Appendix J
Computer Arithmetic

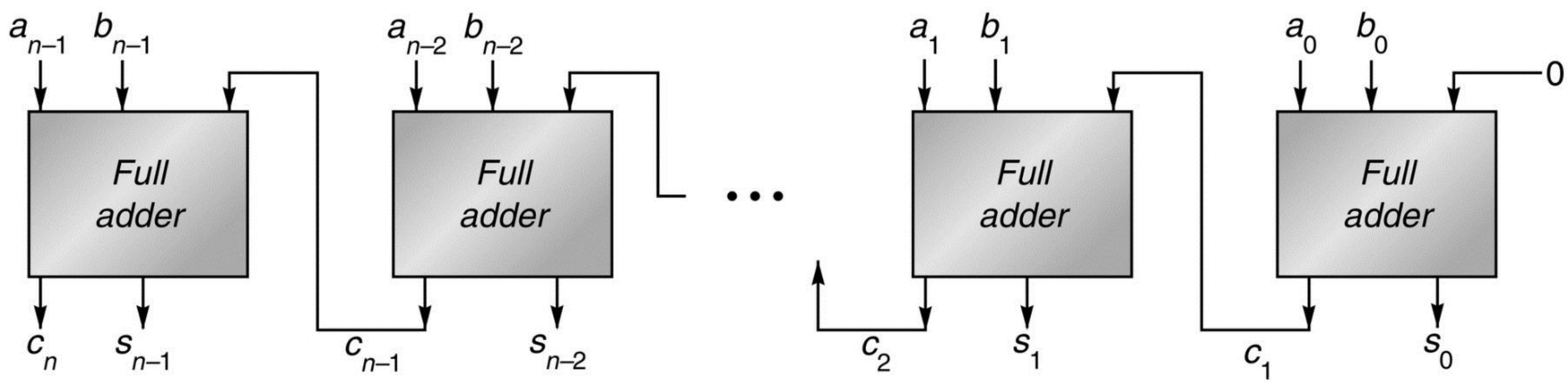


Figure J.1 Ripple-carry adder, consisting of *n* full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

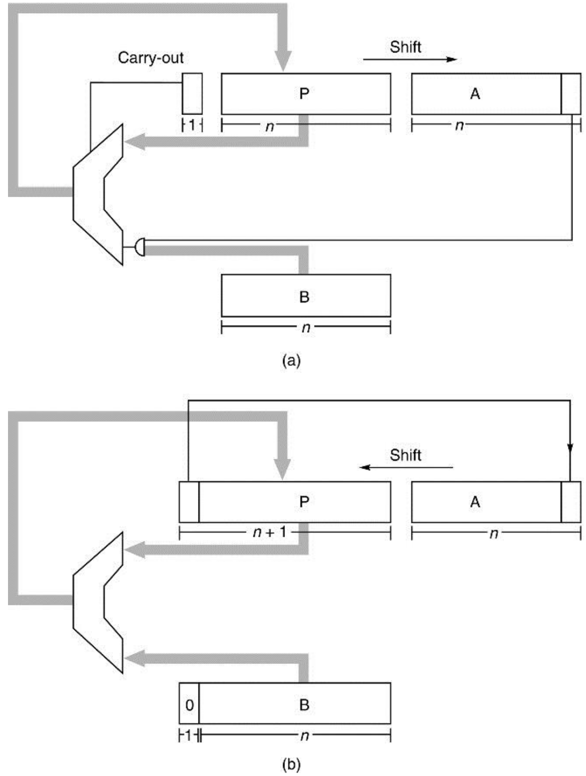


Figure J.2 Block diagram of (a) multiplier and (b) divider for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

P	A	
00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	step 1(i): shift.
−00011		step 1(ii): subtract.
−00010	110**0**	step 1(iii): result is negative, set quotient bit to 0.
00001	110**0**	step 1(iv): restore.
00011	10**0**	step 2(i): shift.
−00011		step 2(ii): subtract.
00000	10**01**	step 2(iii): result is nonnegative, set quotient bit to 1.
00001	0**01**	step 3(i): shift.
−00011		step 3(ii): subtract.
−00010	0**010**	step 3(iii): result is negative, set quotient bit to 0.
00001	0**010**	step 3(iv): restore.
00010	**010**	step 4(i): shift.
−00011		step 4(ii): subtract.
−00001	**0100**	step 4(iii): result is negative, set quotient bit to 0.
00010	**0100**	step 4(iv): restore. The quotient is 0100_2 and the remainder is 00010_2.

(a)

00000	1110	Divide $14 = 1110_2$ by $3 = 11_2$. B always contains 0011_2.
00001	110	step 1(i-b): shift.
+11101		step 1(ii-b): subtract b (add two's complement).
11110	110**0**	step 1(iii): P is negative, so set quotient bit to 0.
11101	10**0**	step 2(i-a): shift.
+00011		step 2(ii-a): add b.
00000	10**01**	step 2(iii): P is nonnegative, so set quotient bit to 1.
00001	0**01**	step 3(i-b): shift.
+11101		step 3(ii-b): subtract b.
11110	0**010**	step 3(iii): P is negative, so set quotient bit to 0.
11100	**010**	step 4(i-a): shift.
+00011		step 4(ii-a): add b.
11111	**0100**	step 4(iii): P is negative, so set quotient bit to 0.
+00011		Remainder is negative, so do final restore step.
00010		The quotient is 0100_2 and the remainder is 00010_2.

(b)

Figure J.3 Numerical example of (a) restoring division and (b) nonrestoring division.

P	A	
0000	1010	Put $-6 = 1010_2$ into A, $-5 = 1011_2$ into B.
0000	1010	step 1(i): $a_0 = a_{-1} = 0$, so from rule I add 0.
0000	0101	step 1(ii): shift.
+0101		step 2(i): $a_1 = 1$, $a_0 = 0$.Rule III says subtract b (or add $-b = -1011_2 = 0101_2$).
0101	0101	
0010	1010	step 2(ii): shift.
+ 1011		step 3(i): $a_2 = 0$, $a_1 = 1$. Rule II says add b (1011).
1101	1010	
1110	1101	step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)
+ 0101		step 4(i): $a_3 = 1$, $a_2 = 0$. Rule III says subtract b.
0011	1101	
0001	1110	step 4(ii): shift. Final result is $00011110_2 = 30$.

Figure J.4 Numerical example of Booth recoding. Multiplication of $a = -6$ by $b = -5$ to get 30.

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
IBM 370	If enable is on	No	Yes. `Add` sets cond code.	Yes. Logical `add` sets cond code.
Intel 8086	No	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
MIPS R3000	Two `add` instructions; one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	`Addcc` sets V bit. `Add` does not.	`Addcc` sets C bit. `Add` does not.

Figure J.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

Language	Division	Remainder
FORTRAN	$-5/3 = -1$	$\text{MOD}(-5, 3) = -2$
Pascal	-5 DIV $3 = -1$	-5 MOD $3 = 1$
Ada	$-5/3 = -1$	-5 MOD $3 = 1$ -5 REM $3 = -2$
C	$-5/3$ undefined	-5% 3 undefined
Modula-3	-5 DIV $3 = -2$	-5 MOD $3 = 1$

Figure J.6 Examples of integer division and integer remainder in various programming languages.

