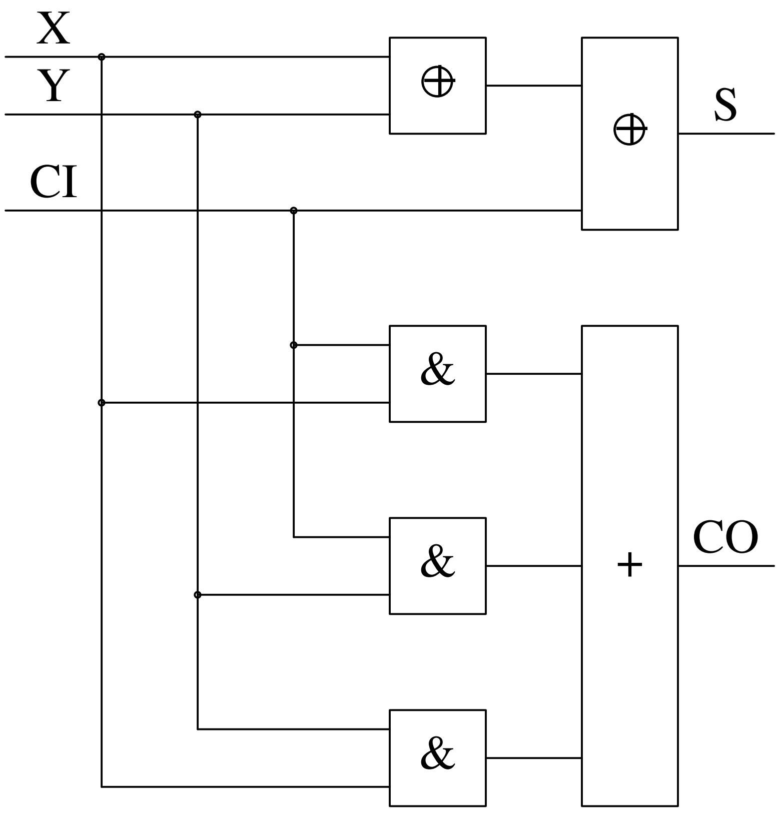
To add operads with more than one bit it is necessary to allow for carries between bit positions. The basic building block is a *full adder*.

A full adder allows a carry-in CI to be added to X and Y.
The possible sums are 0, 1, 2 or 3.
These can be expressed with two bits, S and CO.

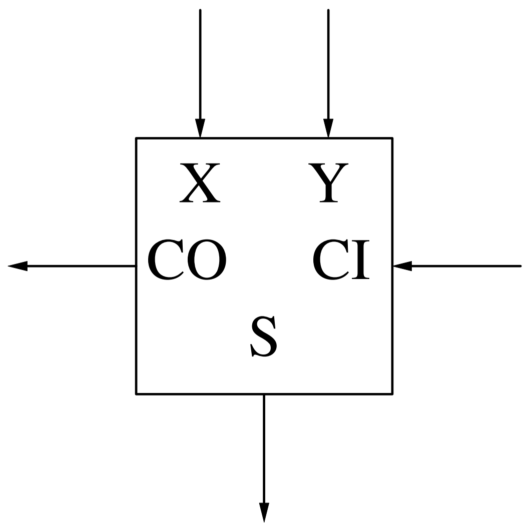
X	Y	CI	CO	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = X \oplus Y \oplus CI = X \cdot Y' \cdot CI' + X' \cdot Y \cdot CI' +$$
$$X' \cdot Y' \cdot CI + X \cdot Y \cdot CI$$
$$CO = X \cdot Y + X \cdot CI + Y \cdot CI.$$

A possible circuit:

