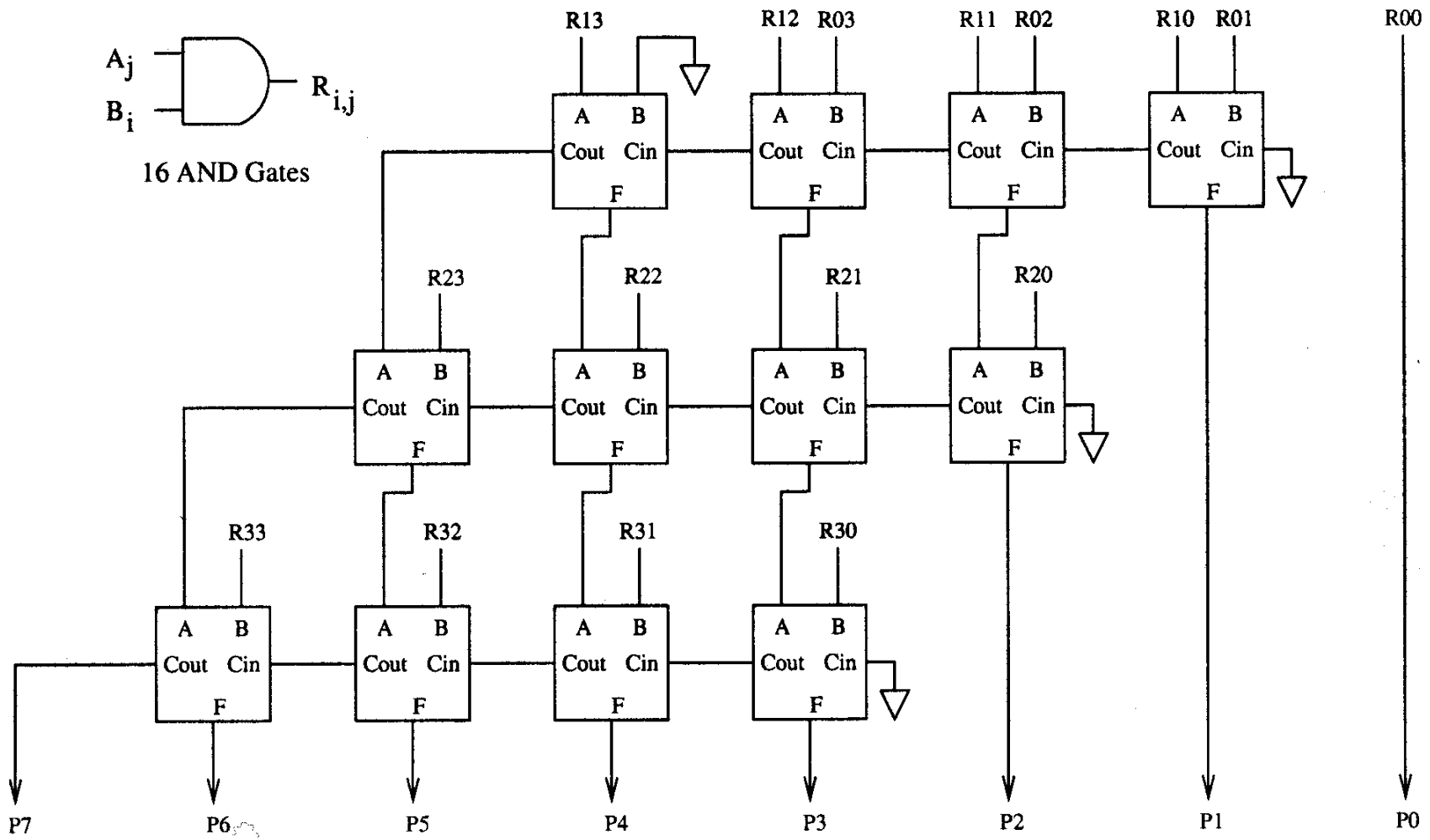
