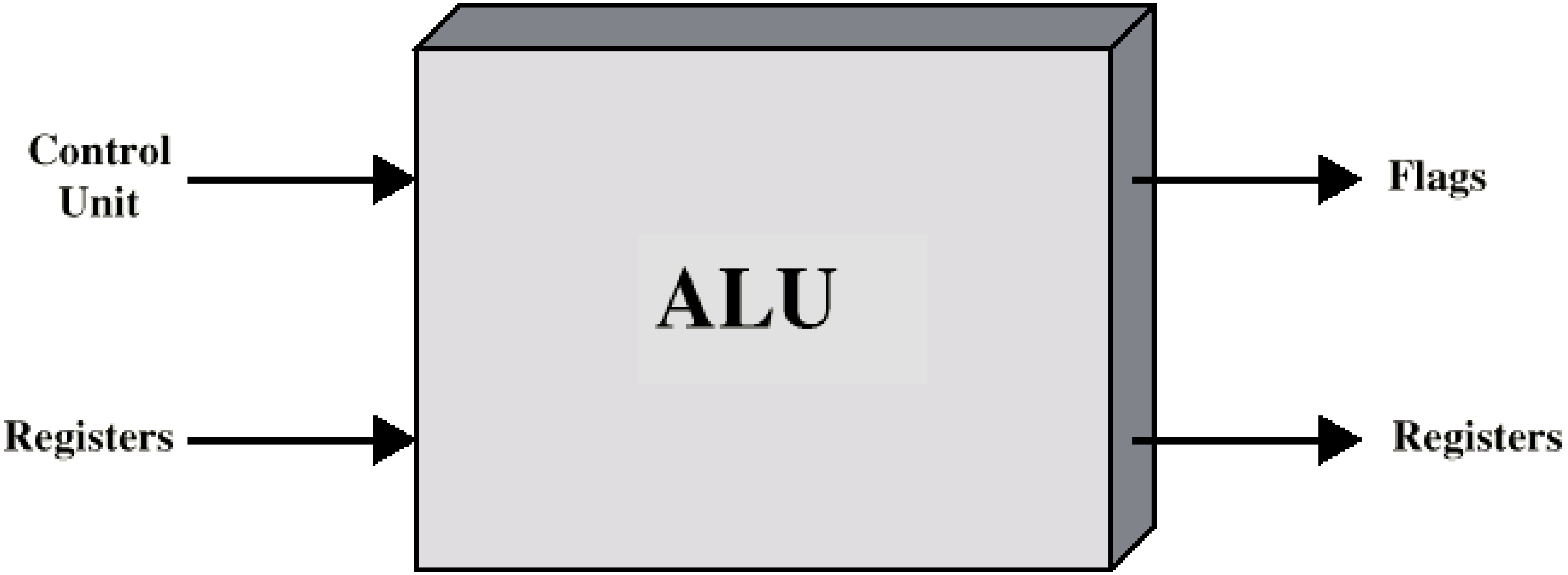
Module No-2

Basics of ALU Design

Module No-2: Basics of ALU Design:

- Binary number representation;
- Fixed and Floating point representation of numbers.
- Adders: Serial and Parallel adders, Ripple Carry / Carry Look ahead / Carry Save;
- Multipliers & Divider Circuits: Multiplication of signed binary numbers Booth Multipliers;

ALU Inputs and Outputs



Integer Representation

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
  - e.g. 41=00101001
- No minus sign
- No period
- Sign-Magnitude
- Two’s compliment

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)

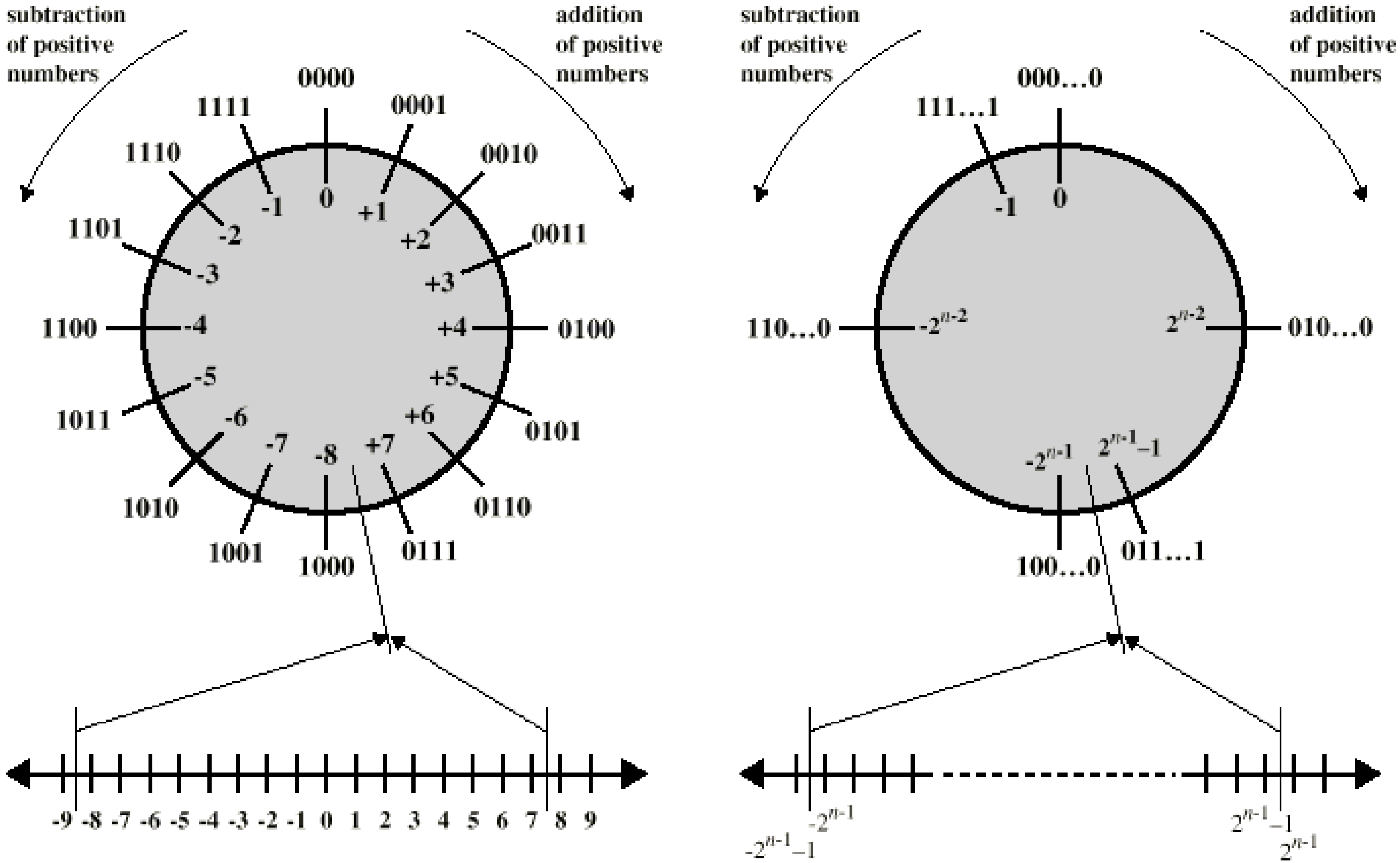
Two’s Compliment

- +3 = 00000011
- +2 = 00000010
- +1 = 00000001
- +0 = 00000000
- -1 = 11111111
- -2 = 11111110
- -3 = 11111101

Benefits

- One representation of zero
- Arithmetic works easily (see later)
- Negating is fairly easy
  - 3 = 00000011
  - Boolean complement gives 11111100
  - Add 1 to LSB 11111101

Geometric Depiction of Twos Complement Integers



(a) 4-bit numbers

(b) n-bit numbers

Negation Special Case 1

- 0 = 00000000
- Bitwise not 11111111
- Add 1 to LSB +1
- Result 1 00000000
- Overflow is ignored, so:
- - 0 = 0 √

Negation Special Case 2

- -128 = 10000000
- bitwise not 01111111
- Add 1 to LSB +1
- Result 10000000
- So:
- -(-128) = -128 X
- Monitor MSB (sign bit)
- It should change during negation

Range of Numbers

- 8 bit 2s compliment
  - +127 = 01111111 = 2^7 -1
  - -128 = 10000000 = -2^7
- 16 bit 2s compliment
  - +32767 = 011111111 11111111 = 2^{15} - 1
  - -32768 = 100000000 00000000 = -2^{15}

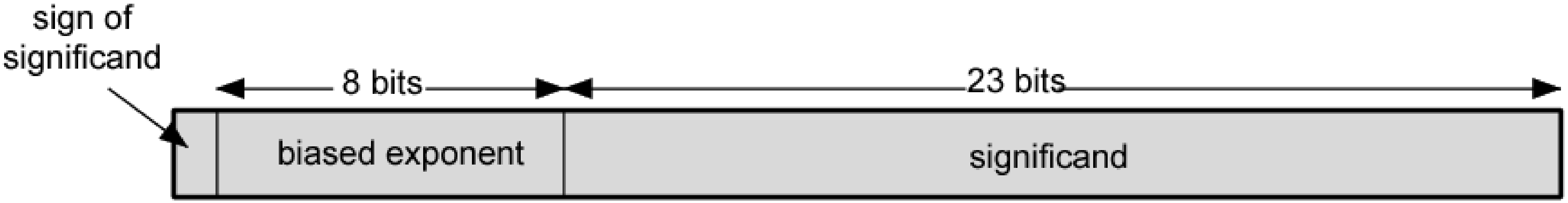
Conversion Between Lengths

- Positive number pack with leading zeros
- +18 = 00010010
- +18 = 00000000 00010010
- Negative numbers pack with leading ones
- -18 = 10010010
- -18 = 11111111 10010010
- i.e. pack with MSB (sign bit)

Real Numbers

- Numbers with fractions
- Could be done in pure binary
  - $1001.1010 = 2^{3} + 2^{0} + 2^{-1} + 2^{-3} = 9.625$
- Where is the binary point?
- Fixed?
  - Very limited
- Moving?
  - How do you show where it is?

Floating Point

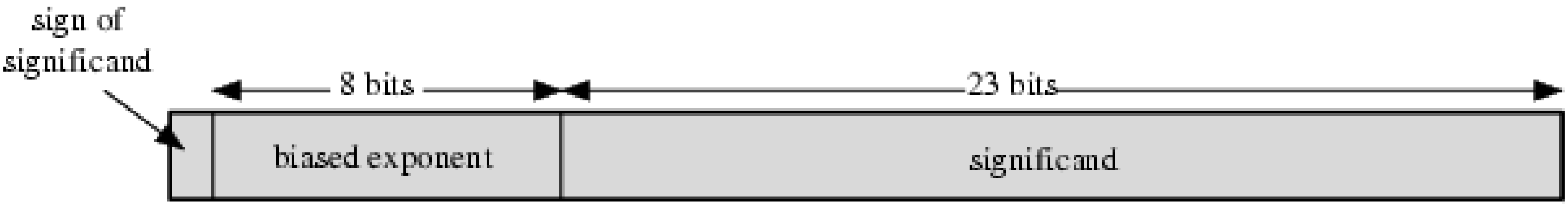


(a) Format

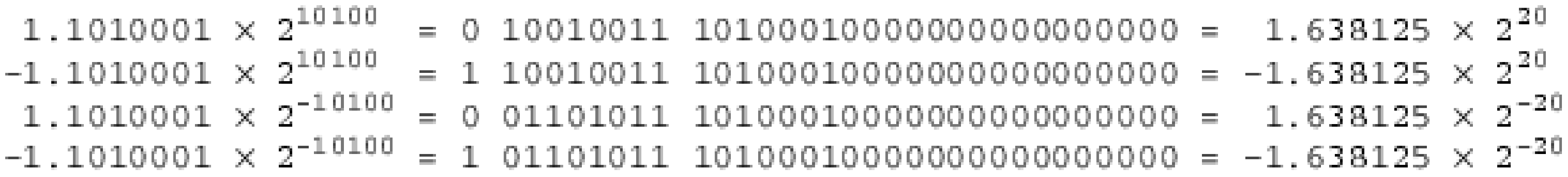
- +/- .significand x 2^{exponent}
- Point is actually fixed between sign bit and body of mantissa
- Exponent indicates place value (point position)

Misnomer:เรียกชื่อผิด, ใช้ชื่อ

Floating Point Examples



(a) Format



$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$

$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$

(b) Examples

Signs for Floating Point

- Mantissa is stored in 2s compliment
- Exponent is in excess or biased notation
  - e.g. Excess (bias) 128 means
  - 8 bit exponent field
  - Pure value range 0-255
  - Subtract 128 to get correct value
  - Range -128 to +127

Normalization

- FP numbers are usually normalized
- i.e. exponent is adjusted so that leading bit (MSB) of mantissa is 1
- Since it is always 1 there is no need to store it
- (c.f. Scientific notation where numbers are normalized to give a single digit before the decimal point e.g. 3.123 x 10^3)

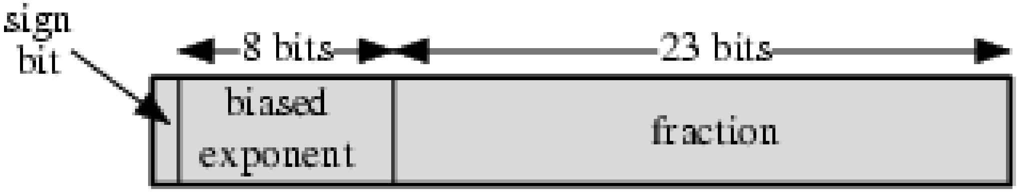
FP Ranges

- For a 32 bit number
  - 8 bit exponent
  - +/- $2^{256} \approx 1.5 \times 10^{77}$
- Accuracy
  - The effect of changing lsb of mantissa
  - 23 bit mantissa $2^{-23} \approx 1.2 \times 10^{-7}$
  - About 6 decimal places

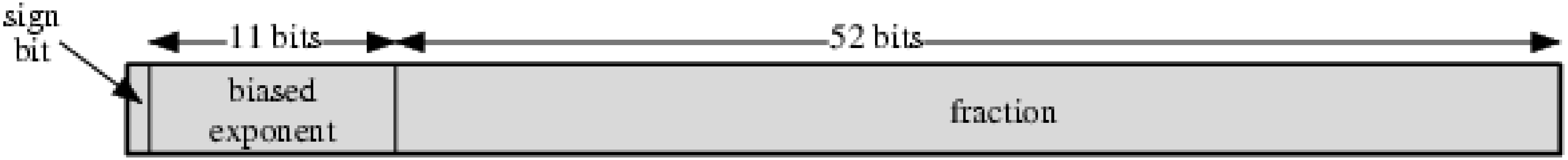
IEEE 754

- Standard for floating point storage
- 32 and 64 bit standards
- 8 and 11 bit exponent respectively
- Extended formats (both mantissa and exponent) for intermediate results

IEEE 754 Formats



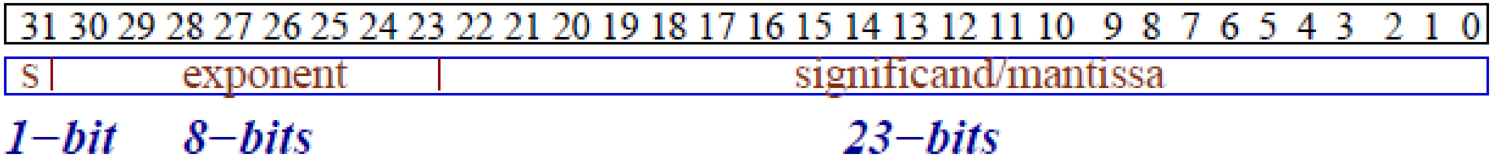
(a) Single format



(b) Double format

IEEE 754 Formats

- Most of the binary floating-point representations follow the IEEE-754 standard.
- The data type float uses IEEE 32-bit single precision format and the data type double uses IEEE 64-bit double precision format.
- A floating-point constant is treated as a double precision number by GCC.

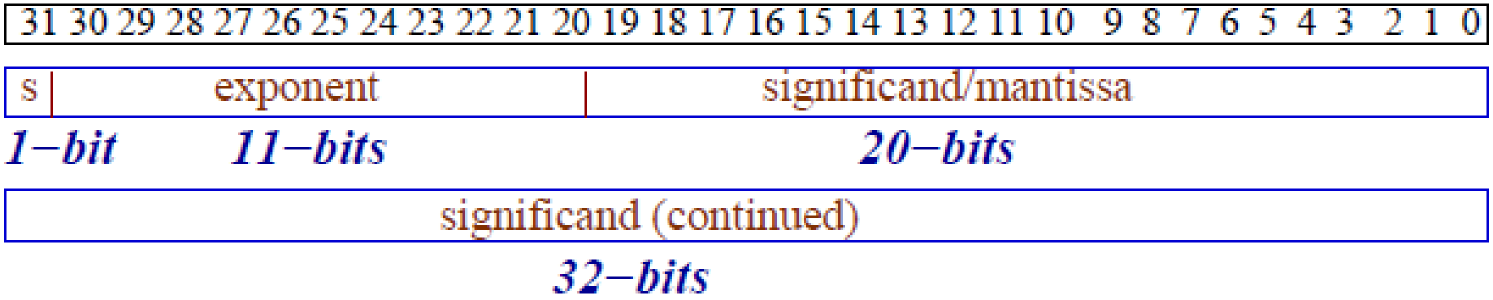


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

s | exponent | significand/mantissa

1−bit *8−bits* *23−bits*

Single Precession (32−bit)



31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

s | exponent | significand/mantissa

1−bit *11−bits* *20−bits*

significand (continued)

32−bits

Double Precession (64−bit)

Interpretation of Bits

- The most significant bit indicates the sign of the number - one is negative and zero is positive.
- The next eight bits (11 in case of double precession) store the value of the signed exponent of two ($2^{biasedExp}$).
- Remaining 23 bits (52 in case of double precession) are for the significand (mantissa).