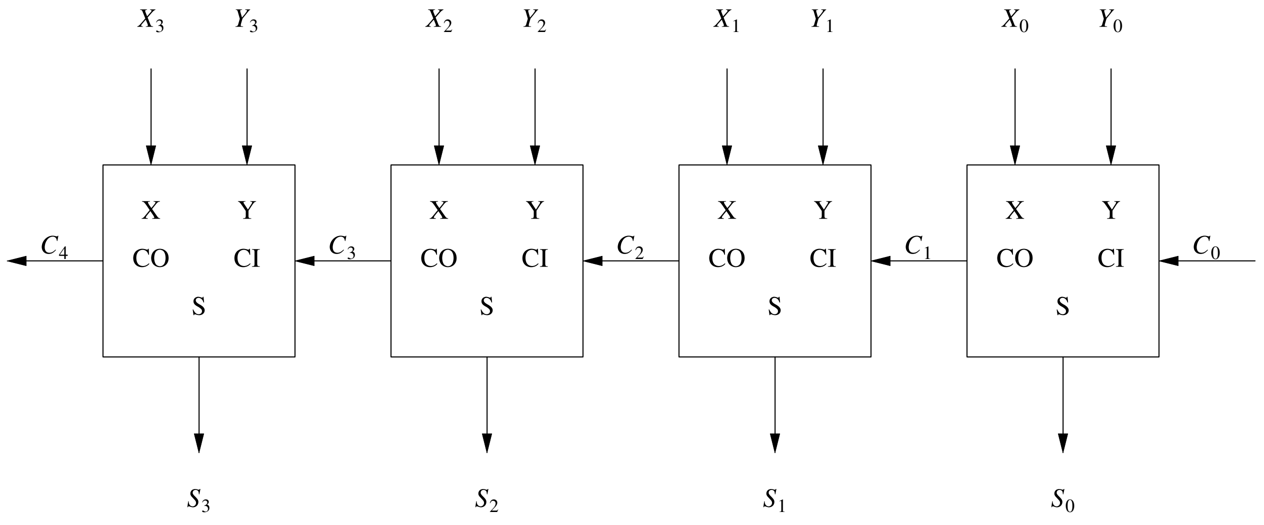


A symbol for a full adder:



Ripple Adders

Two *n*-bit operands can be added by cascading *n* full adders.
The resulting adder is called a *ripple adder* or a *ripple carry adder*.



This is another example of an iterative circuit.

With $c_0 = 0$, the sum of X and Y is obtained as $(C_n S_{n-1} \cdots S_0)$.

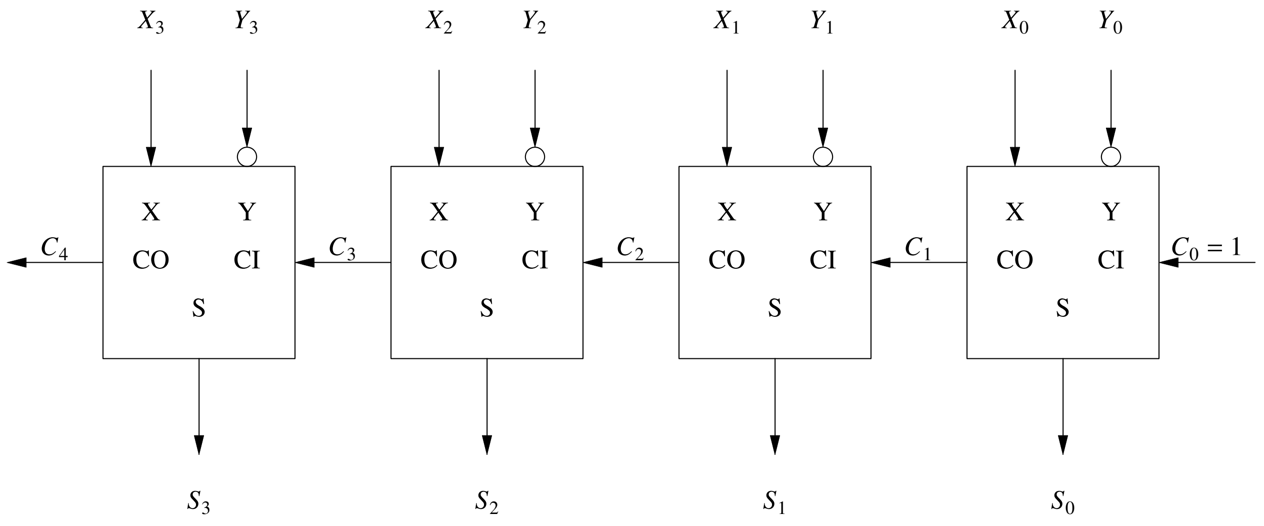
The ripple adder is slow since a carry may have to propagate from the least-significant full adder to the most-significant full adder.

The total delay can be written as
$t_{ADD} = t_{X,Y \to CO} + (n-2)t_{CI \to CO} + t_{CI \to S,CO}$

It is possible to design faster adders as we will see later.

Subtractors

A subtractor can be implemented using an adder.
$X - Y = X + (-Y)$
We need to substitute the two's-complement of Y for $-Y$.
The two's-complement of Y is obtained by complementing each bit of Y to obtain Y', and adding 1.
$X - Y = X + (-Y) = X + Y' + 1$
Addition of 1 is implemented by using a carry-in of 1



It is also possible to implement a subtractor directly from a logic block analogous to a full adder.

X	Y	BI	BO	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$D = X \oplus Y \oplus BI$$
$$BO = X' \cdot Y + X' \cdot BI + Y \cdot BI$$

The resulting circuit can be implemented using a full adder.

Magnitude Comparators

A magnitude comparator can be obtained from a subtractor by computing the following borrow-outs:
BO_{X-Y} for the computation $X - Y$.
BO_{Y-X} for the computation $Y - X$.

This requires only the borrow-out to be computed. The sum can be omitted.