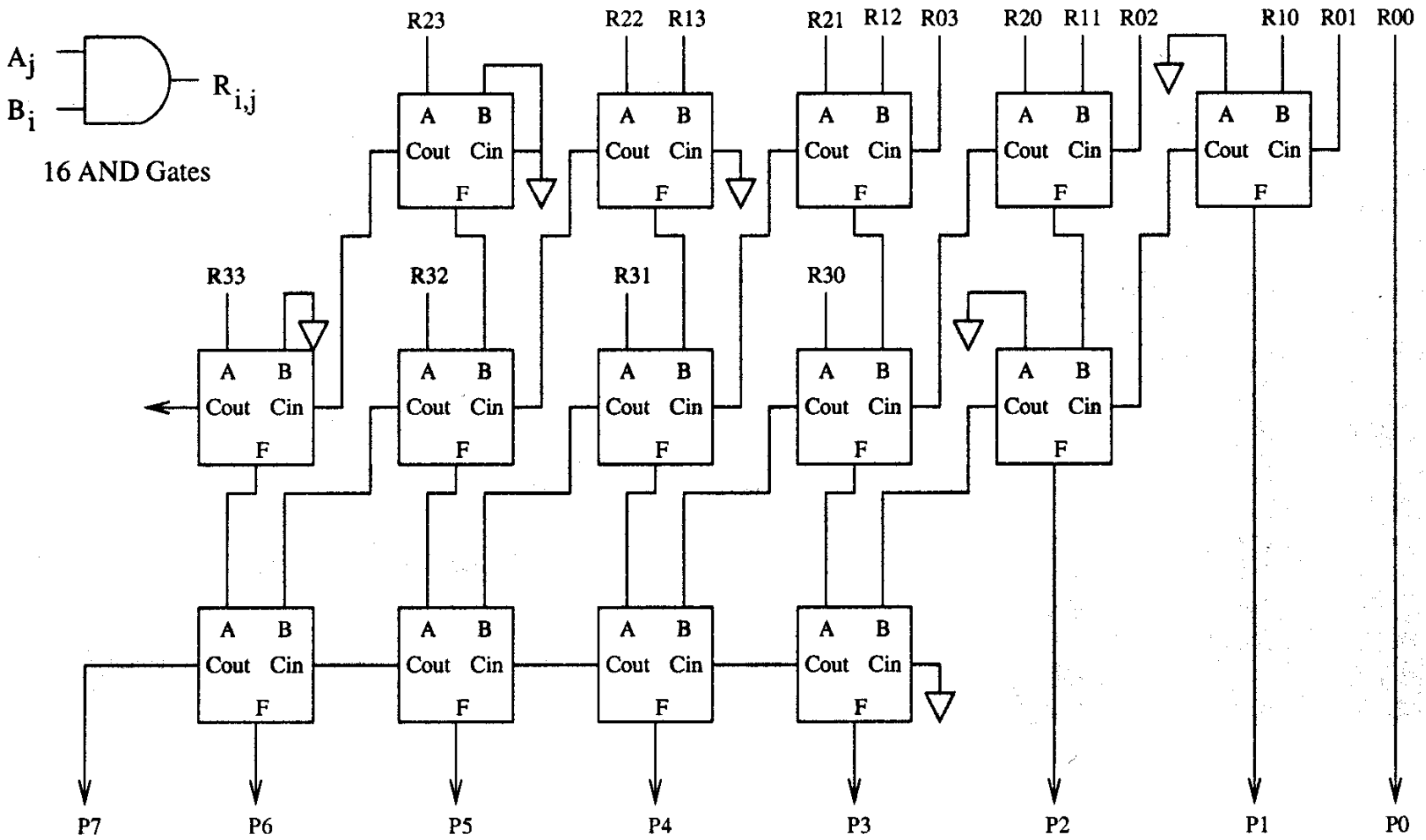
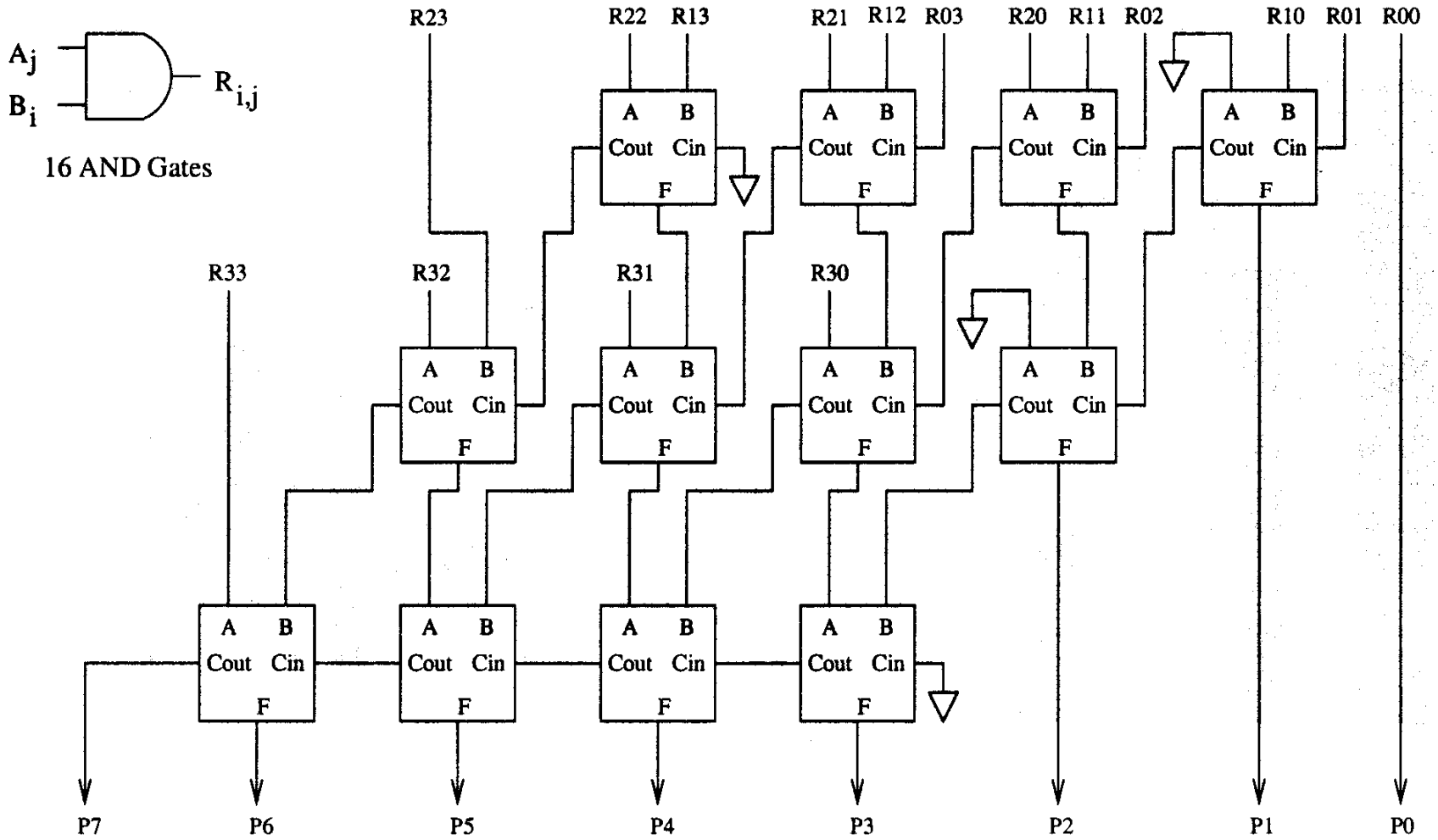
4-BIT MULTIPLIER USING FULL ADDERS/RIPPLE CARRY ADDERS