Types of Data

- Data represented in this format are classified in five groups.
- Normalized numbers,
- Zeros,
- Subnormal(denormal) numbers,
- Infinity and not-a-number (nan).

Single Precession Data: Interpretation

Single Precision		Data Type
Exponent	*Significand*	
0	0	± 0
0	nonzero	± subnormal number
1 - 254	anything	± normalized number
255	0	$\pm\infty$
255	nonzero	NaN (not a number)

Double Precession Data

Double Precision		Data Type
Exponent	*Significand*	
0	0	±0
0	nonzero	± subnormal number
1 - 2046	anything	± normalized number
2047	0	$\pm\infty$
2047	nonzero	NaN (not a number)

Single Precession Normalized Number

- Let the sign bit (31) be s, the exponent (30-23)

be e and the mantissa (significand or fraction)

(22-0) be m.

The valid range of the exponents is 1 to 254 (if e is treated as an unsigned number).

- The actual exponent is biased by 127
- To get e i.e. the actual value of the exponent is e − 127.
- This gives the range: $2^{1-127} = 2^{-126}$ to $2^{254-127} = 2^{127}$.

Single Precession Normalized Number

- The normalized significand is 1.m (binary dot).

The binary point is before bit-22 andthe 1 (one) is not present explicitly.

- The sign bit s = 1 for a −ve number is zero (0) for a +ve number.
- The value of a normalized number is

$$\boxed{(-1)^s \times 1.m \times 2^{e-127}}$$

An Example

Consider the following 32-bit pattern

1 1011 0110 011 0000 0000 0000 0000 0000

The value is

$$
\begin{aligned}
& (-1)^{1} \times 2^{10110110-01111111} \times 1.011 \\
= \; & -1.375 \times 2^{55} \\
= \; & -49539595901075456.0 \\
= \; & -4.9539595901075456 \times 10^{16}
\end{aligned}
$$

An Example

Consider the decimal number: +105.625. The equivalent binary representation is

$$
\begin{aligned}
& +1101001.101 \\
= \;& +1.101001101 \times 2^{6} \\
= \;& +1.101001101 \times 2^{133-127} \\
= \;& +1.101001101 \times 2^{10000101-01111111}
\end{aligned}
$$

In IEEE 754 format:

$$\boxed{0\ 1000\ 0101\ 101\ 0011\ 0100\ 0000\ 0000\ 0000}$$

An Example

Consider the decimal number: +2.7. The equivalent binary representation is

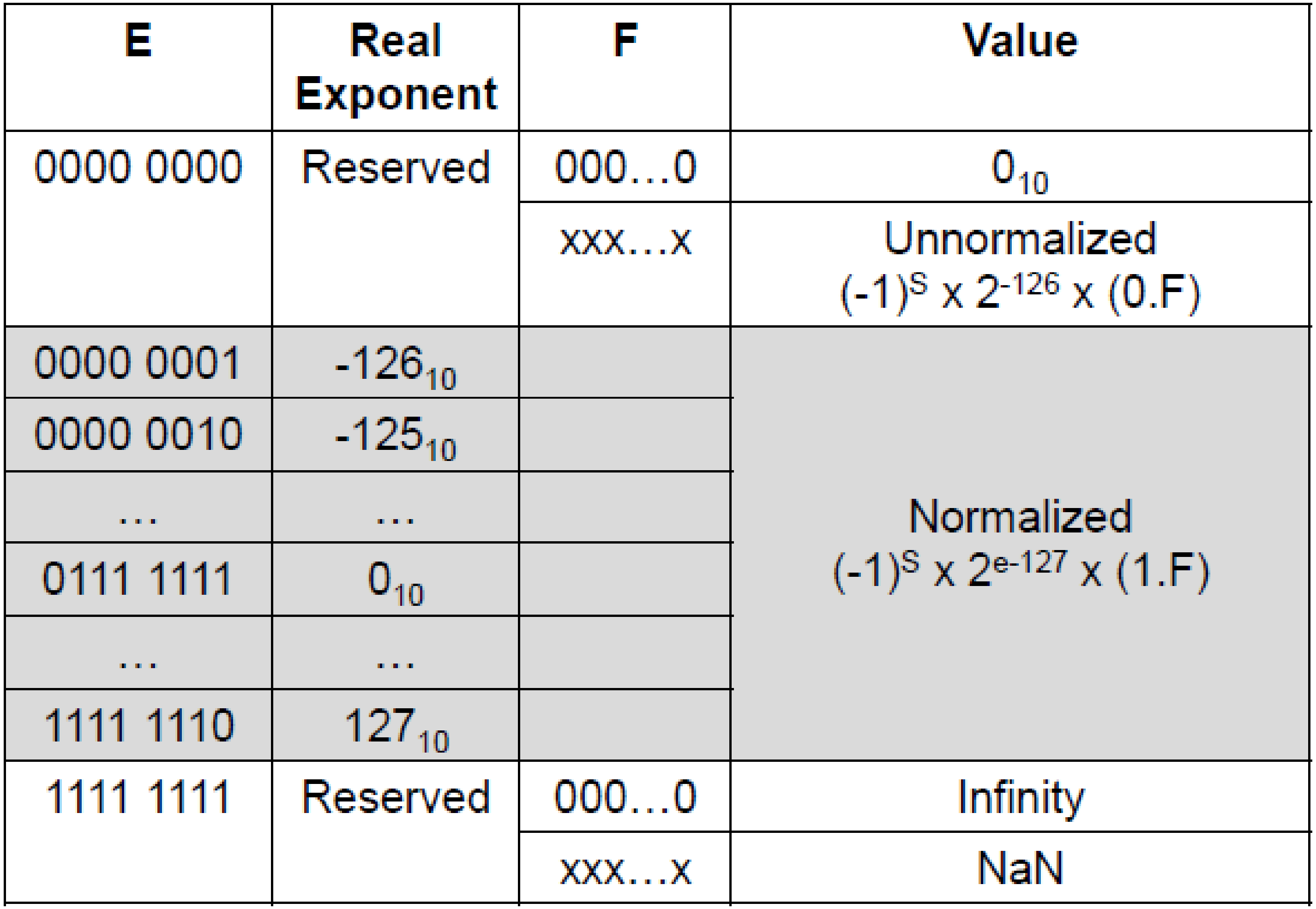
$$
\begin{aligned}
& +10.10\;1100\;1100\;1100\cdots \\
= \; & +1.010\;1100\;1100\cdots \times 2^{1} \\
= \; & +1.010\;1100\;1100\cdots \times 2^{128-127} \\
= \; & +1.010\;1100\cdots \times 2^{10000000-01111111}
\end{aligned}
$$

In IEEE 754 format (approximate):

$$\boxed{0\;1000\;0000\;010\;1100\;1100\;1100\;1100\;1101}$$

e		Actual Exponent
0000 0000		Reserved
0000 0001	1-127 = -126	-126_{10}
0000 0010	2-127 = -125	-125_{10}
…		…
0111 1111		0_{10}
…		…
1111 1110	254-127=127	127_{10}
1111 1111		Reserved

E	F	meaning	Notes
00000000	0…0	0	+0.0 and -0.0
00000000	X…X	Valid number	Unnormalized $=(-1)^S \times 2^{-126} \times (0.F)$
11111111	0…0	Infinity	
11111111	X…X	Not a Number	



E	Real Exponent	F	Value
0000 0000	Reserved	000…0	0_{10}
		xxx…x	Unnormalized $(-1)^S$ x 2^{-126} x (0.F)
0000 0001	-126_{10}		Normalized $(-1)^S$ x 2^{e-127} x (1.F)
0000 0010	-125_{10}		
…	…		
0111 1111	0_{10}		
…	…		
1111 1110	127_{10}		
1111 1111	Reserved	000…0	Infinity
		xxx…x	NaN

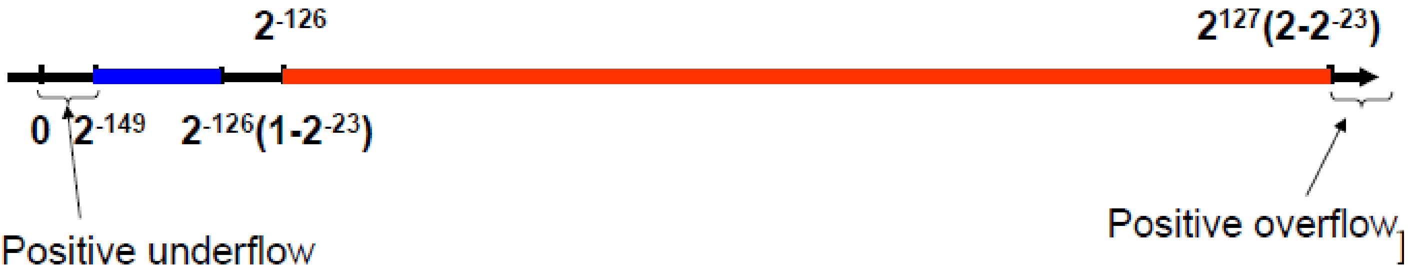
Range of numbers

- ❑ Normalized (positive range; negative is symmetric)

smallest	0 00000001 00000000000000000000000	$+2^{-126}(1+0) = 2^{-126}$
largest	0 11111110 11111111111111111111111	$+2^{127}(2-2^{-23})$

- ❑ Unnormalized

smallest	0 00000000 00000000000000000000001	$+2^{-126}(2^{-23}) = 2^{-149}$
largest	0 00000000 11111111111111111111111	$+2^{-126}(1-2^{-23})$

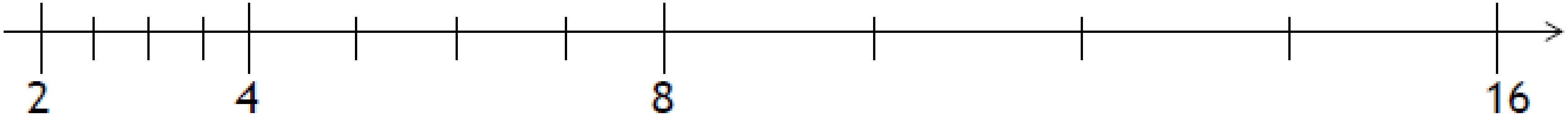


- The smallest magnitude of a normalized number in single precession is
- ± 0000 0001 000 0000 0000 0000 0000 0000, whose value is 1.0 × 2^{-126} .
- The largest magnitude of a normalized number in single precession is
- ± 1111 1110 111 1111 1111 1111 1111 1111, whose value is
- $1.99999988 \times 2^{127} \approx 3.403 \times 10^{38}$

- The smallest magnitude of a subnormal number in single precession is ± 0000 0000 000 0000 0000 0000 0000 0001, whose value is $2^{-126+(-23)} = 2^{-149}$.
- The largest magnitude of a subnormal number in single precession is ± 0000 0000 111 1111 1111 1111 1111 1111, whose value is $0.99999988 \times 2^{-126}$.

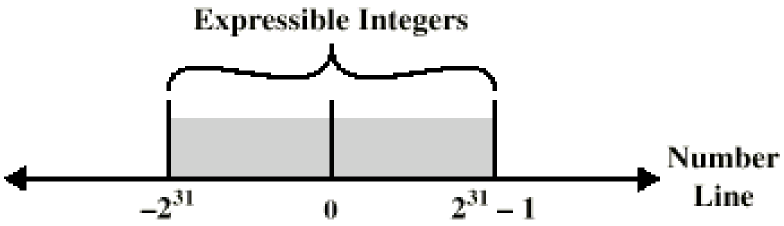
- The smallest subnormal 2^{-149} is closer to zero.
- The largest subnormal $0.99999988 \times 2^{-126}$ is closer to the smallest normalized number 1.0×2^{-126}.

- ❑ There *aren't* more IEEE numbers.
- ❑ With 32 bits, there are 2^{32}, or about 4 billion, different bit patterns.
  - These can represent 4 billion integers *or* 4 billion reals.
  - But there are an infinite number of reals, and the IEEE format can only represent *some* of the ones from about -2^{128} to $+2^{128}$.
  - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}

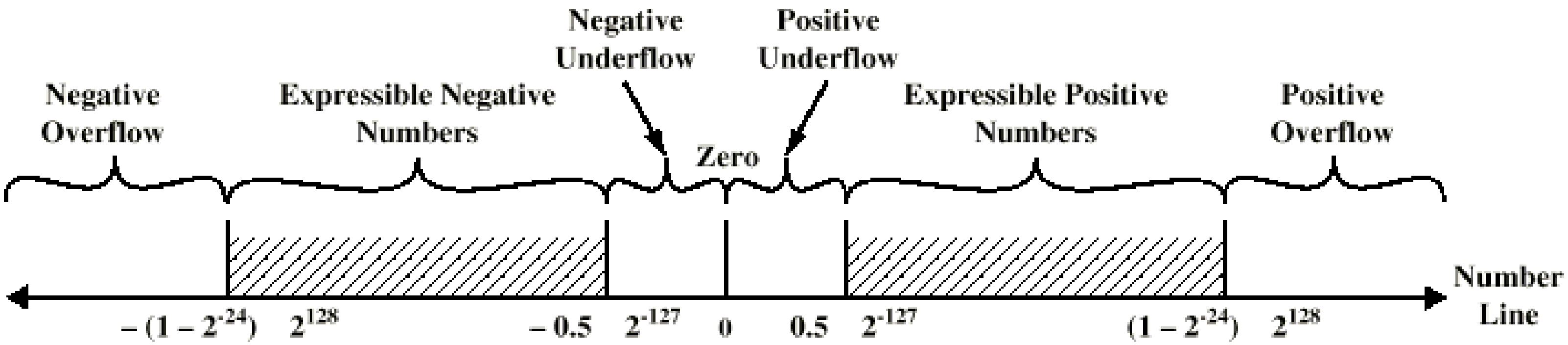


- ❑ Thus, floating-point arithmetic has "issues"
  - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
  - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- ❑ The IEEE 754 standard guarantees that all machines will produce the same results—but those results may not be mathematically accurate!