We have the following possibilities:

BO_{X-Y}	BO_{Y-X}	result
0	0	$X = Y$
0	1	$X > Y$
1	0	$X < Y$

Carry-Lookahead Adders

A carry lookahead adder speeds up the longest path through the adder by using the following basic idea:

Cell i of an adder has a carry-in C_i.
C_i is a function of $X_{i-1} \cdots X_0$, $Y_{i-1} \cdots Y_0$ and C_0.

Calculate C_i from $X_{i-1} \cdots X_0$, $Y_{i-1} \cdots Y_0$ and C_0 using fewer logic levels than those included in $i-1$ full adders.
Then S_i can be calculated from the equation
$S_i = X_i \oplus Y_i \oplus C_i$

Carry lookahead logic is based on two definitions:

Cell i of an adder is said to *generate* a carry if $C_{i+1} = 1$ based only on X_i and Y_i, and independent of $X_{i-1} \cdots X_0, Y_{i-1} \cdots Y_0, C_0$.

The carry-generate function is denoted by G_i. We have $G_i = X_i \cdot Y_i$.

X_i	Y_i	C_{i-1}	C_i	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cell i of an adder is said to *propagate* a carry if $C_i = 1$ results in $C_{i+1} = 1$ based only on X_i and Y_i, and independent of $X_{i-1} \cdots X_0$, $Y_{i-1} \cdots Y_0$ and C_0.

The carry-propagate function is denoted by P_i.
Interpreting the definition strictly,
$P_i = X_i \oplus Y_i$
It is also correct to use
$P_i = X_i + Y_i$

X_i	Y_i	C_{i-1}	C_i	S_i
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

Using G_i and P_i, we obtain
$C_{i+1} = G_i + P_i \cdot C_i$.

To avoid ripple carry it is possible to expand these equations.

$$C_1 = G_0 + P_0 \cdot C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 \cdot C_1 \\ &= G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0) \\ &= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \end{aligned}$$

$$\begin{aligned} C_3 &= G_2 + P_2 \cdot C_2 \\ &= G_2 + P_2 \cdot (G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0) \\ &= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + \\ &\qquad + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \end{aligned}$$

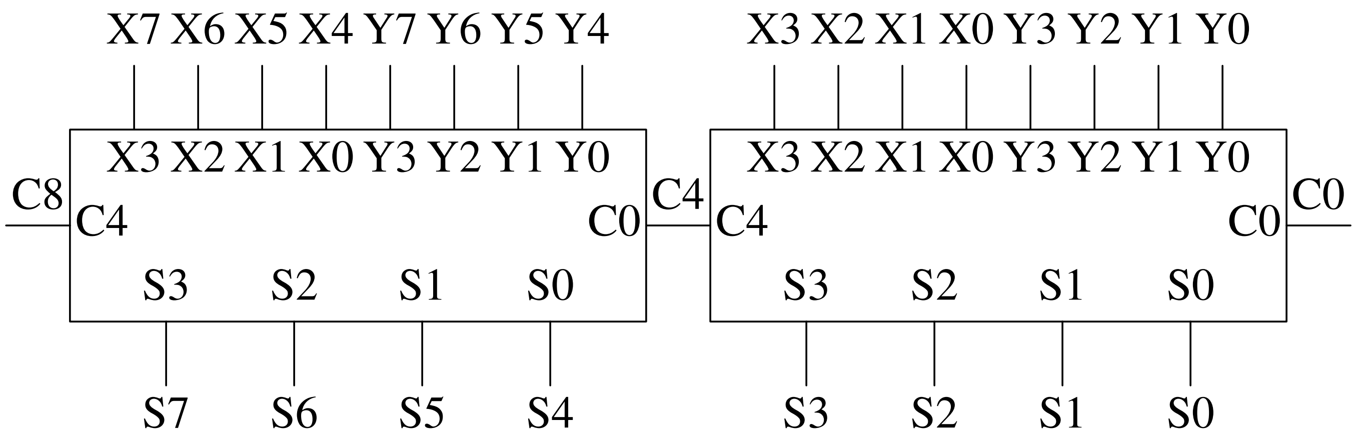
and so on

These equations require three levels of logic
(two-level equations,
one more level for G_i and P_i).

$S_0 = X_0 \oplus Y_0 \oplus C_0$
$S_1 = X_1 \oplus Y_1 \oplus C_1$
$S_2 = X_2 \oplus Y_2 \oplus C_2$
and so on
These equations require two additional levels of logic.

However, the number of gates grows with *i*. Therefore, carry lookahead is used for a limited number of levels, for example, four.

In a *group ripple adder*, logic blocks implementing 4-bit carry lookahead are cascaded. Between the groups, ripple carry is used.



