Principles of Computer Architecture

Miles Murdocca and Vincent Heuring

Chapter 3: Arithmetic

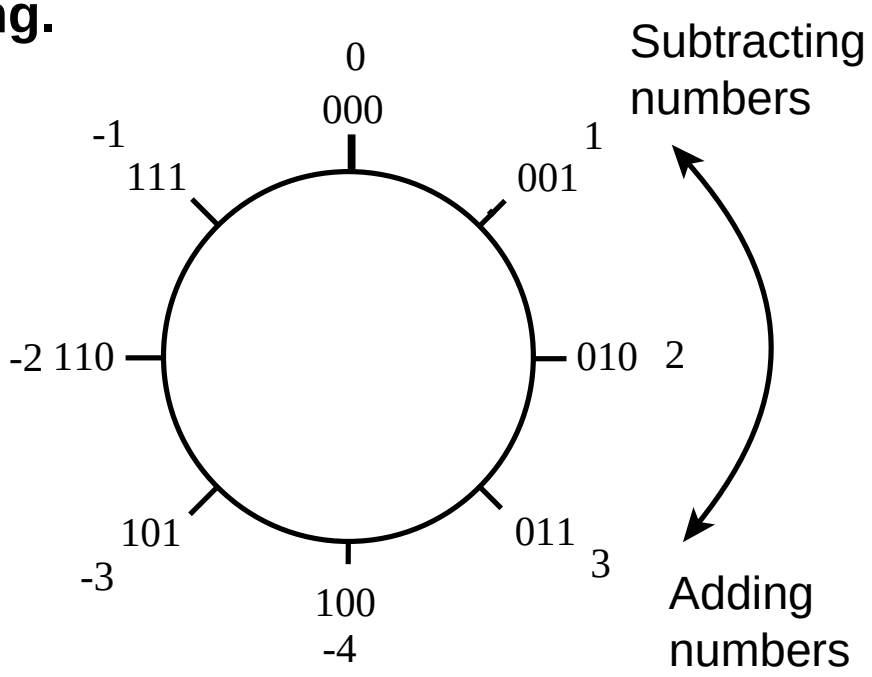
Chapter Contents

Computer Arithmetic

- Using number representations from Chapter 2, we will explore four basic arithmetic operations: addition, subtraction, multiplication, division.
- Significant issues include: fixed point vs. floating point arithmetic, overflow and underflow, handling of signed numbers, and performance.
- We look first at fixed point arithmetic, and then at floating point arithmetic.

Number Circle for 3-Bit Two's Complement Numbers

- **Numbers can be added or subtracted by traversing the number circle clockwise for addition and counterclockwise for subtraction.**
- **Overflow occurs when a transition is made from +3 to -4 while proceeding around the number circle when adding, or from -4 to +3 while subtracting.**



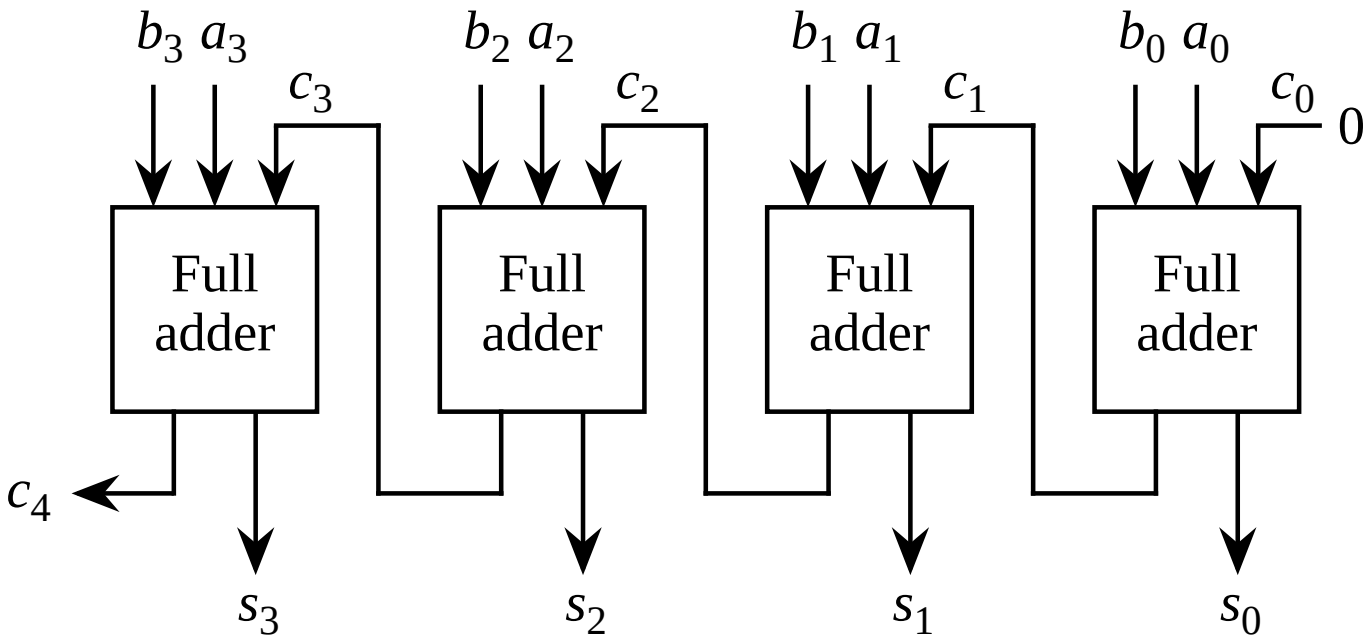
Overflow

- **Overflow occurs when adding two positive numbers produces a negative result, or when adding two negative numbers produces a positive result. Adding operands of unlike signs never produces an overflow.**
- **Notice that discarding the carry out of the most significant bit during two's complement addition is a normal occurrence, and does not by itself indicate overflow.**
- **As an example of overflow, consider adding $(80 + 80 = 160)_{10}$, which produces a result of -96_{10} in an 8-bit two's complement format:**

```
  01010000 =  80
+ 01010000 =  80
----------
  10100000 = -96 (not 160 because the sign bit is 1.)
```

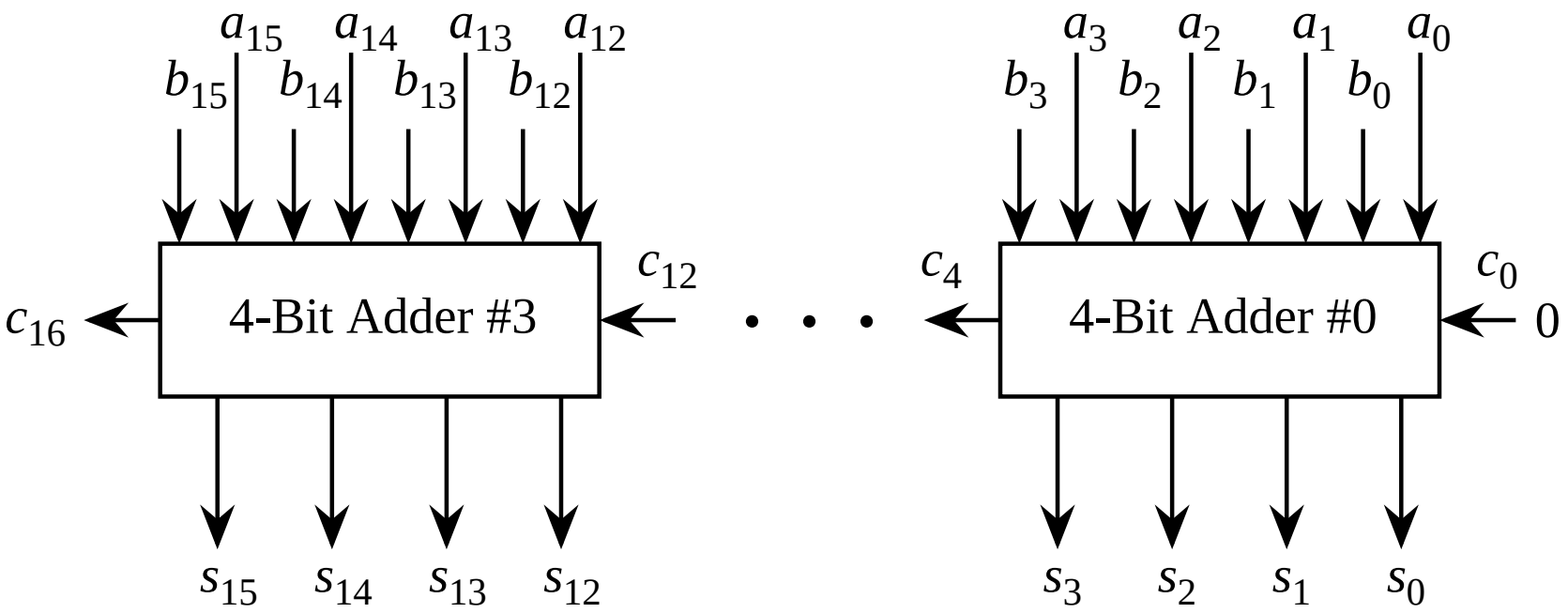
Ripple Carry Adder

- **Two binary numbers *A* and *B* are added from right to left, creating a sum and a carry at the outputs of each full adder for each bit position.**