Expressible Numbers

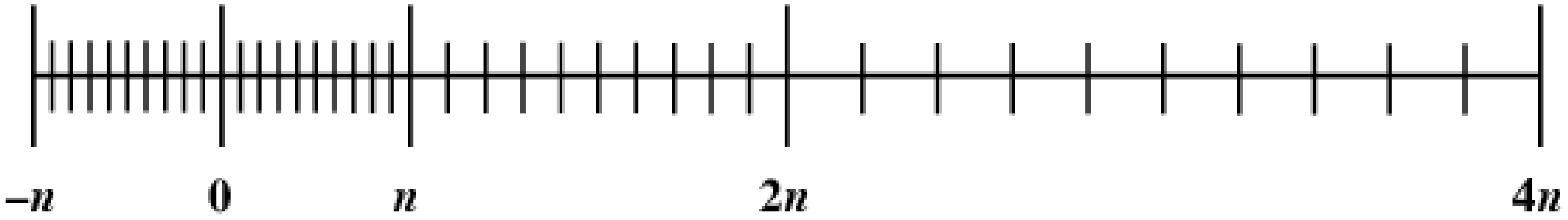


(a) Twos Complement Integers



(b) Floating-Point Numbers

Density of Floating Point Numbers



Arithmetic

Addition/subtraction of signed numbers

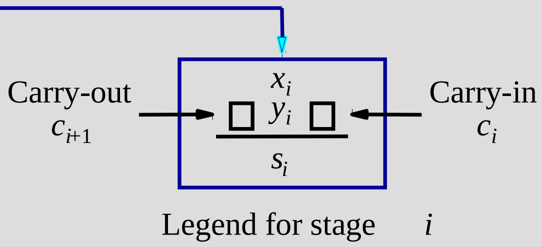
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

$$\begin{array}{r} X \\ +\ Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\ 6 \\ \hline 13 \end{array} = \begin{array}{rrrrr} & 0 & 1 & \boxed{1} & 1 \\ +\ {}_0 & 0 & {}_1\ 1 & {}_1\ \boxed{1} & {}_0\ 0\ {}_0 \\ \hline & 1 & 1 & \boxed{0} & 1 \end{array}$$



Legend for stage i

At the i^{th} stage:

Input:

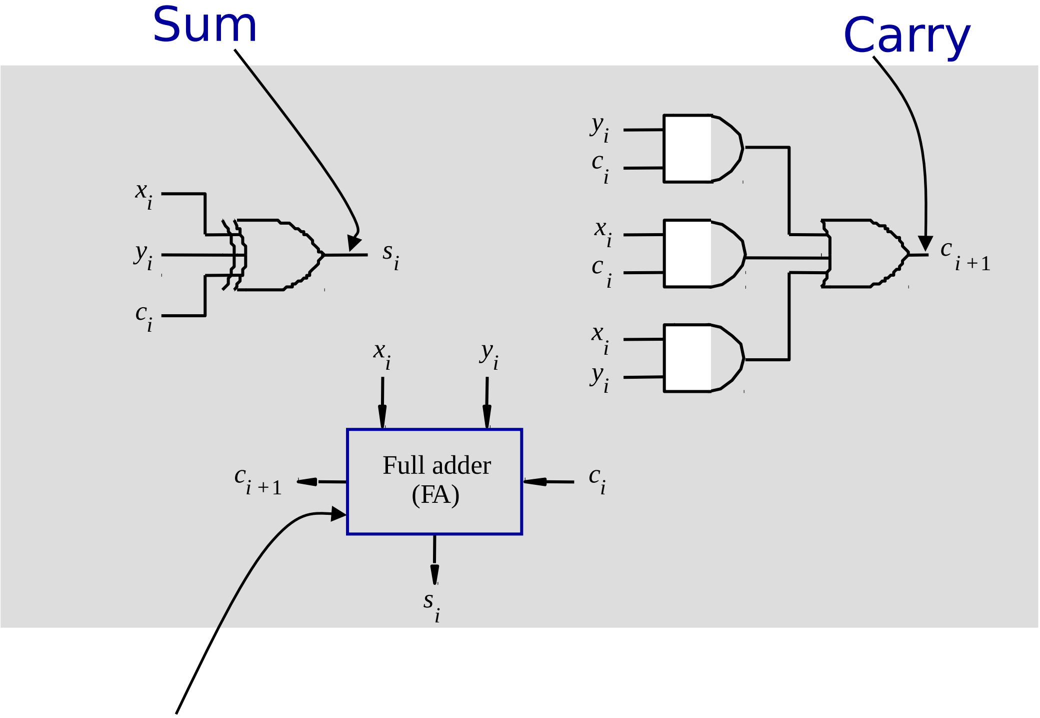
c_i is the carry-in

Output:

s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

Addition logic for a single stage

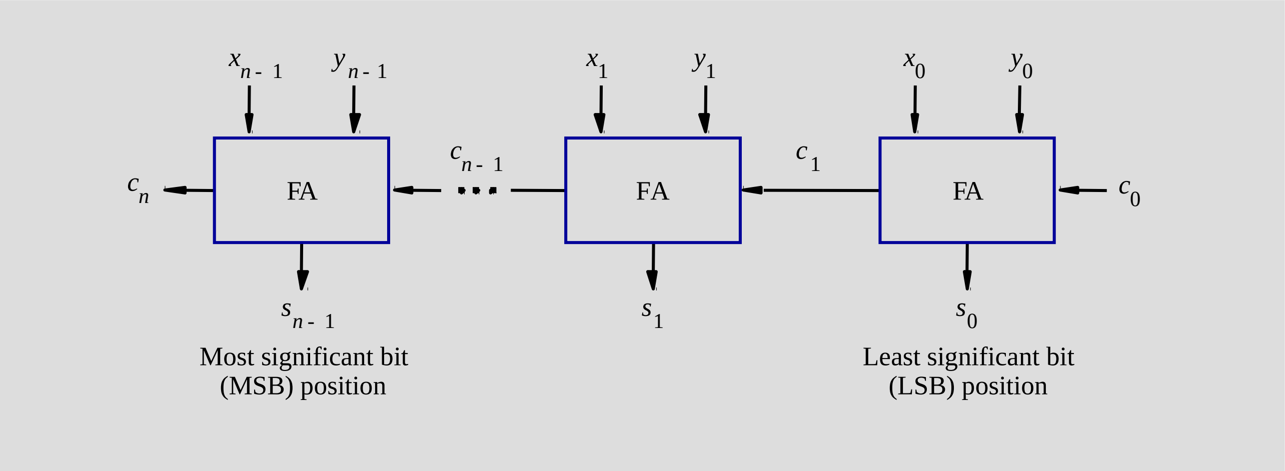


Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n-bit adder

Cascade *n* full adder (FA) blocks to form a *n*-bit adder.

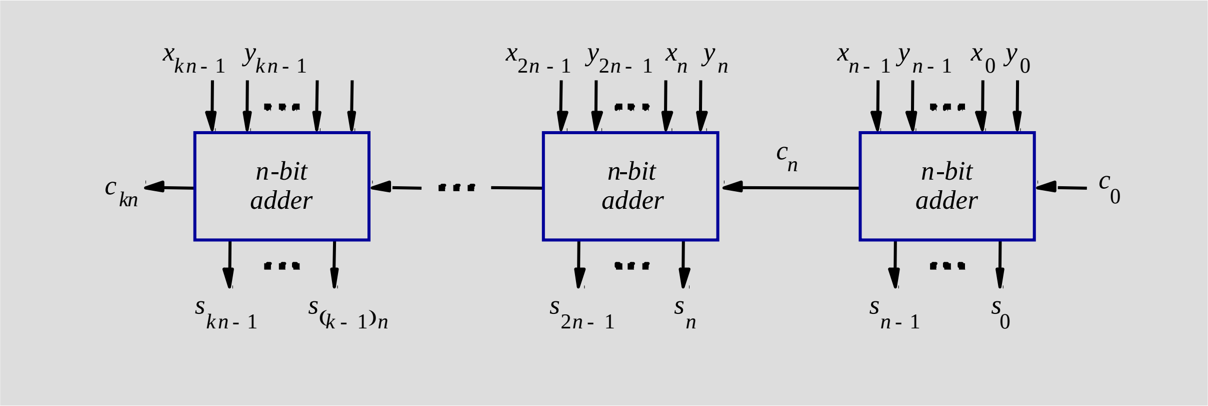
Carries propagate or ripple through this cascade, *n*-bit ripple carry adder.



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

K n-bit adder

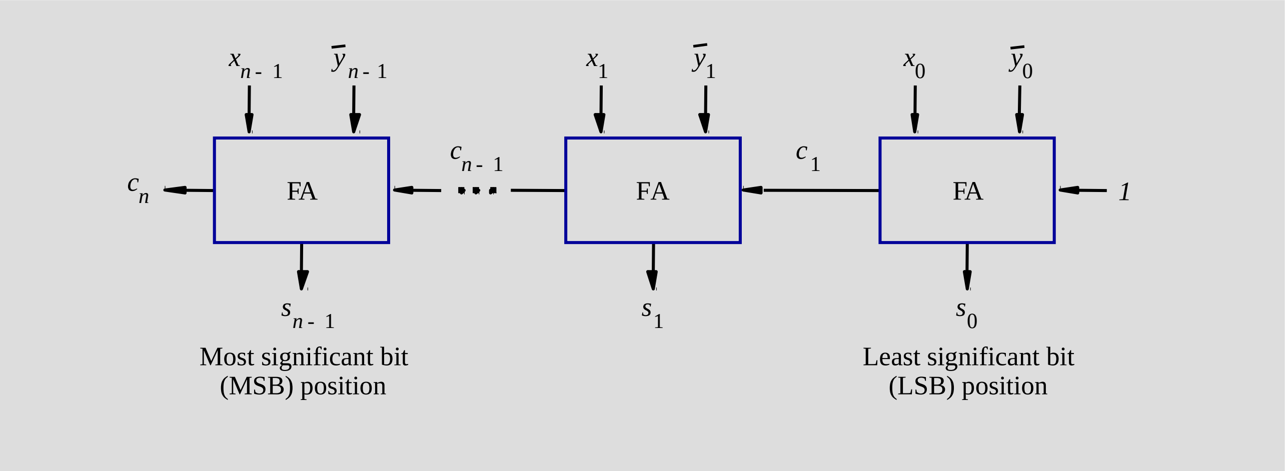
K n-bit numbers can be added by cascading *k n*-bit adders.



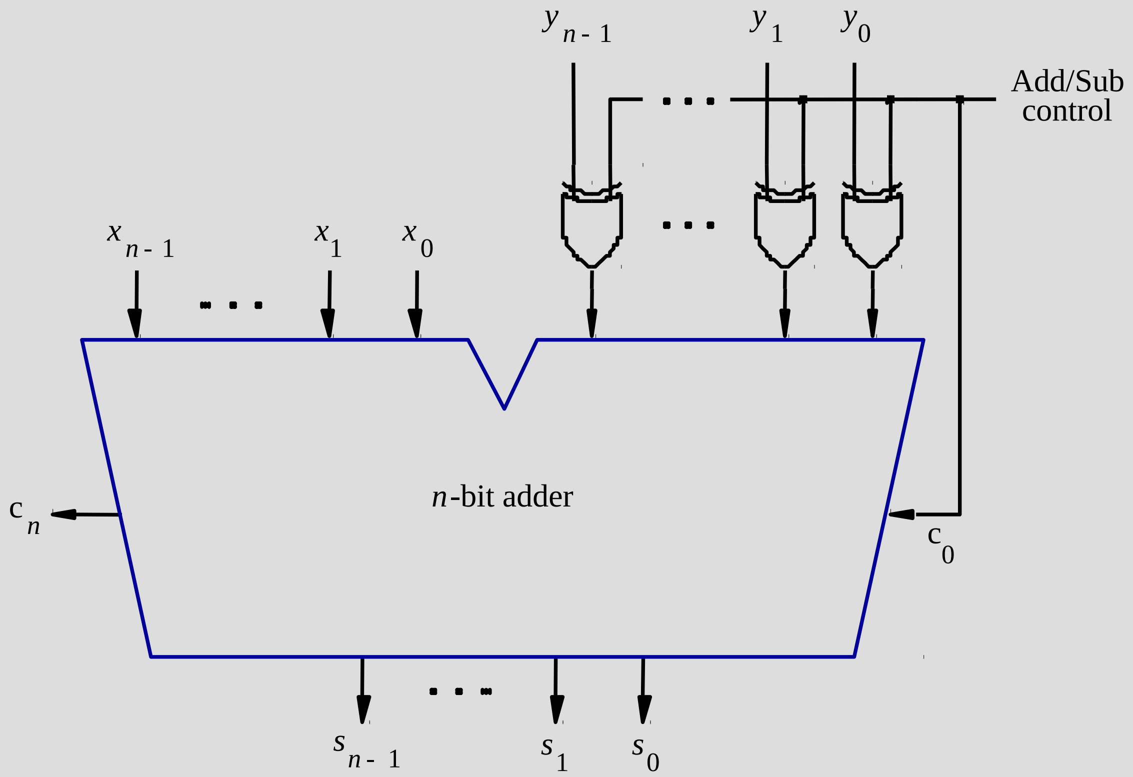
Each *n*-bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

n-bit subtractor

- Recall *X* – *Y* is equivalent to adding 2’s complement of *Y* to *X*.
- 2’s complement is equivalent to 1’s complement + 1.
- $X - Y = X + \overline{Y} + 1$
- 2’s complement of positive and negative numbers is computed similarly.



n-bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Detecting overflows

Overflows can only occur when the sign of the two operands is the same.

Overflow occurs if the sign of the result is different from the sign of the operands.

Recall that the MSB represents the sign.

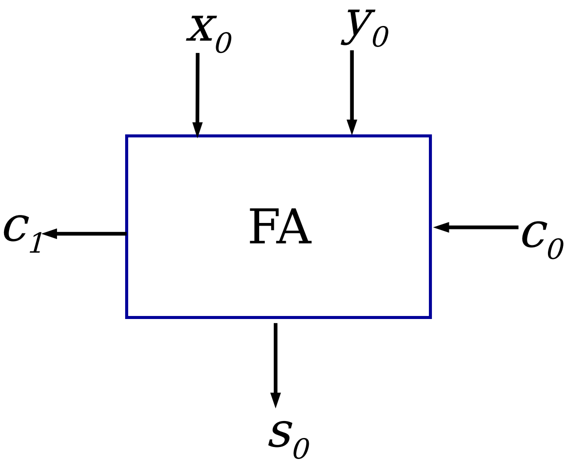
- x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x*, operand *y* and result *s* respectively.

Circuit to detect overflow can be implemented by the following logic expressions:

$$Overflow = x_{n-1} y_{n-1} \overline{s}_{n-1} + \overline{x}_{n-1} \overline{y}_{n-1} s_{n-1}$$

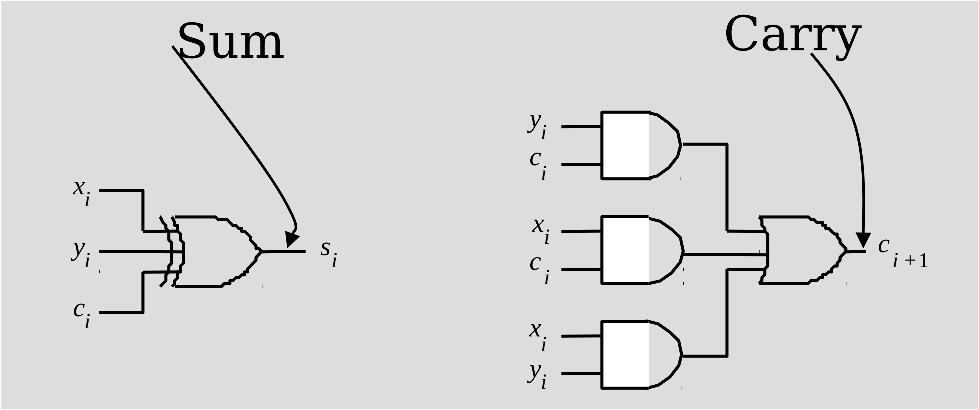
$$Overflow = c_n \oplus c_{n-1}$$

Computing the add time



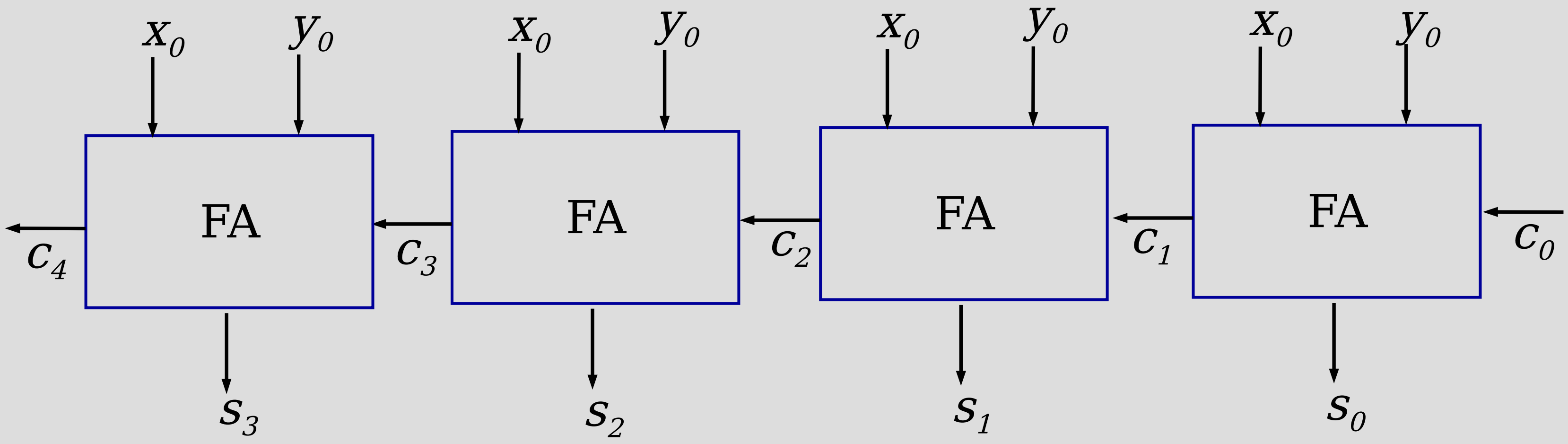
Consider *0th* stage:

- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Computing the add time (contd..)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an n-bit adder, s_{n-1} is available after $2n-1$ gate delays
c_n is available after $2n$ gate delays.

Fast addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

We can write:

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

$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Carry lookahead

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

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow\ c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

continuing

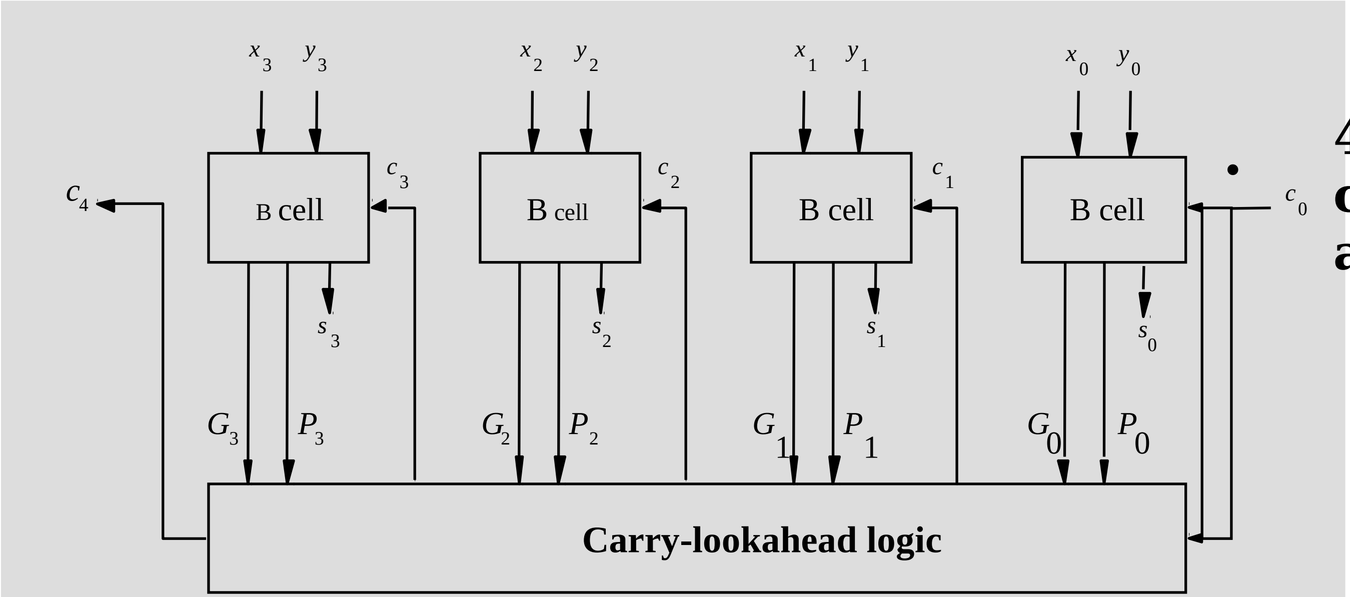
$$\Rightarrow\ c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

until

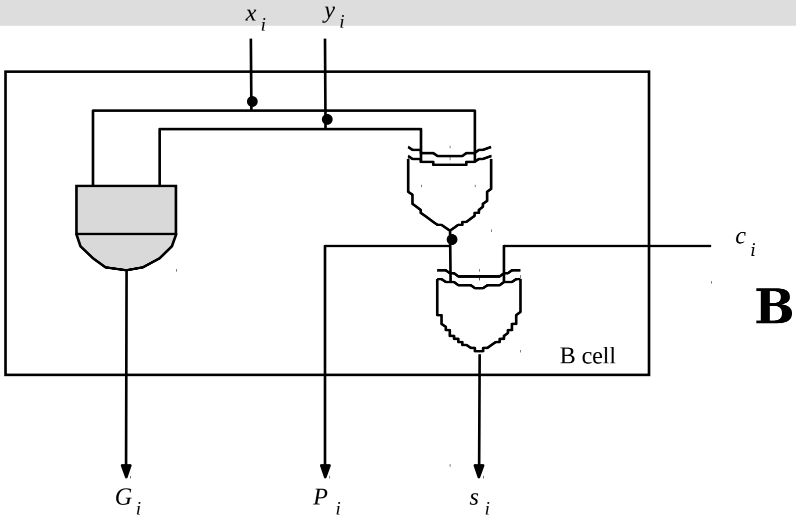
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- All carries can be obtained 3 gate delays after X, Y and c_0 are applied.
  - -One gate delay for P_i and G_i
  - -Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n, n-bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.

Carry-lookahead adder



4-bit carry-lookahead adder



B-cell for a single stage

Carry lookahead adder (contd..)

Performing n-bit addition in 4 gate delays independent of n is good only theoretically because of fan-in constraints.

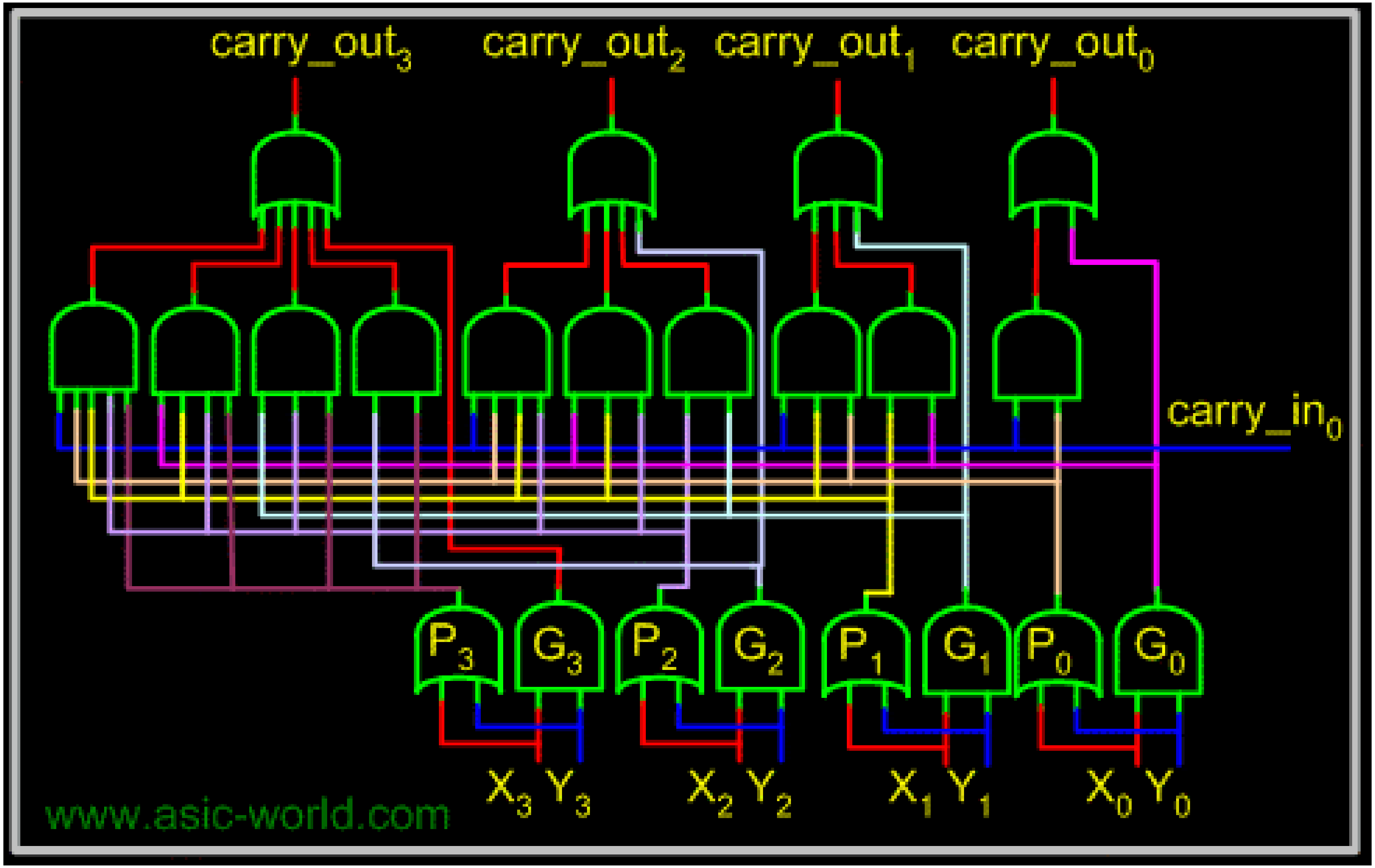
$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + .. + P_iP_{i-1}..P_1G_0 + P_iP_{i-1}...P_0c_0$$

Last AND gate and OR gate require a fan-in of (n+1) for a n-bit adder.

- For a 4-bit adder ($n=4$) fan-in of 5 is required.
- Practical limit for most gates.

In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

4-bit carry-lookahead Adder



Blocked Carry-Lookahead adder

Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

Rewrite this as:

$$P_0^I = P_3 P_2 P_1 P_0$$

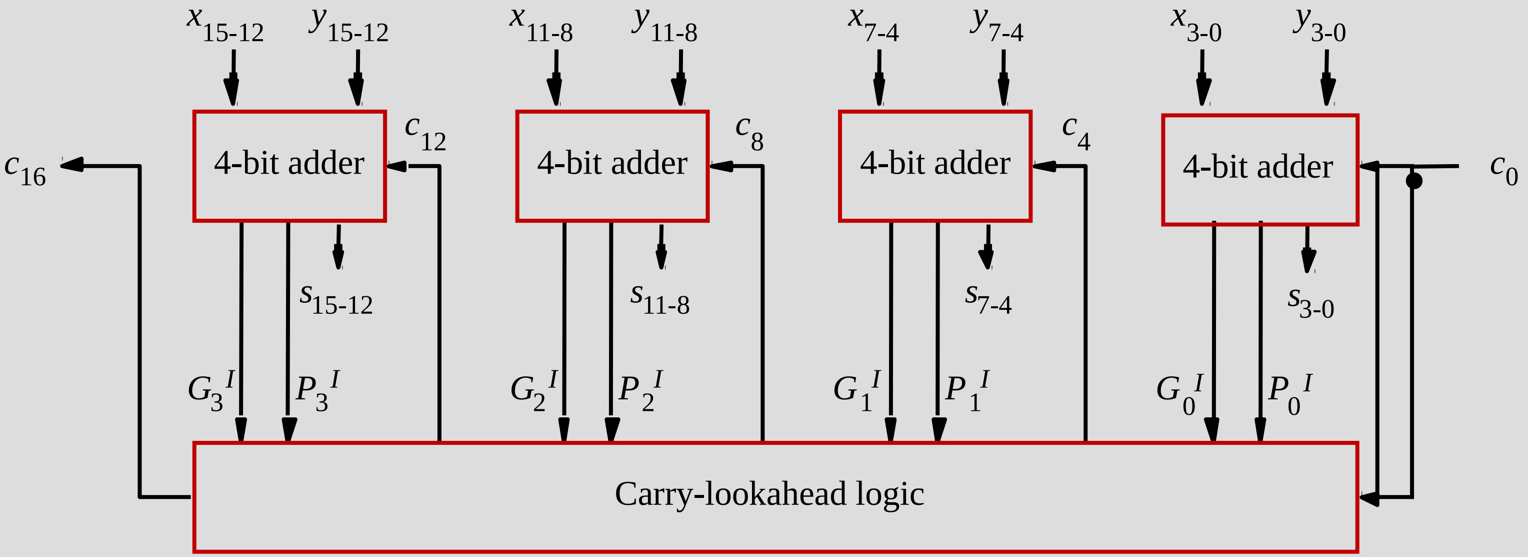
$$G_0^I = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 *4*-bit adders, c_{16} can be expressed as:

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^0 G_0^I + P_3^I P_2^I P_1^0 P_0^0 c_0$$

Blocked Carry-Lookahead adder

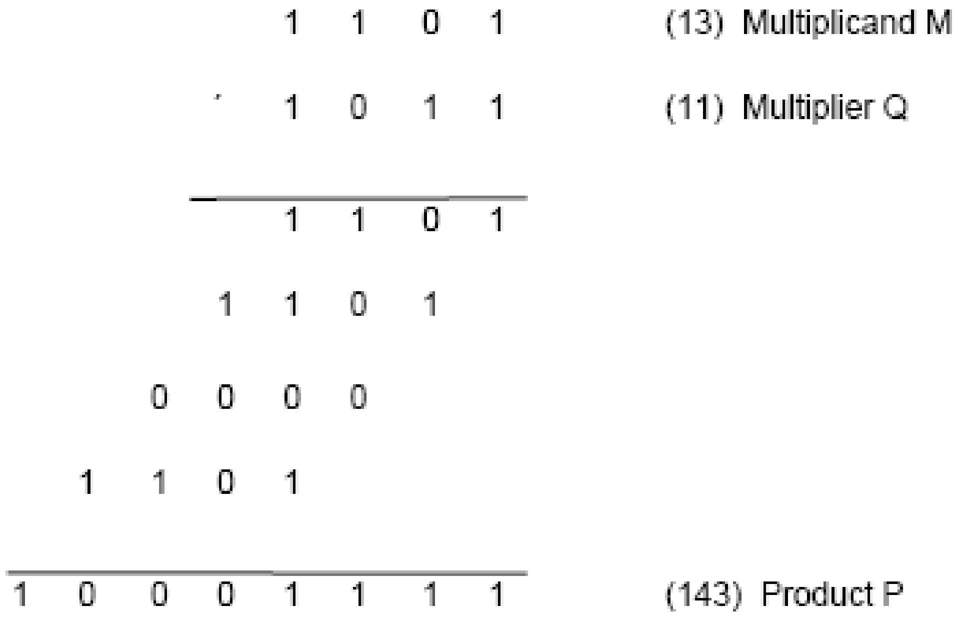


After x_i, y_i and c_0 are applied as inputs:

- G_i and P_i for each stage are available after 1 gate delay.
- P^I is available after 2 and G^I after 3 gate delays.
- All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.
- s_{15} which depends on c_{12} is available after 8 (5+3)gate delays (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

Multiplication

Multiplication of unsigned numbers



```
            1   1   0   1      (13)  Multiplicand M
            1   0   1   1      (11)  Multiplier Q
        -------------------
            1   1   0   1
        1   1   0   1
    0   0   0   0
1   1   0   1
-------------------------------
1   0   0   0   1   1   1   1  (143)  Product P
```

roduct of 2 *n*-bit numbers is at most a *2n*-bit number.

nsigned multiplication can be viewed as addition of shifted ersions of the multiplicand.

Multiplication of unsigned numbers (contd..)

We added the partial products at end.

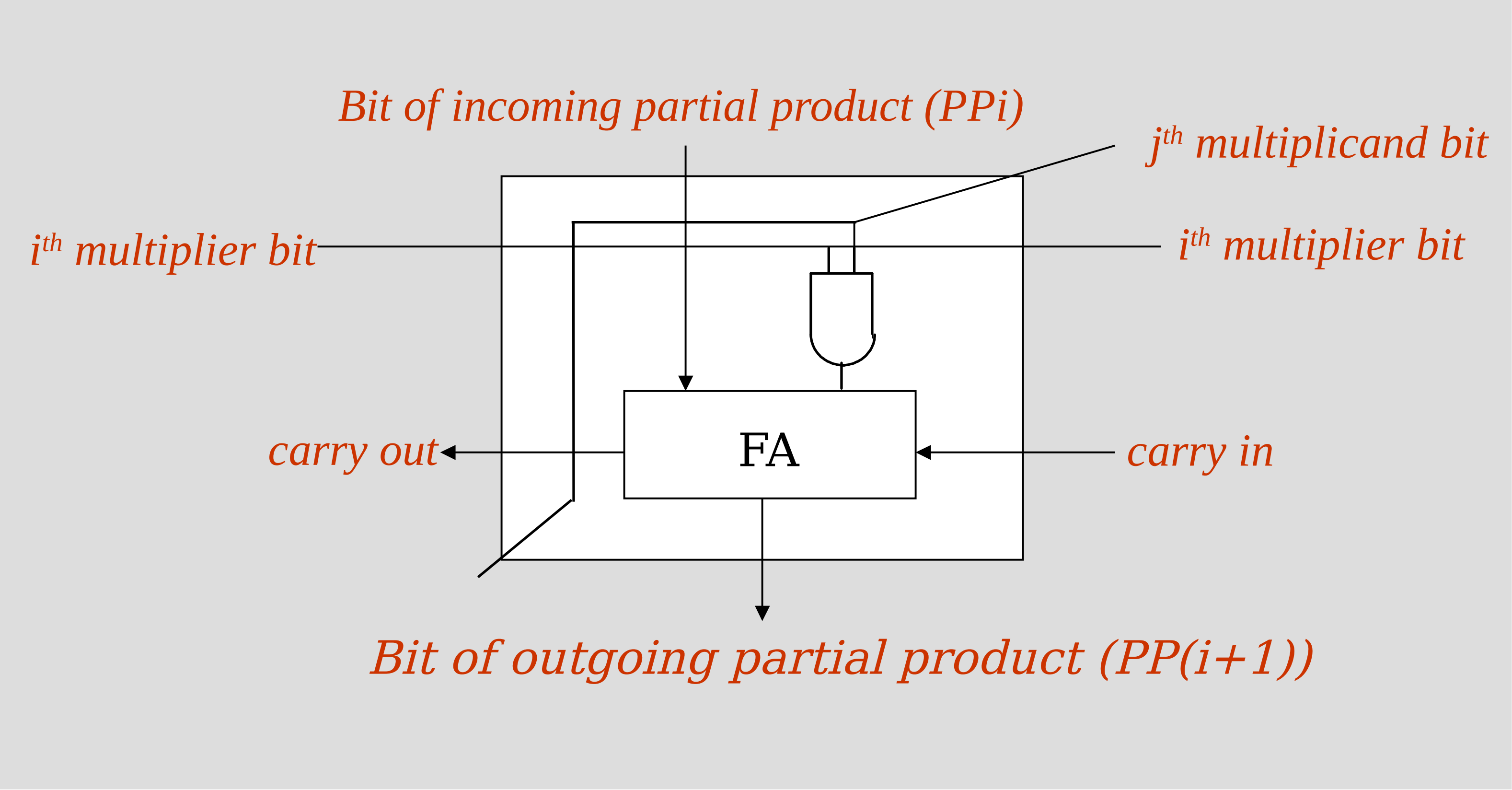
- Alternative would be to add the partial products at each stage.

Rules to implement multiplication are:

- If the i^{th} bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
- Hand over the partial product to the next stage
- Value of the partial product at the start stage is 0.

Multiplication of unsigned numbers

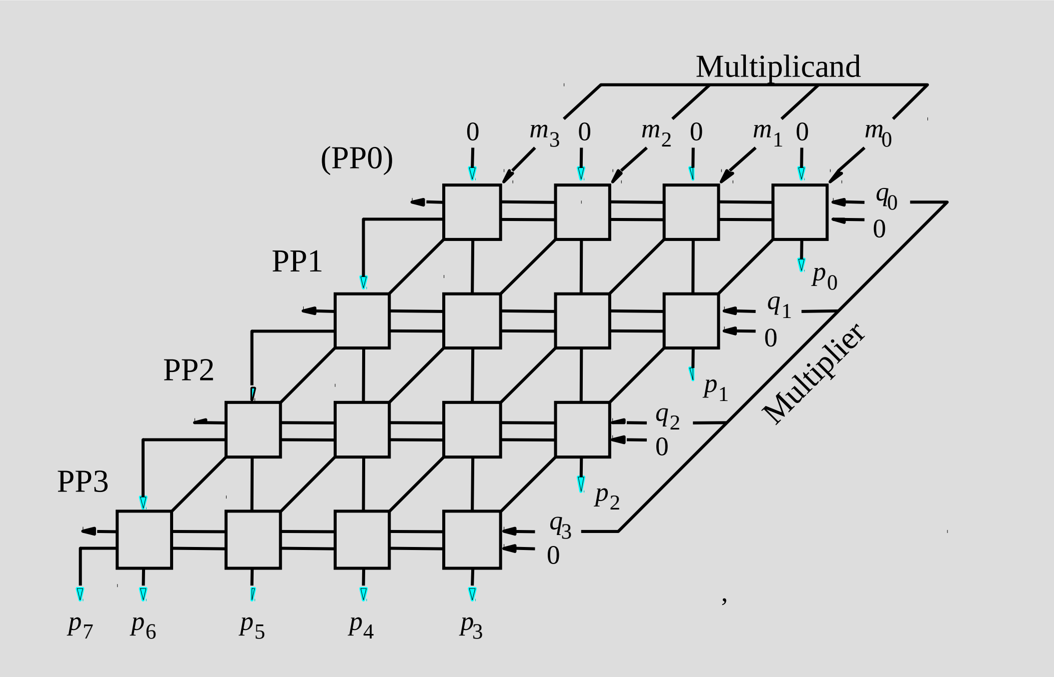
Typical multiplication cell



Bit of incoming partial product (PPi)
jth multiplicand bit
ith multiplier bit
ith multiplier bit
FA
carry out
carry in
Bit of outgoing partial product (PP(i+1))

Combinatorial array multiplier

Combinatorial array multiplier



Product is: $p_7, p_6, ..p_0$

Multiplicand is shifted by displacing it through an array of adders.

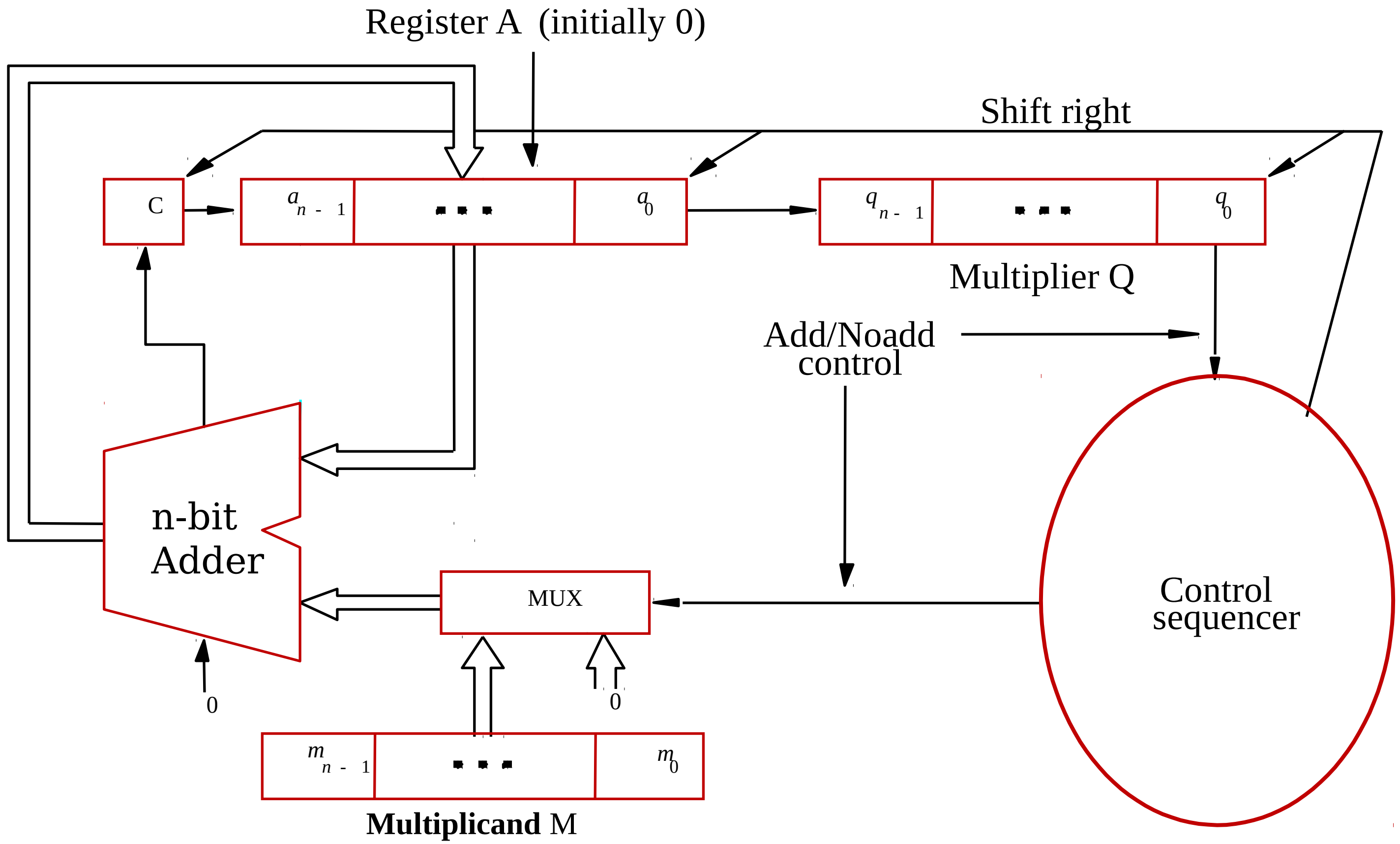
Combinatorial array multiplier (contd..)

- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

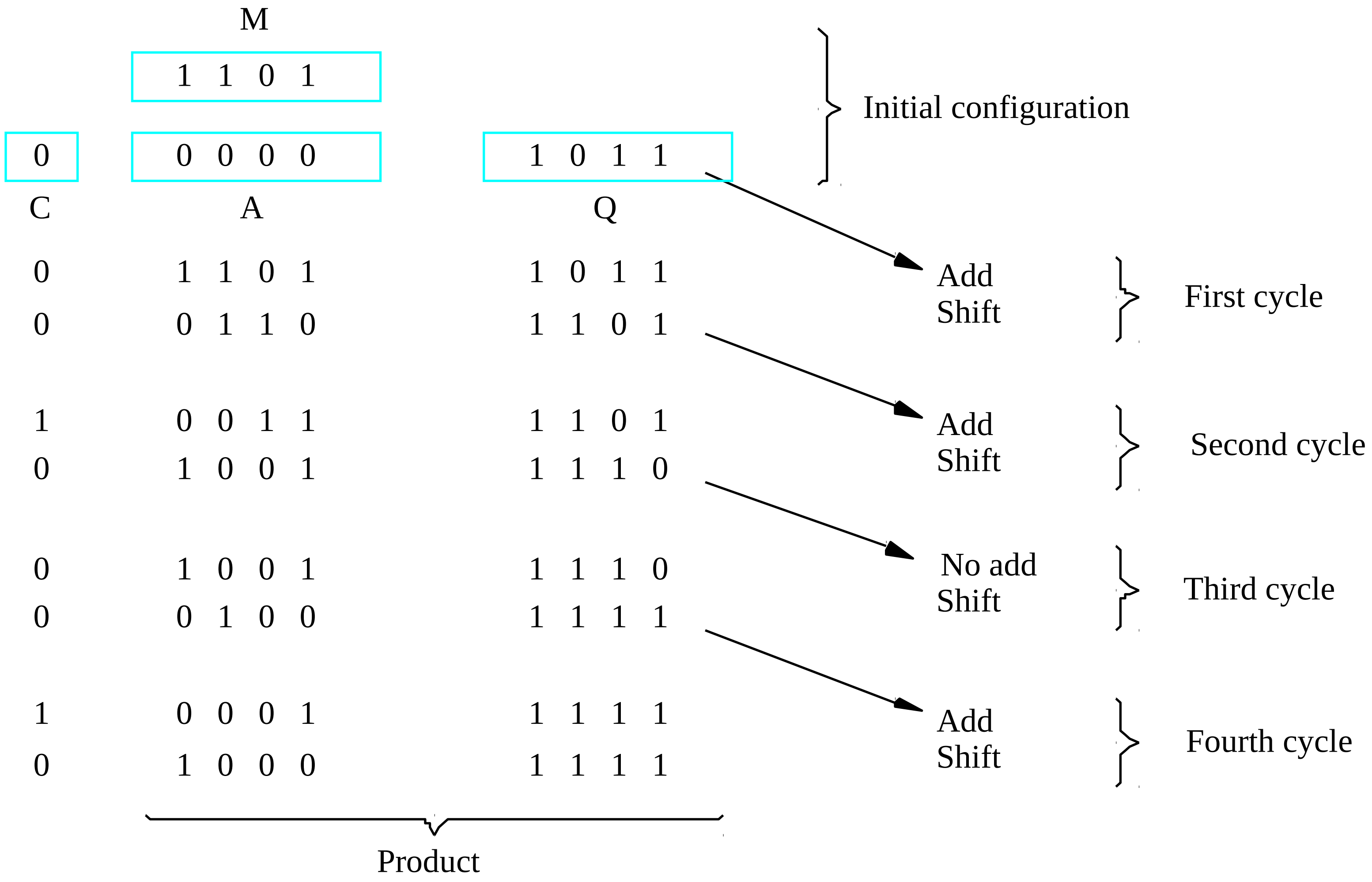
Sequential multiplication

- Recall the rule for generating partial products:
  - If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.

Sequential Circuit Multiplier



Sequential multiplication (contd..)



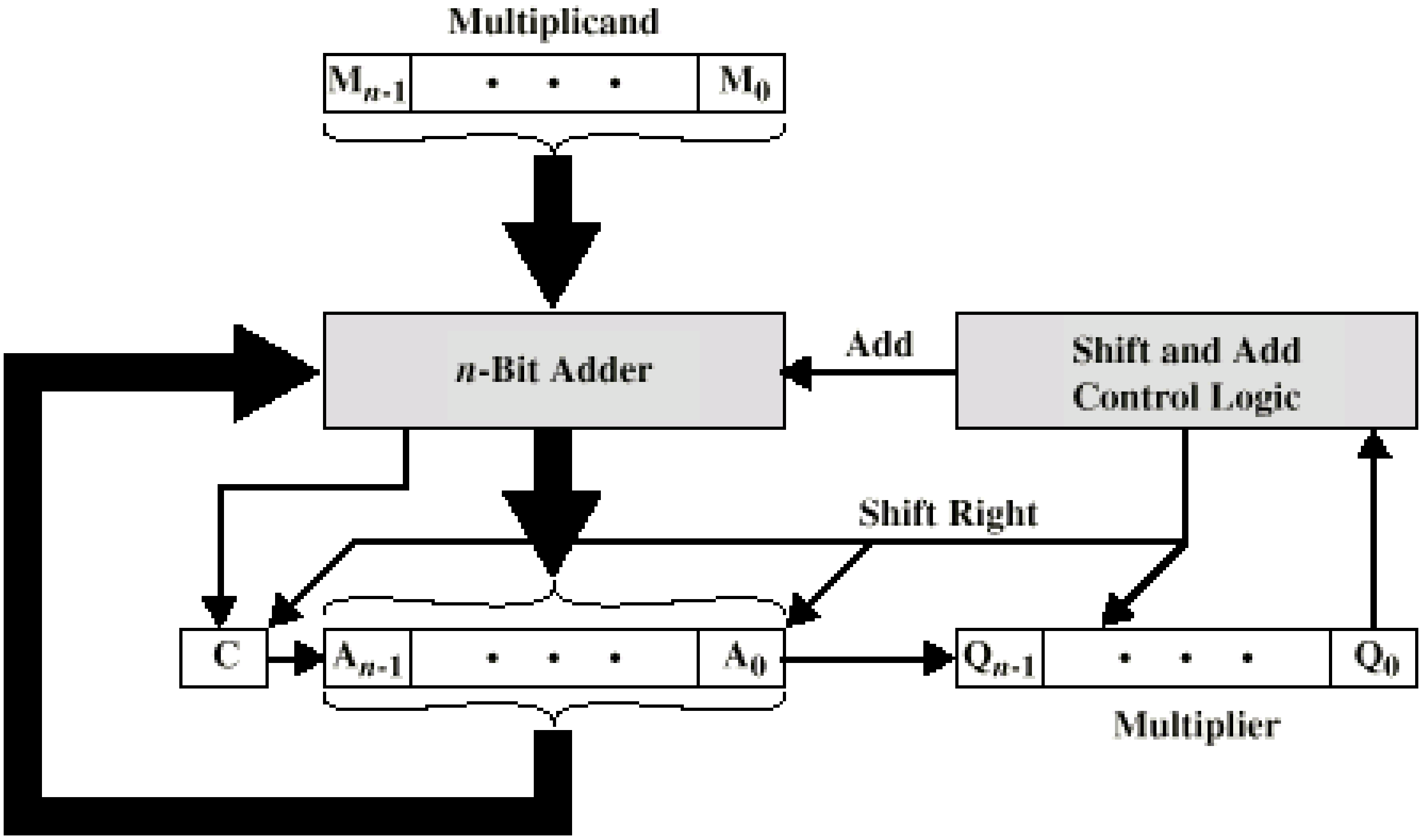
M

1 1 0 1

C	A	Q		
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

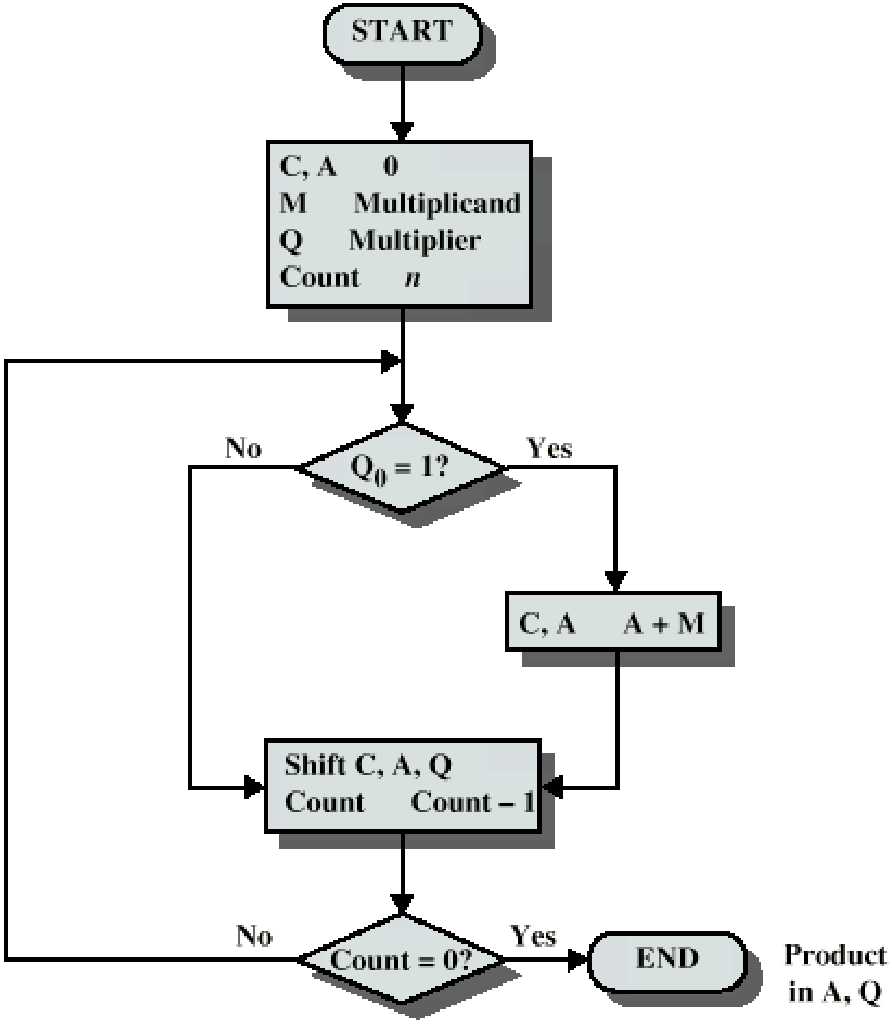
Product

Unsigned Binary Multiplication

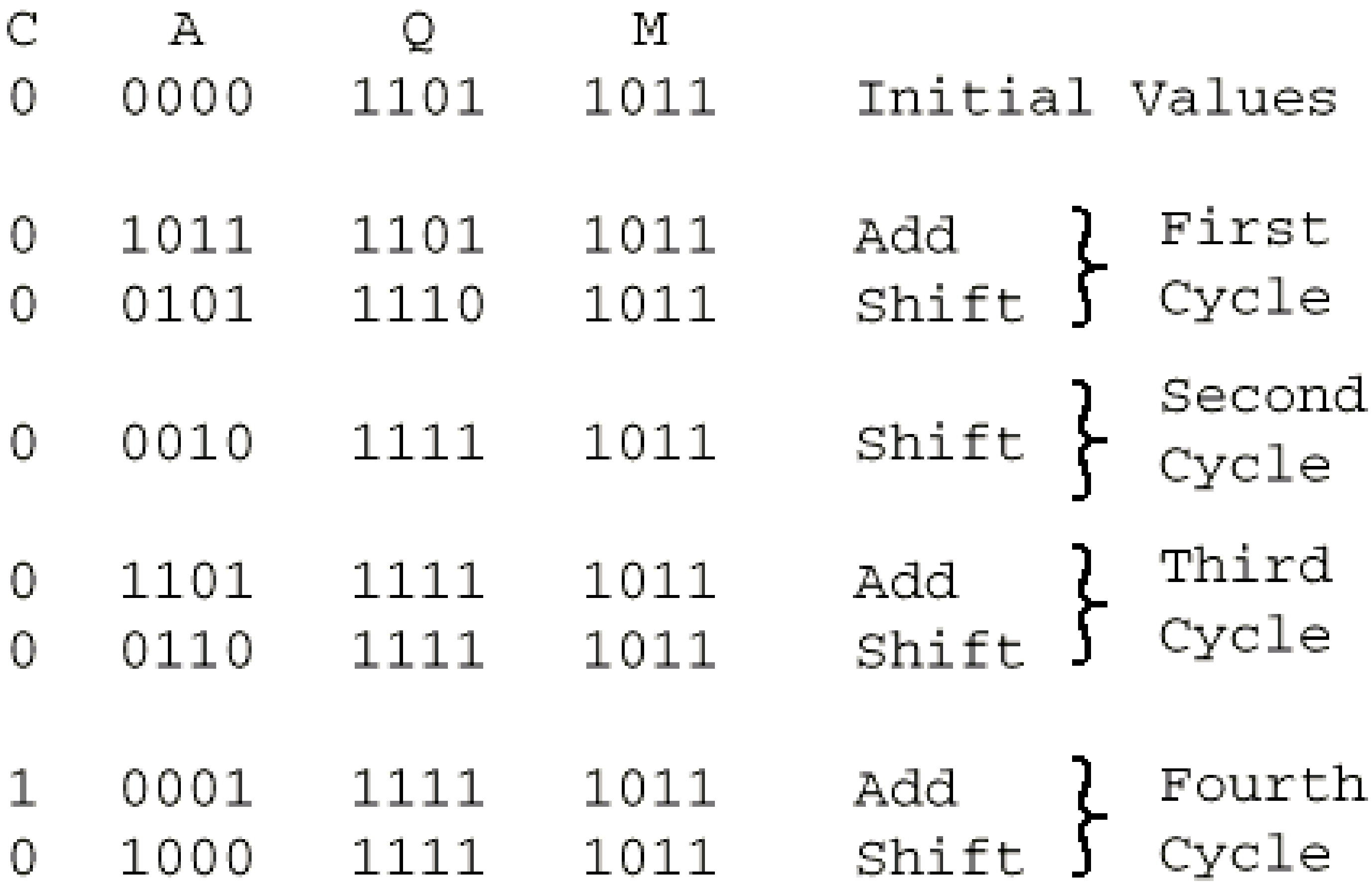


(a) Block Diagram

Flowchart for Unsigned Binary Multiplication



Execution of Example



C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Signed Multiplication

Signed Multiplication

- Considering 2’s-complement signed operands, what will happen to (-13)×(+11) if following the same method of unsigned multiplication?

					1	0	0	1	1	(- 13)
					0	1	0	1	1	(+ 11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	(- 143)

Sign extension is shown in blue

Sign extension of negative multiplicand.

Signed Multiplication

- For a negative multiplier, a straightforward solution is to form the 2’s-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.
- This is possible because complementation of both operands does not change the value or the sign of the product.
- A technique that works equally well for both negative and positive multipliers – Booth algorithm.

Booth Algorithm

- Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure?

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

Booth Algorithm

- Since 0011110 = 0100000 – 0000010, if we use the expression to the right, what will happen?

```
                        0  1  0  1  1  0  1
                        0 +1  0  0  0 -1  0
                     ----------------------
 0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  1  1  0  1  0  0  1  1     <---   2's complement of
                                                   the multiplicand
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  1  0  1  1  0  1
 0  0  0  0  0  0  0  0
-----------------------------------------
 0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Booth Algorithm

- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0

⇓

0 +1 -1 +1 0 -1 0 +1 0 0 -1 +1 -1 +1 0 -1 0 0

Booth recoding of a multiplier.

Booth Algorithm

$$\begin{array}{rrrrrrl}
 & 0 & 1 & 1 & 0 & 1 & (+13) \\
X & 1 & 1 & 0 & 1 & 0 & (-6) \\ \cline{1-6}
\end{array}$$

$$\Longrightarrow$$

$$\begin{array}{rrrrrrrrrrl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & -1 & +1 & -1 & 0 & \\ \cline{6-10}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\ \cline{1-10}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}$$

Booth multiplication with a negative multiplier.

Booth Algorithm

Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i*-1	
0	0	0 X M
0	1	+ 1 X M
1	0	- 1 X M
1	1	0 X M

Booth multiplier recoding table.

Booth Algorithm

- Best case – a long string of 1’s (skipping over 1s)
- Worst case – 0’s and 1’s are alternating

Worst-case multiplier

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

⇓

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

Ordinary multiplier

1 1 0 0 0 1 0 1 1 0 1 1 1 1 0 0

⇓

0 -1 0 0 +1 -1 +1 0 -1 +1 0 0 0 -1 0 0

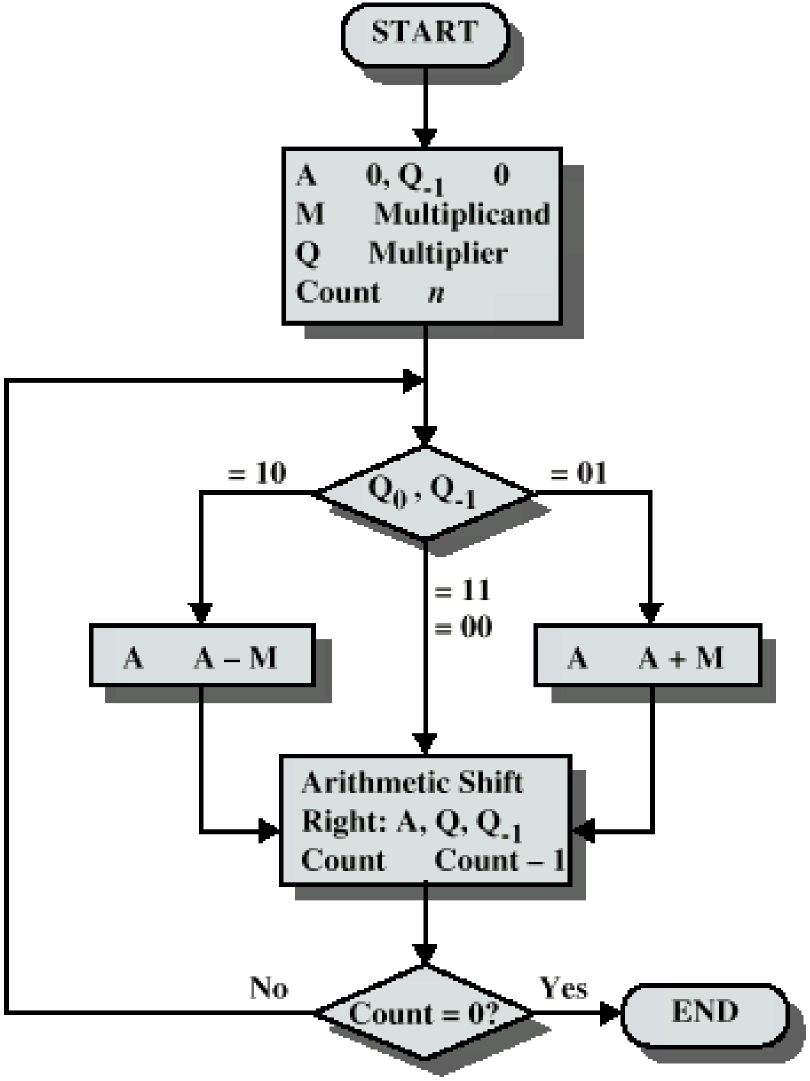
Good multiplier

0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1

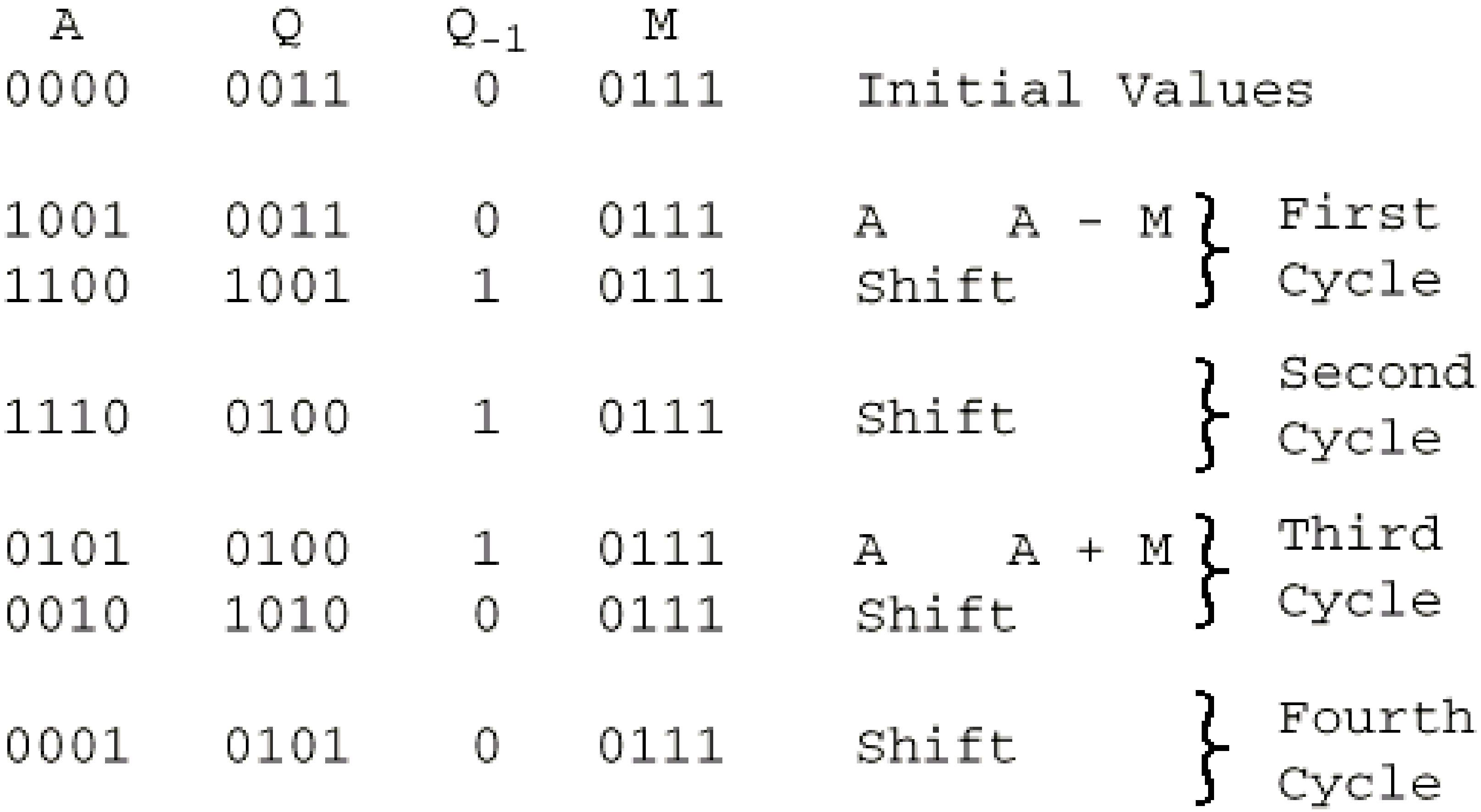
⇓

0 0 0 +1 0 0 0 0 -1 0 0 0 +1 0 0 -1

Booth’s Algorithm



Example of Booth's Algorithm

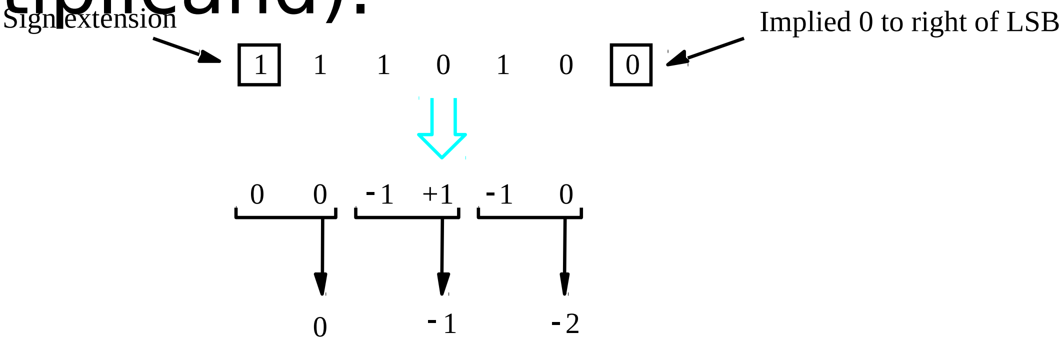


A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Fast Multiplication

Bit-Pair Recoding of Multipliers

- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



(a) Example of bit-pair recoding derived from Booth recoding

Bit-Pair Recoding of Multipliers

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position *i*
i + 1	*i*	*i* - 1	
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	- 2 X M
1	0	1	- 1 X M
1	1	0	- 1 X M
1	1	1	0 X M

(b) Table of multiplicand selection decisions

Bit-Pair Recoding of Multipliers

```
     0  1  1  0  1   (+13)
  ×  1  1  0  1  0   (-6)
  ----------------
```

⇓

```
                 0  1  1  0  1
                 0 -1 +1 -1  0
              ----------------
  0  0  0  0  0  0  0  0  0  0
  1  1  1  1  1  0  0  1  1
  0  0  0  0  1  1  0  1
  1  1  1  0  0  1  1
  0  0  0  0  0  0
  ----------------------------
  1  1  1  0  1  1  0  0  1  0   (-78)
```

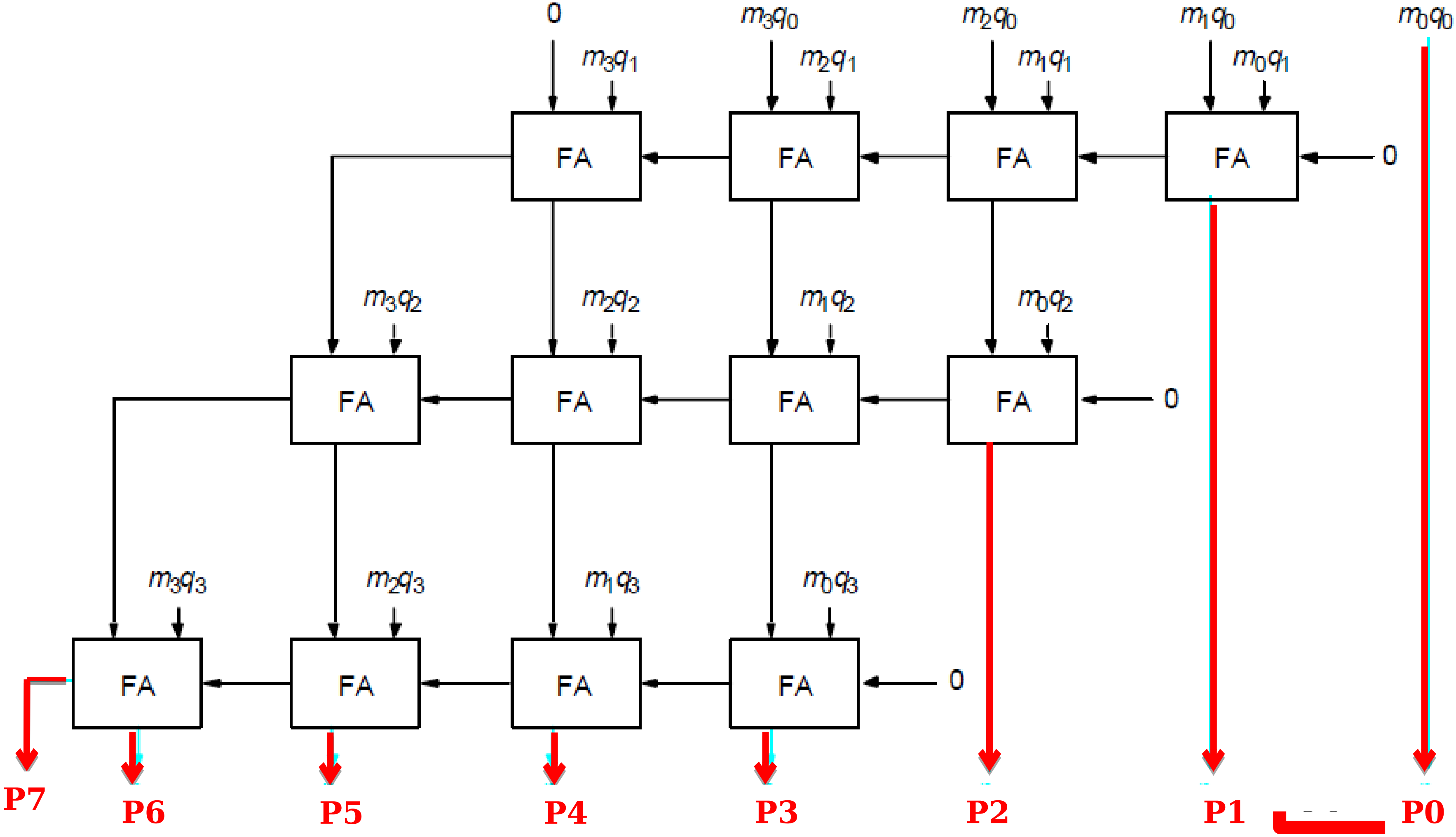
⇓

```
                 0  1  1  0  1
                 0    -1    -2
              ----------------
  1  1  1  1  1  0  0  1  1  0
  1  1  1  1  0  0  1  1
  0  0  0  0  0  0
  ----------------------------
  1  1  1  0  1  1  0  0  1  0
```

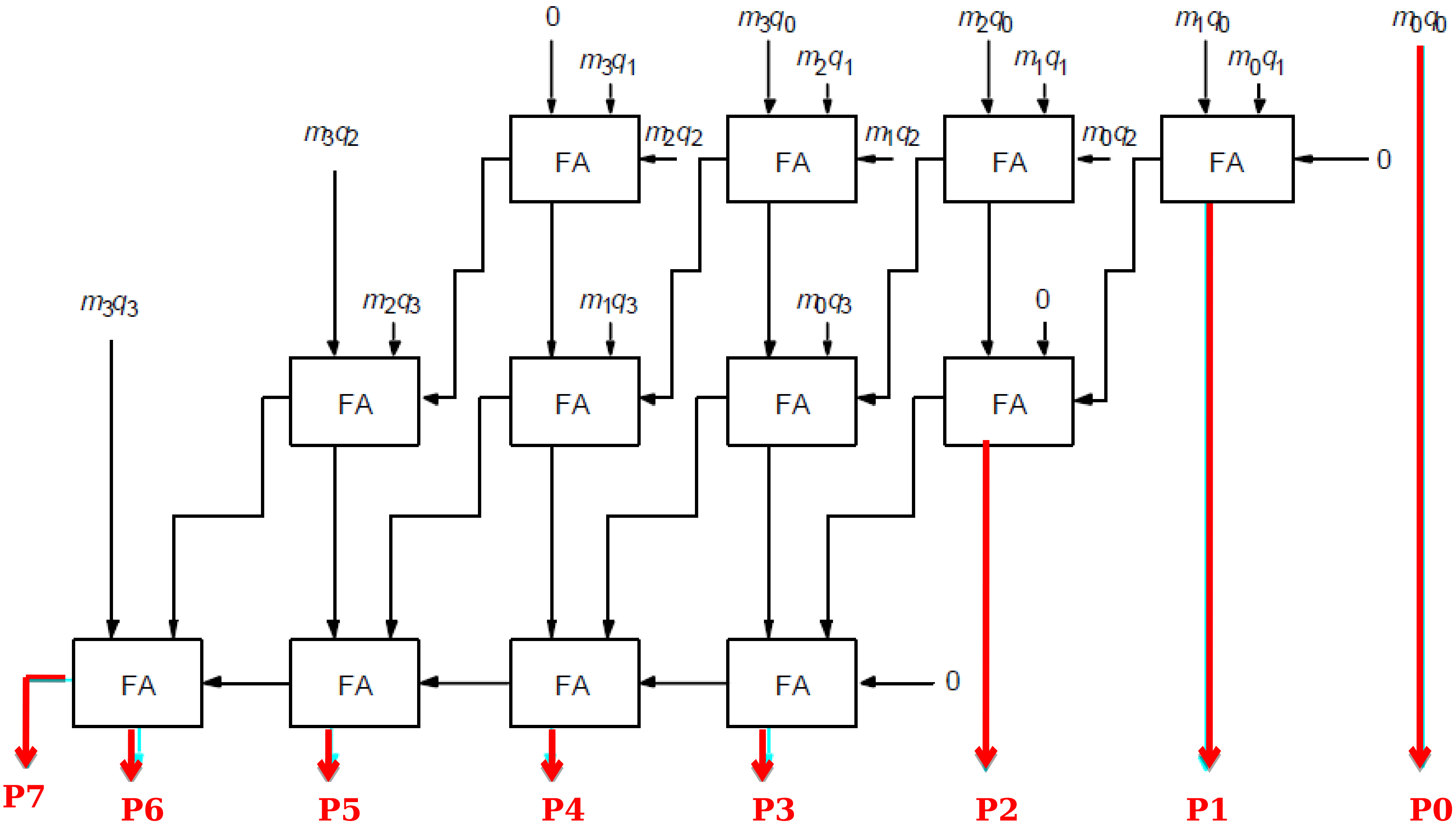
Figure 6.15. Multiplication requiring only *n*/2 summands.

Carry-Save Addition of Summands

- CSA speeds up the addition process.



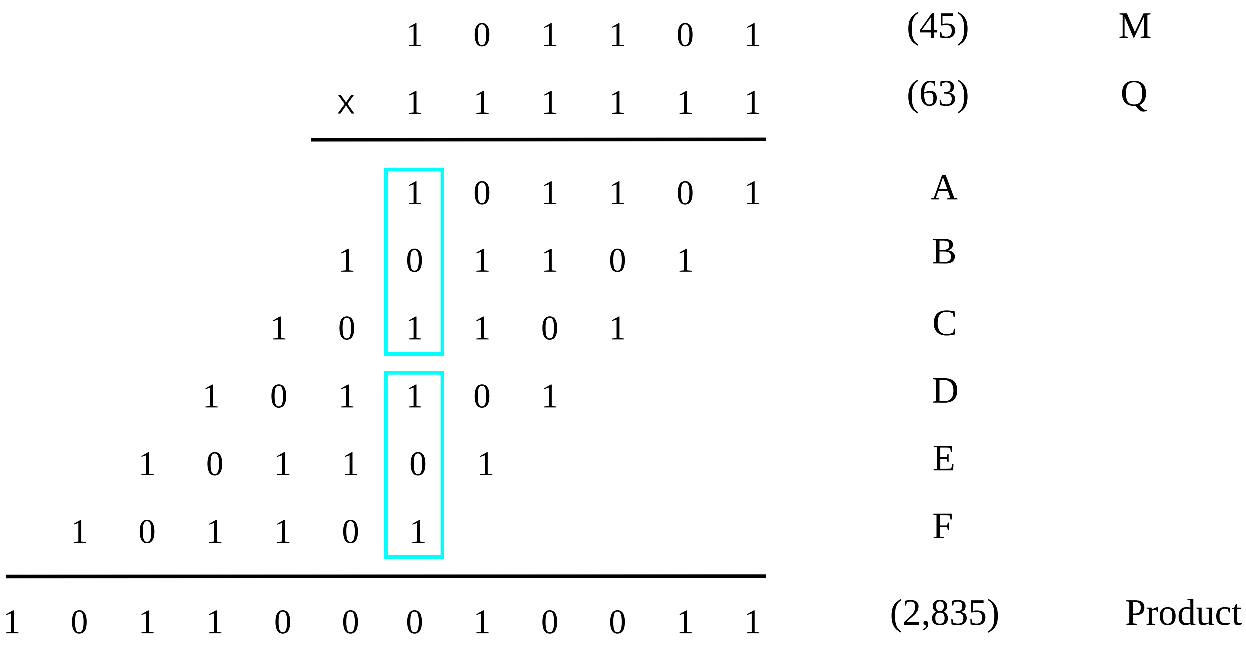
Carry-Save Addition of Summands(Cont.,)



Carry-Save Addition of Summands(Cont.,)

- Consider the addition of many summands, we can:
  - Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
  - Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
  - Continue with this process until there are only two vectors remaining
  - They can be added in a RCA or CLA to produce the desired product

Carry-Save Addition of Summands



						1	0	1	1	0	1	(45)	M
					x	1	1	1	1	1	1	(63)	Q
						1	0	1	1	0	1	A	
					1	0	1	1	0	1		B	
				1	0	1	1	0	1			C	
			1	0	1	1	0	1				D	
		1	0	1	1	0	1					E	
	1	0	1	1	0	1						F	
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 6.17. A multiplication example used to illustrate carry-save addition as shown in Figure 6.18.

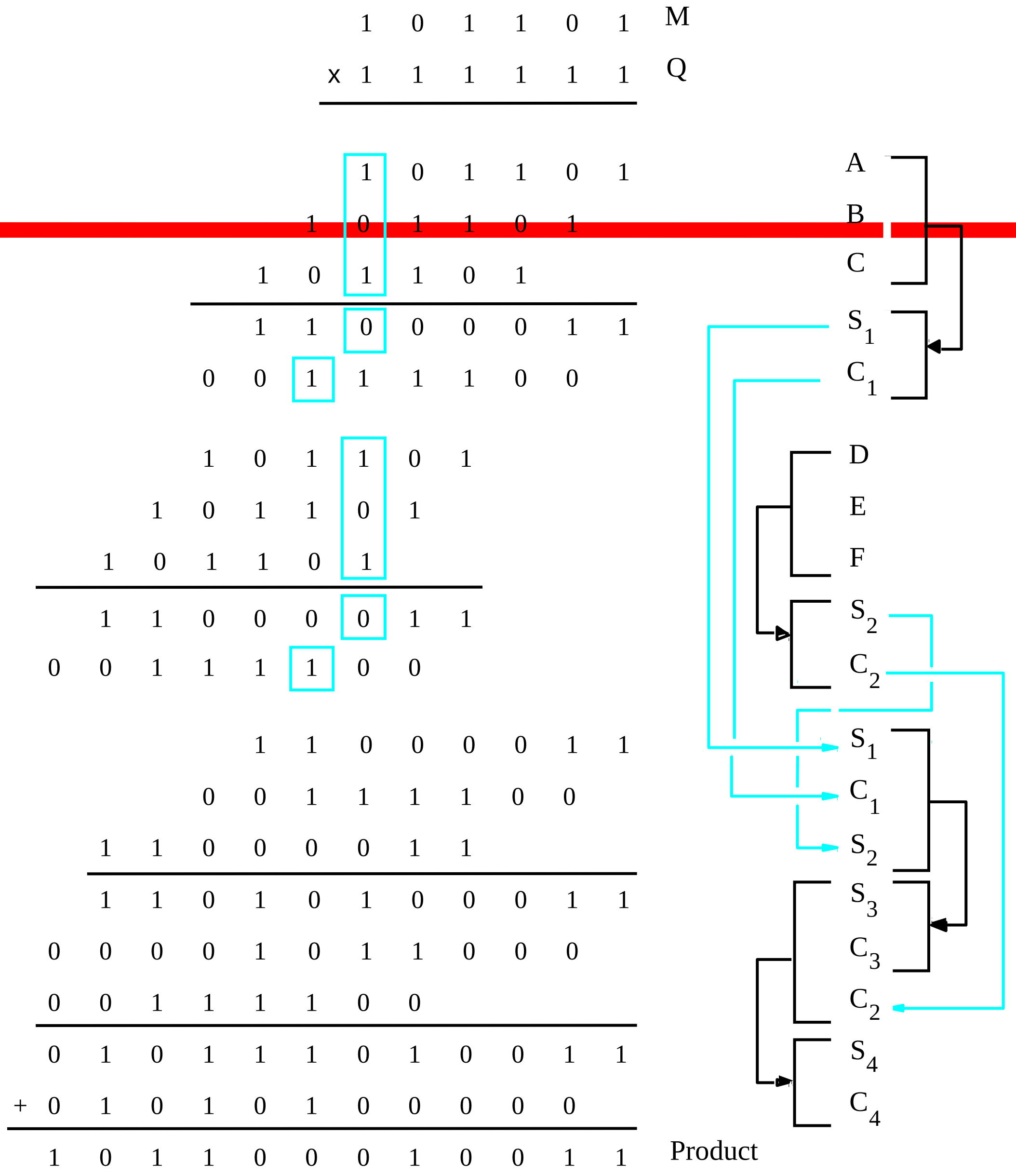


Figure 6.18. The multiplication example from Figure 6.17 performed using carry-save addition.

Integer Division

Manual Division

$$\begin{array}{r}
21\\
13\;\overline{)\;274}\\
26\\
\hline
14\\
13\\
\hline
1
\end{array}
\qquad\qquad
\begin{array}{r}
10101\\
1101\;\overline{)\;100010010}\\
1101\\
\hline
10000\\
1101\\
\hline
1110\\
1101\\
\hline
1
\end{array}$$

Longhand division examples.

Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Circuit Arrangement

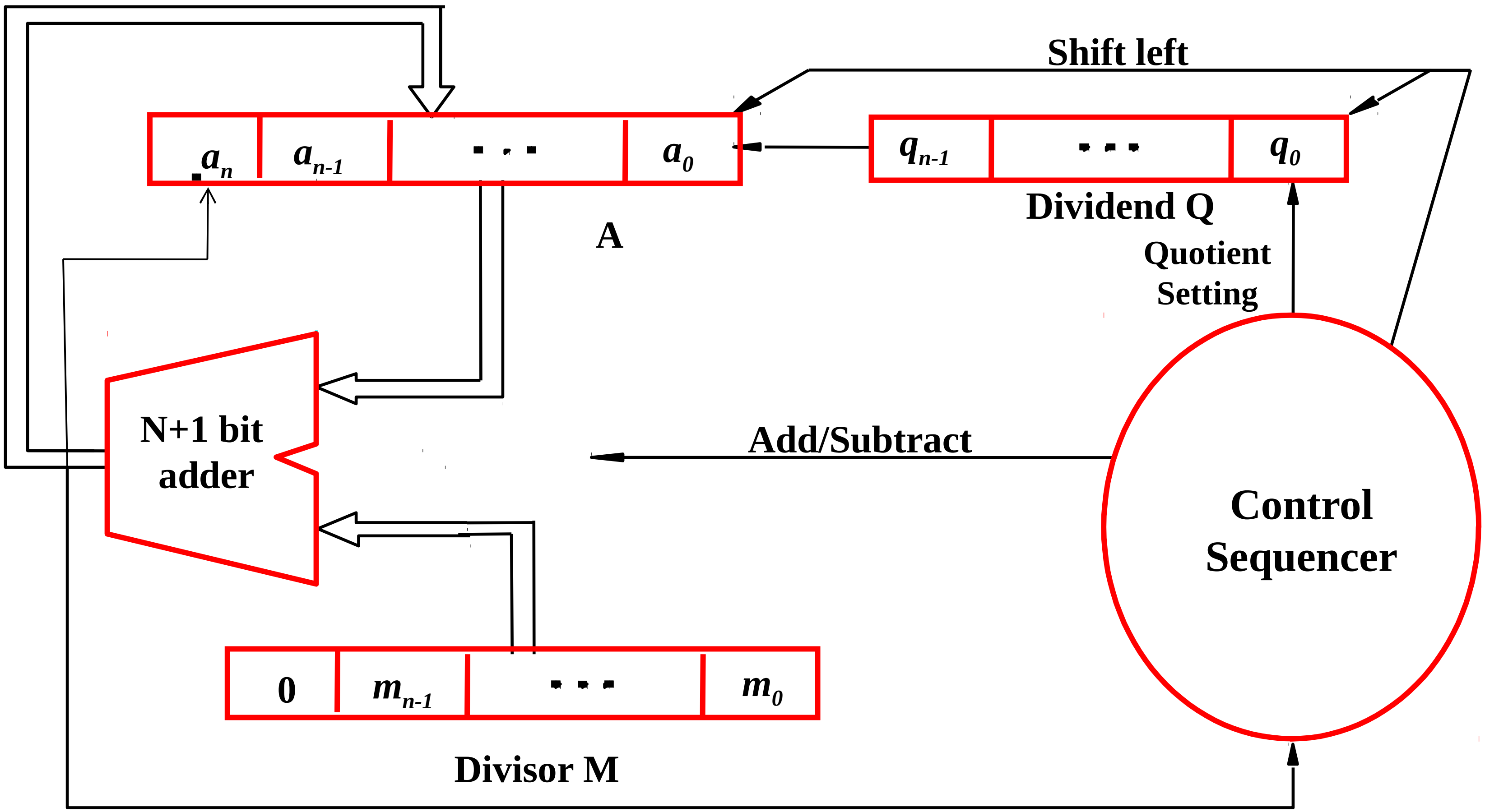
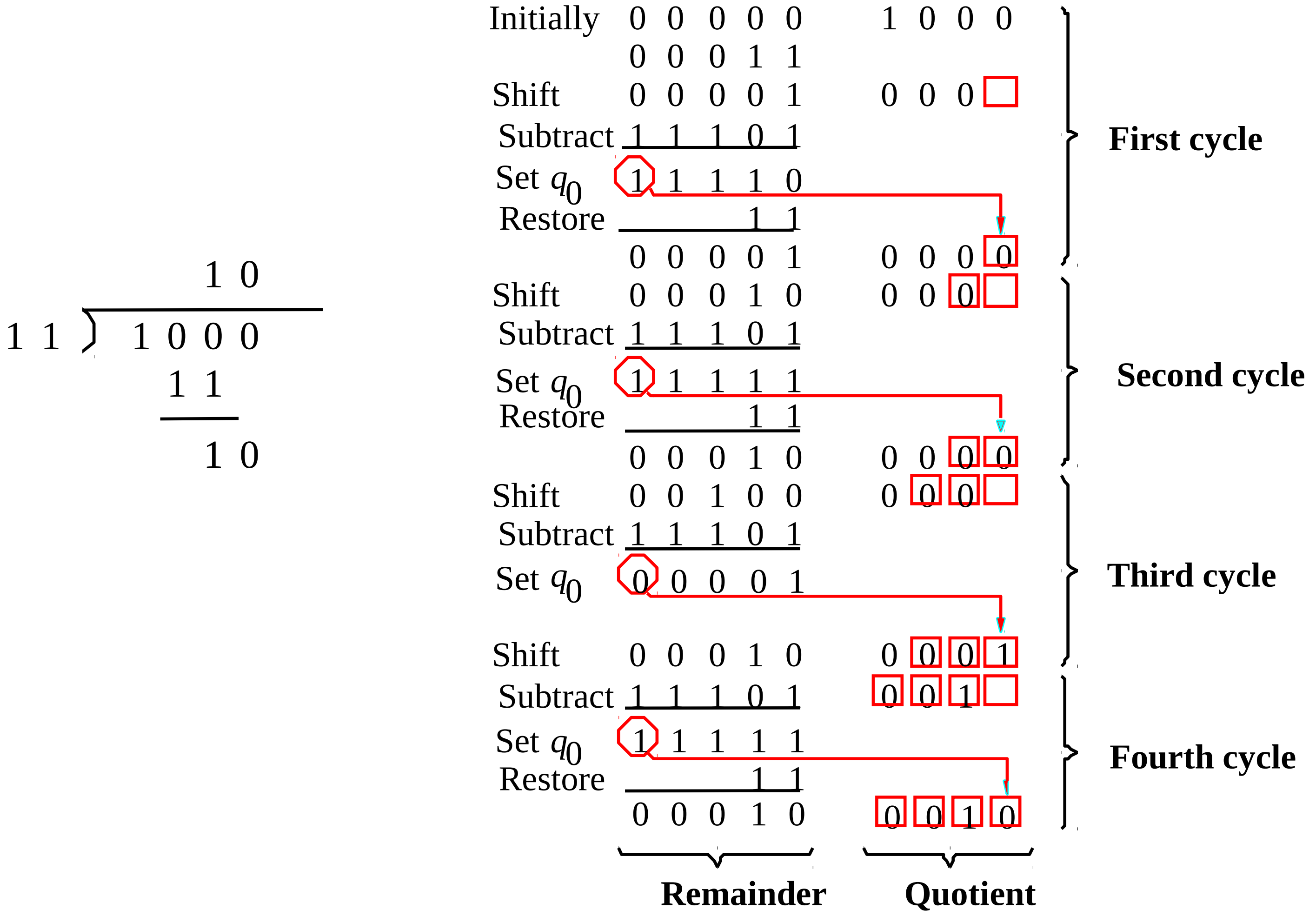


Figure 6.21. Circuit arrangement for binary division.

Restoring Division

- Shift A and Q left one binary position
- Subtract M from A, and place the answer back in A
- If the sign of A is 1, set q_0 to 0 and add M back to A (restore A); otherwise, set q_0 to 1
- Repeat these steps *n* times

Examples



```
        1 0
1 1 ) 1 0 0 0
        1 1
        1 0
```

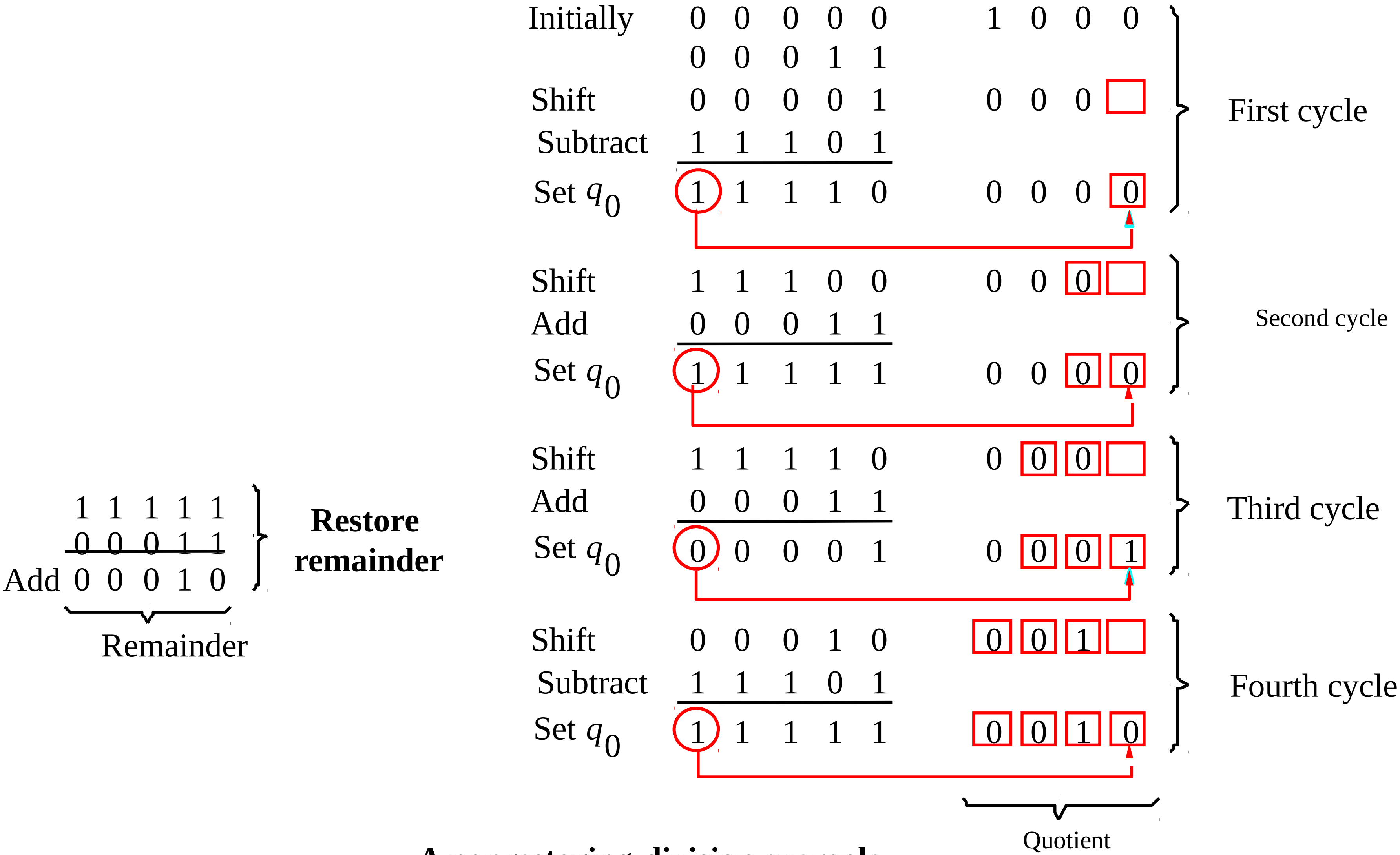
Initially	0 0 0 0 0	1 0 0 0	First cycle
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 0		
Restore	1 1		
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 □	Second cycle
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 □	Third cycle
Subtract	1 1 1 0 1		
Set q_0	0 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract	1 1 1 0 1	0 0 1 □	Fourth cycle
Set q_0	1 1 1 1 1		
Restore	1 1		
	0 0 0 1 0	0 0 1 0	
	Remainder	**Quotient**	

Figure 6.22. A restoring-division example.

Nonrestoring Division

- Avoid the need for restoring A after an unsuccessful subtraction.
- Any idea?
- Step 1: (Repeat *n* times)
  - If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
  - Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.
- Step2: If the sign of A is 1, add M to A

Examples



Initially	0	0	0	0	0	1	0	0	0	First cycle
	0	0	0	1	1					
Shift	0	0	0	0	1	0	0	0	□	
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	0	0	0	0	0	
Shift	1	1	1	0	0	0	0	0	□	Second cycle
Add	0	0	0	1	1					
Set q_0	1	1	1	1	1	0	0	0	0	
Shift	1	1	1	1	0	0	0	0	□	Third cycle
Add	0	0	0	1	1					
Set q_0	0	0	0	0	1	0	0	0	1	
Shift	0	0	0	1	0	0	0	1	□	Fourth cycle
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	1	0	0	1	0	
						Quotient				

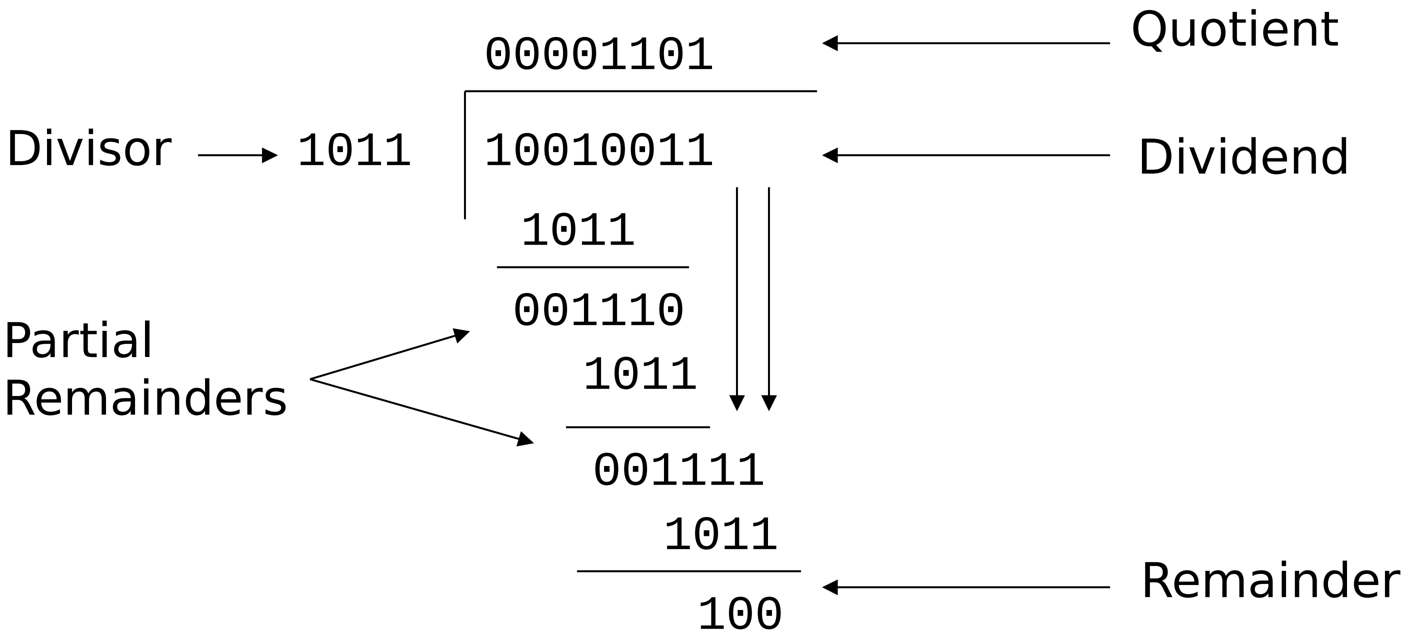
	1	1	1	1	1	Restore remainder
	0	0	0	1	1	
Add	0	0	0	1	0	
	Remainder					

A nonrestoring-division example.

Division

- More complex than multiplication
- Negative numbers are really bad!
- Based on long division

Division of Unsigned Binary Integers



Quotient

Divisor

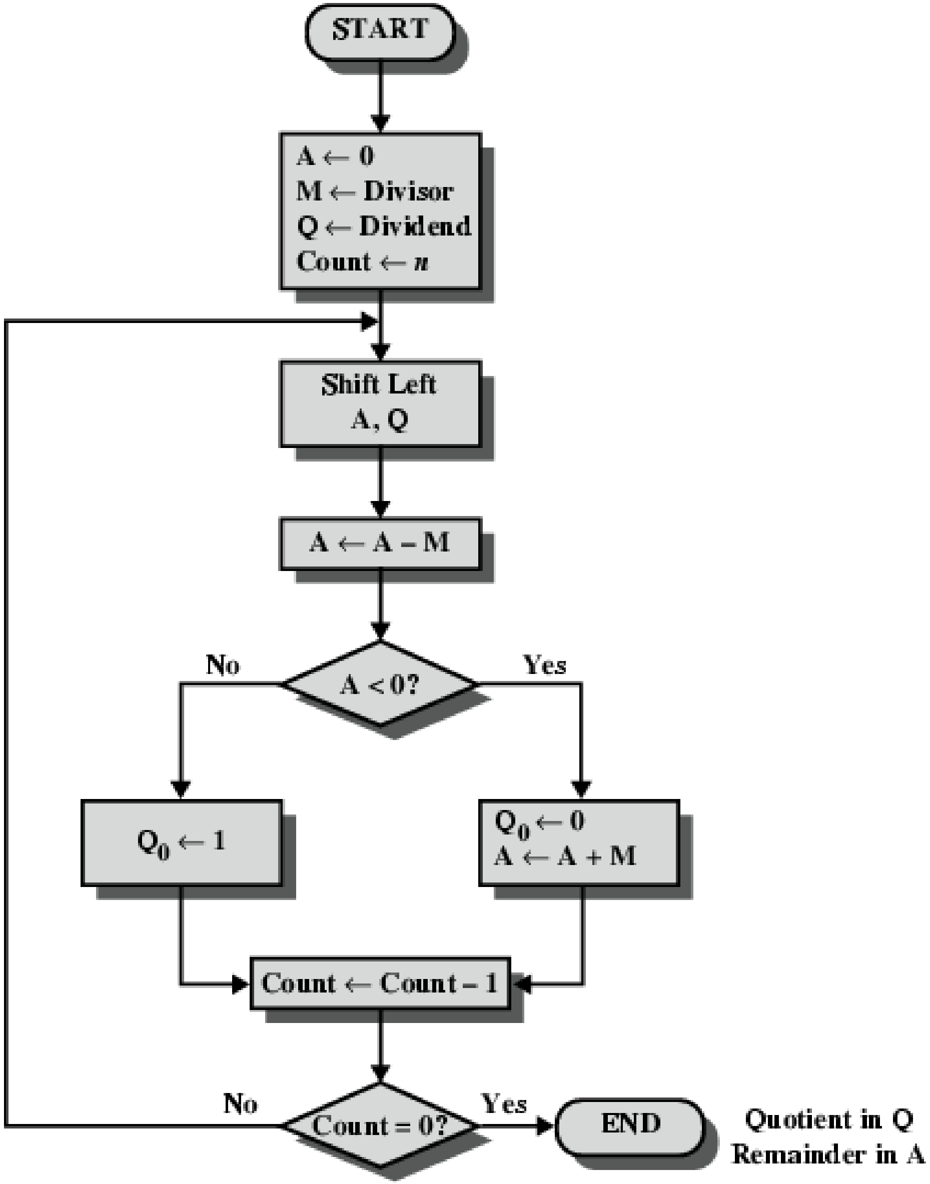
Dividend

Partial Remainders

Remainder

```
              00001101
          ------------
Divisor 1011 | 10010011
               1011
               ------
               001110
                  1011
                 ------
                 001111
                    1011
                   ------
                      100
```

Flowchart for Unsigned Binary Division



START
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left
A, Q
A ← A – M
No
A < 0?
Yes
Q_0 ← 1
Q_0 ← 0
A ← A + M
Count ← Count – 1
No
Count = 0?
Yes
END
Quotient in Q
Remainder in A