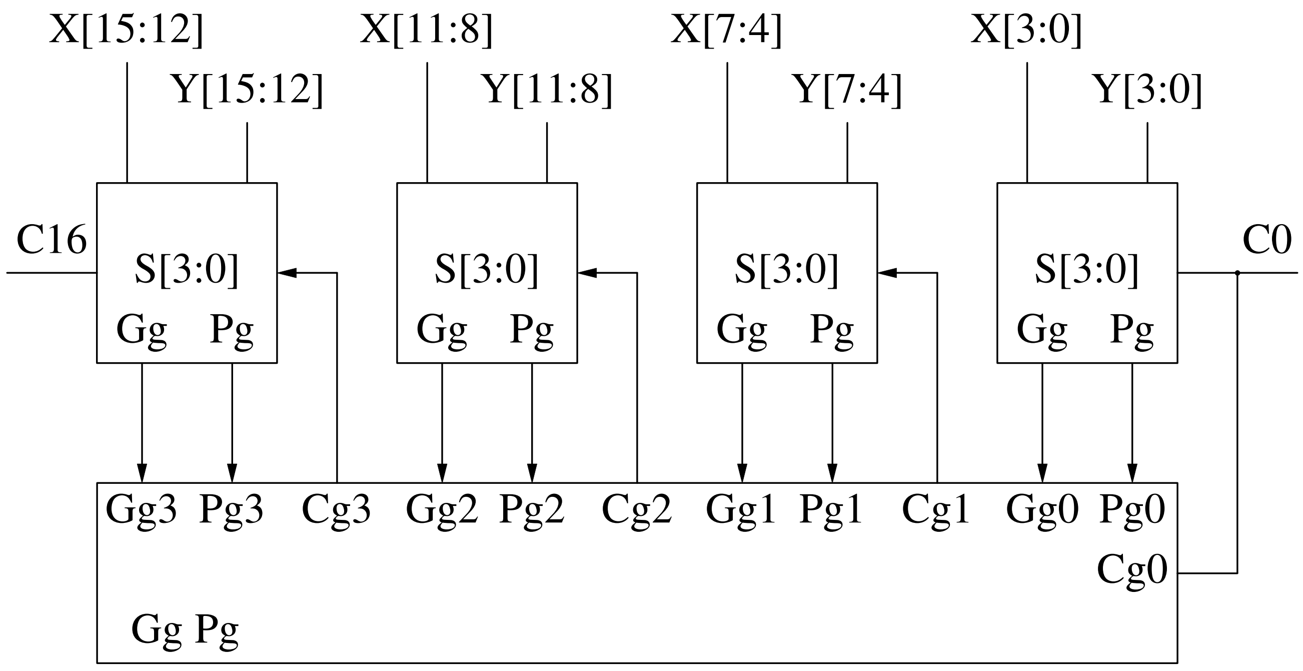
In a *group-carry lookahead adder*, the groups are connected using another level of carry-lookahead.
For this, each four-bit adder needs to produce carry-lookahead signals for its entire group.

These are computed as:

$$G_g = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + \\ + P_3 \cdot P_2 \cdot P_1 \cdot G_0$$

$$P_g = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$

When two or more adders are cascaded, the G and P functions can be used to compute carry-outs through the same equations as before.


X[15:12]
Y[15:12]
X[11:8]
Y[11:8]
X[7:4]
Y[7:4]
X[3:0]
Y[3:0]
C16
C0
S[3:0]
Gg
Pg
S[3:0]
Gg
Pg
S[3:0]
Gg
Pg
S[3:0]
Gg
Pg
Gg3
Pg3
Cg3
Gg2
Pg2
Cg2
Gg1
Pg1
Cg1
Gg0
Pg0
Cg0
Gg Pg

