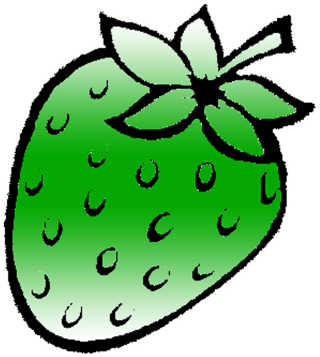
STRAWBERRY



/strawberrydevelopers

/strawberry_app

For more visit:

Strawberrydevelopers.weebly.com

UNIT IV: DATA PATH DESIGN

Agenda

- Introduction
- Fixed Point Arithmetic
  - Addition
  - Subtraction
  - Multiplication & Serial multiplier
  - Division & Serial Divider
  - Two's Complement (Addition, Subtraction)
- Booth's algorithm
- Design of basic serial adders
- Serial & parallel(ripple) Subtractors
- High speed adders
- Floating Point Arithmetic
  - Addition & Subtraction
  - Guard, Round & Sticky bits
  - Multiplication & Division
- Combinational & Sequential ALU: Assignment 2

Data Representation

- The basic form of information handled by a computer are instructions and data
- Data can be in the form of numbers or nonnumeric data
- Data in the number form can further classified as fixed point and floating point

Fixed Point Numbers

- Fixed point number actually symbolizes the real data types.

- Fixed point numbers are those which have a defined numbers after and before the decimal point.

Fixed Point Numbers: Decimal to Binary Conversion

For example:

$$2.375 = (10.011)_{two} = (1\times2^{1}) + (0\times2^{0}) + (0\times2^{-1}) + (1\times2^{-2}) + (1\times2^{-3})$$

Fixed Point Numbers: Unsigned Integers

- Unsigned integers represent positive numbers
- The decimal range of unsigned 8-bit binary numbers is 0 - 255

Signed Integers

- We dealt with representing the natural numbers
- Signed or directed whole numbers = integers

  { . . . , –3, –2, –1, 0, 1, 2, 3, . . . }

- Signed magnitude for 8 bit numbers ranges from +127 to -127

Fixed Point Arithmetic

- Four basic arithmetic instructions:
  - Addition
  - Subtraction
  - Multiplication
  - Division

Addition

Rules of Binary Addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0, and carry 1 to the next more significant bit

For example,

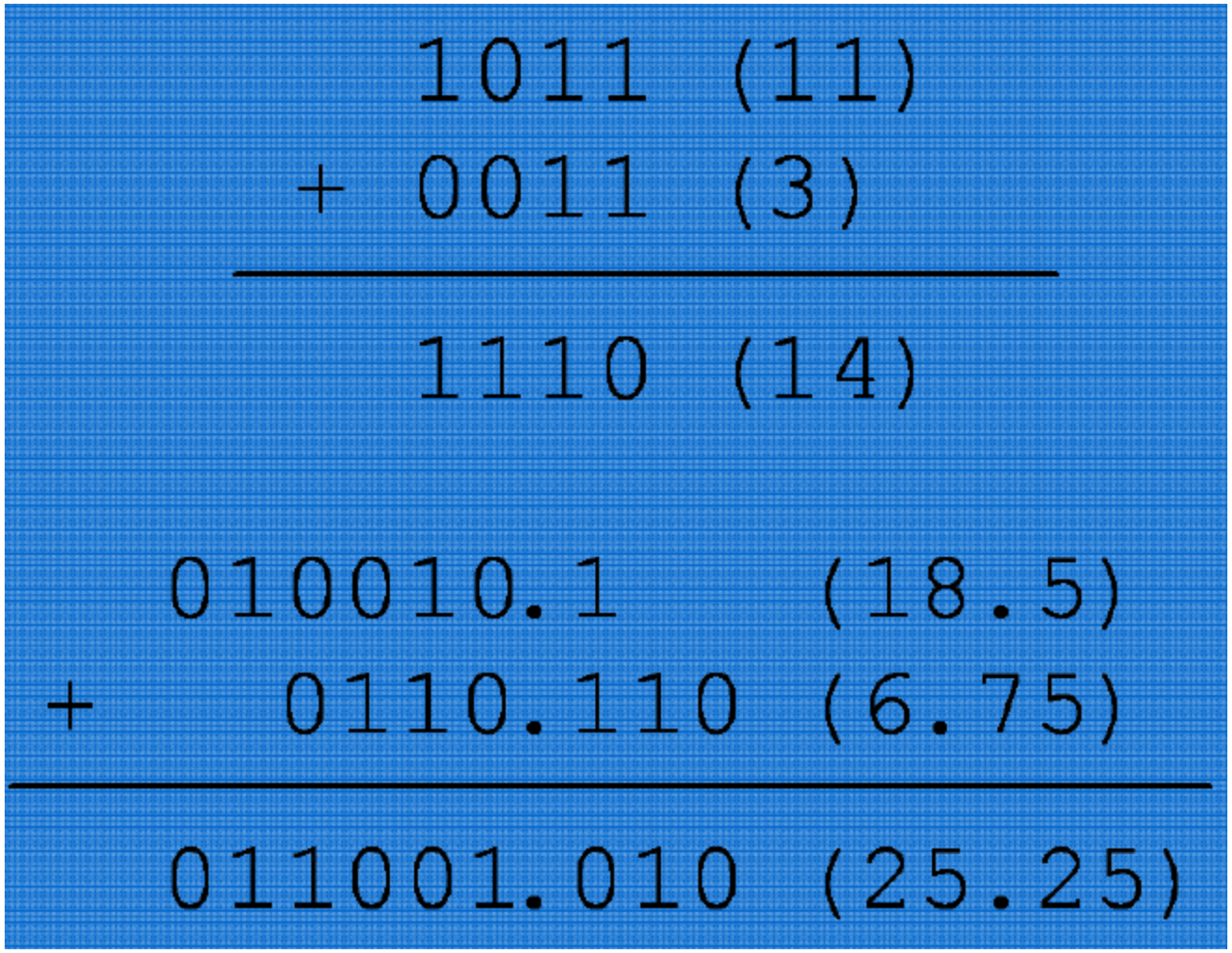
00011010 + 00001100 = 00100110

			1	*1*					*carries*
	0	0	0	1	1	0	1	0	= $26_{(\text{base } 10)}$
+	0	0	0	0	1	1	0	0	= $12_{(\text{base } 10)}$
	0	0	1	0	0	1	1	0	= $38_{(\text{base } 10)}$

00010011 + 00111110 = 01010001

		1	*1*	*1*	*1*	*1*			*carries*
	0	0	0	1	0	0	1	1	= $19_{(\text{base } 10)}$
+	0	0	1	1	1	1	1	0	= $62_{(\text{base } 10)}$
	0	1	0	1	0	0	0	1	= $81_{(\text{base } 10)}$

Addition


1011 (11)
+ 0011 (3)
1110 (14)
010010.1 (18.5)
+ 0110.110 (6.75)
011001.010 (25.25)

Subtraction

Rules of Binary Subtraction

- 0 - 0 = 0
- 0 - 1 = 1, and borrow 1 from the next more significant bit
- 1 - 0 = 1
- 1 - 1 = 0

For example,

00100101 - 00010001 =

```
        0                          borrows
  0  0  1  10  0  1  0  1  =  37(base 10)
- 0  0  0  1  0  0  0  1  =  17(base 10)
-------------------------
  0  0  0  1  0  1  0  0  =  20(base 10)
```

Subtraction: Example

00110011 - 00010110 =

		0	*^{1}0*	*1*					*borrows*
	0	0	~~1~~	~~1~~	~~0~~	^{1}0	1	1	= $51_{(\text{base } 10)}$
-	0	0	0	1	0	1	1	0	= $22_{(\text{base } 10)}$
	0	0	0	1	1	1	0	1	= $29_{(\text{base } 10)}$

Multiplication

Rules of Binary Multiplication

- 0 x 0 = 0
- 0 x 1 = 0
- 1 x 0 = 0
- 1 x 1 = 1, and no carry or borrow bits

For example,

00101001 x 00000110 = 11110110

```
      0  0  1  0  1  0  0  1  =   41 (base 10)
   x  0  0  0  0  0  1  1  0  =    6 (base 10)
-------------------------------
      0  0  0  0  0  0  0  0
   0  0  1  0  1  0  0  1
0  0  1  0  1  0  0  1
-------------------------------
0  0  1  1  1  1  0  1  1  0  = 246 (base 10)
```

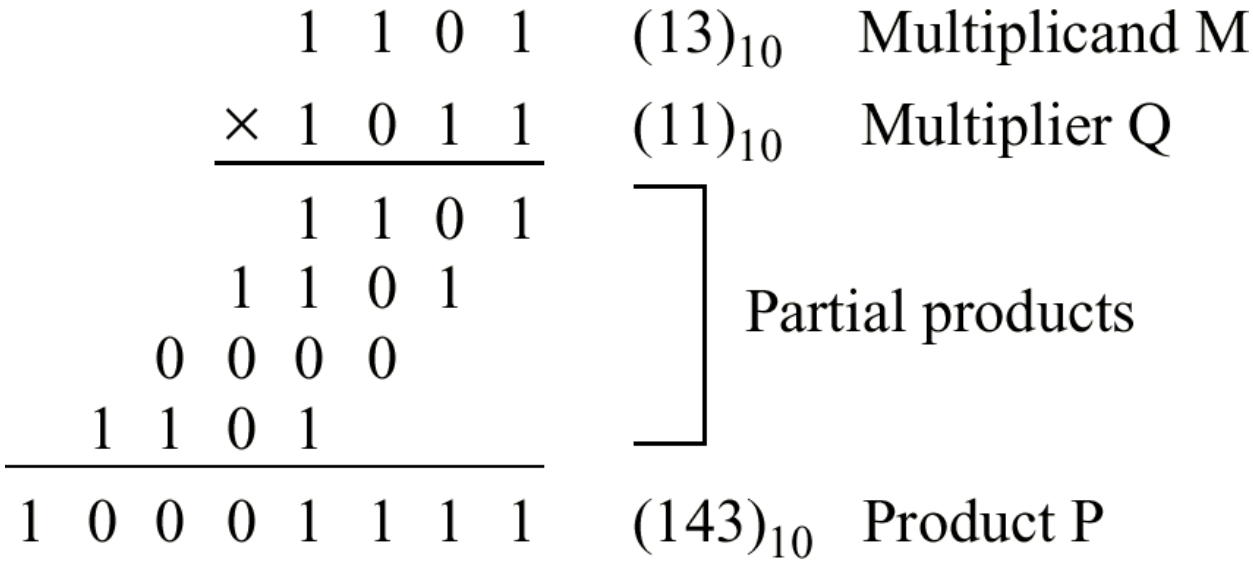
Multiplication Example

00010111 x 00000011 = 01000101

	0	0	0	1	0	1	1	1	=	$23_{(\text{base } 10)}$
x	0	0	0	0	0	0	1	1	=	$3_{(\text{base } 10)}$
		1	*1*	*1*	*1*	*1*				*carries*
	0	0	0	1	0	1	1	1		
0	0	0	1	0	1	1	1			
0	0	1	0	0	0	1	0	1	=	$69_{(\text{base } 10)}$