	Single	Single extended	Double	Double extended
p (bits of precision)	24	≥ 32	53	≥ 64
E_{max}	127	≥ 1023	1023	≥ 16383
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent bias	127		1023	

Figure J.7 Format parameters for the IEEE 754 floating-point standard. The first row gives the number of bits in the significand. The blanks are unspecified parameters.

Exponent	Fraction	Represents
$e = E_{\min} - 1$	$f = 0$	± 0
$e = E_{\min} - 1$	$f \neq 0$	$0.f \times 2^{E_{\min}}$
$E_{\min} \leq e \leq E_{\max}$	—	$1.f \times 2^{e}$
$e = E_{\max} + 1$	$f = 0$	$\pm\infty$
$e = E_{\max} + 1$	$f \neq 0$	NaN

Figure J.8 Representation of special values. When the exponent of a number falls outside the range $E_{\min} \leq e \leq E_{\max}$, then that number has a special interpretation as indicated in the table.

(a)

$$\begin{array}{r} 1.23 \\ \times\ 6.78 \\ \hline \mathbf{8.33}94 \\ \uparrow \end{array}$$

$r = 9 > 5$ so round up
rounds to 8.34

(b)

$$\begin{array}{r} 2.83 \\ \times\ 4.47 \\ \hline \mathbf{12.6}501 \\ \uparrow \end{array}$$

$r = 5$ and a following digit $\neq 0$ so round up
rounds to 1.27×10^1

(c)

$$\begin{array}{r} 1.28 \\ \times\ 7.81 \\ \hline 0\mathbf{9.99}68 \\ \uparrow \end{array}$$

$r = 6 > 5$ so round up
rounds to 1.00×10^1

Figure J.9 Examples of rounding a multiplication. Using base 10 and $p = 3$, parts (a) and (b) illustrate that the result of a multiplication can have either $2p - 1$ or $2p$ digits; hence, the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

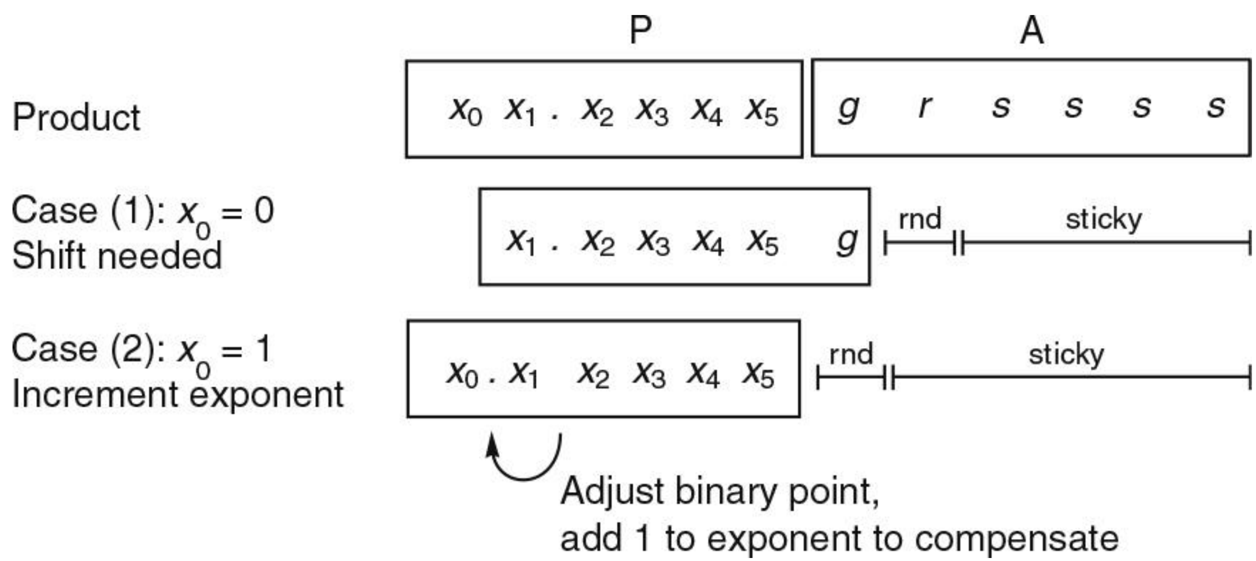


Figure J.10 The two cases of the floating-point multiply algorithm. The top line shows the contents of the P and A registers after multiplying the significands, with $p = 6$. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked s.

Rounding mode	Sign of result ≥ 0	Sign of result < 0
$-\infty$		$+1$ if $r \vee s$
$+\infty$	$+1$ if $r \vee s$	
0		
Nearest	$+1$ if $r \wedge p_0$ or $r \wedge s$	$+1$ if $r \wedge p_0$ or $r \wedge s$

Figure J.11 Rules for implementing the IEEE rounding modes. Let S be the magnitude of the preliminary result. Blanks mean that the p most-significant bits of S are the actual result bits. If the condition listed is true, add 1 to the pth most-significant bit of S. The symbols r and s represent the round and sticky bits, while p_0 is the pth most-significant bit of S.

swap	**compl**	**sign(a_1)**	**sign(a_2)**	**sign(result)**
Yes		+	–	–
Yes		–	+	+
No	No	+	–	+
No	No	–	+	–
No	Yes	+	–	–
No	Yes	–	+	+

Figure J.12 Rules for computing the sign of a sum when the addends have different signs. The *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are "don't care."

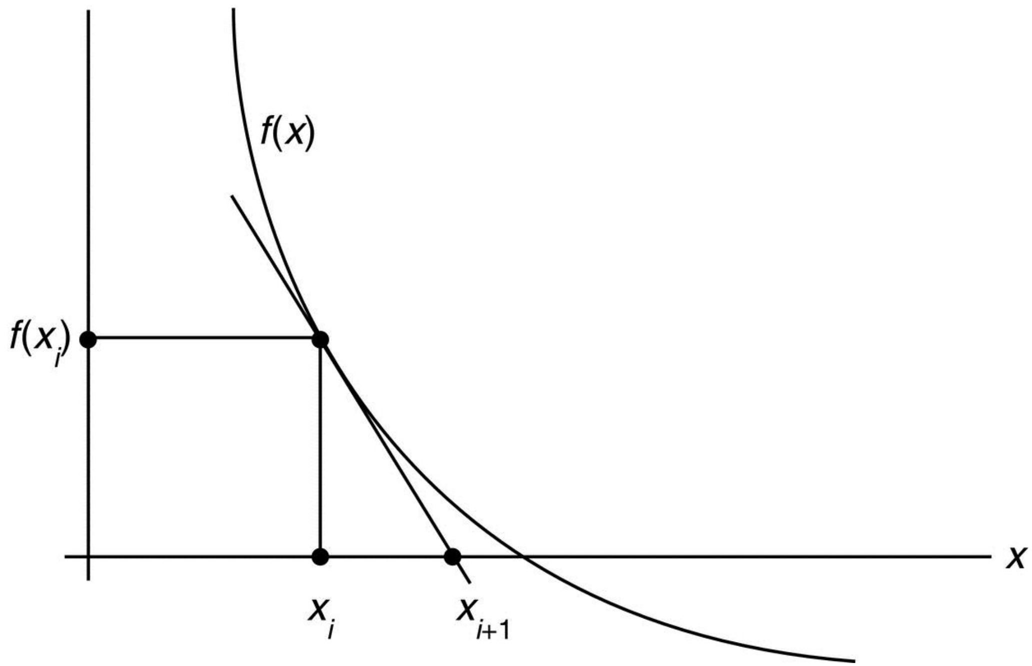


Figure J.13 Newton's iteration for zero finding. If x_i is an estimate for a zero of *f*, then x_{i+1} is a better estimate. To compute x_{i+1}, find the intersection of the *x*-axis with the tangent line to *f* at $f(x_i)$.

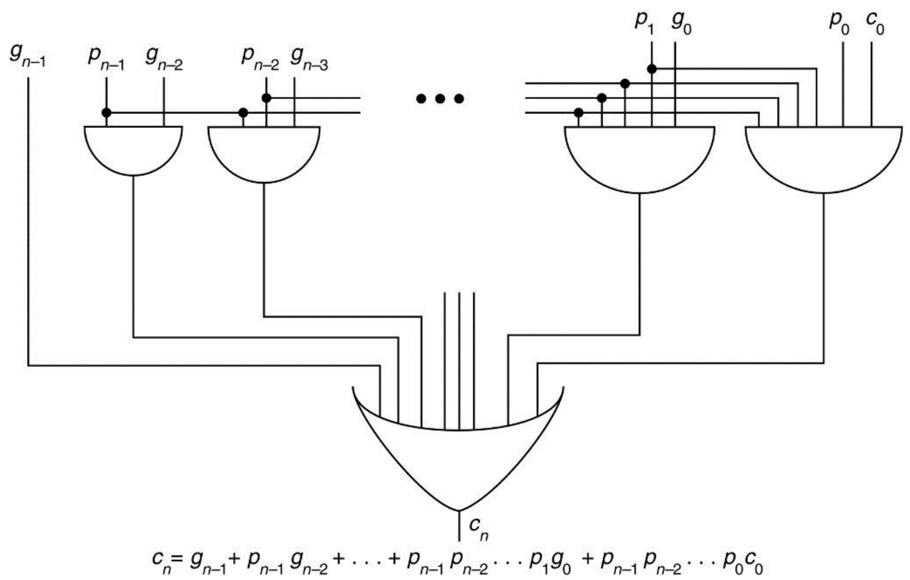


Figure J.14 Pure carry-lookahead circuit for computing the carry-out c_n of an *n*-bit adder.

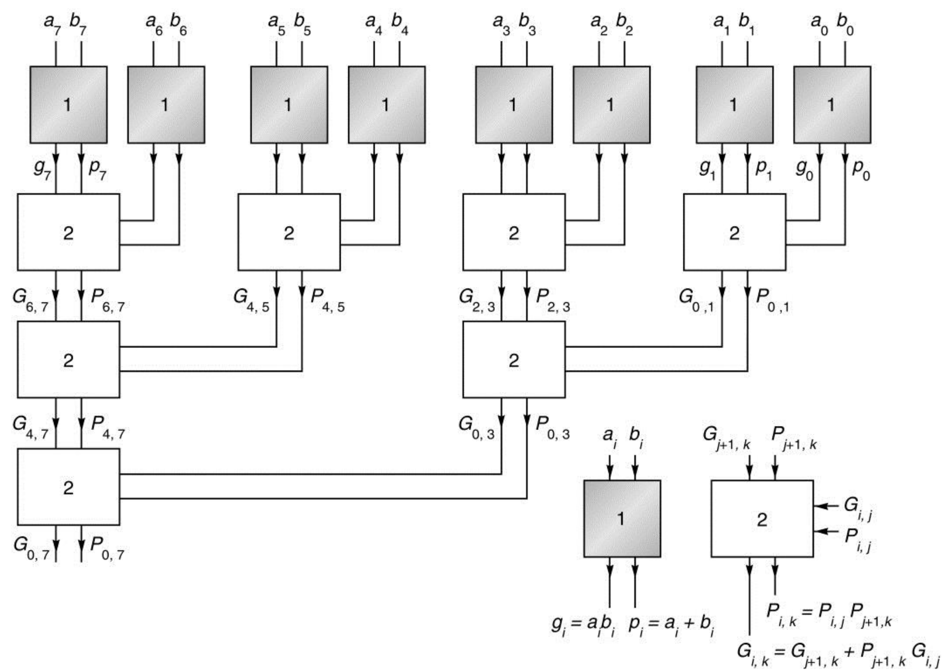


Figure J.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of *P* and *G* are computed.

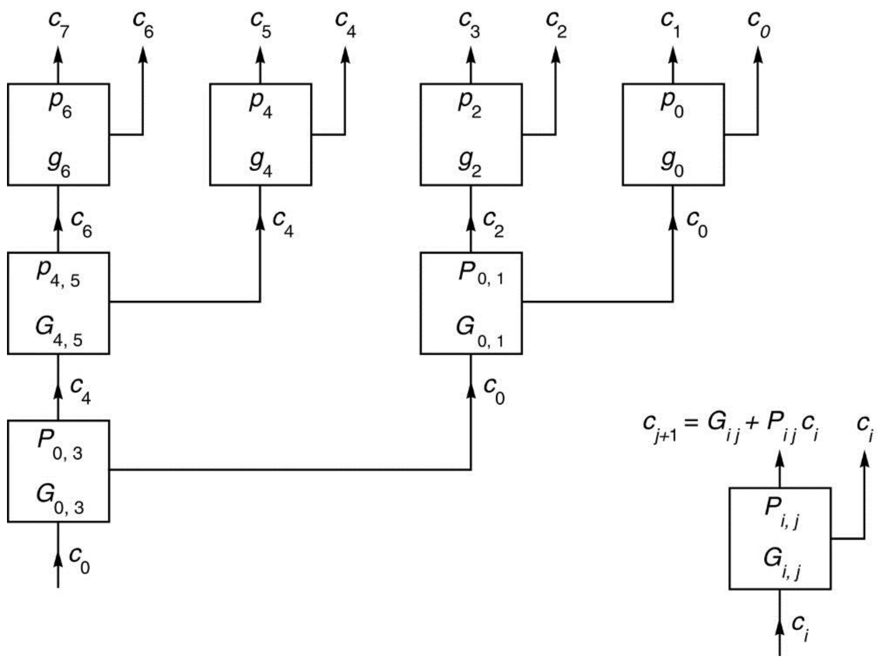


Figure J.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

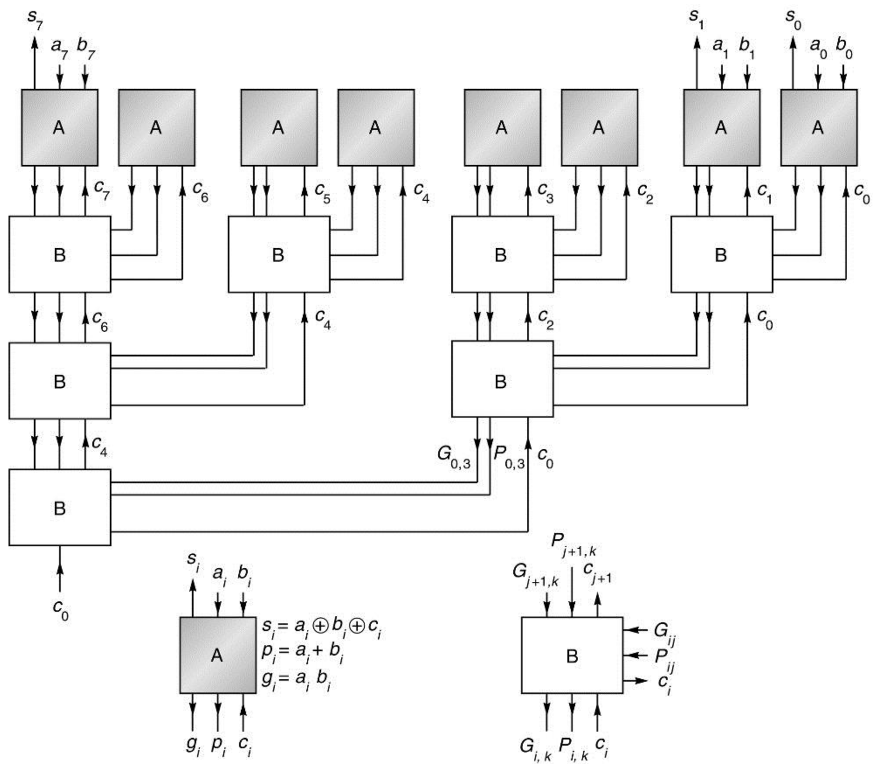


Figure J.17 Complete carry-lookahead tree adder. This is the combination of Figures J.15 and J.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

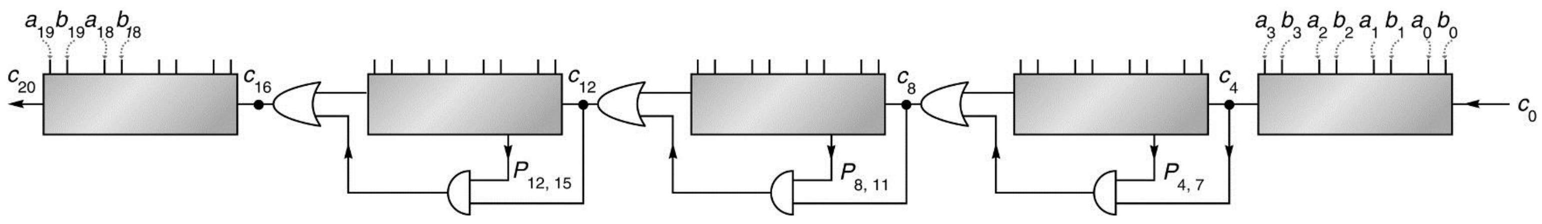


Figure J.18 Carry-skip adder. This is a 20-bit carry-skip adder (n = 20) with each block 4 bits wide (k = 4).

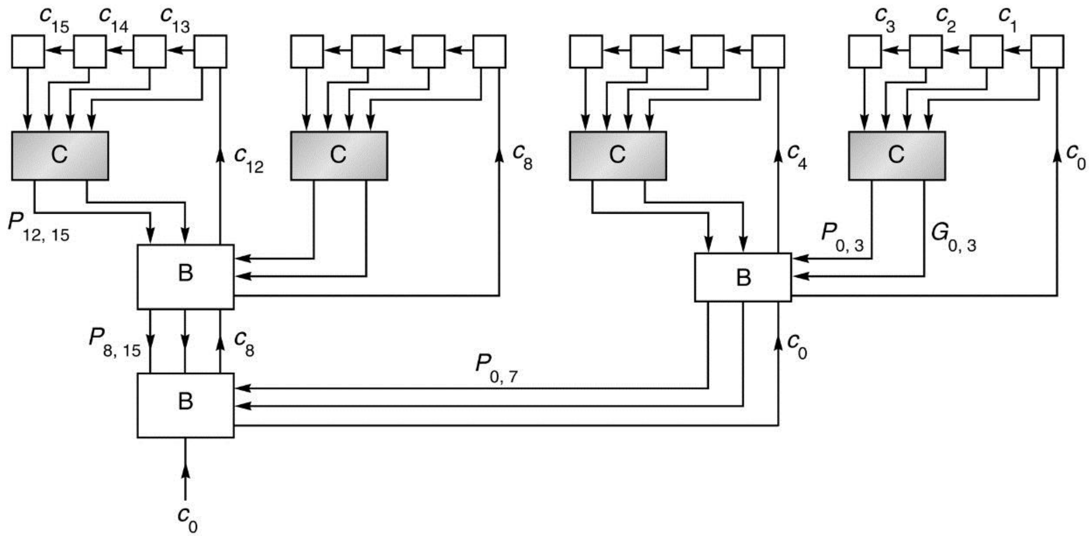


Figure J.19 Combination of CLA and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

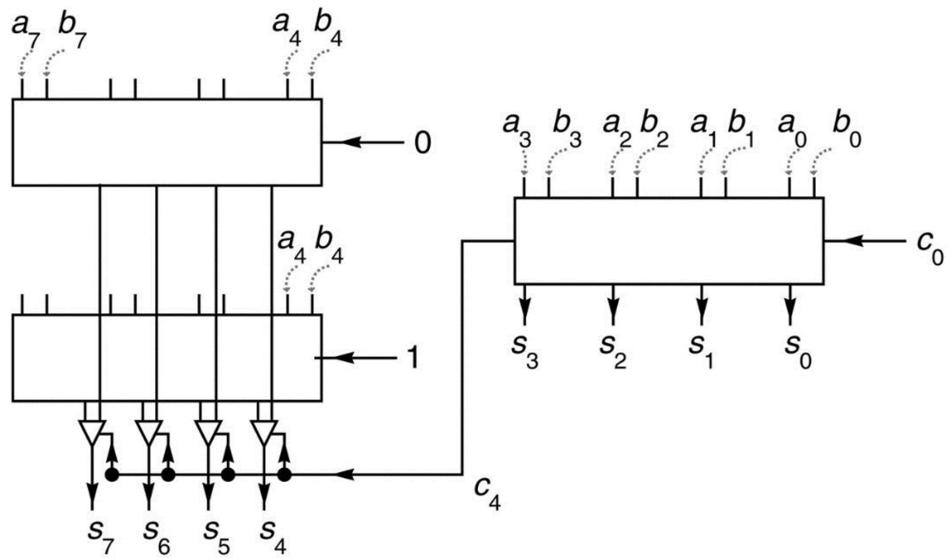


Figure J.20 Simple carry-select adder. At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.

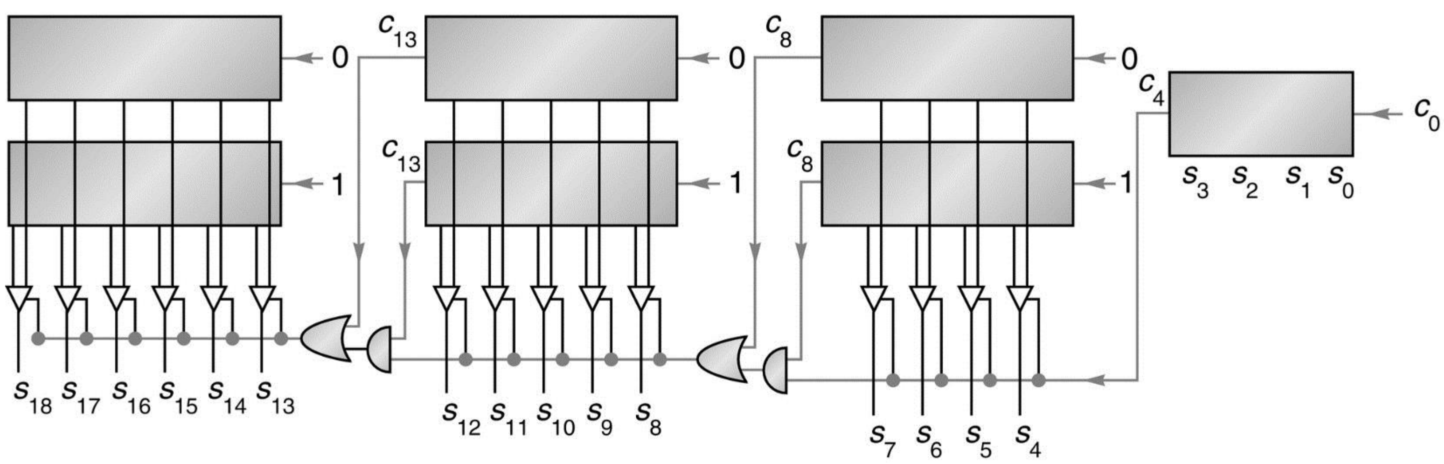


Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

P	A	
00000	1000	Divide 8 = 1000 by 3 = 0011. B contains 0011.
00010	0000	Step 1: B had two leading 0 s, so shift left by 2. B now contains 1100.
		Step 2.1: Top three bits are equal. This is case (a), so
00100	000**0**	set $q_0 = 0$ and shift.
		Step 2.2: Top three bits not equal and $P \geq 0$ is case (c), so
01000	00**01**	set $q_1 = 1$ and shift.
+ 10100		Subtract B.
11100	00**01**	Step 2.3: Top bits equal is case (a), so
11000	0**010**	set $q_2 = 0$ and shift.
		Step 2.4: Top three bits unequal is case (b), so
10000	**010$\bar{1}$**	set $q_3 = -1$ and shift.
+ 01100		Add B.
11100		Step 3. remainder is negative so restore it and subtract 1 from q.
+ 01100		
01000		Must undo the shift in step 1, so right-shift by 2 to get true remainder. Remainder = 10, quotient = $010\bar{1} - 1 = 0010$.

Figure J.23 SRT division of $1000_2/0011_2$. The quotient bits are shown in bold, using the notation 1 for − 1.

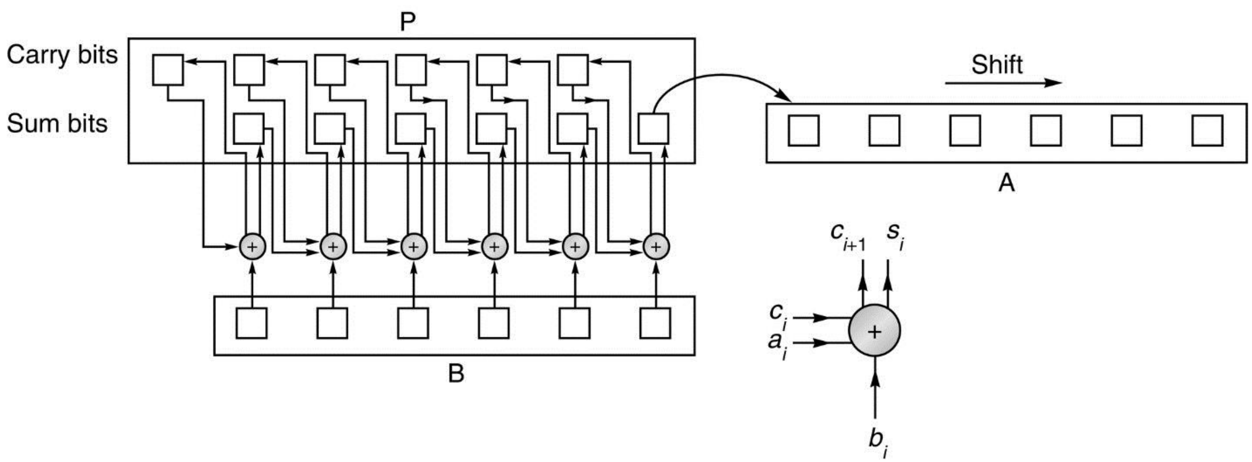


Figure J.24 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

Low-order bits of A		Last bit shifted out	
$2i+1$	$2i$	$2i-1$	Multiple
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

Figure J.25 Multiples of *b* to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is − *b*, obtained from the second-to-last row of the table.

P	A	L	
00000	1001		Multiply $-7 = 1001$ times $-5 = 1011$. B contains 1011.
+ 11011			Low-order bits of A are 0, 1; $L = 0$, so add B.
11011	1001		
11110	1110	0	Shift right by two bits, shifting in 1 s on the left.
+ 01010			Low-order bits of A are 1, 0; $L = 0$, so add $-2b$.
01000	1110	0	
00010	0011	1	Shift right by two bits.
			Product is $35 = 0100011$.

Figure J.26 Multiplication of − 7 times − 5 using radix-4 Booth recoding. The column labeled L contains the last bit shifted out the right end of A.

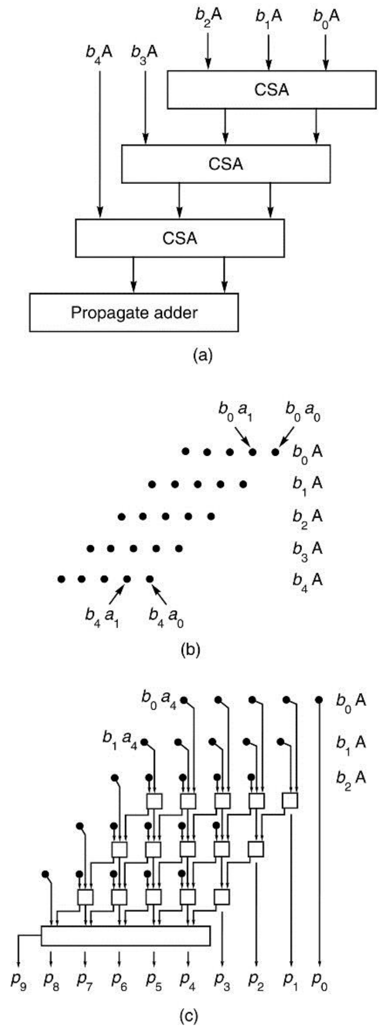


Figure J.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

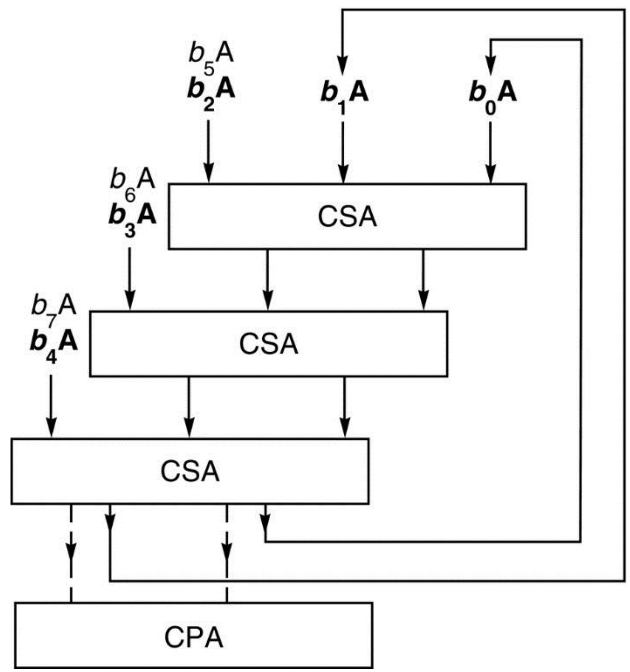


Figure J.28 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure J.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.

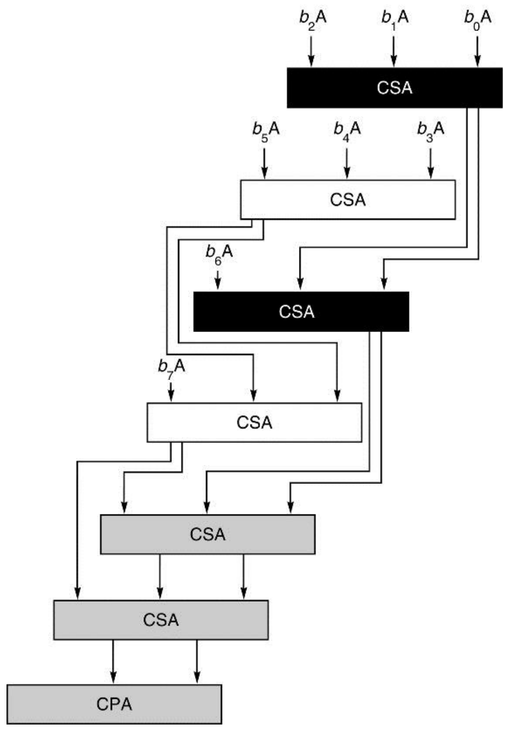


Figure J.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

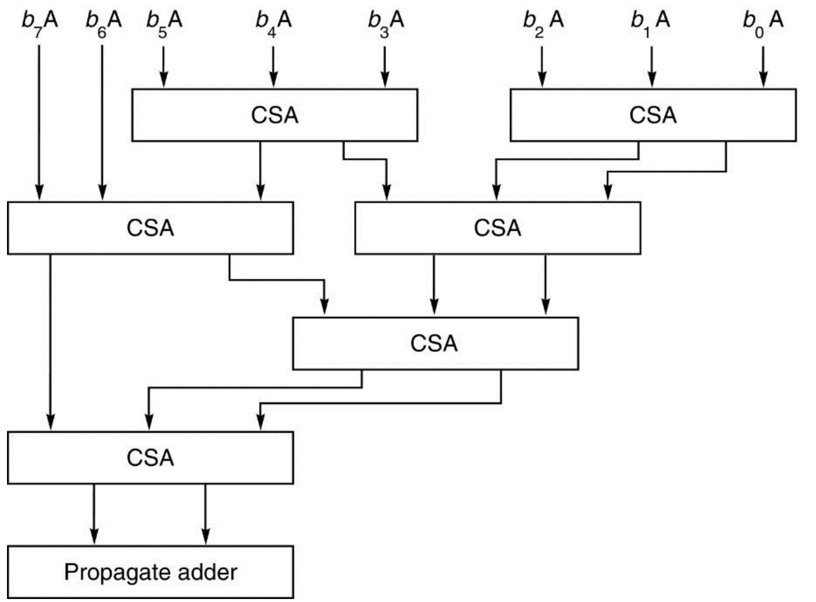


Figure J.30 Wallace tree multiplier. An example of a multiply tree that computes a product in 0(log *n*) steps.

1	1	$\overline{1}$	0	$\overline{1}\ x$		$\overline{1}\ x$	
$+\ 1$	$+\ \overline{1}$	$+\ \overline{1}$	$+\ 0$	$+\ 0\ y$		$+\ 0\ y$	
$1\ 0$	$0\ 0$	$\overline{1}\ 0$	$0\ 0$	$1\ \overline{1}$	if $x \geq 0$ and	$1\ \overline{1}$	if $x \geq 0$ and
				$0\ 1$	$y \geq 0$ otherwise	$\overline{1}\ 1$	$y \geq 0$ otherwise

Figure J.31 Signed-digit addition table. The leftmost sum shows that when computing 1 + 1, the sum bit is 0 and the carry bit is 1.

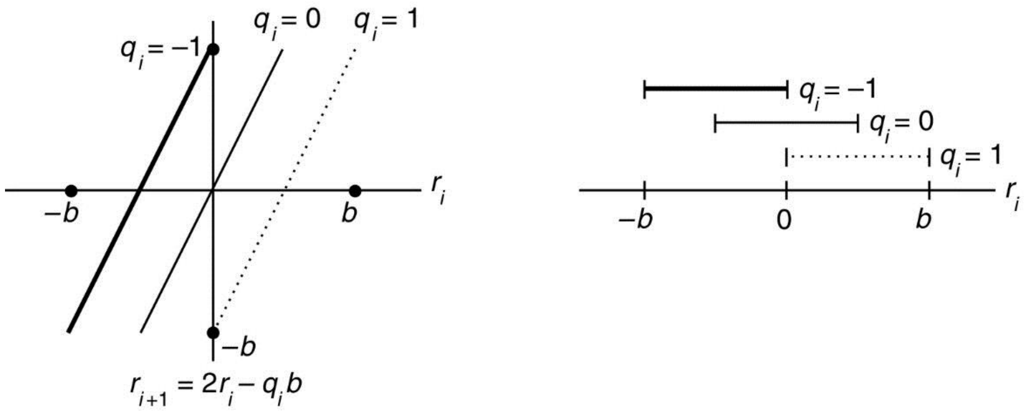


Figure J.32 Quotient selection for radix-2 division. The *x*-axis represents the *i*th remainder, which is the quantity in the (P,A) register pair. The *y*-axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

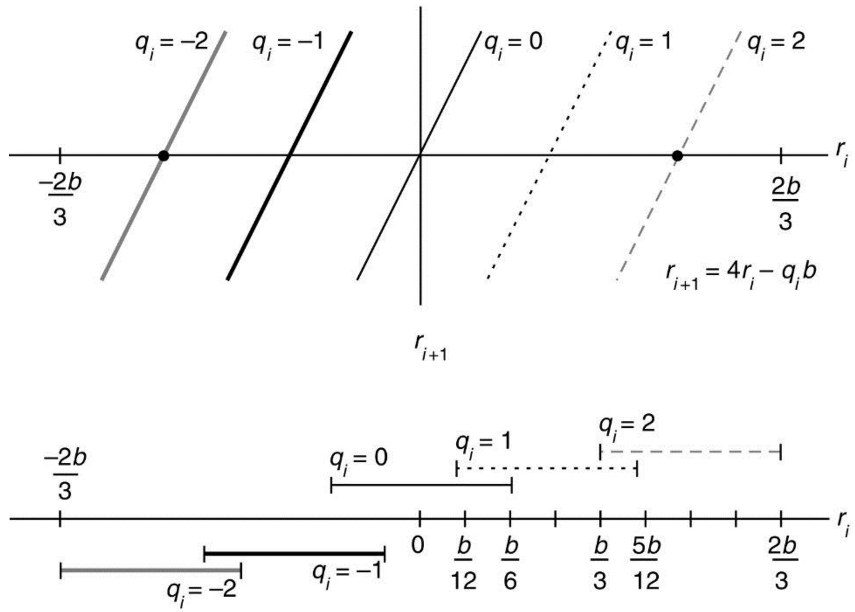


Figure J.33 Quotient selection for radix-4 division with quotient digits − 2, − 1, 0, 1, 2.

b	Range of P		q	b	Range of P		q
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

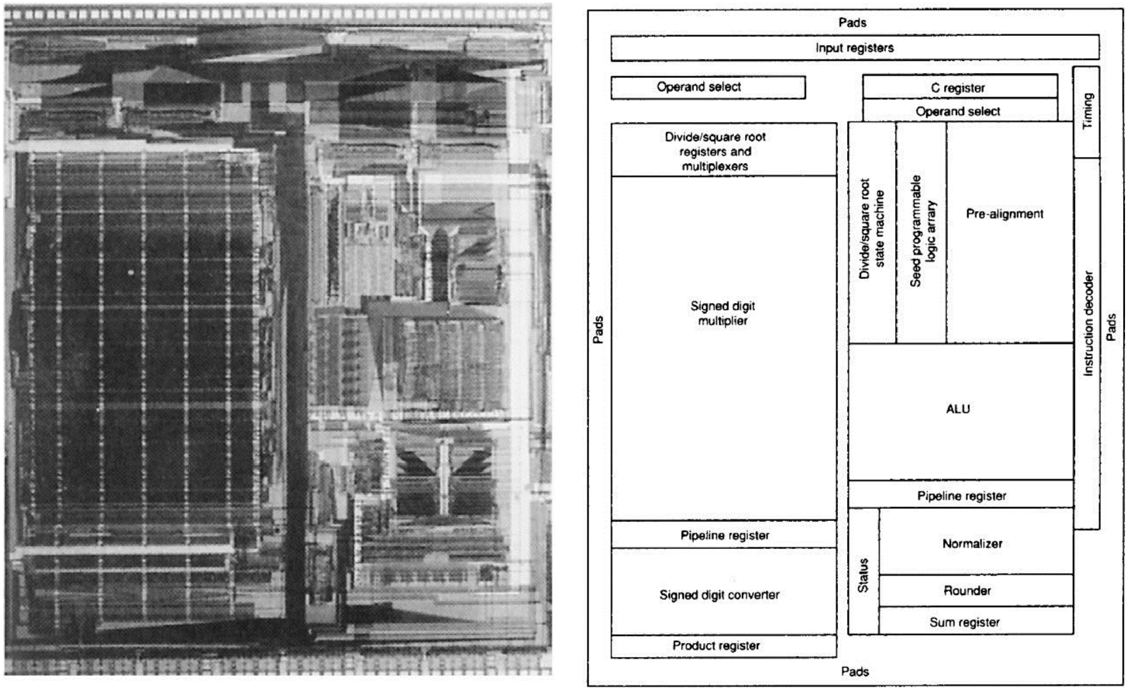
Figure J.34 Quotient digits for radix-4 SRT division with a propagate adder. The top row says that if the high-order 4 bits of *b* are $1000_2 = 8$, and if the top 6 bits of P are between $110100_2 = -12$ and $111001_2 = -7$, then -2 is a valid quotient digit.

P	A		
000000000	10010101	Divide 149 by 5. B contains 00000101.	
000010010	10100000	Step 1:	B had 5 leading 0s, so shift left by 5. B now contains 10100000, so use $b = 10$ section of table.
001001010	100000**0**	Step 2.1:	Top 6 bits of P are 2, so shift left by 2. From table, can pick q to be 0 or 1. Choose $q_0 = 0$.
		Step 2.2:	Top 6 bits of P are 9, so shift left 2. $q_1 = 2$.
100101010	0000**02**		
+ 011000000			Subtract $2b$.
111101010	0000**02**	Step 2.3:	Top bits $= -3$, so shift left 2. Can pick 0 or -1 for q, pick $q_2 = 0$.
110101000	00**020**		
		Step 2.4:	Top bits $= -11$, so shift left 2. $q_3 = -2$.
010100000	**0202**		
+ 101000000			Add $2b$.
111100000		Step 3:	Remainder is negative, so restore by adding b and subtract 1 from q.
+ 010100000			
010000000		Answer:	$q = 020\overline{2} - 1 = 29$
			To get remainder, undo shift in step 1 so $\text{remainder} = 010000000 >> 5 = 4$.

Figure J.35 Example of radix-4 SRT division. Division of 149 by 5.

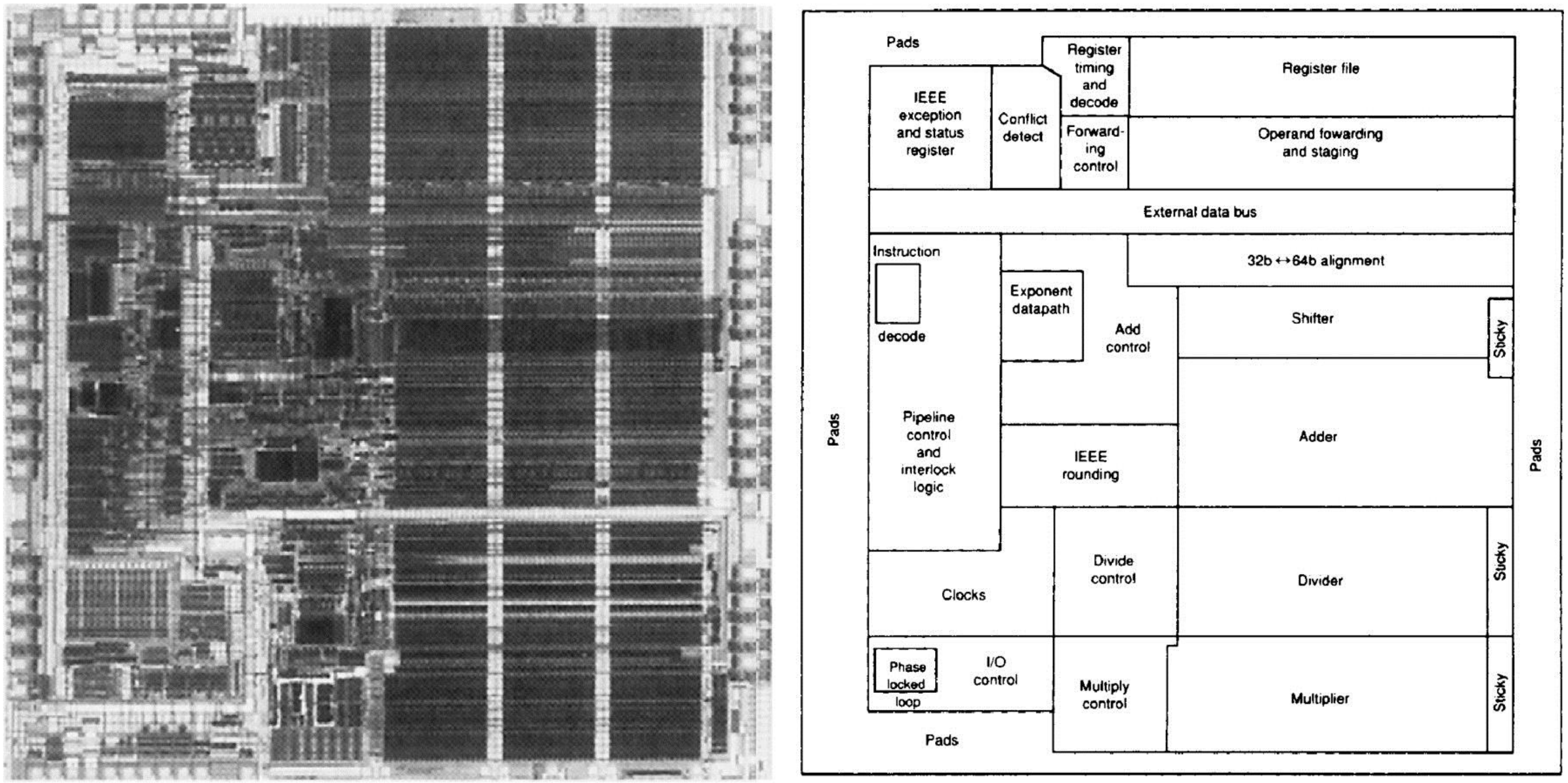
Features	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/square root	–	30	14

Figure J.36 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.



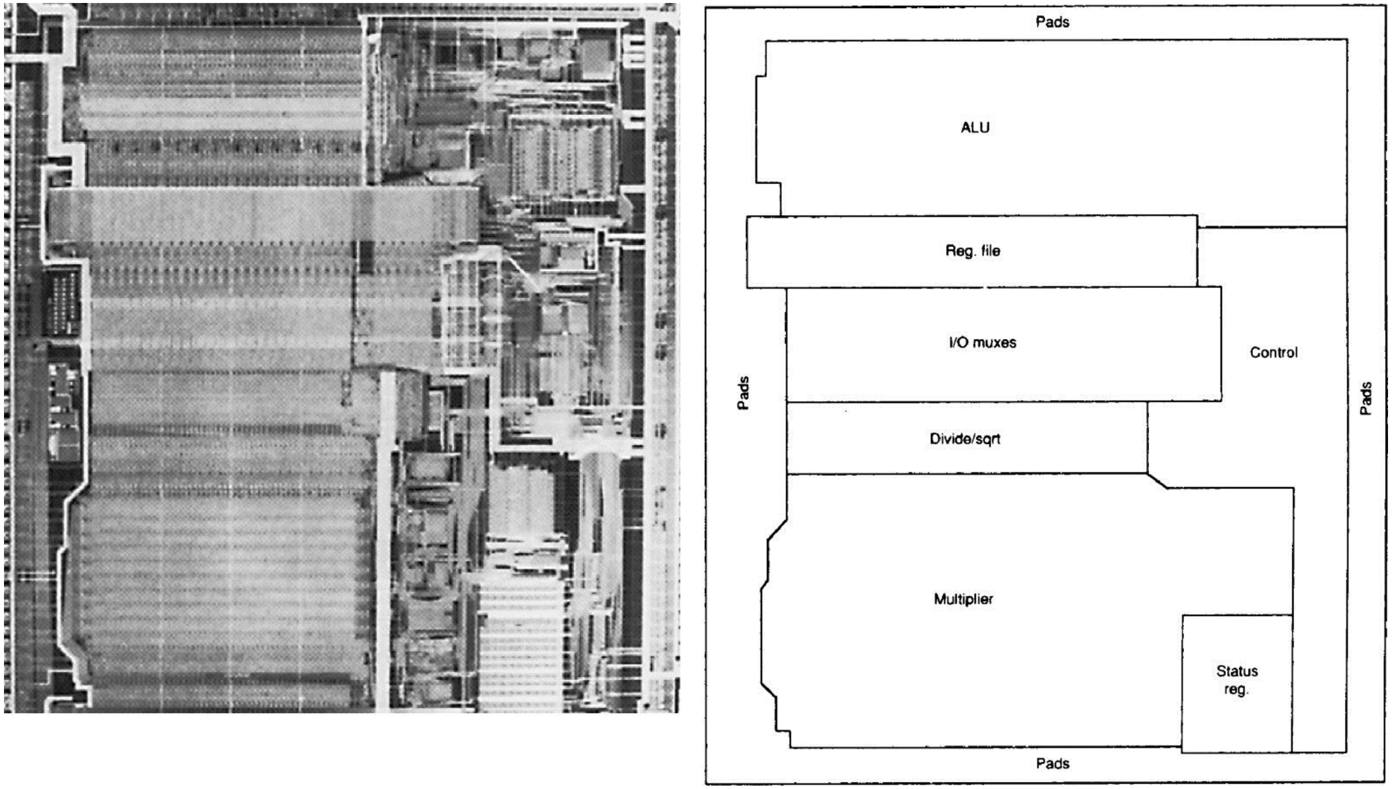
TI 8847

Figure J.37 Chip layout for the TI 8847, MIPS R3010, and Weitek 3364. In the left-hand columns are the photomicrographs; the right-hand columns show the corresponding floor plans.



MIPS R3010

Figure J.37 (Continued)



Weitek 3364

Figure J.37 (Continued)