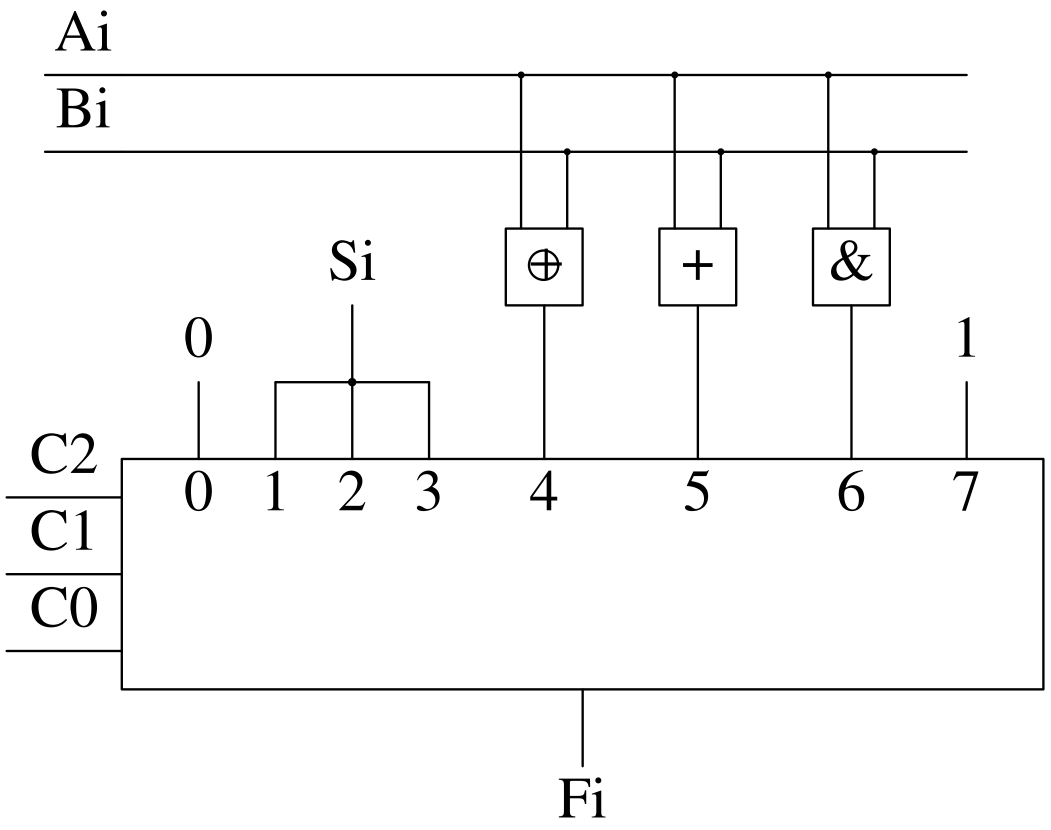
Arithmetic and Logic Units (ALUs)

An *arithmetic and logic unit* (*ALU*) is a combinational circuit that can perform a number of different arithmetic and logical operations on a pair of *b*-bit operands.

A set of function-select inputs determine which operation will be performed.

Example: A 4-bit ALU.
Inputs A[3:0], B[3:0] and CIN.
Outputs F[3:0] and COUT.

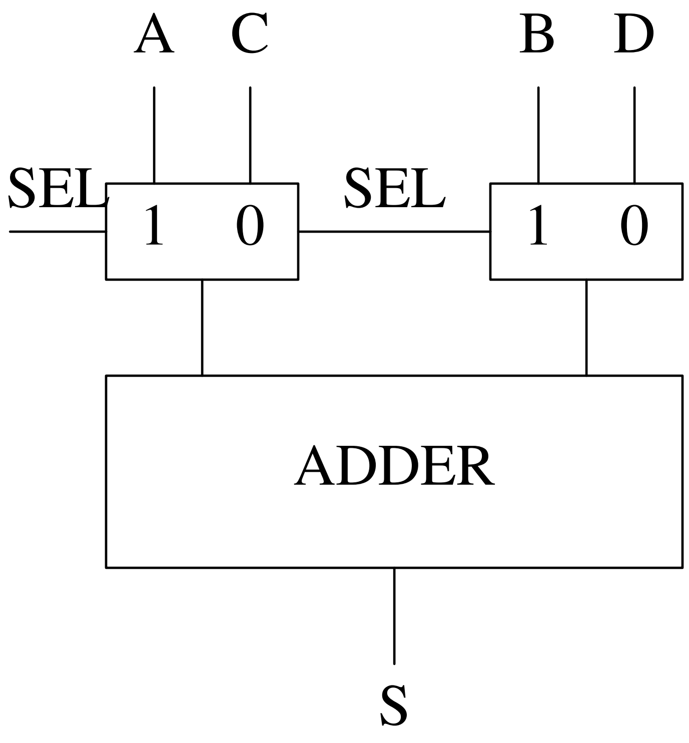
C2	C1	C0	Function
0	0	0	F=0000
0	0	1	F=A-B-1+CIN (arith)
0	1	0	F=B-A-1+CIN (arith)
0	1	1	F=A+B+CIN (arith)
1	0	0	F=A$\oplus$B (logical)
1	0	1	F=A+B (logical)
1	1	0	F=A$\cdot$B (logical)
1	1	1	F=1111


Ai
Bi
Si
⊕
+
&
0
1
C2
C1
C0
0
1
2
3
4
5
6
7
Fi

A Verilog Example:

```
module Vradders(SEL,A,B,C,D,S);
    input SEL;
    input [7:0] A,B,C,D;;
    output reg [7:0] S;
    always@(*)
        if(SEL) S=A+B;
        else S=C+D;
endmodule
```

The synthesis tool will use a single adder with multiplexers to select operands.



8.3. Multipliers

Binary multiplication is done similar to decimal multiplication.

Example with unsigned numbers:

$$\begin{array}{cccccccc}
 & & & & 1 & 0 & 1 & 1 \\
 & & & & 1 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 0 & 1 & 1 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 1 & 0 & 1 & 1 & & \\
 & 1 & 0 & 1 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

For n-bit operands the largest product is
$(2^n - 1) \cdot (2^n - 1)$.
It can be represented using $2n$ bits.

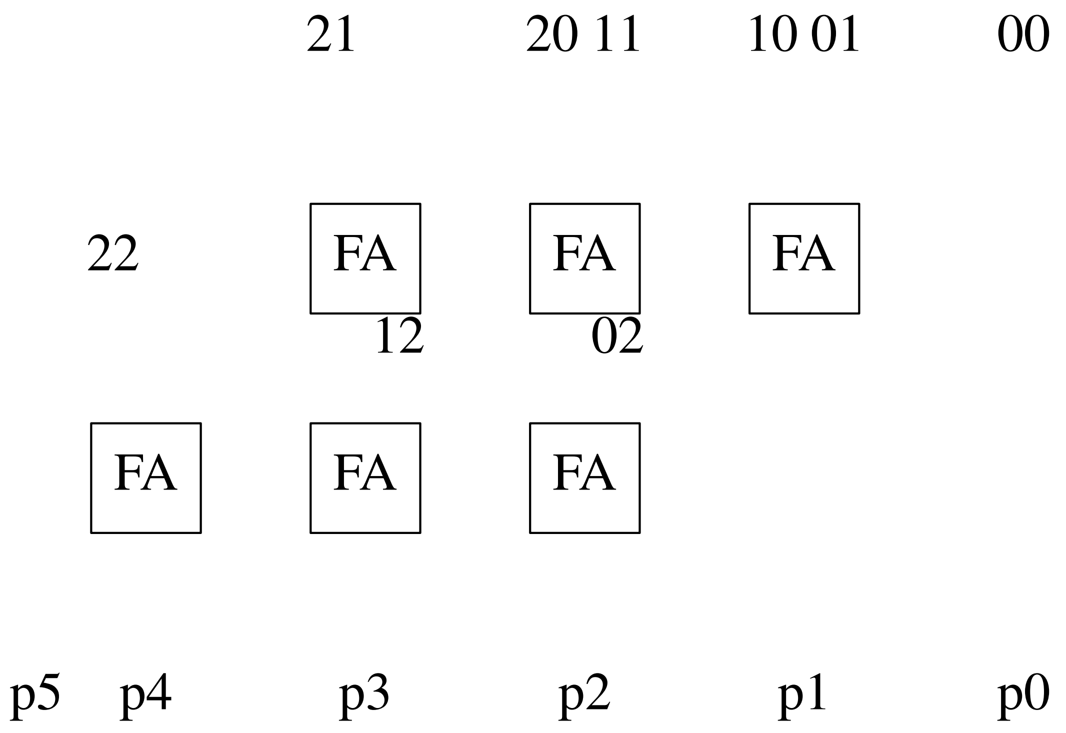
For signed numbers, it is possible to use absolute values and unsigned multiplication.
It is then necessary to complement the result if only one operand was negative.

The circuit for implementing a multiplier can be combinational or sequential.

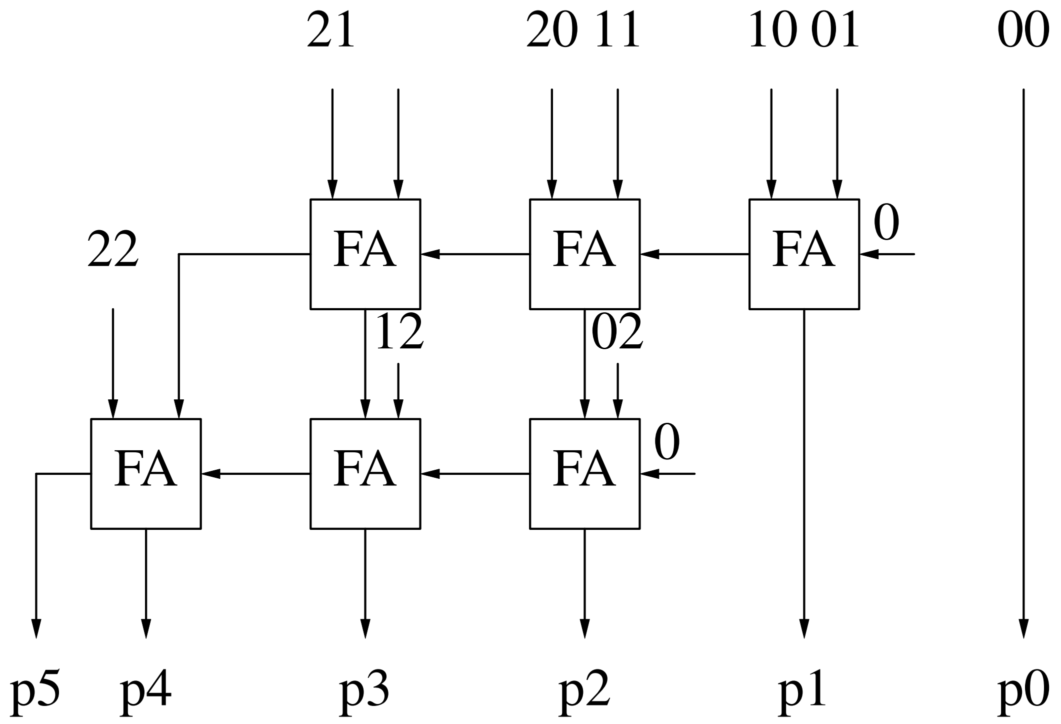
A sequential multiplier is based on an iterative addition of a shifted operand.
It requires a shifter, a single adder, and a register to store the intermediate results.

A combinational multiplier for n-bit operands is based on the use of $n-1$ n-bit adders.

$$
\begin{array}{cccccc}
 & & & x_2 & x_1 & x_0 \\
 & & & y_2 & y_1 & y_0 \\
\hline
 & & & x_2 \cdot y_0 & x_1 \cdot y_0 & x_0 \cdot y_0 \\
 & & x_2 \cdot y_1 & x_1 \cdot y_1 & x_0 \cdot y_1 & \\
 & x_2 \cdot y_2 & x_1 \cdot y_2 & x_0 \cdot y_2 & & \\
\hline
p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$



$$\begin{array}{cccccc}
 & & & x_2 & x_1 & x_0 \\
 & & & y_2 & y_1 & y_0 \\
\hline
 & & & x_2 \cdot y_0 & x_1 \cdot y_0 & x_0 \cdot y_0 \\
 & & x_2 \cdot y_1 & x_1 \cdot y_1 & x_0 \cdot y_1 & \\
 & x_2 \cdot y_2 & x_1 \cdot y_2 & x_0 \cdot y_2 & & \\
\hline
p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$



Extension:

$$\begin{array}{r}
x_{n-1} \cdots x_1 x_0 \\
y_{m-1} \;\; \cdots \;\; y_1 y_0 \\
\hline
(x_{n-1} \cdots x_1 x_0) \cdot y_0 \\
(x_{n-1} \cdots x_1 x_0) \cdot y_1 \quad\;\; \\
\cdots \qquad\qquad \\
(x_{n-1} \cdots x_1 x_0) \cdot y_{m-1} \qquad\qquad\qquad\quad \\
\hline
p_{n+m-1} \qquad\qquad \cdots \qquad\qquad p_1 p_0
\end{array}$$

We need to add m numbers, each one having n-bits.
Each addition requires an adder with n stages (n full adders).
Addition of m numbers requires $m-1$ adders.

The total number of full adders is $(m-1)n$.

Delay in terms of full adders:

Let FA(i,j) be the full adder in row i column j.

i=0 and j=0 for the rightmost top full adder.

The maximum delay path is

FA(0,0) $\rightarrow$FA(0,1) $\rightarrow$FA(1,1) $\rightarrow$FA(1,2)$\cdots$

Each row not including the last one adds two full-adder delays.

There are $m-2$ such rows.

The last row adds n units.

Total: $2(m-2)+n$

Note that the carry-out of full adder i,j is connected to the carry-in of full adder i,j+1.

It is possible to speed up the multiplier by connecting the carry-out of full adder i,j to the carry-in of full adder i+1,j+1.
This is called carry-save addition.
It works until the row before last.
The last row can use carry-lookahead.

Delay without carry lookahead:
The maximum delay path is
FA(0,0) $\rightarrow$FA(1,1) $\rightarrow$FA(2,2)$\cdots$
Each row not including the last one adds one full-adder delay.
There are $m - 2$ such rows.
The last row adds n units.
Total: $m - 2 + n$

BCD Adder

BCD stands for binary-coded decimal.
It is the code where the digits 0 to 9 are encoded through their 4-bit unsigned binary representations, 0000 to 1001.
The code words 1010 to 1111 are not used.

A packed-BCD representation places two BCD digits in a byte.
A byte represents the numbers 0 to 99.

In general, digits in BCD have the following weights:

$$\begin{array}{c|c|c} 100 & 10 & 1 \\ \cdots d_9 d_8 & d_7 d_6 d_5 d_4 & d_3 d_2 d_1 d_0 \end{array}$$

Addition can be done similar to adding 4-bit binary numbers.
However, if the result exceeds 9 it is necessary to correct it as follows.
Suppose that the result is $R \geq 10$.

A carry-out of 1 has value 10 in BCD, but value 16 in binary.

If we add 6 in binary it will produce the following result:
R = 10 + (R-10)
R + 6 = 10 + (R-10) + 6 = 16 + (R-10)
The 16 is the binary carry.
We will be left with R-10 in the least-significant four digits, as needed.

Examples:

$$\begin{array}{rr} 3 & 0011 \\ 6 & 0110 \\ \hline 9 & 1001 \end{array}$$

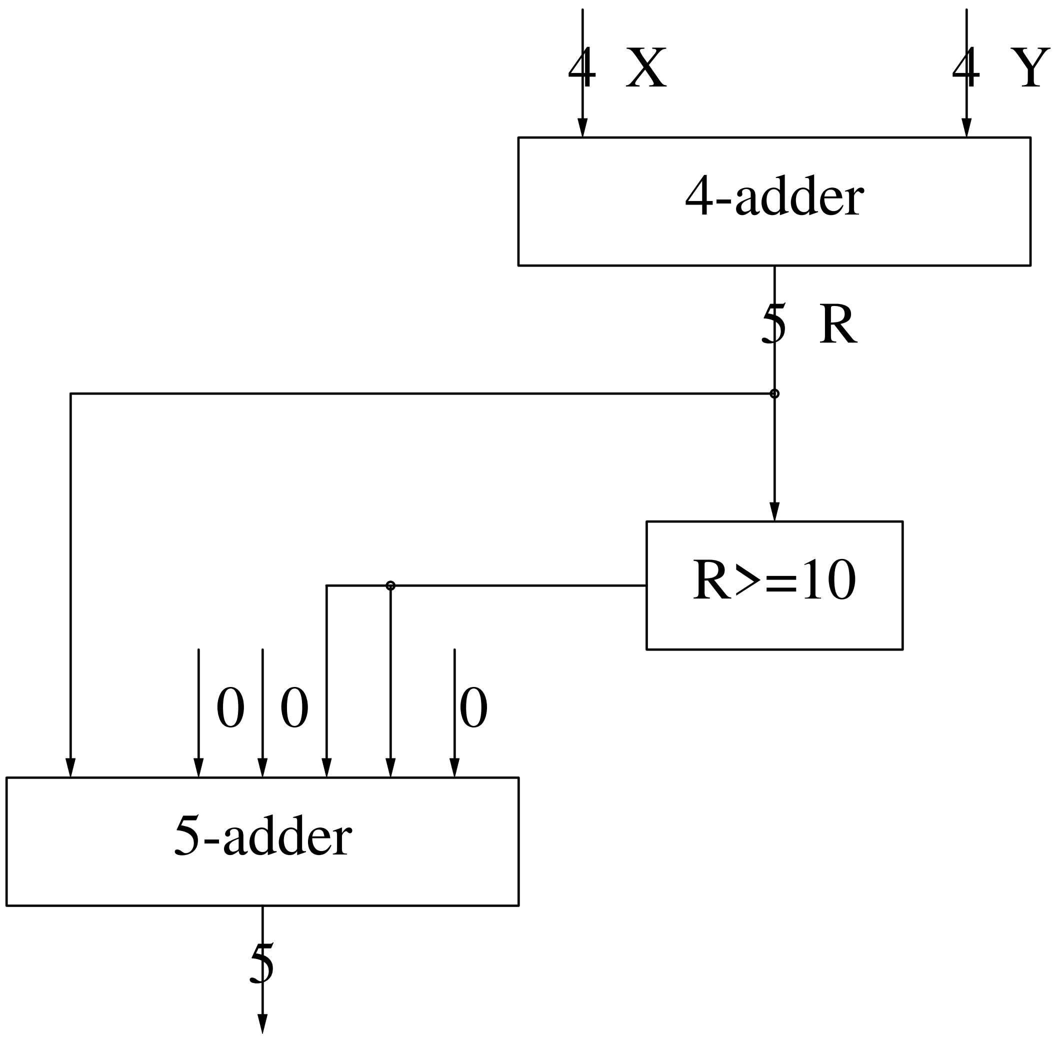
$$\begin{array}{rr} 3 & 0011 \\ 7 & 0111 \\ \hline & 1010 \\ & 0110 \\ \hline 10 & 10000 \end{array}$$

$$\begin{array}{rr} 9 & 1001 \\ 9 & 1001 \\ \hline & 10010 \\ & 0110 \\ \hline 18 & 11000 \end{array}$$

An adder needs to identify the case where the result is larger than nine and add six.

R_4	R_3	R_2	R_1	R_0	$R_{\geq 10}$
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1
1	x	x	x	x	1

$$\begin{aligned} R_{\geq 10} &= R_4 + R_3 \cdot R_2' \cdot R_1 + R_3 \cdot R_2 \\ &= R_4 + R_3 \cdot R_1 + R_3 \cdot R_2 \end{aligned}$$


4 X
4 Y
4-adder
5 R
R>=10
0
0
0
5-adder
5