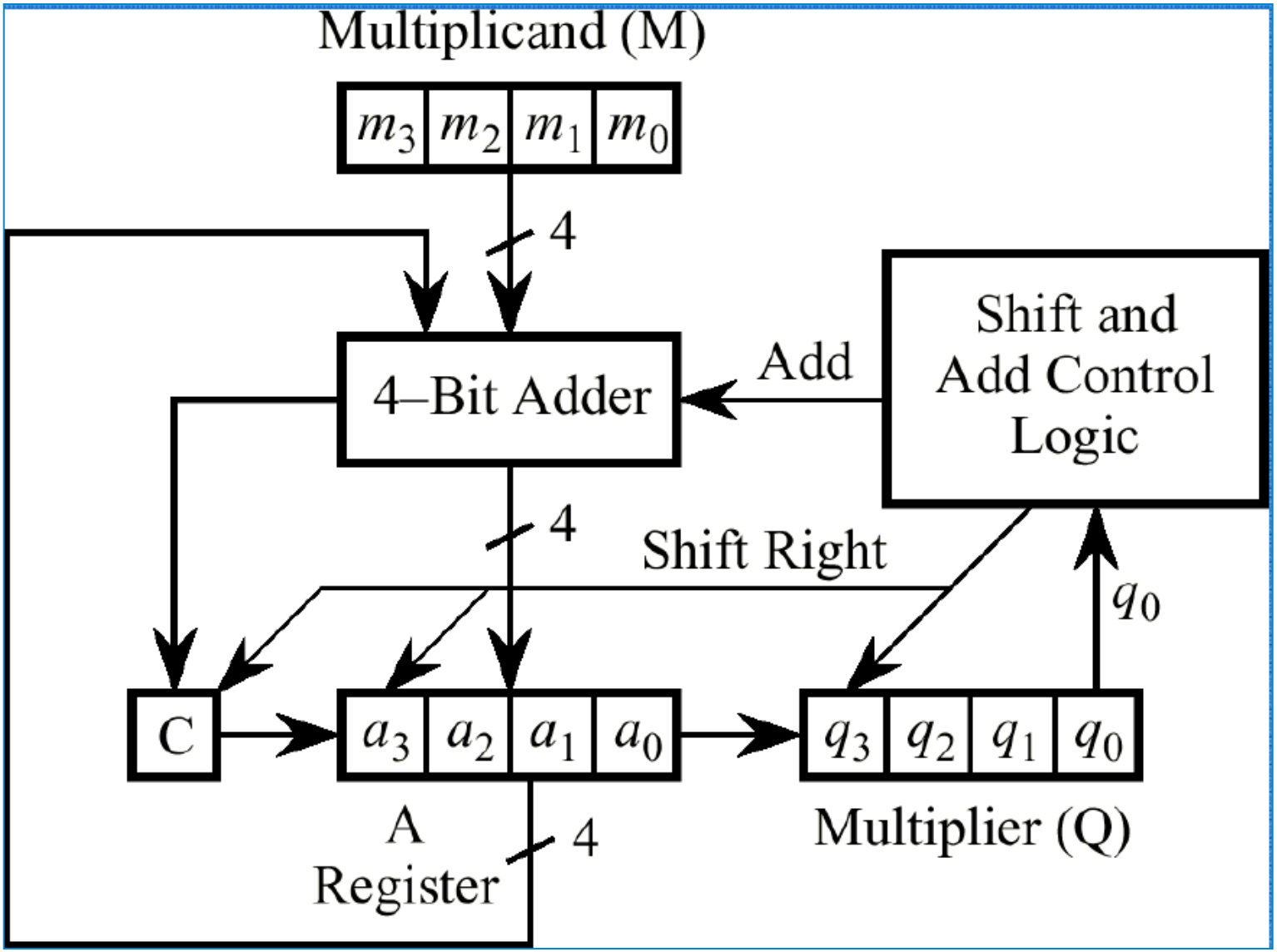
Multiplication Example

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

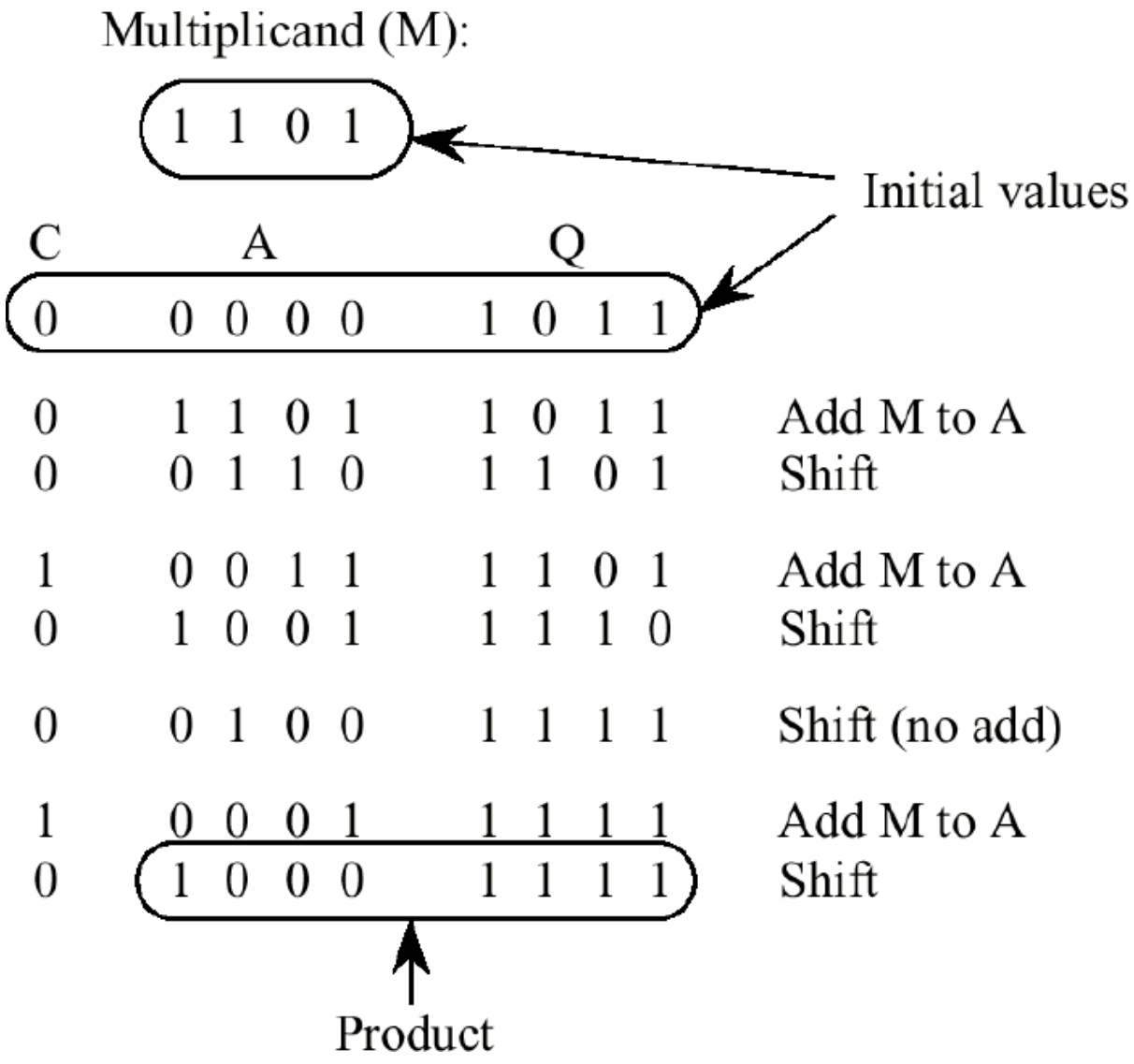


- Multiplication of two 4-bit signed binary integers produces only a 7-bit result (each operand reduces to a sign bit and a 3-bit magnitude for each operand, producing a sign-bit and a 6-bit result).

Multiplication: A serial Multiplier



Multiplication: Example



Division: Example

For example,

00101010 ÷ 00000110 = **00000111**

```
                          1  1  1   =   7(base 10)
                 ------------------
1  1  0 ) 0  0  ±¹⁰ 1  0  1  0   =  42(base 10)
             -   1  1  0           =   6(base 10)
          ---------------
                1                      borrows
             ±  0  ¹⁰ 1
          -     1  1  0
          ---------------
                   1  1  0
             -     1  1  0
          ------------------
                         0
```

Division: Example

10000111 ÷ 00000101 = 00011011

```
                    1  1  0  1  1   =  27 (base 10)
              ---------------------
1  0  1  ) 1  0  0  1  0  1  1  1   =  135 (base 10)
           -  1  0  1                =    5 (base 10)
           ----------
              1  1  0
           -  1  0  1
           ----------
                 1  1
              -     0
              -------
                 1  1  1
              -  1  0  1
              ----------
                    1  0  1
                 -  1  0  1
                 ----------
                          0
```

Division: Example of Base 2

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

$$
\begin{array}{r}
0\,0\,1\,0\ \ \mathrm{R}\,1 \\
\hline
1\,1 \;\big|\; 0\,1\,1\,1 \\
\quad 1\,1 \\
\hline
\quad\ \ 0\,1
\end{array}
$$

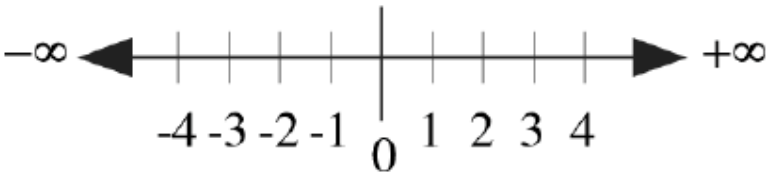
Sign-Magnitude

- Left most bit is sign bit
- 0 means positive
- 1 means negative
- +18 = 00010010
- -18 = 10010010
- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0)

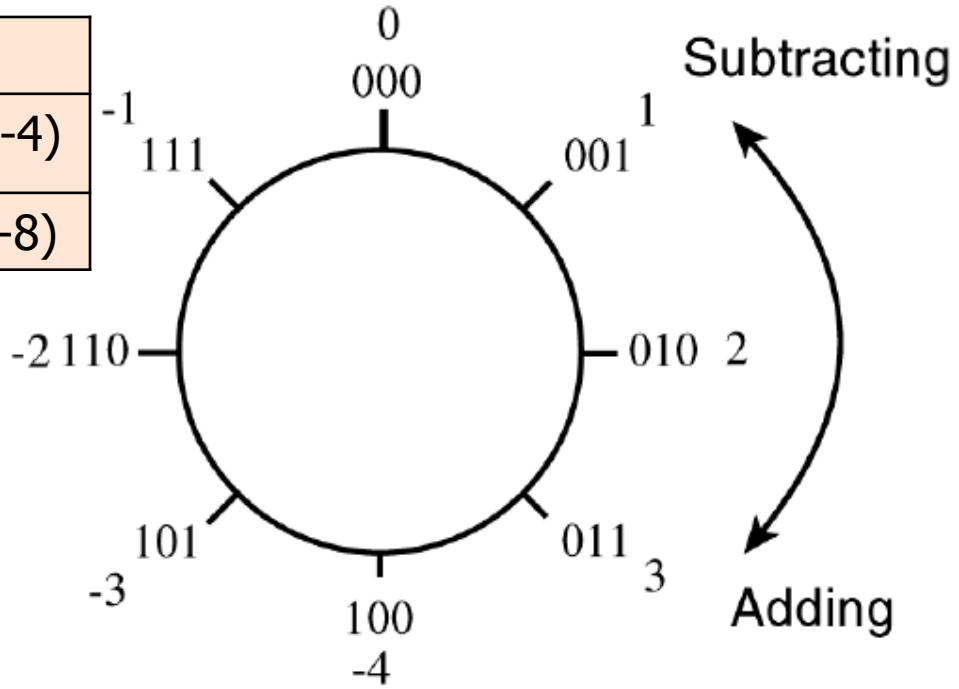
Two's Complement: Signed Integers

- Numbers are added or subtracted on the number circle by traversing clockwise for addition and counterclockwise for subtraction.
- Unlike the number line (a) where overflow never occurs, 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.