4-BIT MULTIPLIER USING CARRY SAVE ADDERS/3-to-2 ROW REDUCTION UNITS (RRUs)



4-BIT MULTIPLIER USING CARRY SAVE ADDERS/3-to-2 ROW REDUCTION UNITS (RRUs)



This approach allows the time delay of each (intermediate) stage to reduce to 2 gate delays!
The idea of "saving" the carry to the next stage gives rise to the name "carry save adder," or CSA.
A CSA is no different than a FA, it's just the way we use it!
The carrys must be added - this approach just puts it off to the last stage where it's done by a CPA (carry propagate adder).

The CSA can be thought of as a 3 row-to-2 row reduction device:

$R_{2,x}$, $R_{1,x}$, and $R_{0,x}$ (rows) are reduced to two "rows:"

→ The row of F outputs and the row of carrys

→ These two rows must be added to get the sum of the first 3 partial products

The second CSA row again does a 3-to-2 reduction, and the CPA adds these two rows to get the final result (product). Note that the addition of the final two rows could be done using an adder with carry lookahead as discussed previously!

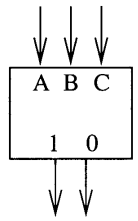
The CSA functions as a 3-to-2 row reduction unit, but other forms are possible: 7-to-3
15-to-4
31-to-5
2^{k-1}-to-k

Theoretically, each of these RRUs could be implemented in 2-level logic so that the delay per reduction stage was simply 2 gate delays.

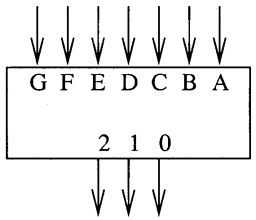
→ A 7-to-3 RRU has a 128-line truth table and over 140 minterms in the 3 output equations!

→ The "middle" bit has 42 product terms, for example, in the SOP implementation!

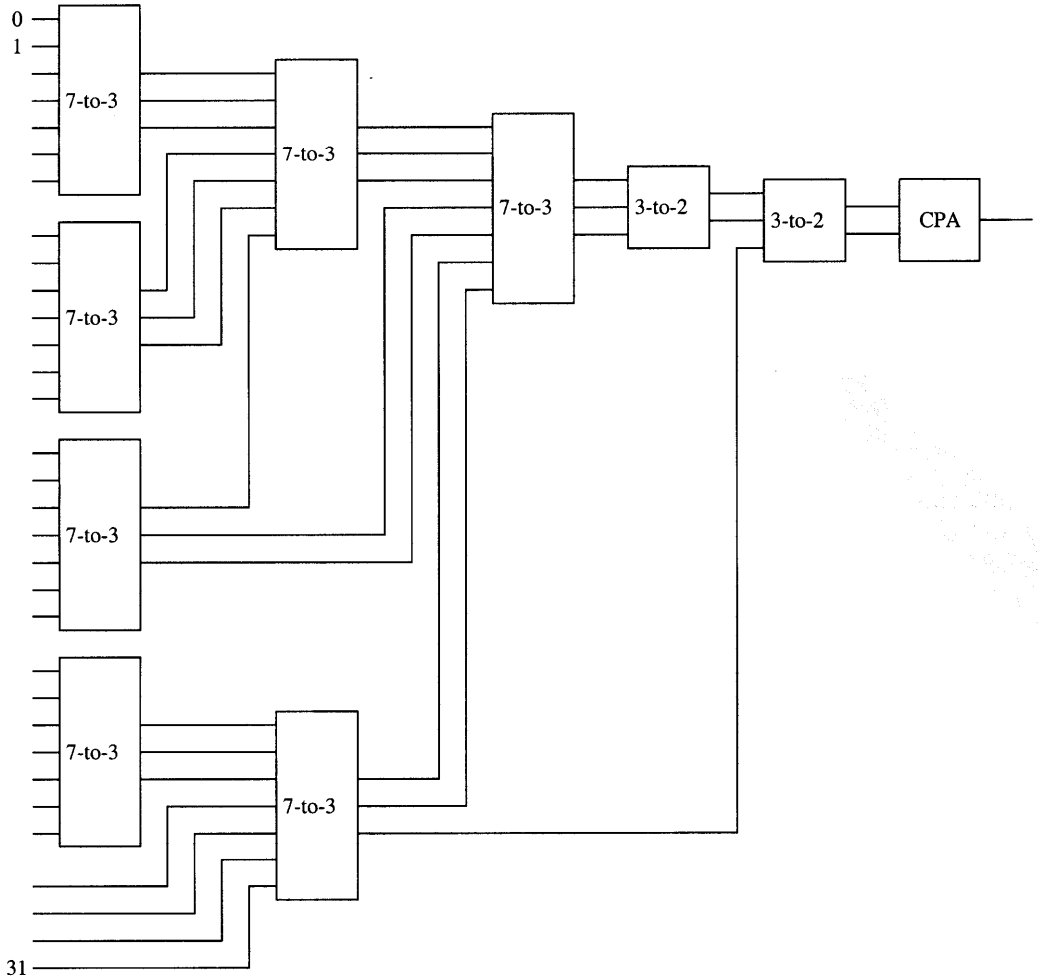
Let's look at the FA/CSA as a row reduction unit (RRU):