Constructing Larger Adders

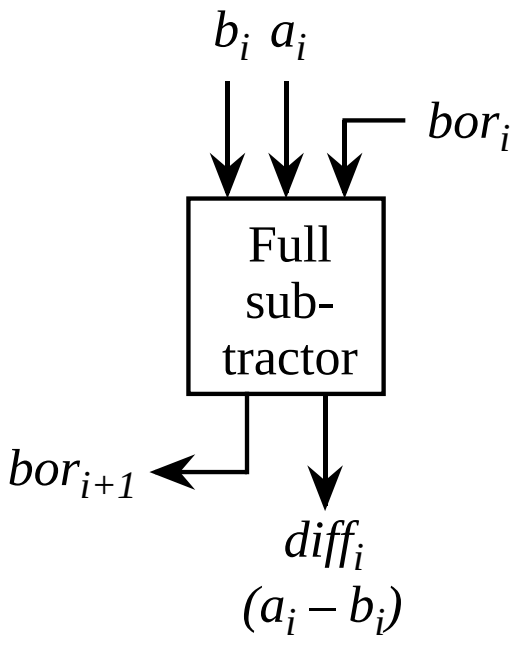
- **A 16-bit adder can be made up of a cascade of four 4-bit ripple-carry adders.**



Full Subtractor

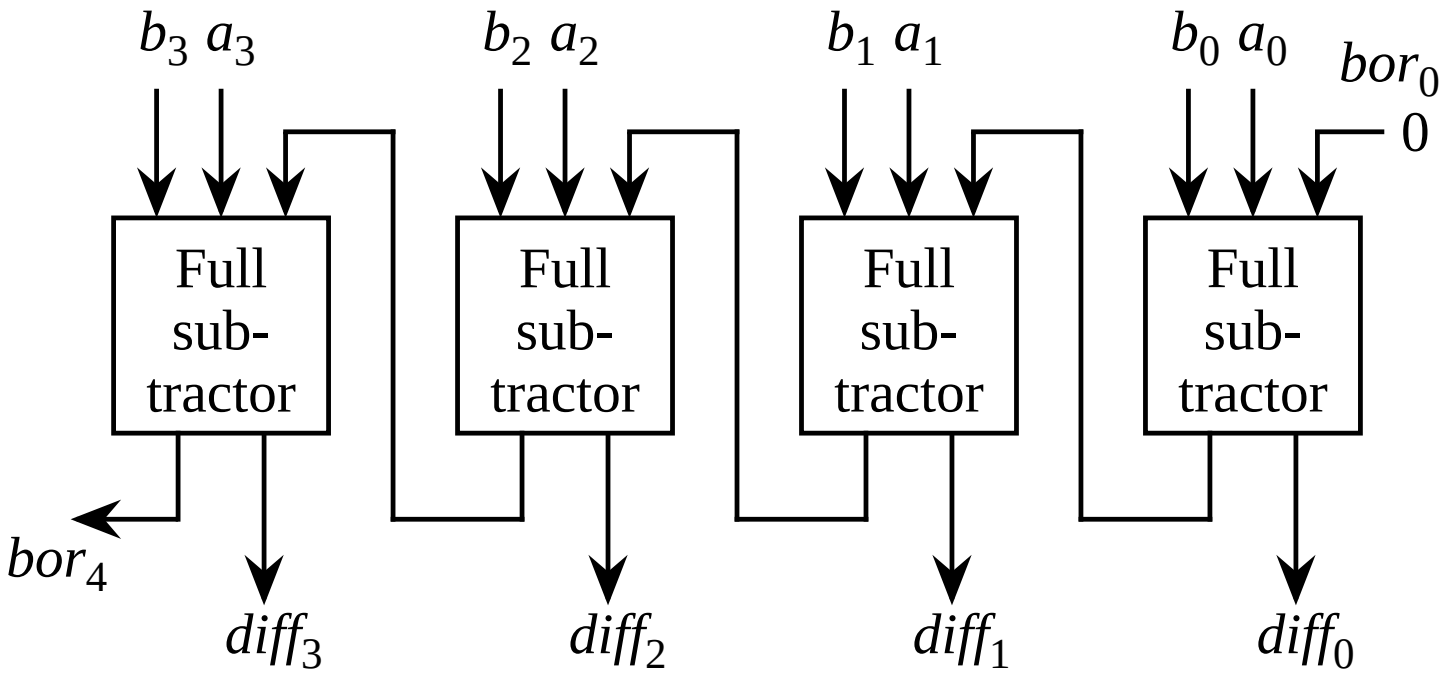
- **Truth table and schematic symbol for a ripple-borrow subtractor:**

a_i	b_i	bor_i	$diff_i$	bor_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



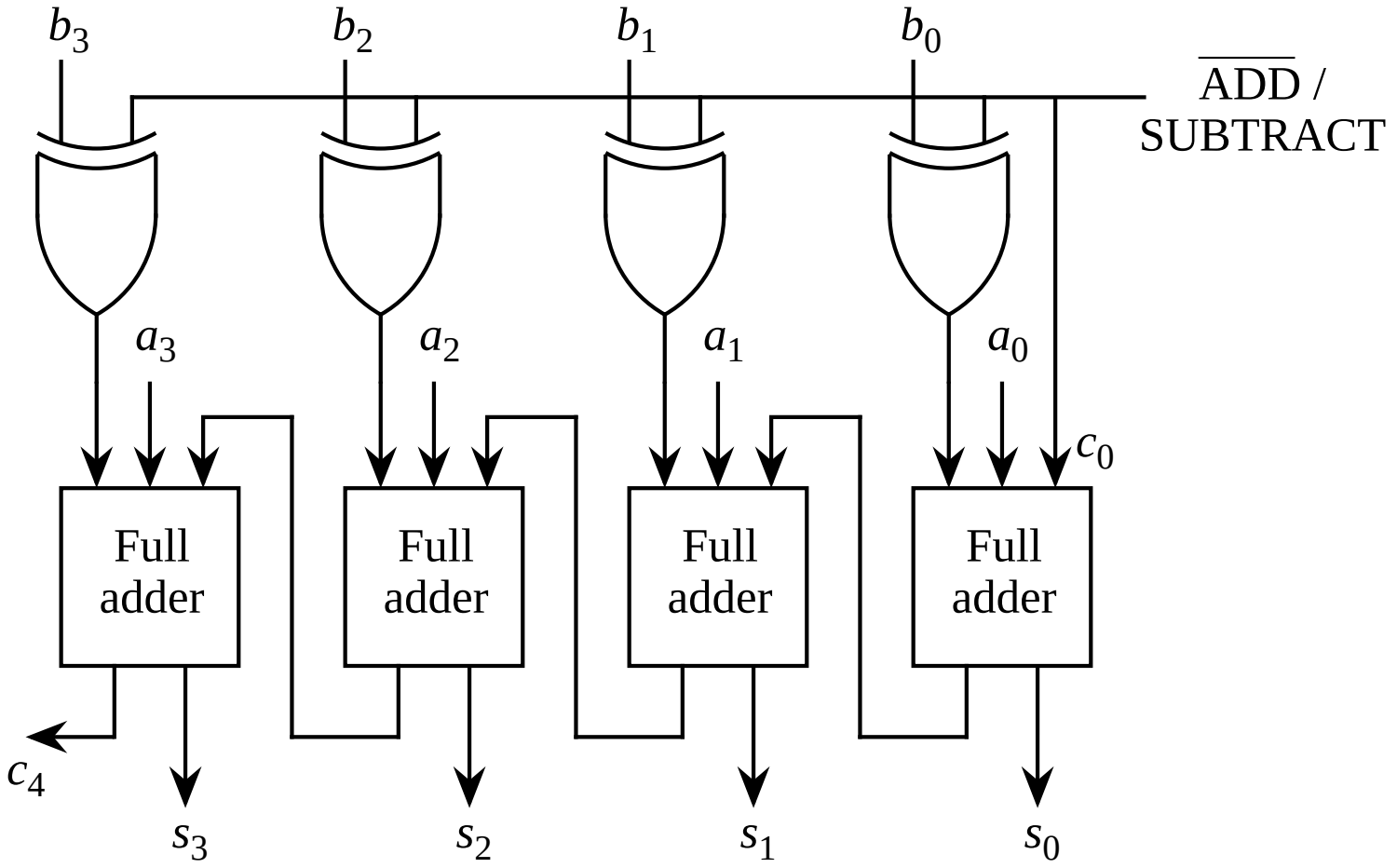
Ripple-Borrow Subtractor

- **A ripple-borrow subtractor can be composed of a cascade of full subtractors.**
- **Two binary numbers *A* and *B* are subtracted from right to left, creating a difference and a borrow at the outputs of each full subtractor for each bit position.**



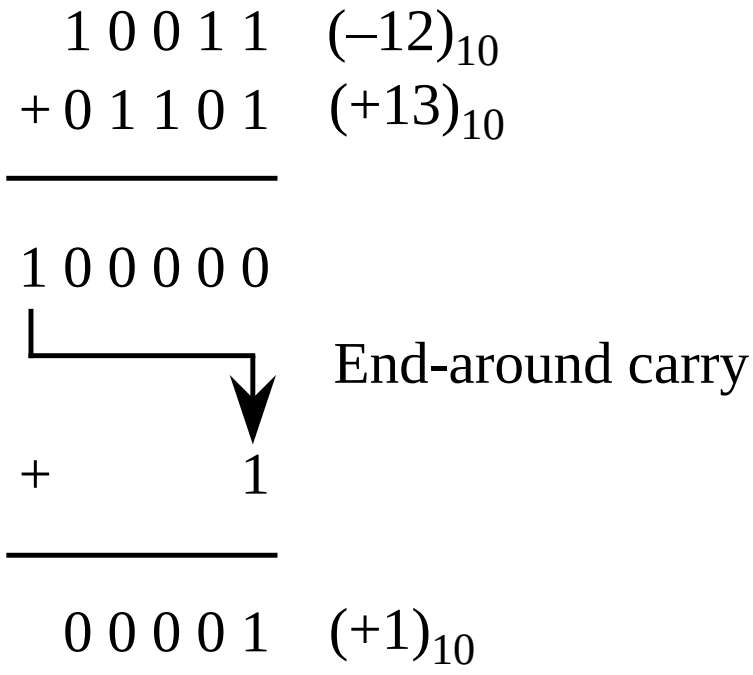
Combined Adder/Subtractor

- **A single ripple-carry adder can perform both addition and subtraction, by forming the two's complement negative for *B* when subtracting. (Note that +1 is added at c_0 for two's complement.)**



One's Complement Addition

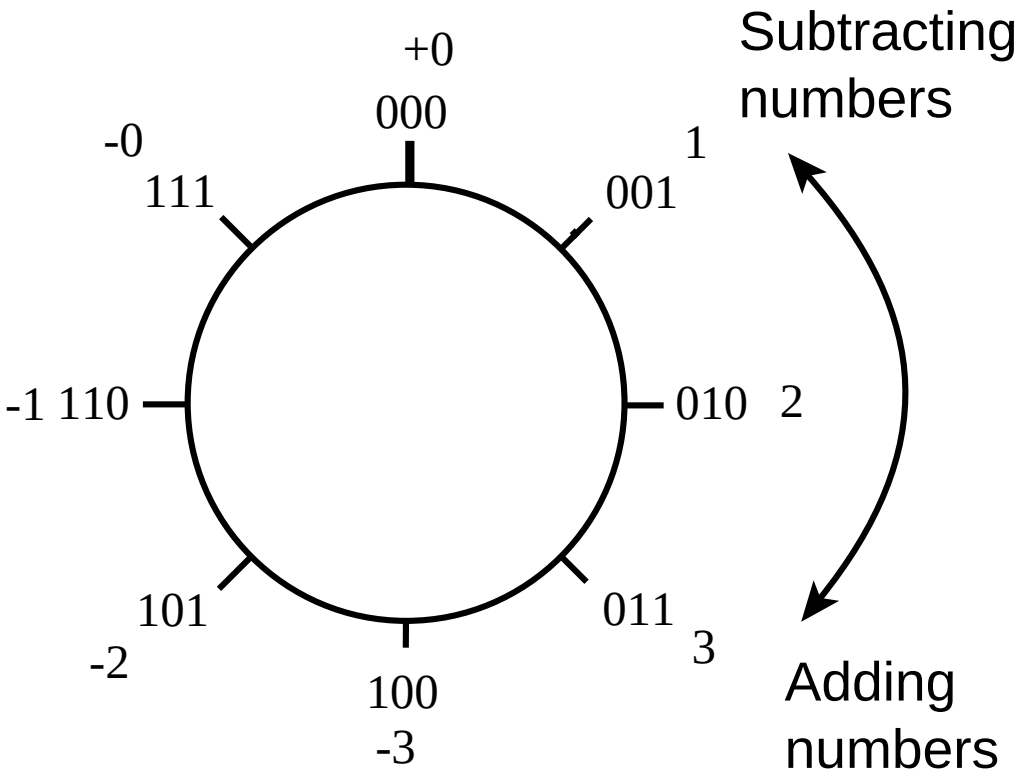
- **An example of one's complement integer addition with an end-around carry:**



- **The end-around carry is needed because there are two representations for 0 in one's complement. Both representations for 0 are visited when one or both operands are negative.**

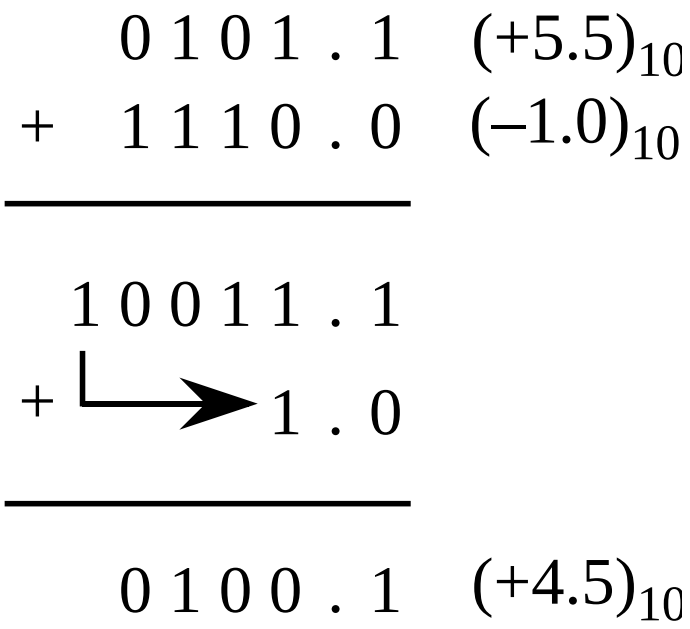
Number Circle (Revisited)

- **Number circle for a three-bit signed one's complement representation. Notice the two representations for 0.**



End-Around Carry for Fractions

- **The end-around carry complicates one's complement addition for non-integers, and is generally not used for this situation.**
- **The issue is that the distance between the two representations of 0 is 1.0, whereas the rightmost fraction position is less than 1.**



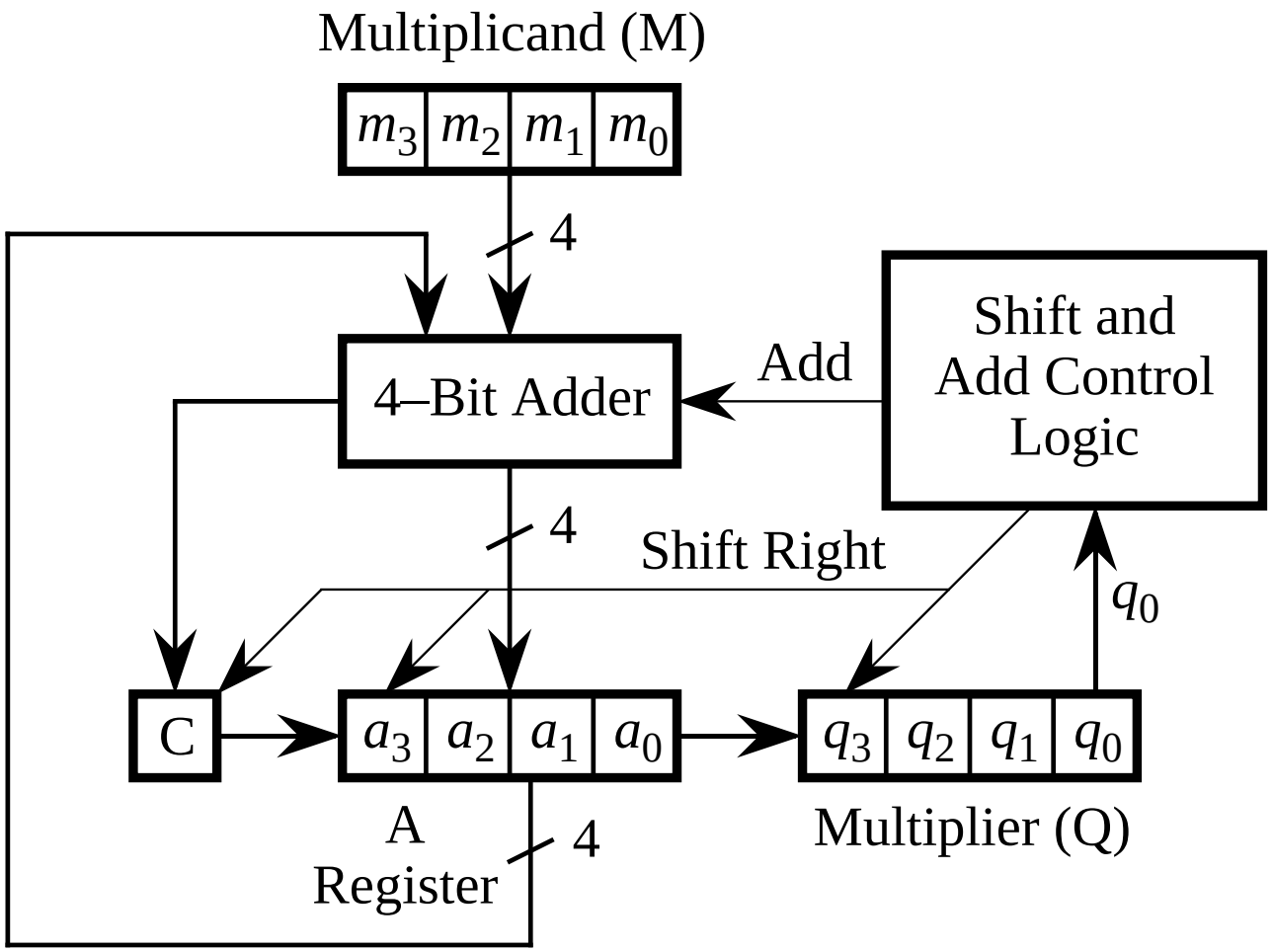
Multiplication Example

- **Multiplication of two 4-bit unsigned binary integers produces an 8-bit result.**

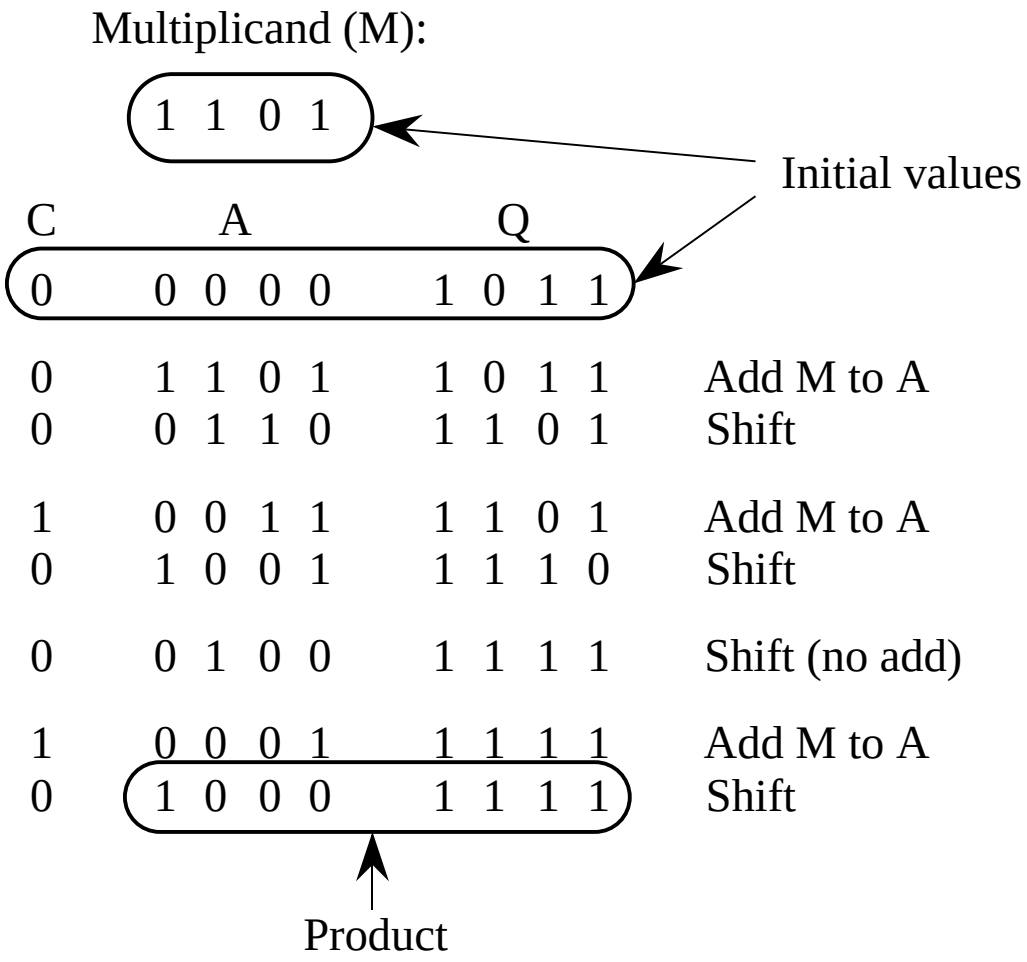
			1	1	0	1	$(13)_{10}$	Multiplicand M
		×	1	0	1	1	$(11)_{10}$	Multiplier Q
			1	1	0	1		
		1	1	0	1			Partial products
	0	0	0	0				
1	1	0	1					
1	0	0	0	1	1	1	1	$(143)_{10}$	Product P

- **Multiplication of two 4-bit signed binary integers produces only a 7-bit result (each operand reduces to a sign bit and a 3-bit magnitude for each operand, producing a sign-bit and a 6-bit result).**

A Serial Multiplier



Example of Multiplication Using Serial Multiplier

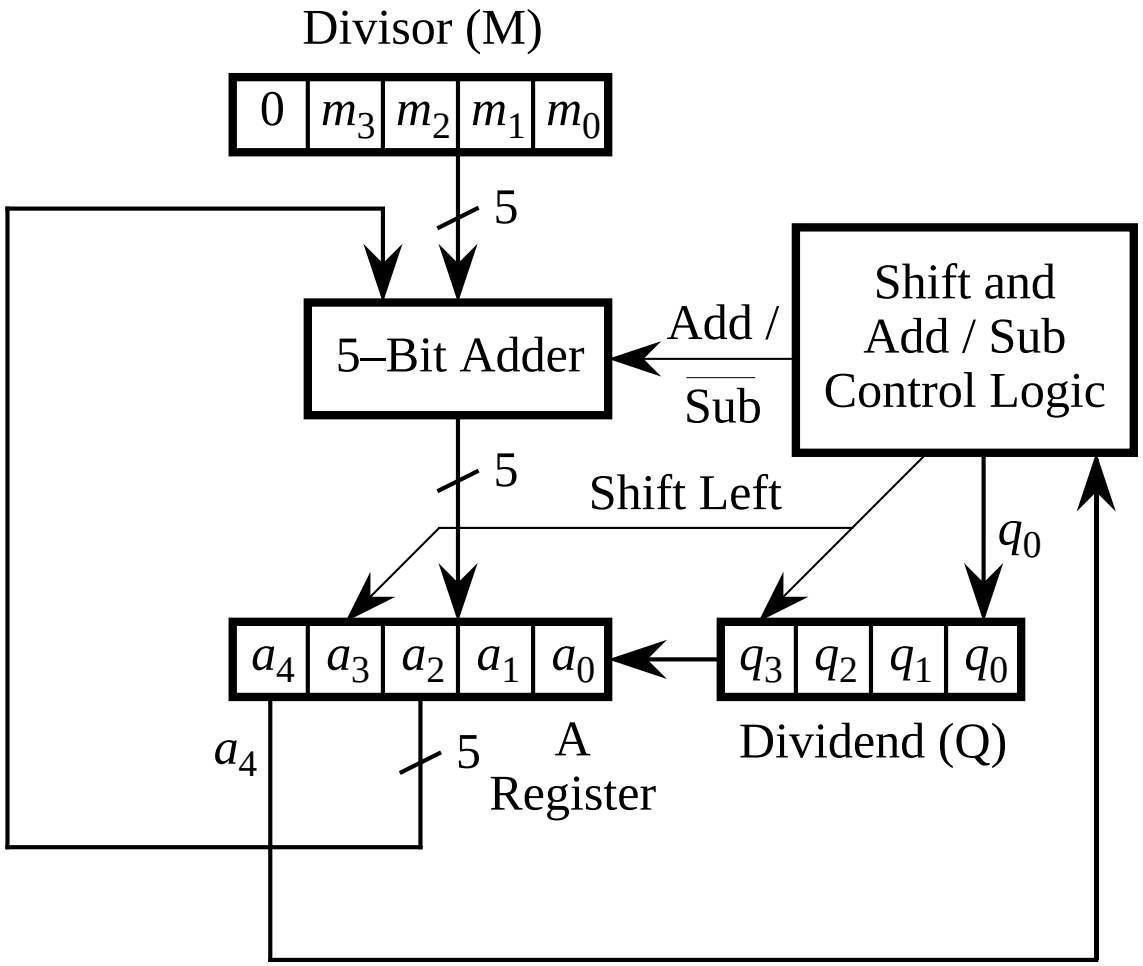


Example of Base 2 Division

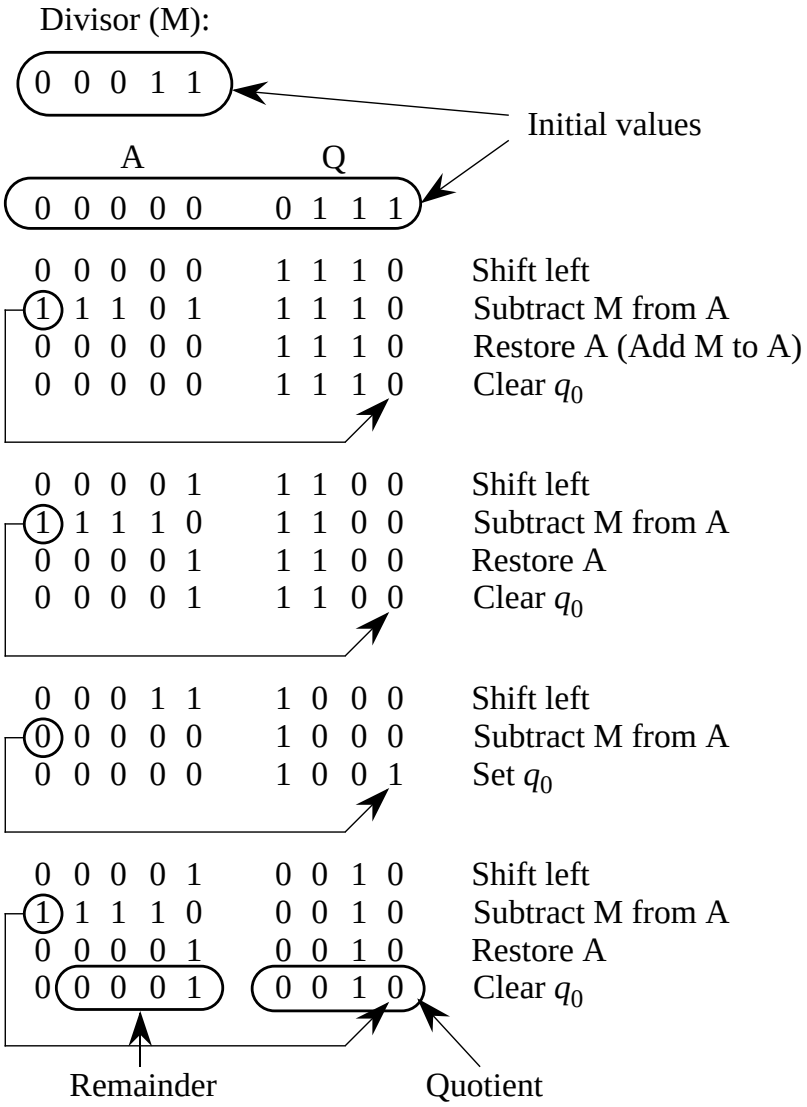
- $(7 / 3 = 2)_{10}$ with a remainder R of 1.
- Equivalently, $(0111 / 11 = 10)_2$ with a remainder R of 1.

```
        0 0 1 0  R 1
    ┌──────────────
1 1 │ 0 1 1 1
      1 1
    ──────
        0 1
```

Serial Divider



Division Example Using Serial Divider



Multiplication of Signed Integers

- **Sign extension to the target word size is needed for the negative operand(s).**
- **A target word size of 8 bits is used here for two 4-bit signed operands, but only a 7-bit target word size is needed for the result.**

$$
\begin{array}{rcl}
 & 1\ 1\ 1\ 1 & (-1)_{10} \\
\times & 0\ 0\ 0\ 1 & (+1)_{10} \\
\hline
 & 1\ 1\ 1\ 1 & \\
 & 0\ 0\ 0\ 0 & \\
 & 0\ 0\ 0\ 0 & \\
0\ 0\ 0\ 0 & & \\
\hline
0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 & & (+15)_{10}
\end{array}
$$

(Incorrect; result should be –1)

$$
\begin{array}{rcl}
1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & & (-1)_{10} \\
\times\ \ 0\ 0\ 0\ 1 & & (+1)_{10} \\
\hline
1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & & \\
0\ 0\ 0\ 0\ 0\ 0\ 0 & & \\
0\ 0\ 0\ 0\ 0\ 0 & & \\
0\ 0\ 0\ 0\ 0 & & \\
\hline
1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 & & (-1)_{10}
\end{array}
$$

Carry-Lookahead Addition

$$s_i = \bar{a}_i\bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i\bar{b}_i\bar{c}_i + a_i b_i c_i$$

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

$$c_{i+1} = a_i b_i + (a_i + b_i)c_i$$

$$c_{i+1} = G_i + P_i c_i$$

- **Carries are represented in terms of G_i (generate) and P_i (propagate) expressions.**

$$G_i = a_i b_i \quad \text{and} \quad P_i = a_i + b_i$$

$$c_0 = 0$$

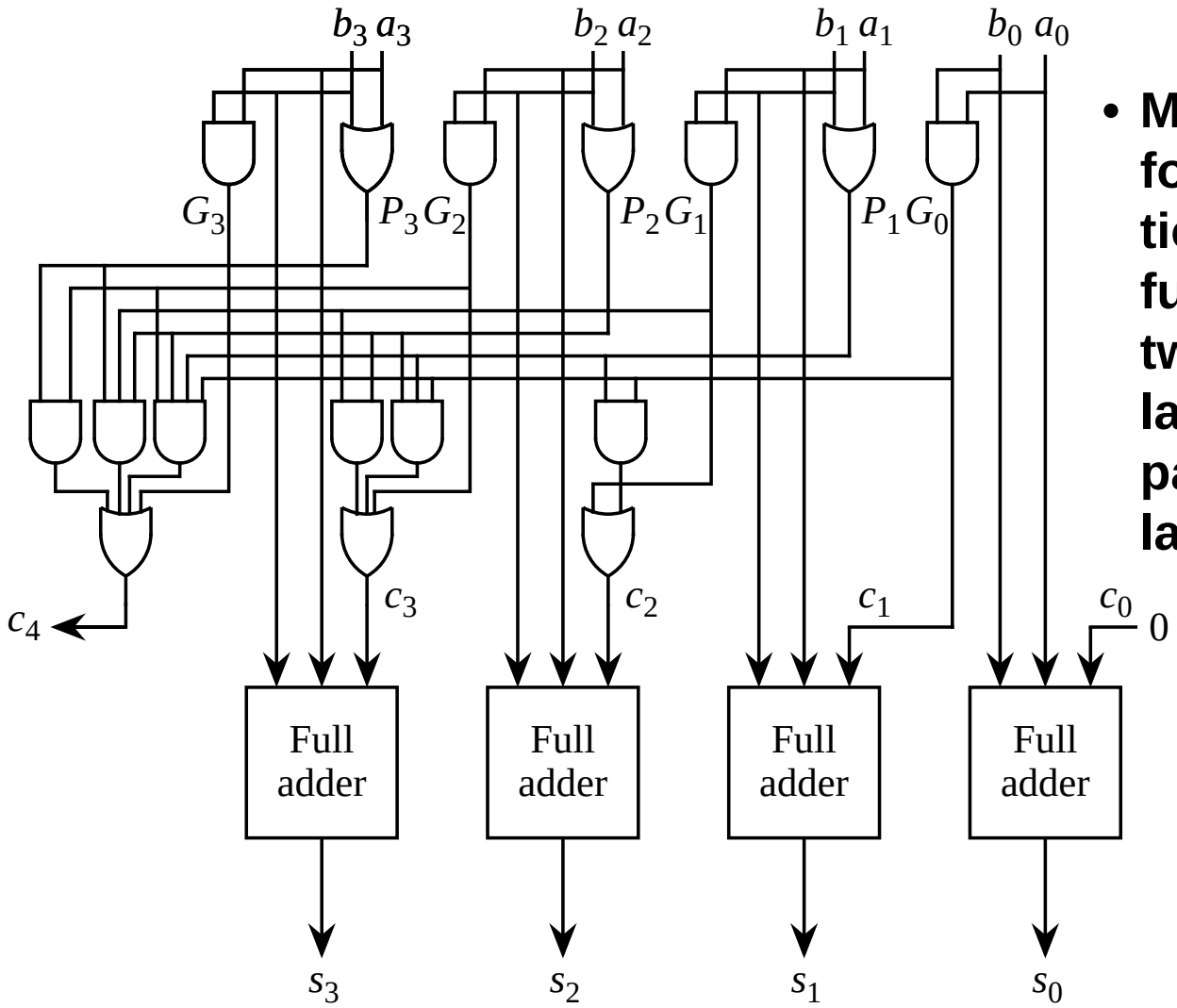
$$c_1 = G_0$$

$$c_2 = G_1 + P_1 G_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

Carry Lookahead Adder



- **Maximum gate delay for the carry generation is only 3. The full adders introduce two more gate delays. Worst case path is 5 gate delays.**

Floating Point Arithmetic

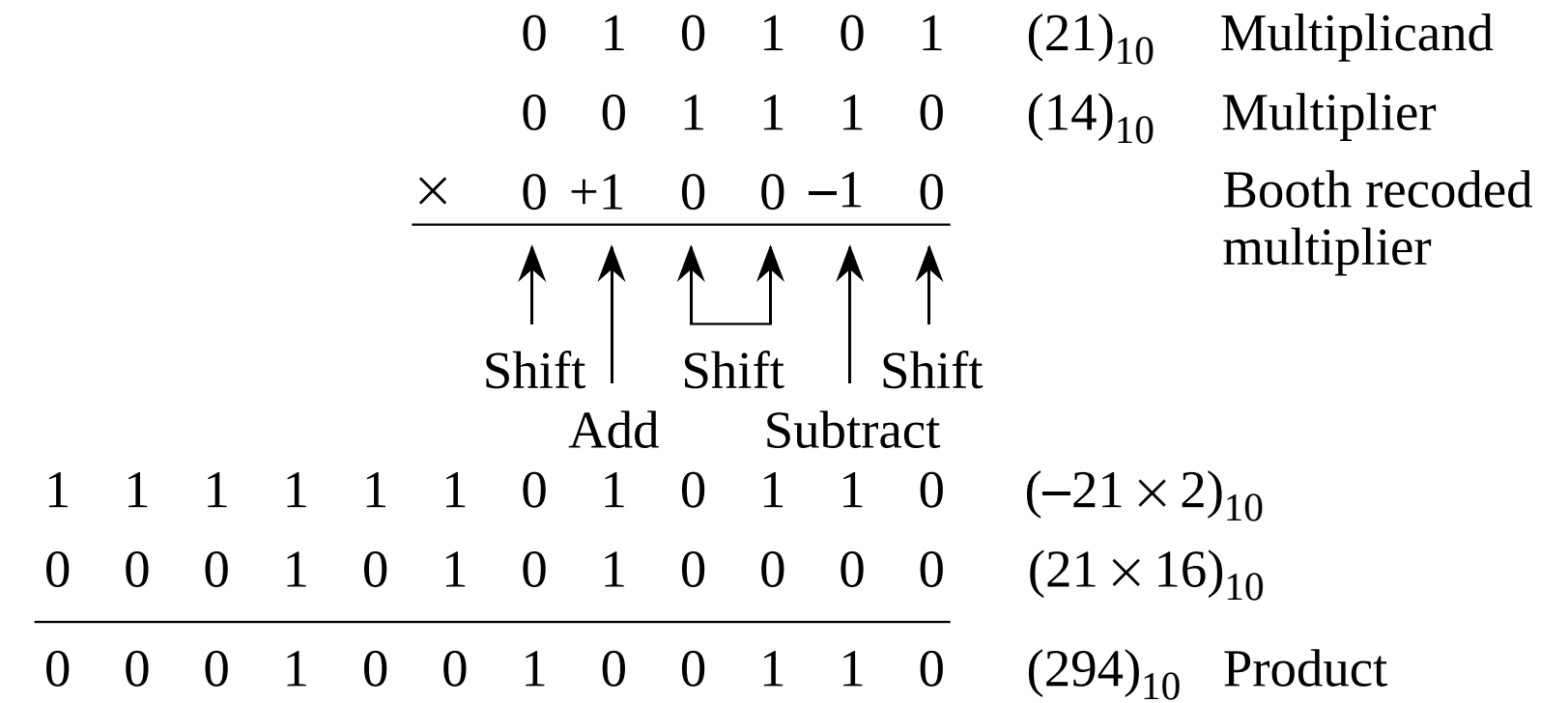
- Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands.
- The exponents of the operands must be made equal for addition and subtraction. The fractions are then added or subtracted as appropriate, and the result is normalized.
- Ex: Perform the floating point operation: $(.101 \times 2^3 + .111 \times 2^4)_2$
- Start by adjusting the *smaller* exponent to be equal to the larger exponent, and adjust the fraction accordingly. Thus we have $.101 \times 2^3 = .010 \times 2^4$, losing $.001 \times 2^3$ of precision in the process.
- The resulting sum is $(.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5$, and rounding to three significant digits, $.100 \times 2^5$, and we have lost another 0.001×2^4 in the rounding process.

Floating Point Multiplication/Division

- Floating point multiplication/division are performed in a manner similar to floating point addition/subtraction, except that the sign, exponent, and fraction of the result can be computed separately.
- Like/unlike signs produce positive/negative results, respectively. Exponent of result is obtained by adding exponents for multiplication, or by subtracting exponents for division. Fractions are multiplied or divided according to the operation, and then normalized.
- Ex: Perform the floating point operation: $(+.110 \times 2^5) / (+.100 \times 2^4)_2$
- The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is 5 – 4 = 1.
- We divide fractions, producing the result: 110/100 = 1.10.
- Putting it all together, the result of dividing $(+.110 \times 2^5)$ by $(+.100 \times 2^4)$ produces $(+1.10 \times 2^1)$. After normalization, the final result is $(+.110 \times 2^2)$.

The Booth Algorithm

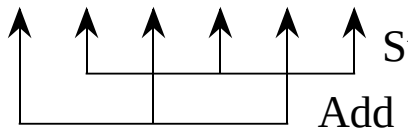
- **Booth multiplication reduces the number of additions for intermediate results, but can sometimes make it worse as we will see.**
- **Positive and negative numbers treated alike.**



A Worst Case Booth Example

- **A worst case situation in which the simple Booth algorithm requires twice as many additions as serial multiplication.**

	0	0	1	1	1	0	$(14)_{10}$ Multiplicand
	0	1	0	1	0	1	$(21)_{10}$ Multiplier
×	+1	−1	+1	−1	+1	−1	Booth recoded multiplier



1	1	1	1	1	1	1	1	0	0	1	0	$(-14 \times 1)_{10}$
0	0	0	0	0	0	0	1	1	1	0	0	$(14 \times 2)_{10}$
1	1	1	1	1	1	0	0	1	0	0	0	$(-14 \times 4)_{10}$
0	0	0	0	0	1	1	1	0	0	0	0	$(14 \times 8)_{10}$
1	1	1	1	0	0	1	0	0	0	0	0	$(-14 \times 16)_{10}$
0	0	0	1	1	1	0	0	0	0	0	0	$(14 \times 32)_{10}$
0	0	0	1	0	0	1	0	0	1	1	0	$(294)_{10}$ Product

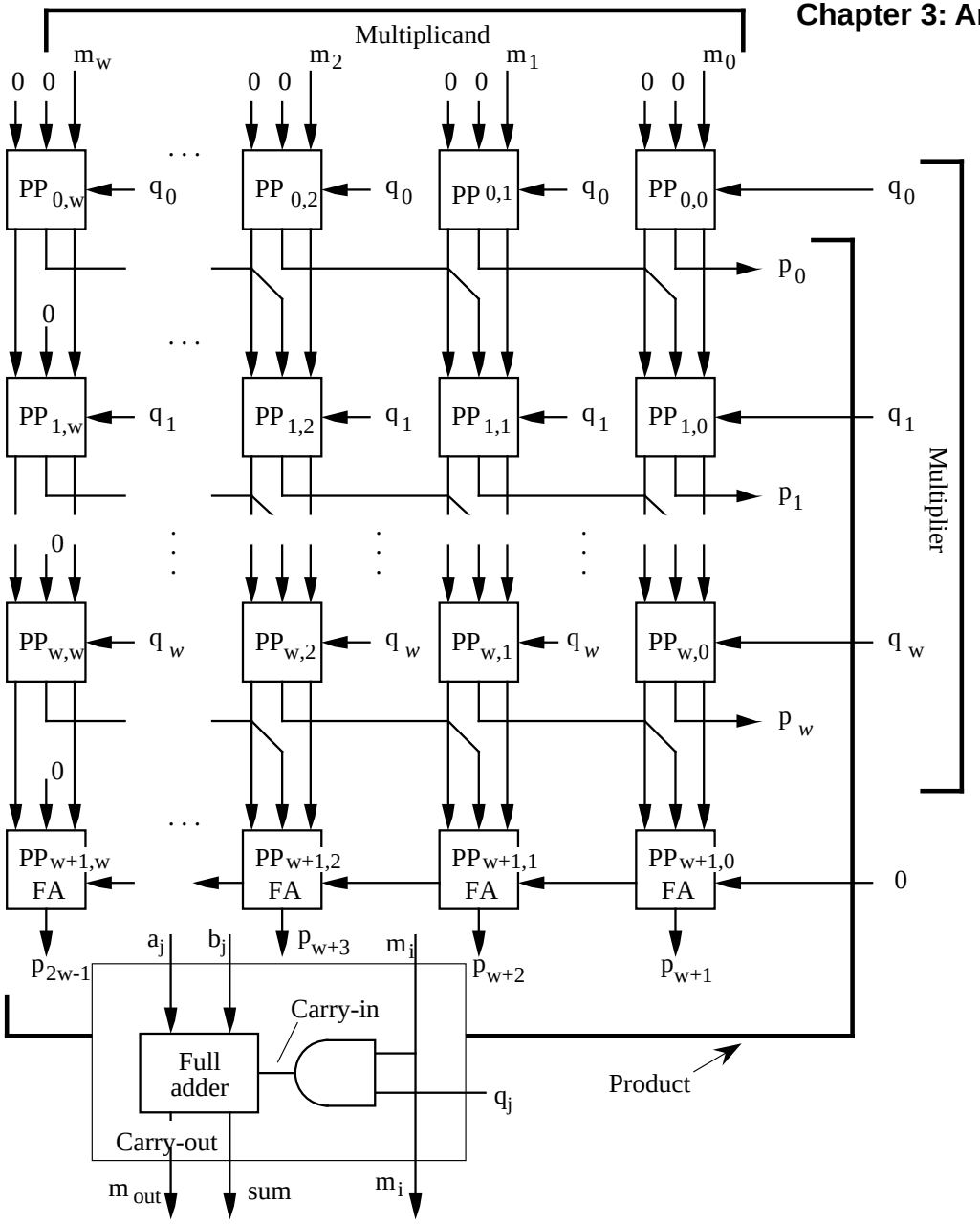
Bit-Pair Recoding (Modified Booth Algorithm)

						0	0	1	1	1	0	$(21)_{10}$ Multiplicand
						0	1	0	1	0	1	$(14)_{10}$ Multiplier
					×	+1	−1	+1	−1	+1	−1	Booth recoded multiplier
							+1		+1		+1	Bit pair recoded multiplier
0	0	0	0	0	0	0	0	1	1	1	0	$(14 \times 1)_{10}$
0	0	0	0	0	0	1	1	1	0	0	0	$(14 \times 4)_{10}$
0	0	0	0	1	1	1	0	0	0	0	0	$(14 \times 16)_{10}$
0	0	0	1	0	0	1	0	0	1	1	0	$(294)_{10}$ Product

Coding of Bit Pairs

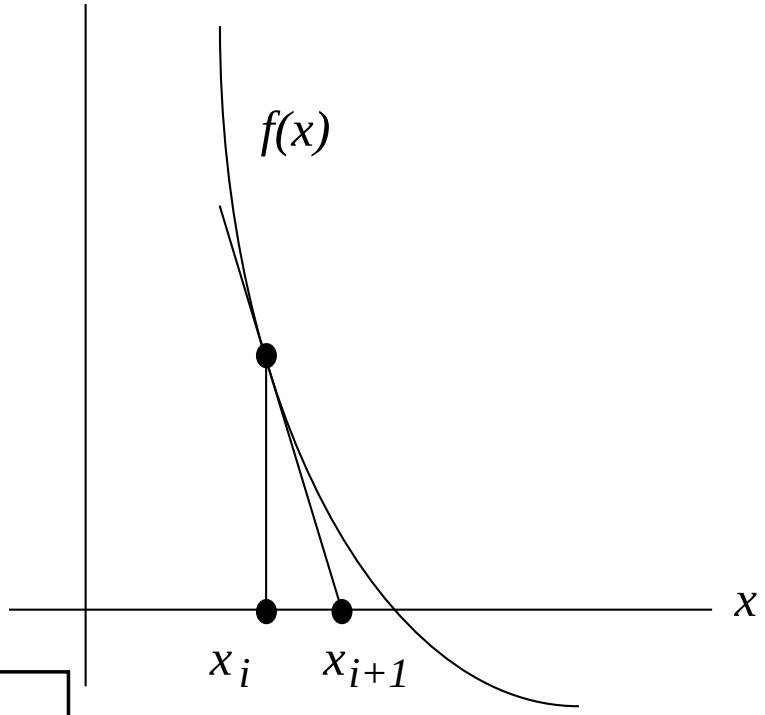
Booth pair ($i+1$, i)		Recoded bit pair (i)	Corresponding multiplier bits ($i+1$, i, $i-1$)
0	0	0	000 or 111
0	+1	+1	001
0	–1	–1	110
+1	0	+2	011
+1	+1	—	
+1	–1	+1	010
–1	0	–2	100
–1	+1	–1	101
–1	–1	—	

Parallel Pipelined Array Multiplier



Newton's Iteration for Zero Finding

- **The goal is to find where the function *f(x)* crosses the *x* axis by starting with a guess x_i and then using the error between $f(x_i)$ and zero to refine the guess.**
- **A three-bit lookup table for computing x_0:**



B = First three bits of *b*	Actual base 10 value of 1/B	Corresponding lookup table entry
.100	2	10
.101	1 3/5	01
.110	1 1/3	01
.111	1 1/7	01

- **The division operation *a*/*b* is computed as $a \times 1/b$. Newton's iteration provides a fast method of computing 1/*b*.**

Residue Arithmetic

- **Implements carryless arithmetic (thus fast!), but comparisons are difficult without converting to a weighted position code.**
- **Representation of the first twenty decimal integers in the residue number system for the given moduli:**

Decimal	Residue 5794	Decimal	Residue 5794
0	0000	10	0312
1	1111	11	1423
2	2222	12	2530
3	3333	13	3641
4	4440	14	4052
5	0551	15	0163
6	1662	16	1270
7	2073	17	2381
8	3180	18	3402
9	4201	19	4513

Examples of Addition and Multiplication in the Residue Number System

29 + 27 = 56	
Decimal	Residue 5794
29	4121
27	2603
56	1020

$10 \times 17 = 170$	
Decimal	Residue 5794
10	0312
17	2381
170	0282

16-bit Group Carry Lookahead Adder

- A16-bit GCLA is composed of four 4-bit CLAs, with additional logic that generates the carries between the four-bit groups.

$$GG_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

$$GP_0 = P_3P_2P_1P_0$$

$$c_4 = GG_0 + GP_0c_0$$

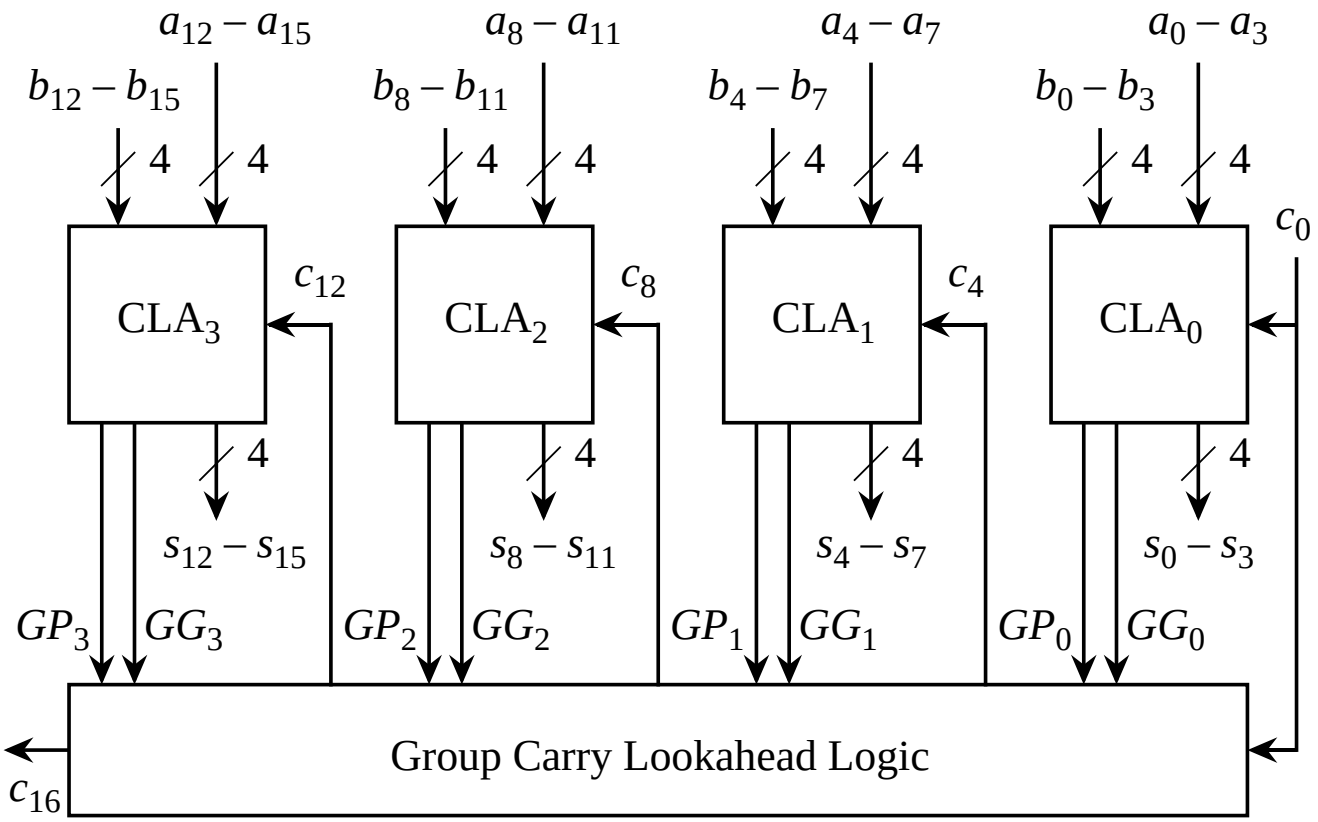
$$c_8 = GG_1 + GP_1c_4 = GG_1 + GP_1GG_0 + GP_1GP_0c_0$$

$$c_{12} = GG_2 + GP_2c_8 = GG_2 + GP_2GG_1 + GP_2GP_1GG_0 + GP_2GP_1GP_0c_0$$

$$c_{16} = GG_3 + GP_3c_{12} = GG_3 + GP_3GG_2 + GP_3GP_2GG_1 + GP_3GP_2GP_1GG_0 + GP_3GP_2GP_1GP_0c_0$$

16-Bit Group Carry Lookahead Adder

- **Each CLA has a longest path of 5 gate delays.**



- **In the GCLL section, GG and GP signals are generated in 3 gate delays; carry signals are generated in 2 more gate delays, resulting in 5 gate delays to generate the carry out of each GCLA group and 10 gates delays on the worst case path (which is s_{15} – *not* c_{16}).**

HP 9100 Series Desktop Calculator

- **Source: http://www.teleport.com/ ~dgh/91003q.jpg.**
- **Uses binary coded decimal (BCD) arithmetic.**

Addition Example Using BCD

- **Addition is performed digit by digit (*not* bit by bit), in 4-bit groups, from right to left.**
- **Example $(255 + 63 = 318)_{10}$:**

	0	1	0	0	← Carries
	0 0 0 0	0 0 1 0	0 1 0 1	0 1 0 1	$(+255)_{10}$
	$(0)_{10}$	$(2)_{10}$	$(5)_{10}$	$(5)_{10}$	
+	0 0 0 0	0 0 0 0	0 1 1 0	0 0 1 1	$(+63)_{10}$
	$(0)_{10}$	$(0)_{10}$	$(6)_{10}$	$(3)_{10}$	
	0 0 0 0	0 0 1 1	0 0 0 1	1 0 0 0	$(+318)_{10}$
	$(0)_{10}$	$(3)_{10}$	$(1)_{10}$	$(8)_{10}$	

Subtraction Example Using BCD

- **Subtraction is carried out by adding the ten's complement negative of the subtrahend to the minuend.**
- **Ten's complement negative of subtrahend is obtained by adding 1 to the nine's complement negative of the subtrahend.**
- **Consider performing the subtraction operation $(255 - 63 = 192)_{10}$:**

```
  9999
 −0063
------
  9936

  9936
 +0001
------
  9937
```

	1	1	0	1	0 ← Carries
	0000	0010	0101	0101	$(+255)_{10}$
+	1001	1001	0011	0111	$(-63)_{10}$
	1 0000	0001	1001	0010	$(+192)_{10}$

↑ Discard carry

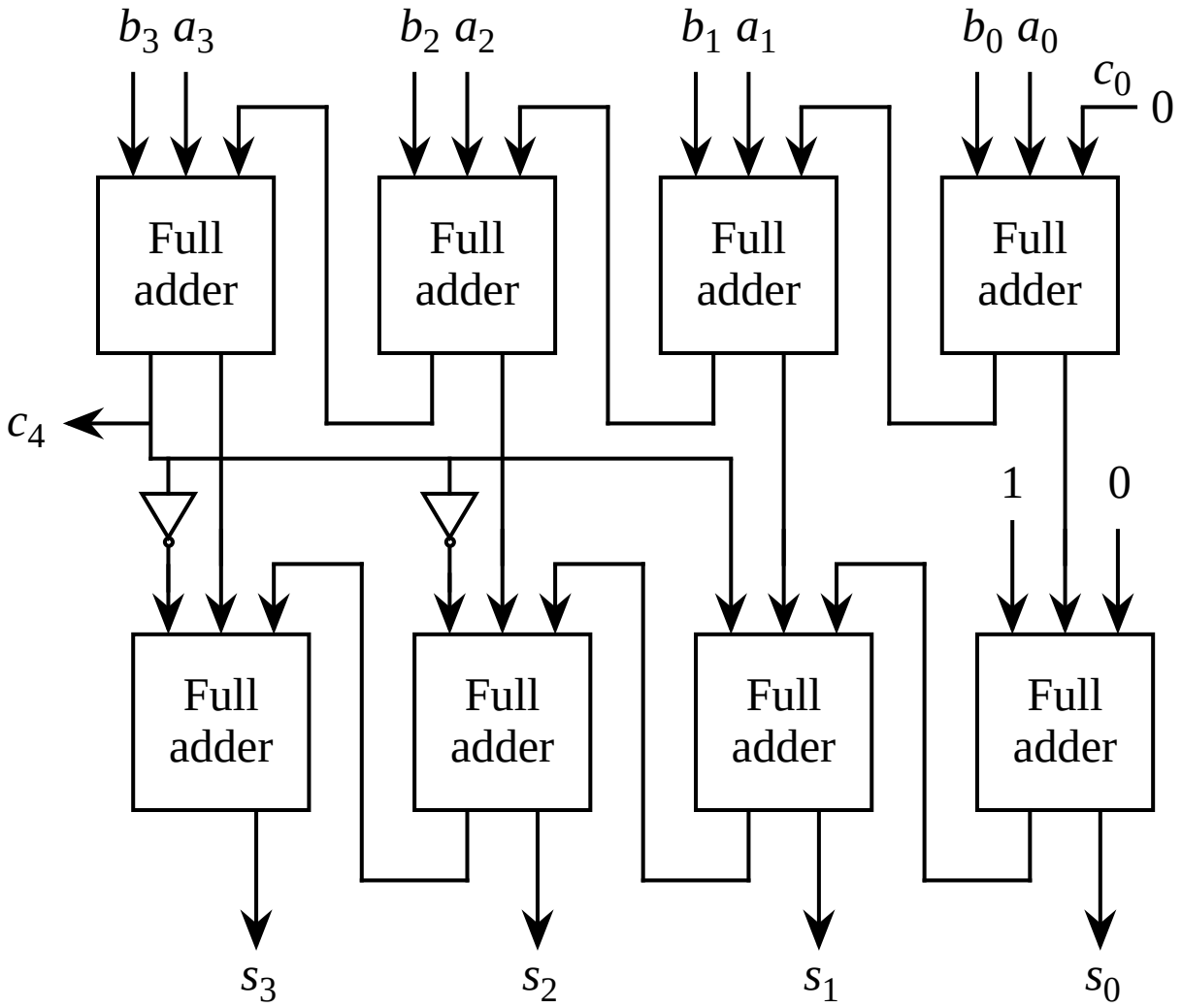
Excess 3 Encoding of BCD Digits

- **Using an excess 3 encoding for each BCD digit, the leftmost bit indicates the sign.**

BCD Bit Pattern	Normal BCD value	Excess 3 value	
0 0 0 0	0	d	
0 0 0 1	1	d	
0 0 1 0	2	d	
0 0 1 1	3	0	Positive numbers
0 1 0 0	4	1	
0 1 0 1	5	2	
0 1 1 0	6	3	
0 1 1 1	7	4	
1 0 0 0	8	5	Negative numbers
1 0 0 1	9	6	
1 0 1 0	d	7	
1 0 1 1	d	8	
1 1 0 0	d	9	
1 1 0 1	d	d	
1 1 1 0	d	d	
1 1 1 1	d	d	

A BCD Full Adder

- **Circuit adds two base 10 digits represented in BCD. Adding 5 and 7 (0101 and 0111) results in 12 (0010 with a carry of 1, and *not* 1100, which is the binary representation of 12_{10}).**



Ten's Complement Subtraction

- **Compare: the traditional signed magnitude approach for adding decimal numbers *vs.* the ten's complement approach, for $(21 - 34 = -13)_{10}$:**

0 0 2 1	0 0 2 1
+ 9 9 6 6	− 0 0 3 4
9 9 8 7	− 0 0 1 3
Ten's Complement	Signed Magnitude

BCD Floating Point Representation

- Consider a base 10 floating point representation with a two digit signed magnitude exponent and an eight digit signed magnitude fraction. On a calculator, a sample entry might look like:

$$-.37100000 \times 10^{-12}$$

- We use a ten's complement representation for the exponent, and a base 10 signed magnitude representation for the fraction. A separate sign bit is maintained for the fraction, so that each digit can take on any of the 10 values 0–9 (except for the first digit, which cannot be zero). We should also represent the exponent in excess 50 (placing the representation for 0 in the middle of the exponents, which range from -50 to +49) to make comparisons easier.
- The example above now looks like this (see next slide):

BCD Floating Point Arithmetic

- The example in the previous slide looks like this:

  Sign bit: 1

  Exponent: 0110 1011

  Fraction: 0110 1010 0100 0011 0011 0011 0011 0011 0011

- Note that the representation is still in excess 3 binary form, with a two digit excess 50 exponent.

- To add two numbers in this representation, as for a base 2 floating point representation, we start by adjusting the exponent and fraction of the smaller operand until the exponents of both operands are the same. After adjusting the smaller fraction, we convert either or both operands from signed magnitude to ten's complement according to whether we are adding or subtracting, and whether the operands are positive or negative, and then perform the addition or subtraction operation.