2-bits	0,1,2,3	
3-bits	0-7	0-3, (-1)-(-4)
4-bits	0-15	0-7, (-1)-(-8)

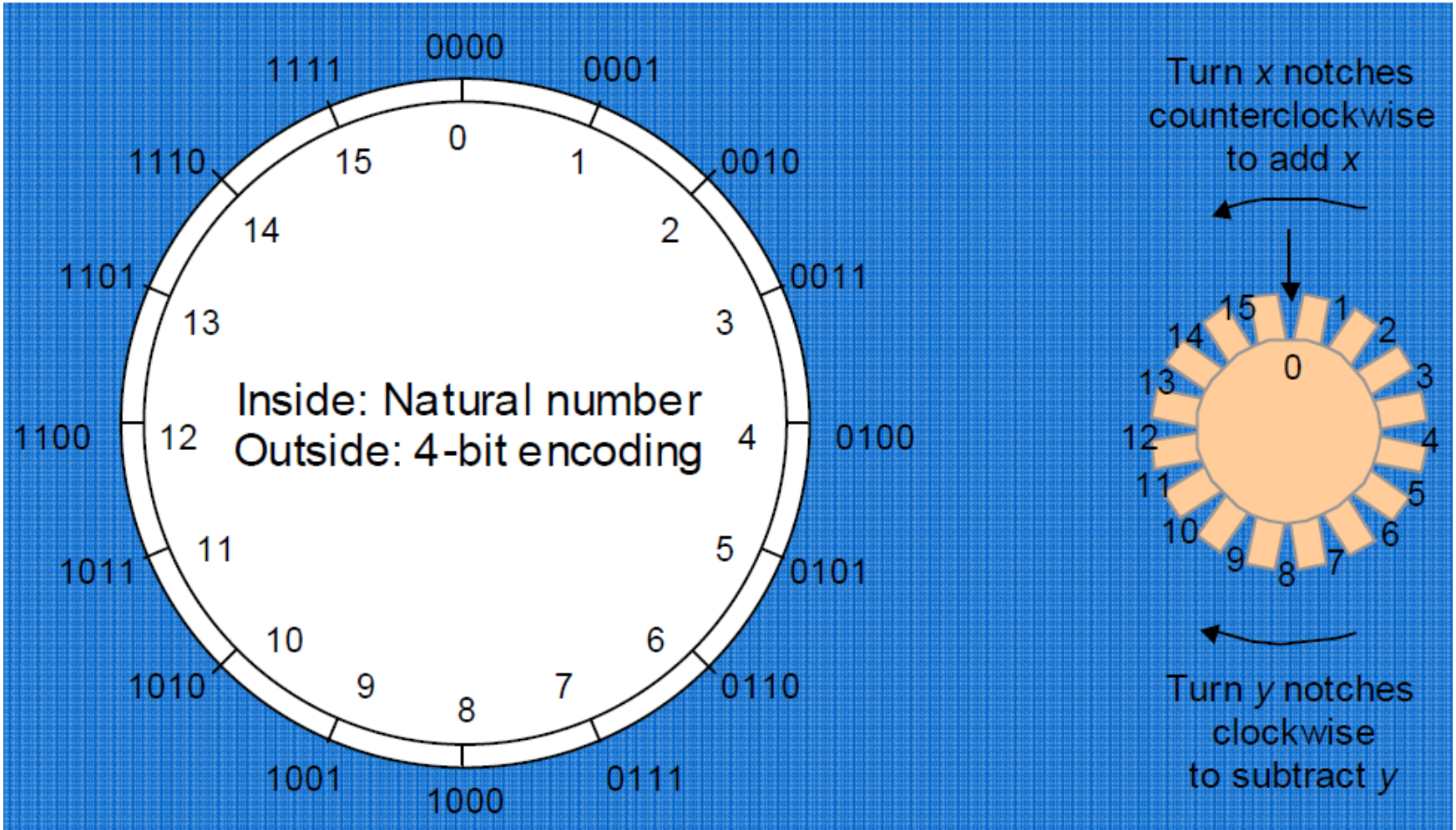


(a)



(b)

Two's Complement: Unsigned Integers



Schematic representation of 4-bit code for integers in [0, 15].

Two's Complement

Integer		2's Complement
Signed	Unsigned	
5	5	0000 0101
4	4	0000 0100
3	3	0000 0011
2	2	0000 0010
1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100
-5	251	1111 1011

Note: The most significant (leftmost) bit indicates the sign of the integer; therefore it is sometimes called the sign bit.

If the sign bit is zero,
then the number is greater than or equal to zero, or positive.
If the sign bit is one,
then the number is less than zero, or negative.

Two's Complement

Calculation of 2's Complement

To calculate the 2's complement of an integer, invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), and then add one.

For example,

```
0001 0001(binary 17)  ⇒  1110 1111(two's complement -17)

      NOT(0001 0001)  =  1110 1110  (Invert bits)
1110 1110 + 0000 0001  =  1110 1111  (Add 1)
```

Two's Complement: Addition

2's Complement Addition

Two's complement addition follows the same rules as binary addition.

For example,

$$5 + (-3) = 2$$

```
  0000 0101  =  +5
+ 1111 1101  =  -3
-------------
  0000 0010  =  +2
```

Two's Complement: Subtraction

2's Complement Subtraction

Two's complement subtraction is the binary addition of the minuend to the 2's complement of the subtrahend (adding a negative number is the same as subtracting a positive one).

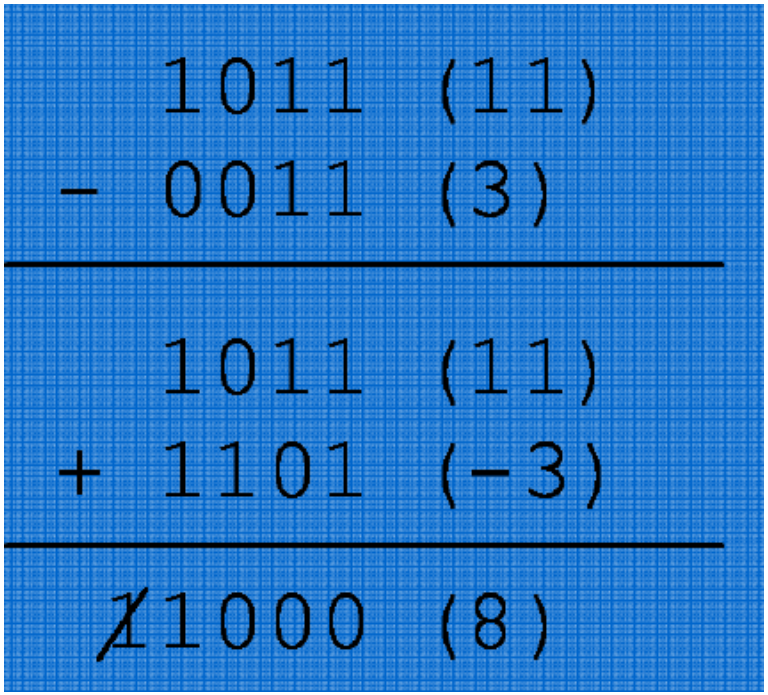
For example,

$$7 - 12 = (-5)$$

```
  0000 0111  =  +7
+ 1111 0100  =  -12
-------------------

  1111 1011  =  -5
```

Two's Complement: Subtraction



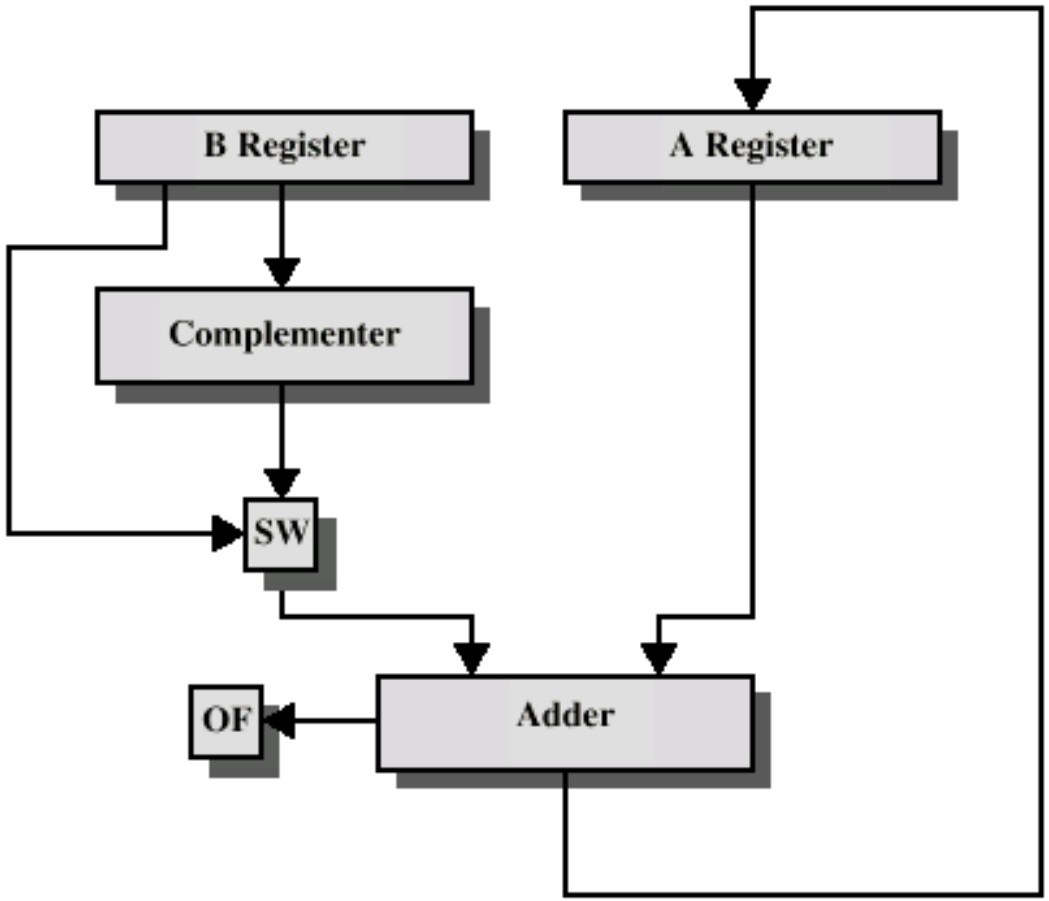
Addition and Subtraction

- Normal binary addition
- Monitor sign bit for overflow

- Take twos compliment of substahend and add to minuend
  - i.e. a - b = a + (-b)

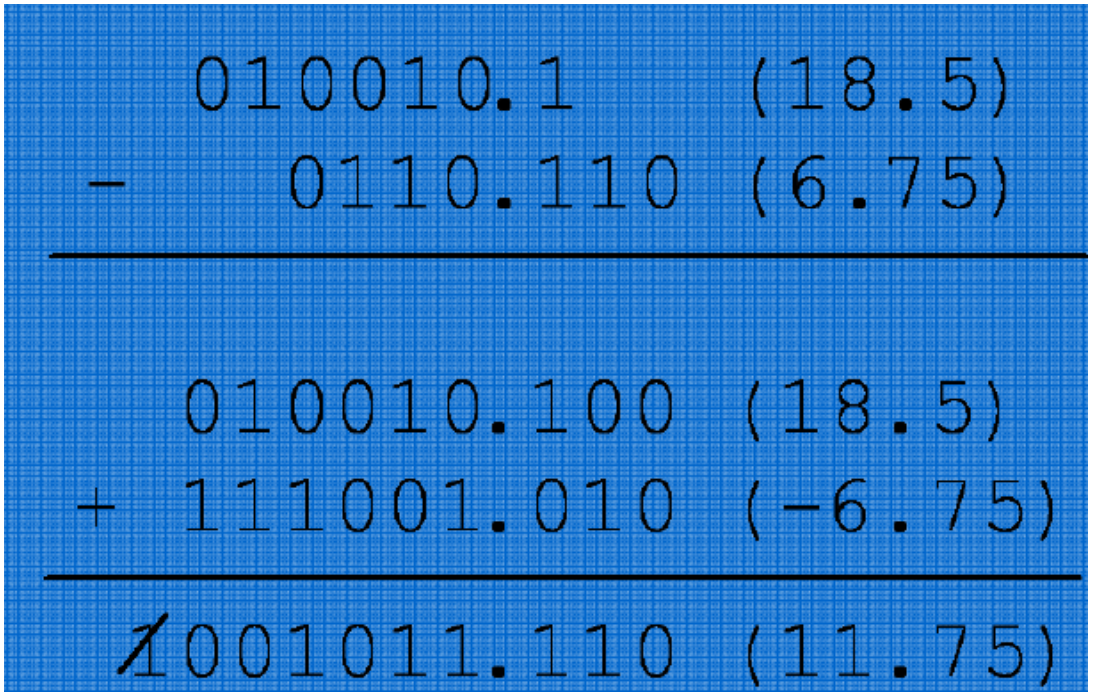
- So we only need addition and complement circuits

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

2's complement of floating point numbers: Example



2's complement of floating point numbers

To create 2's complement:

- Take the number given by you
010111.1100.
- Start on the least significant bit and locate the first 1 marked red
010111.1100.
- Then flip every bit after that first one (1 change to zero and vice verse) 010111.1100-> 101000.0100

2's complement of floating point numbers: Example

- Example:
- 2.50 – 6.75 = -3.75

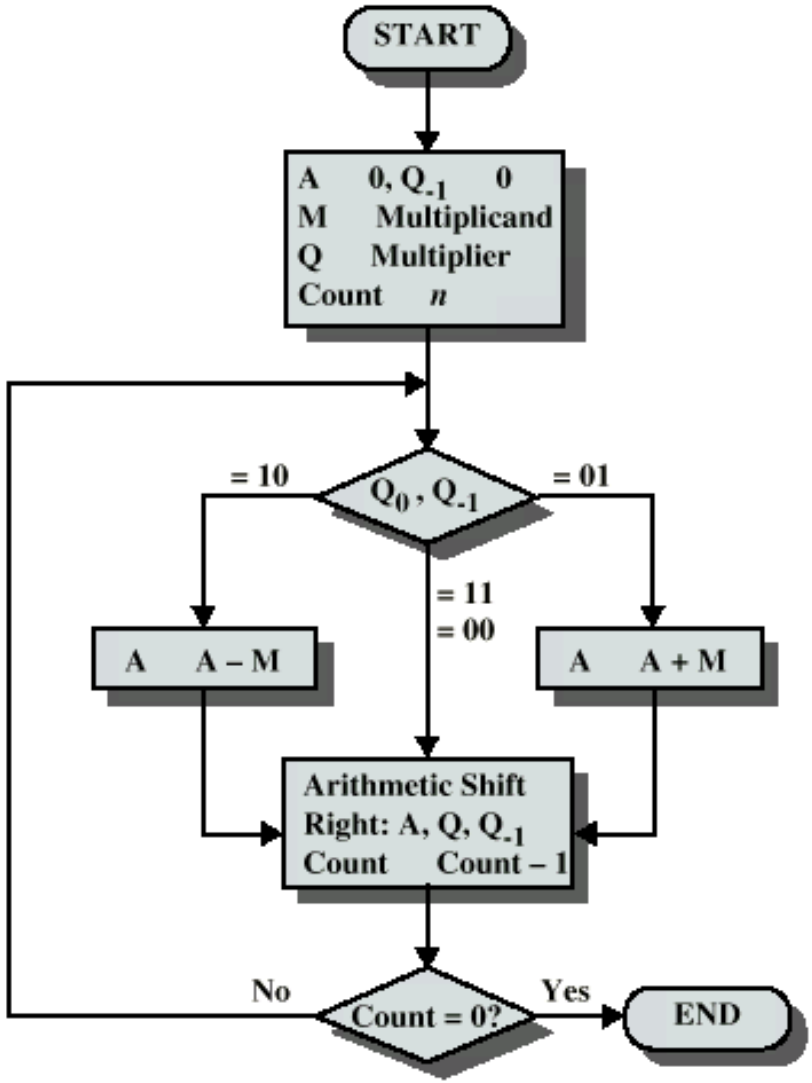
Conversion of 2's complement to decimal

Two's complement properties (including sign change) hold here as well:

$(01.011)_{2's\text{-}compl} = (-0\times2^{1}) + (1\times2^{0}) + (0\times2^{-1}) + (1\times2^{-2}) + (1\times2^{-3}) = +1.375$

$(11.011)_{2's\text{-}compl} = (-1\times2^{1}) + (1\times2^{0}) + (0\times2^{-1}) + (1\times2^{-2}) + (1\times2^{-3}) = -0.625$

Booth's Algorithm :Multiplying Negative Numbers



Example of Booth's Algorithm

Q_0	Q_{-1}	Operation
0	0	Shifting
1	1	Shifting
1	0	A = A – M Shifting
0	1	A = A + M Shifting

Multiplicand (M) = 0111 = 7
Multiplier (Q) = 0011 = 3

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

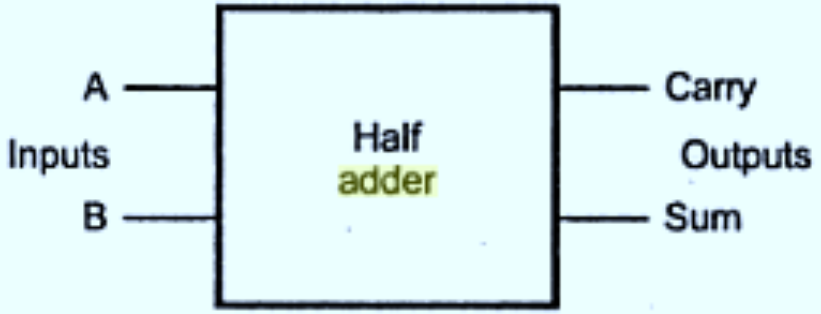
No. of cycles = No. of bits

Half Adder

- It adds two 1 bits but has no provision to include the carry output from previous bit position.

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table for half-adder



Block schematic of half-adder

$$\text{Carry} = AB$$

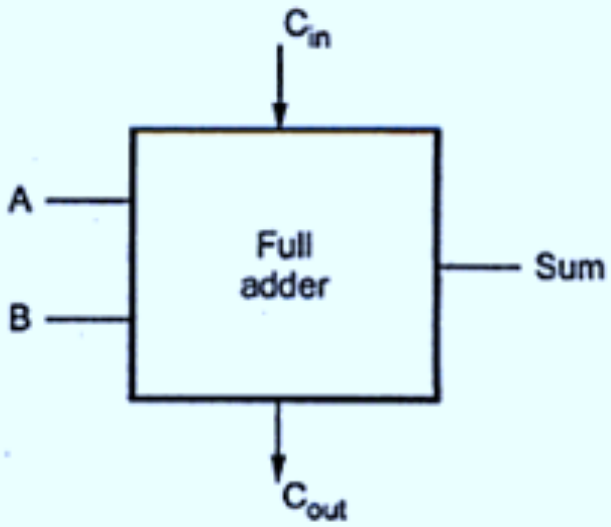
$$\text{Sum} = A\overline{B} + \overline{A}B = A \oplus B$$

Full Adder

- It is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs & two outputs. Two input variables as A & B. The third input Cin represents the carry from the previous bit position.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
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

Truth table for full-adder



Block schematic of full-adder

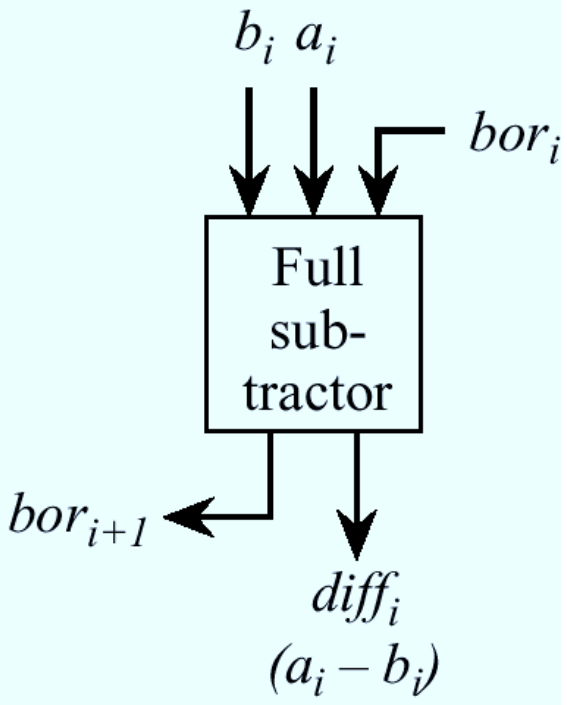
$C_{out} = AB + A C_{in} + B C_{in}$

$Sum = \overline{A}\,\overline{B} C_{in} + \overline{A} B \overline{C}_{in} + A \overline{B}\,\overline{C}_{in} + ABC_{in}$

Full Subtractor

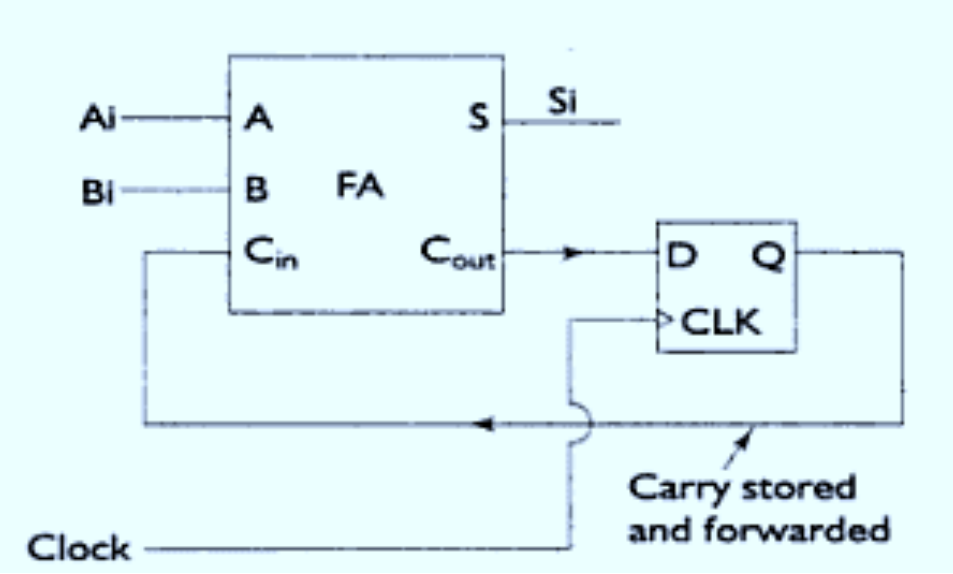
- Truth table and schematic symbol for a ripple-borrow subtractor:

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



Serial Adder

- A serial adder has only a single bit adder. It is used to perform addition of two numbers sequentially bit by bit. Addition of one bit position takes one clock cycle. Thus for n-bit serial adder, n clock cycles are required to complete the addition process & get the result. At each cycle, the carry produced by a bit position is stored in a flip-flop & it is given as an input during the next cycle as carry.



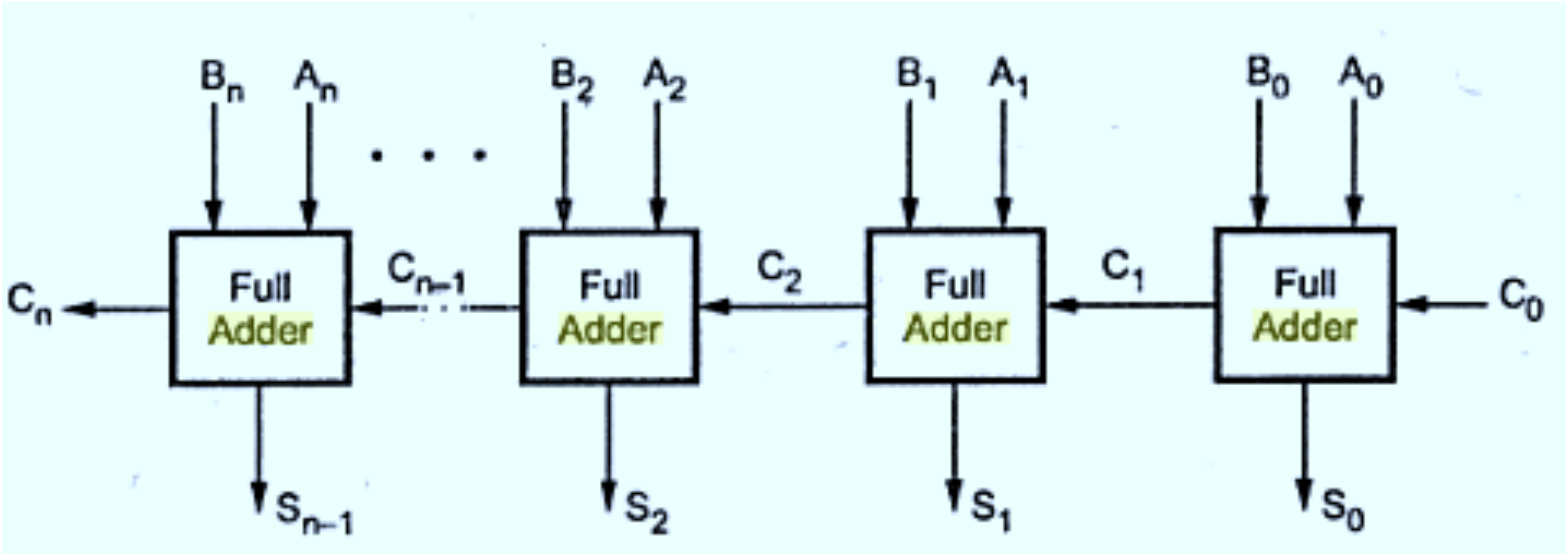
Serial Adder

Merits: The circuits for serial adder is small and hence, it is very cheap irrespective of the number of bits to be added.

Demerits: The serial adder is slow since it takes n clock cycles for completing addition of n bit numbers.

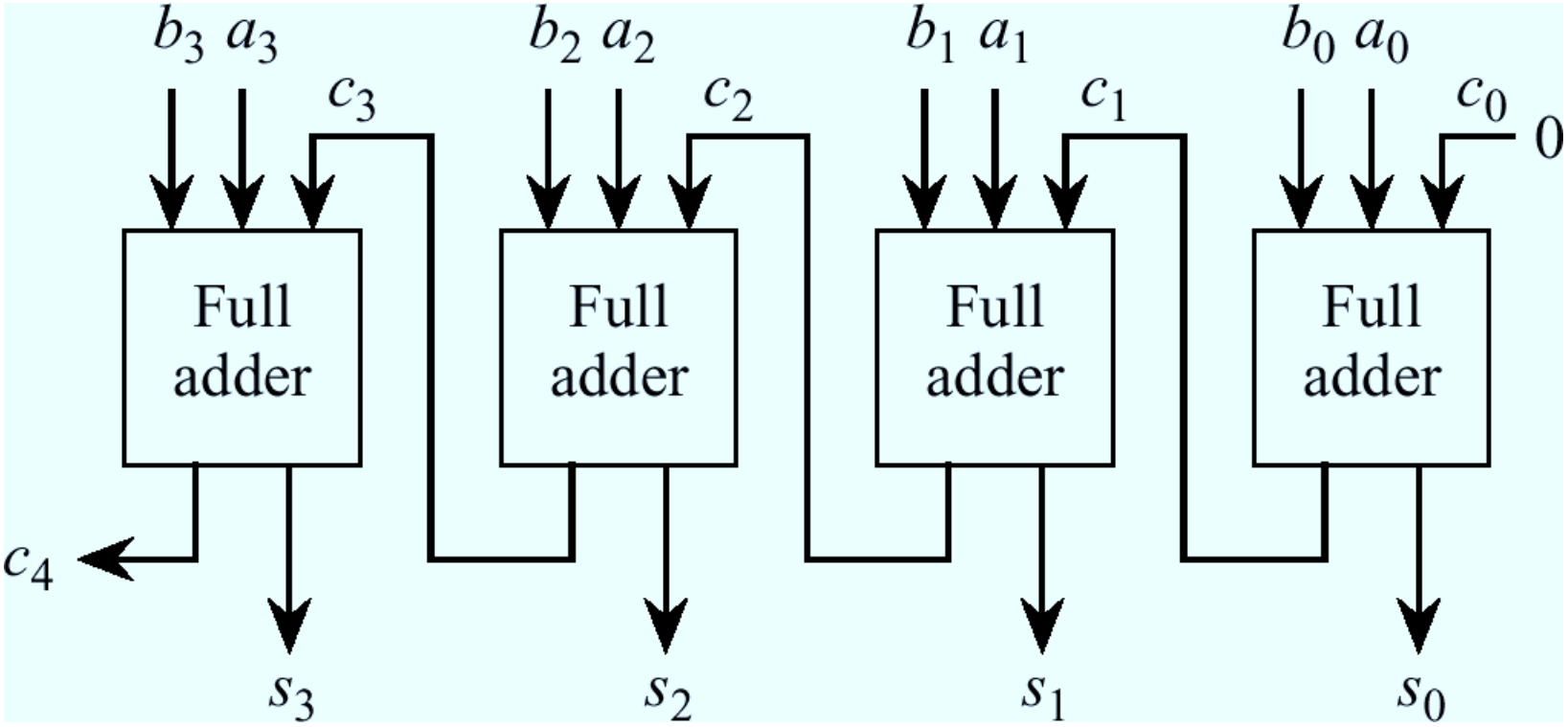
N-bit Ripple/Parallel Adder

- N-bit parallel adder using n number of full-adder circuits connected in cascade.
- The carry output of each adder is connected to the carry input of the next higher-order adder.



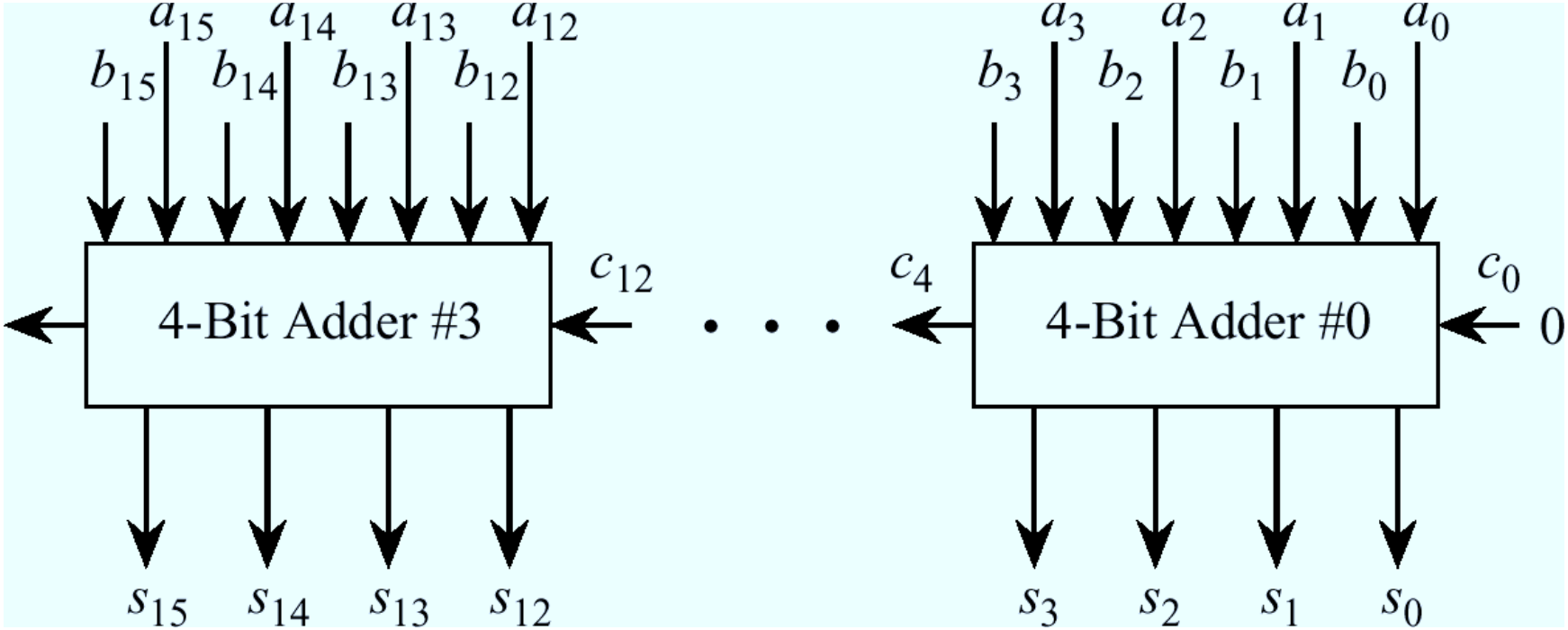
4-bit Ripple Carry Adder

- 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.



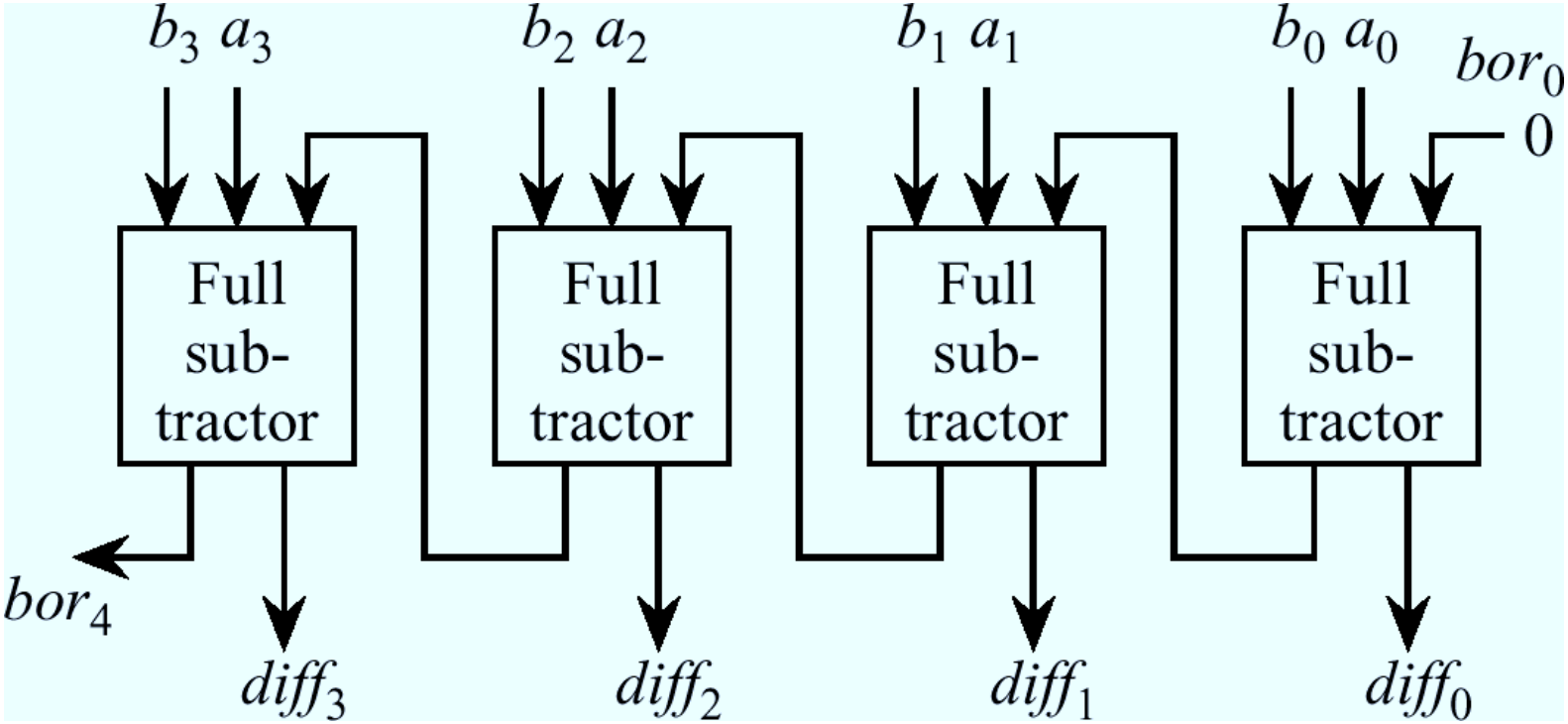
Constructing Larger Adders

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



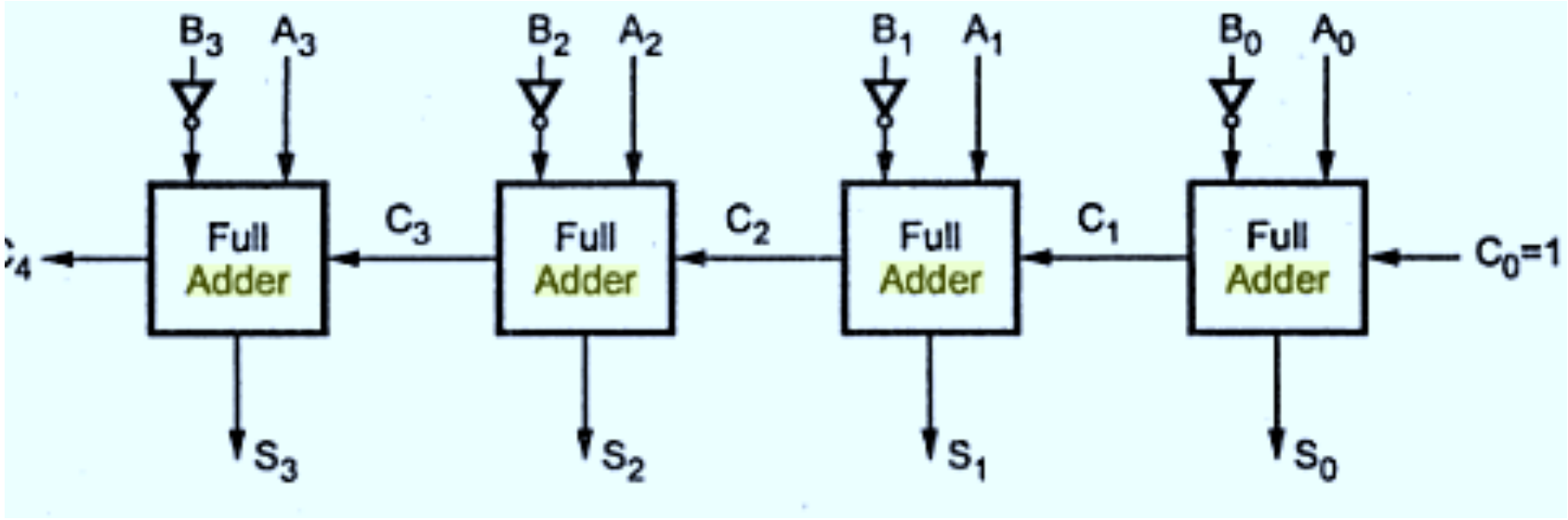
Ripple-Borrow Subtractor

- 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



Ripple Adder/Subtractor (Combined)

- A single ripple-carry adder can perform both addition and subtraction. The subtraction A-B ca be done by taking two's complement of B & adding it to A. 2's complement can be obtained by taking 1's complement & adding one to the least significant pair of bits.
- 1's complement can be implemented with inverters & a one can be added to the sum through the input carry to get 2's complement.



Design of Fast Adders

The n-bit adder discussed in the last section is implemented using full-adder stages. In which the carry output of each full-adder stage is connected to the carry input of the next higher-order stage. Therefore, the sum and carry outputs of any stage cannot be produced until the input carry occurs; this leads to a time delay in the addition process. This delay is known as **carry propagation delay**, which can be best explained by considering the following addition.

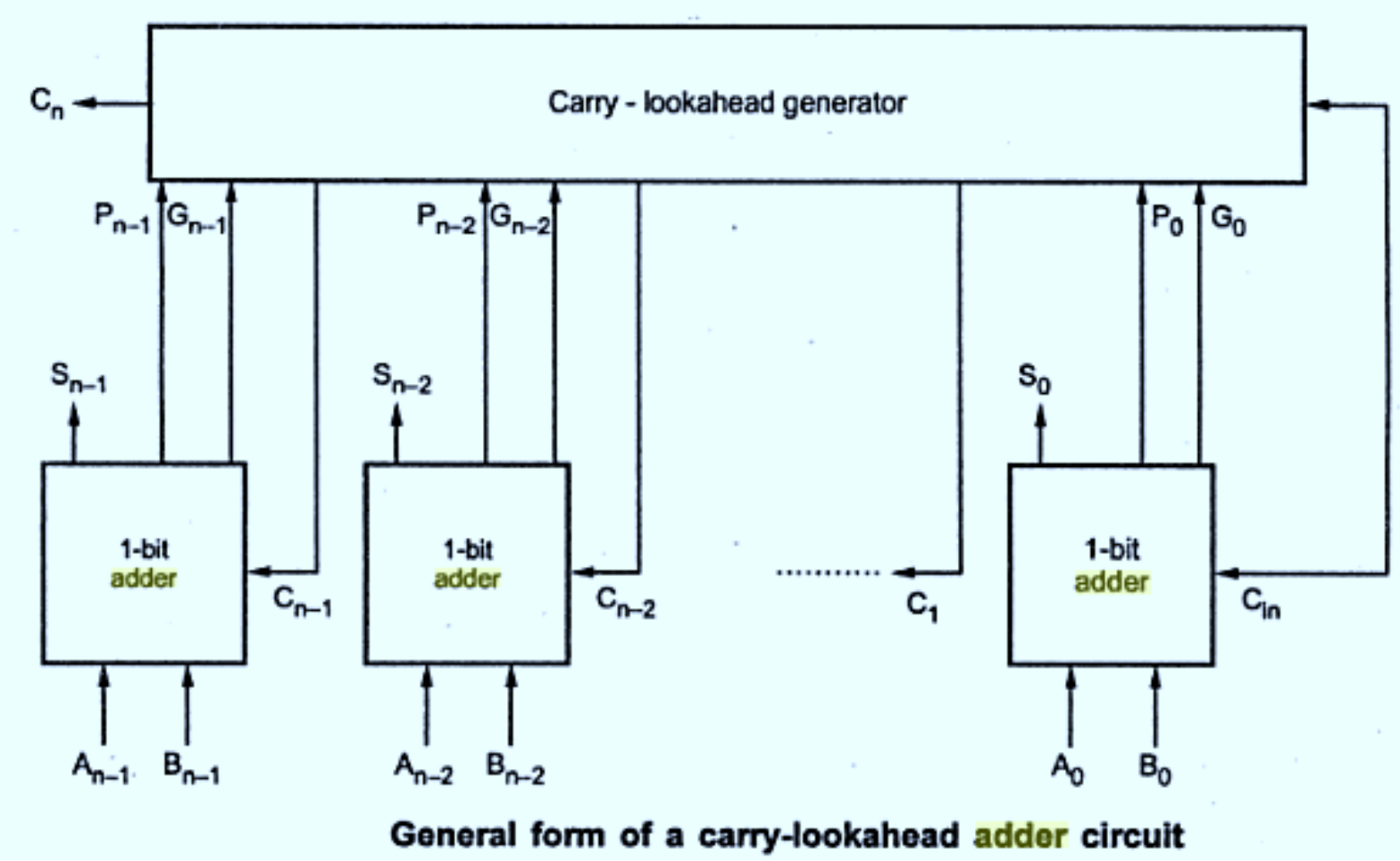
```
  0 1 0 1
+ 0 0 1 1
  1 0 0 0
```

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position (stage), produces a carry into the third position. The latter carry, when added to the bits of the third position, produces a carry into the last position. The key thing to notice in this example is that the sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous positions. This means that, adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in each full-adder. For example, if each full-adder is considered to have a propagation delay of 30 ns, then S_3 will not reach its correct value until 90 ns after LSB carry is generated. Therefore, total time required to perform addition is 90+30 = 120 ns.

Carry Lookahead Adder

- One method of speeding up this process by eliminating inter stage carry delay is called lookahead-carry addition.
- This method look at the lower-order bits of augend & addend to see if a higher order carry is to be generated.
- It uses two functions: carry generate & carry propogate.

N- bit Carry Lookahead Adder



General form of a carry-lookahead adder circuit

Carry-Lookahead Addition

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

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

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

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

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

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

$$c_0 = 0$$

$$c_1 = G_0$$

$$c_2 = G_1 + P_1 G_0$$

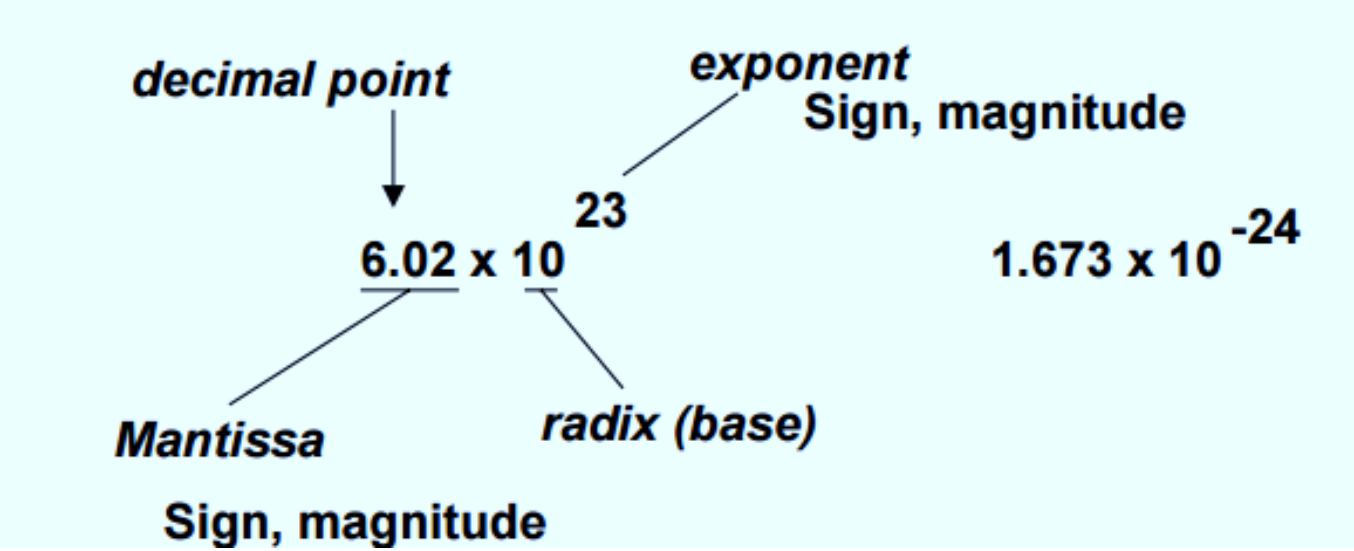
$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0$$

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

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

Floating Point Representation

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576×10^9



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.
- Increase the power of exponent: Right shift
- Decrease the power of exponent: Left shift

Floating Point Arithmetic

- 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 $.101 \times 2^3 = .0101 \times 2^4$,

  rounding off to $.010 \times 2^4$

  $(.0101\text{-}.010) \times 2^4 = .0001 \times 2^4 = .001 \times 2^3$

  Therefore, 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.

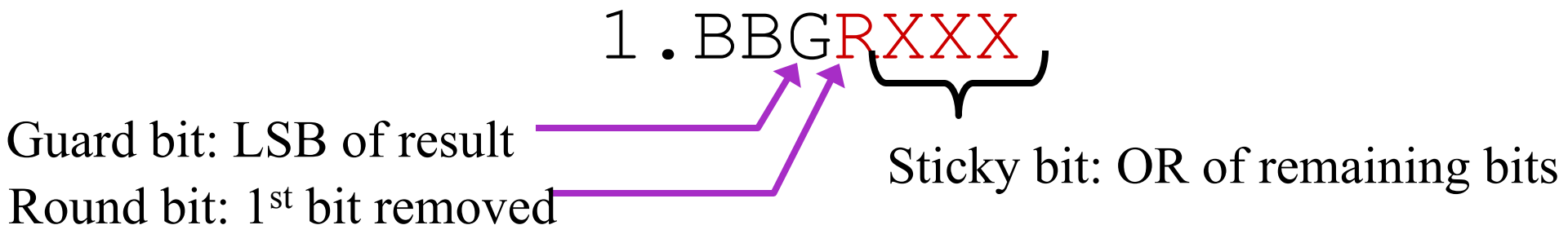
Floating Point Arithmetic (Cont')

- If we simply added the numbers using as much precision as we needed and then applied rounding only in the final normalization step, then the calculation would go like this:

$$.101 \times 2^3 + .111 \times 2^4 =$$
$$.0101 \times 2^4 + .111 \times 2^4 =$$
$$1.0011 \times 2^4.$$

- Normalizing yields .10011 × 2^5, and rounding to three significant digits using the round to nearest even method yields .101 × 2^5.
- Which calculation is correct .100 x 2^5 or .101 x 2^5?
- According to the IEEE 754 standard, the final result should be the same as if the maximum precision needed is used before applying the rounding method, and so the correct result is .101 × 2^5. So what do we do?

Rounding using G, R, S



- If G=1 & R=1, add 1 to LSB
- If G=0 & R=0 or 1, no change
- If G=1 & R=0, look at S
  - If S=1, add 1 to LSB
  - If S=0, round to the nearest even

    (i.e. if LSB =1, then make it 0 (add 1 to LSB) &

    if LSB =0 then no change

    1 = ODD, 0 = EVEN)

Guard, Round, and Sticky Bits

- This raises the issue of how to compute the intermediate results with sufficient accuracy and without requiring too much hardware, and for this we use guard, round, and sticky bits.
- For the previous example, applying guard (g) and round (r) bits with the round toward nearest even method, we have:
-

$$\begin{array}{rll} & .101 & \times 2^3 \\ + & .111 & \times 2^4 \end{array} \quad \Rightarrow \quad \begin{array}{rll} & .0101 & \times 2^4 \\ + & .111 & \times 2^4 \\ \hline & 1.00110 & \times 2^4 \\ & gr & \end{array}$$

Guard, Round, and Sticky Bits

- Only one extra bit is needed for this intermediate result, the guard bit (g), but we also show the round bit (r = 0) to locate its position. As we shift the number to normalize, we set a sticky bit (s) if any of the shifted out bits are nonzero. For this case, there are no nonzero bits to the right of the r bit and so s = 0:

Guard, Round, and Sticky Bits (Cont')

- Now for the rounding step: simply append the sticky bit to the right of the result before rounding. There is no tie as there would be for .100100 and so we round up, otherwise we would have rounded down to the closest even number (.100):

$$1.0011 \times 2^4 = 1.00110 \times 2^4$$

gr

0 is shifted out, so $s = 0$

- For this case, the guard, round, and sticky bits changed our previous result. Note that if r is 0 instead of 1, so that the grs combination is 100, we would have rounded down to .100 because .100 is even whereas .101 is not.

$$.100110 \times 2^5 \cong .101 \times 2^5$$

grs

Examples: Guard, Round, and Sticky Bits

Fraction	GRS	Incr?	Rounded	
1.000000	000	N	1.000	1.101100
100	Y		1.110	1.000010
010	N		1.001	110
Y			1.011	111 Y
			10.000	

If G=1 & R=1, add 1 to LSB
If G=0 & R=0 or 1, no change
If G=1 & R=0, look at S
    If S=1, add 1 to LSB
    If S=0, round to the nearest even
        (i.e. if LSB =1, then make it 0 (add 1 to LSB) &
        if LSB =0 then no change
        1 = ODD, 0 = EVEN)

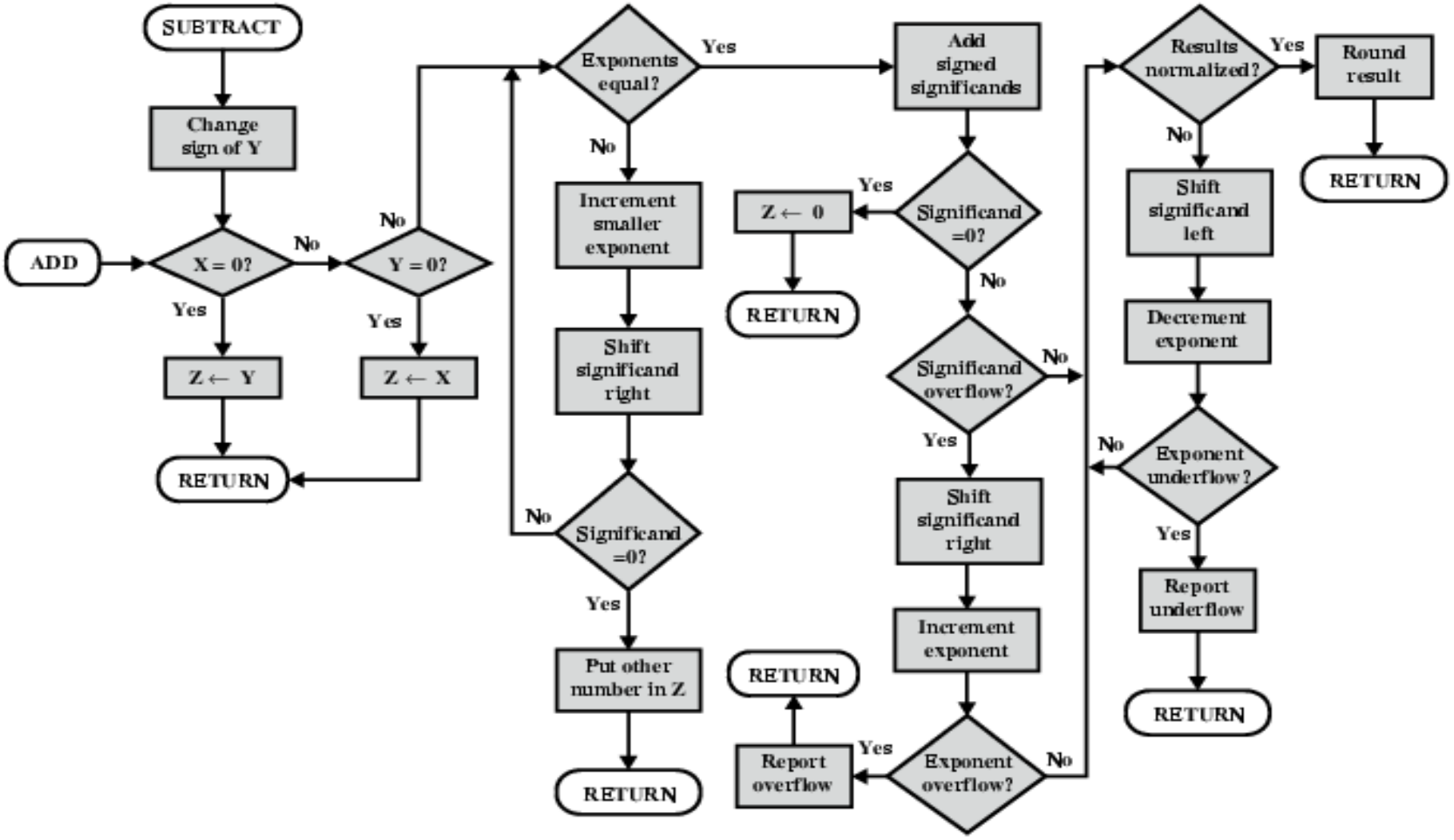
Floating Point Multiplication/Division

- 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 $(+.110 \times 2^2)$.

FP Arithmetic +/-

- Check for zeros
- Align significands (adjusting exponents)
- Add or subtract significands
- Normalize result

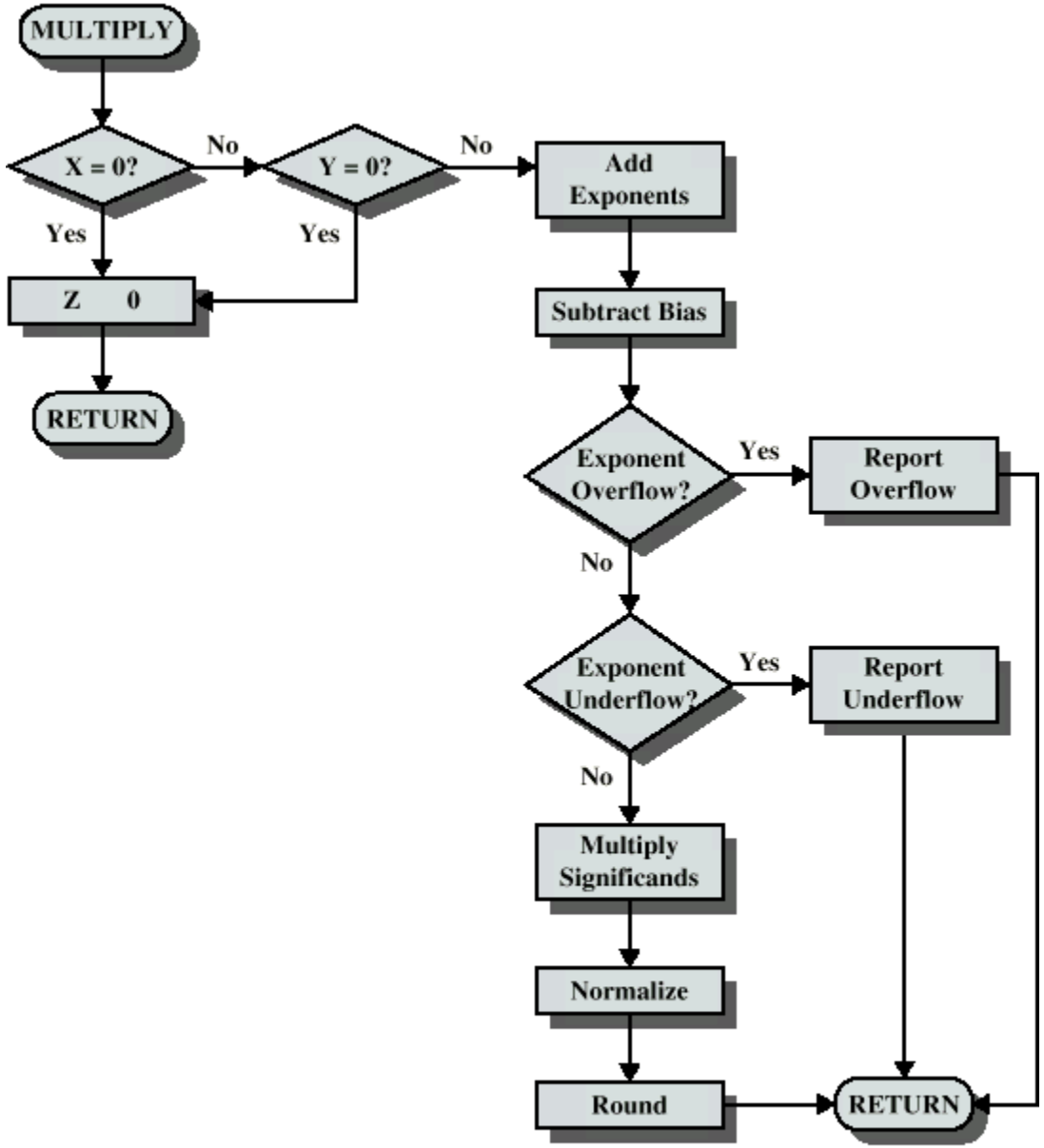
FP Addition & Subtraction Flowchart



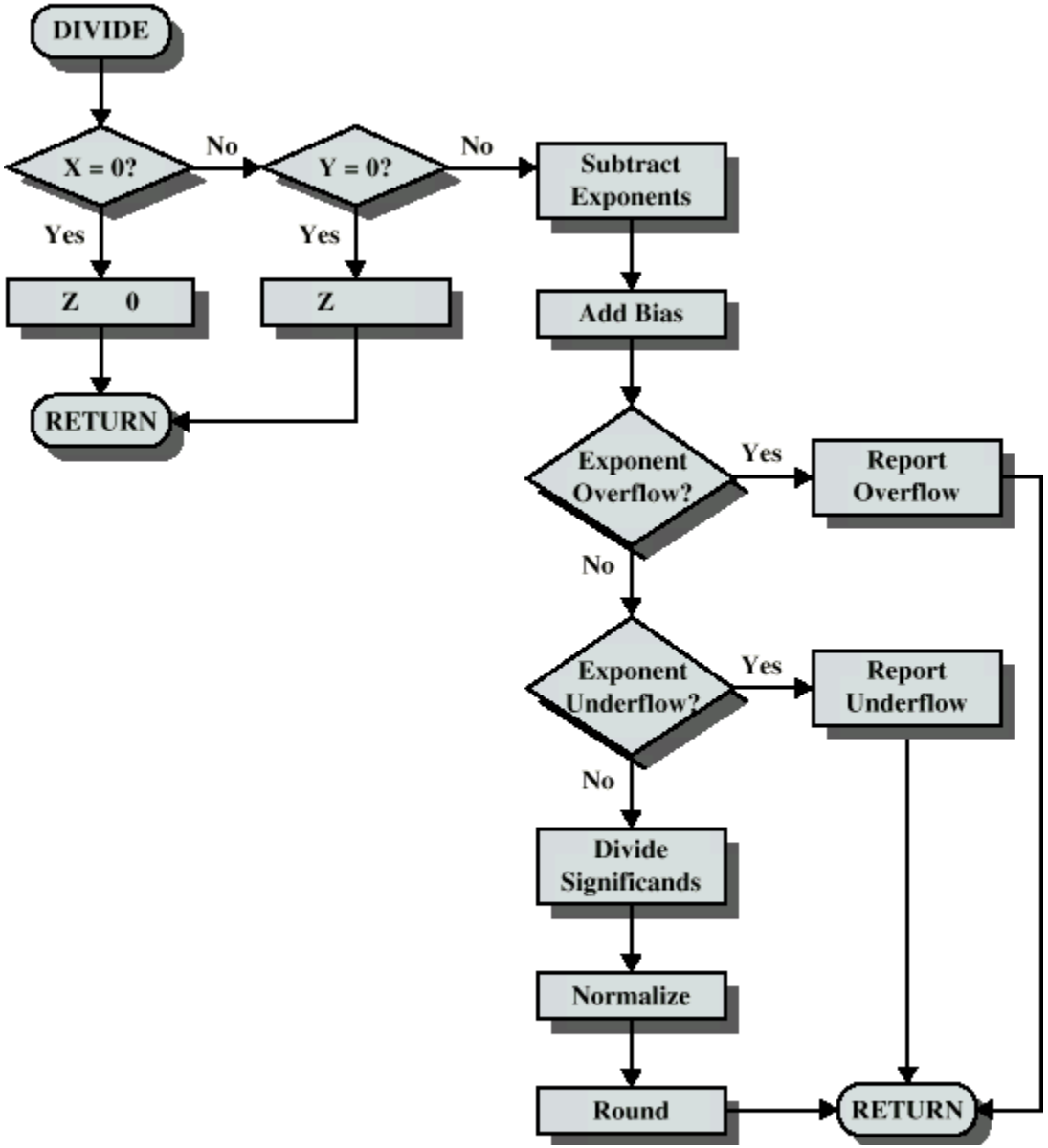
FP Arithmetic x/÷

- Check for zero
- Add/subtract exponents
- Multiply/divide significands (watch sign)
- Normalize
- Round
- All intermediate results should be in double length storage

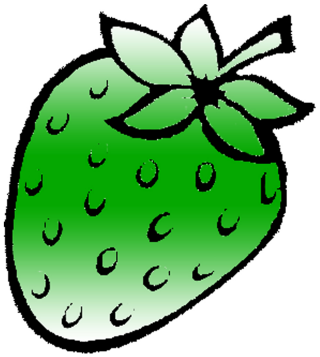
Floating Point Multiplication



Floating Point Division



STRAWBERRY



/strawberrydevelopers

/strawberry_app

For more visit:

Strawberrydevelopers.weebly.com