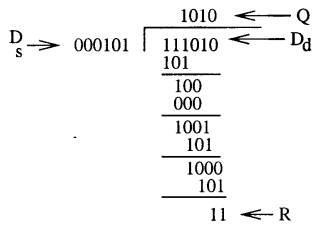
A 7-to-3 would be represented as:

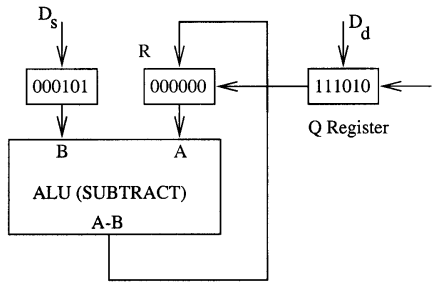


32-BIT MULTIPLIER ARCHITECTURE USING ROW REDUCTION TECHNIQUES



$$\frac{\text{Dividend}}{\text{Divisor}} = \text{Quotient} + \frac{\text{Remainder}}{\text{Divisor}} \quad \text{or} \quad \frac{D_d}{D_s} = Q + \frac{R}{D_s}$$





Division Procedure: If (A-B) > 0, load (A-B) into R and prepare a '1' for the Q register.

Then, shift Q and R.

$D_s = B$	R = A	D_d		
000101	000000	111010	0	(A-B) < 0
	000001	110100		shift
	000001	110100	0	(A-B) < 0
	000011	101000		shift
	000011	101000	0	(A-B) < 0
	000111	010000		shift
	000010	010000	1	(A-B) > 0, R <- (A-B)
	000100	100001		shift
	000100	100001	0	(A-B) < 0
	001001	000010		shift
	000100	000010	1	(A-B) > 0, R <- (A-B)
	001000	000101		shift
	000011	000101	0	(A-B) < 0
	000110	001010		shift
	R	Q		

$$Q = \frac{D_d}{D_s} = \frac{((((D_d) * f_0) * f_1) .....)}{((((D_s) * f_0) * f_1) .....)}$$

If we pick fi so that $((((D_s) * f_0) * f_1) .....) = 1$, then

$Q = ((((D_d) * f_0) * f_1) .....)$

How do we pick fi? One solution, for D_d and D_s being normalized fractions:

$D_s = 1 - x, x < 1$

Pick $f_0 = (1 + x)$ --> $(D_s * f_0) = (1-x)(1+x) = 1 - x^2$

which is closer to 1 than 1-x.

Pick $f_1 = (1 + x^2)$ --> $(D_s * f_0 * f_1) = (1-x^2)(1+x^2) = 1 - x^4$

which is closer to 1 than $1-x^2$.

ETC....

$f_0 = 1 + x = 1 + (1 - D_s) = 2 - D_s$

$f_1 = 1 + x^2 = 1 + (1 - D_s * f_0) = 2 - D_s * f_0$

$x^2 = 1 - (D_s * f_0)$

ETC....

Thus, fi is the "2's complement" of the denominator result to that point!

Example: If $D_s = 0.100$, the 2's complement is 1.100

When the denominator gets close enough to 1, we're done!

- Usually do a fixed number of iterations
- A fixed number of iterations usually requires that we start with a better "seed" than $(2 - D_s)$; this can be done using a ROM.

We could implement division using microcode on a machine that only had a hardware multiplier with enough registers, logic, and microcode states!

Division almost always takes longer on a machine than multiplications (3X-5X).

Division can also be done using an "iterative" hardware approach: