Multiplication

Multiplication of Unsigned Numbers

- Multiplication requires substantially more hardware than addition.
- Multiplication of two *n*-bit number generates a 2*n*-bit product.
- We can use shift-and-add method.
  - Repeated additions of shifted versions of the multiplicand.

```
          1 0 1 0      Multiplicand M  (10)
          1 1 0 1      Multiplier Q    (13)
         --------
          1 0 1 0
        0 0 0 0
      1 0 1 0
    1 0 1 0
 ---------------
1 0 0 0 0 0 1 0        Product P       (130)
```

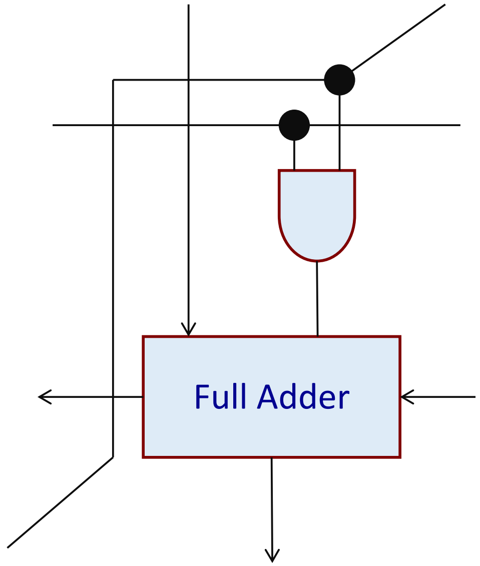
A General Case

				A_3	A_2	A_1	A_0
				B_3	B_2	B_1	B_0
				A_3B_0	A_2B_0	A_1B_0	A_0B_0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1	
		A_3B_2	A_2B_2	A_1B_2	A_0B_2		
	A_3B_3	A_2B_3	A_1B_3	A_0B_3			

- Each $A_i.B_j$ is called a *partial product*.
- Generating the partial products is easy.
  - Requires just an AND gate for each partial product.
- Adding all the *n*-bit partial products in hardware is more difficult.

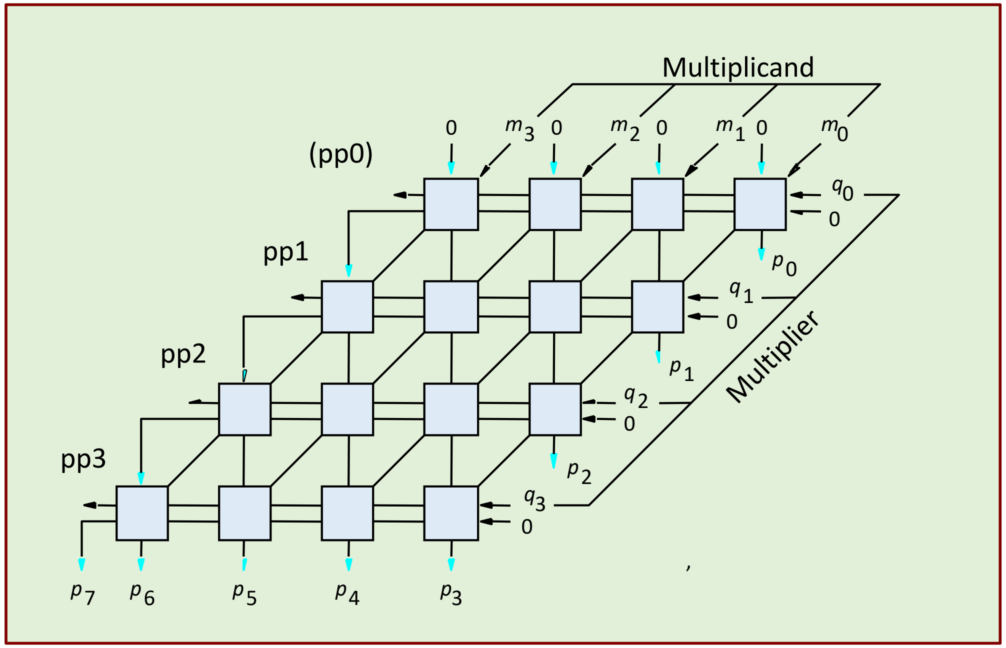
Design of a Combinational Array Multiqlier

- We can directly map the multiplication process as discussed to hardware.
  - We use an array of cells to generate the partial products.
  - Instead of adding the partial products at the end, we add the partial products at every stage of the multiplication.
- The required multiplication cell is as shown.
  - Combines capabilities of partial product generation and also addition of partial products.



- Extremely inefficient, and requires very large amount of hardware.
- Requires n^2 multiplication cells for an $n \times n$ multiplier.
- Advantage is that it is very fast.

			m_3	m_2	m_1	m_0
			q_3	q_2	q_1	q_0
			m_3q_0	m_2q_0	m_1q_0	m_0q_0
		m_3q_1	m_2q_1	m_1q_1	m_0q_1	
	m_3q_2	m_2q_2	m_1q_2	m_0q_2		
m_3q_3	m_2q_3	m_1q_3	m_0q_3			

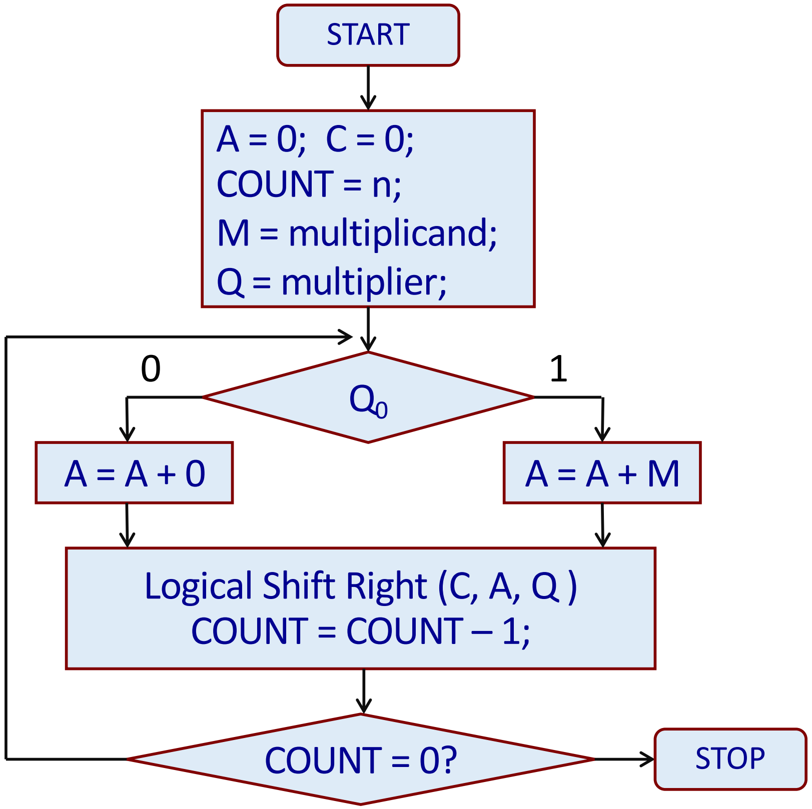


product: $p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

Unsigned Sequential Multiqlication

- Requires much less hardware, but requires several clock cycles to qerform multiplication of two *n*-bit numbers.
  - Typical hardware complexity: O(n)
  - Typical time complexity: O(n)
- In the "*hand multiplication*" that we have seen:
  - If the i-th bit of the multiplier is 1, the multiplicand is shifted left by i bit positions, and added to the partial product.
  - The relative position of the partial products do not change; it is the multiplicand that gets shifted left.

- In the "*shift-and-add*" multiplication that we discuss now, we make the following modifications.
  - We do not shift the multiplicand (i.e., keep its position fixed).
  - We right shift an $2n$-bit partial product at every step.



M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

C: 1-bit carry out from adder

Example 1: (10) x (13)

Assume 5-bit numbers.

M: $(0\ 1\ 0\ 1\ 0)_2$
Q: $(0\ 1\ 1\ 0\ 1)_2$

product = 130
= $(0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0)_2$

C	A	Q		
0	0 0 0 0 0	0 1 1 0 1	Initialization	
0	0 1 0 1 0	0 1 1 0 1	A = A + M	Step 1
0	0 0 1 0 1	0 0 1 1 0	Shift	
0	0 0 1 0 1	0 0 1 1 0	A = A + 0	Step 2
0	0 0 0 1 0	1 0 0 1 1	Shift	
0	0 1 1 0 0	1 0 0 1 1	A = A + M	Step 3
0	0 0 1 1 0	0 1 0 0 1	Shift	
0	1 0 0 0 0	0 1 0 0 1	A = A + M	Step 4
0	0 1 0 0 0	0 0 1 0 0	Shift	
0	0 1 0 0 0	0 0 1 0 0	A = A + 0	Step 5
0	0 0 1 0 0	0 0 0 1 0	Shift	

Example 2: (29) x (21)

Assume 5-bit numbers.

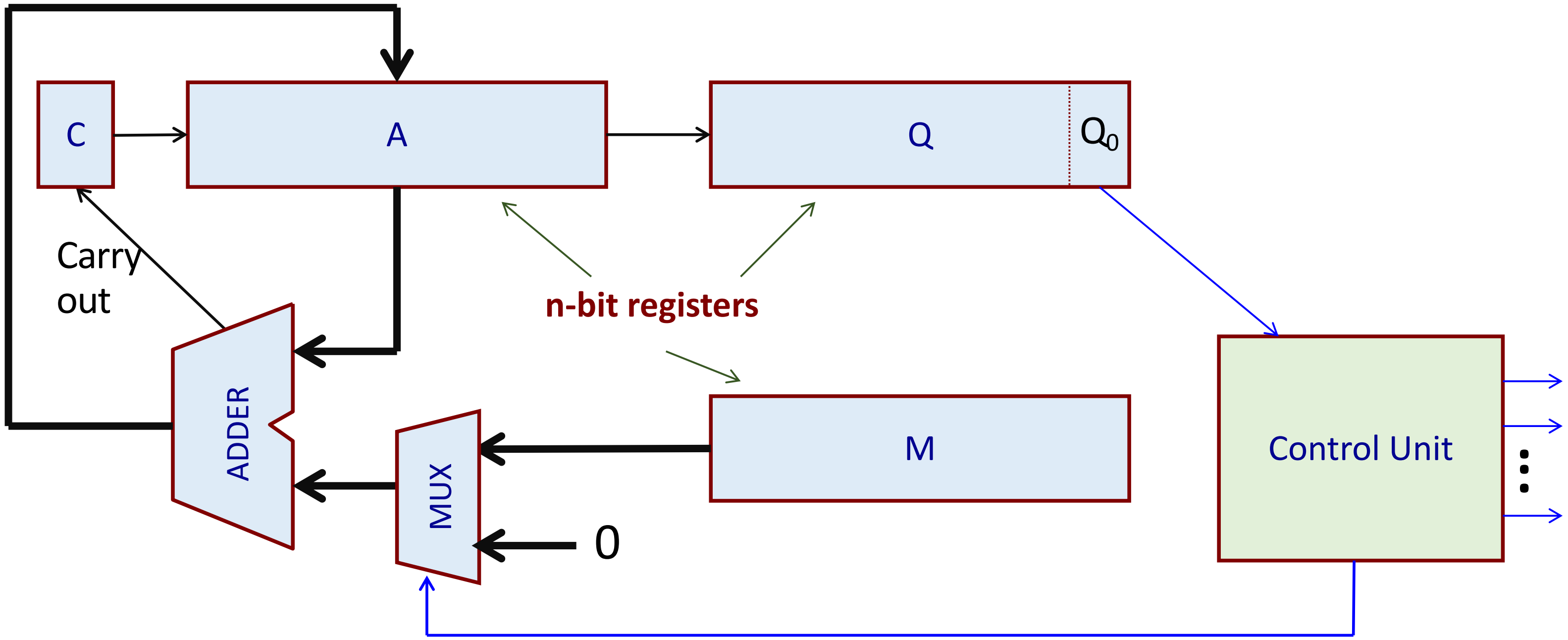
M: $(1\,1\,1\,0\,1)_2$
Q: $(1\,0\,1\,0\,1)_2$

product = 609
= $(1\,0\,0\,1\,1\,0\,0\,0\,0\,1)_2$

C	A	Q		
0	0 0 0 0 0	1 0 1 0 1	Initialization	
0	1 1 1 0 1	1 0 1 0 1	A = A + M	Step 1
0	0 1 1 1 0	1 1 0 1 0	Shift	
0	0 1 1 1 0	1 1 0 1 0	A = A + 0	Step 2
0	0 0 1 1 1	0 1 1 0 1	Shift	
1	0 0 1 0 0	0 1 1 0 1	A = A + M	Step 3
0	1 0 0 1 0	0 0 1 1 0	Shift	
0	1 0 0 1 0	0 0 1 1 0	A = A + 0	Step 4
0	0 1 0 0 1	0 0 0 1 1	Shift	
1	0 0 1 1 0	0 0 0 1 1	A = A + M	Step 5
0	1 0 0 1 1	0 0 0 0 1	Shift	

46

Data path for Shift-and-Add Multiplier



47

Signed Multiplication

- We can extend the basic shift-and-add multiplication method to handle signed numbers.
- One important difference:
  - Require to sign-extend all the partial products before they are added.
  - Recall that for 2's complement representation, sign extension can be done by replicating the sign bit any number of times.

0101 = 0000 0101 = 0000 0000 0000 0101 = 0000 0000 0000 0000 0000 0000 0000 0101

1011 = 1111 1011 = 1111 1111 1111 1011 = 1111 1111 1111 1111 1111 1111 1111 1011

An Example: 6-bit 2's complement multiplication

Note: For *n*-bit multiplication, since we are generating a 2*n*-bit product, overflow can never occur.

```
              1 1 0 1 0 1     (-11)
            x 0 1 1 0 1 0     (+26)
-------------------------
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0 1 0 1
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 0 1 0 1
1 1 1 1 0 1 0 1
0 0 0 0 0 0 0
-------------------------
1 1 1 0 1 1 1 0 0 0 1 0   (-286)
```


49

Booth's Algorithm for Signed Multiplication

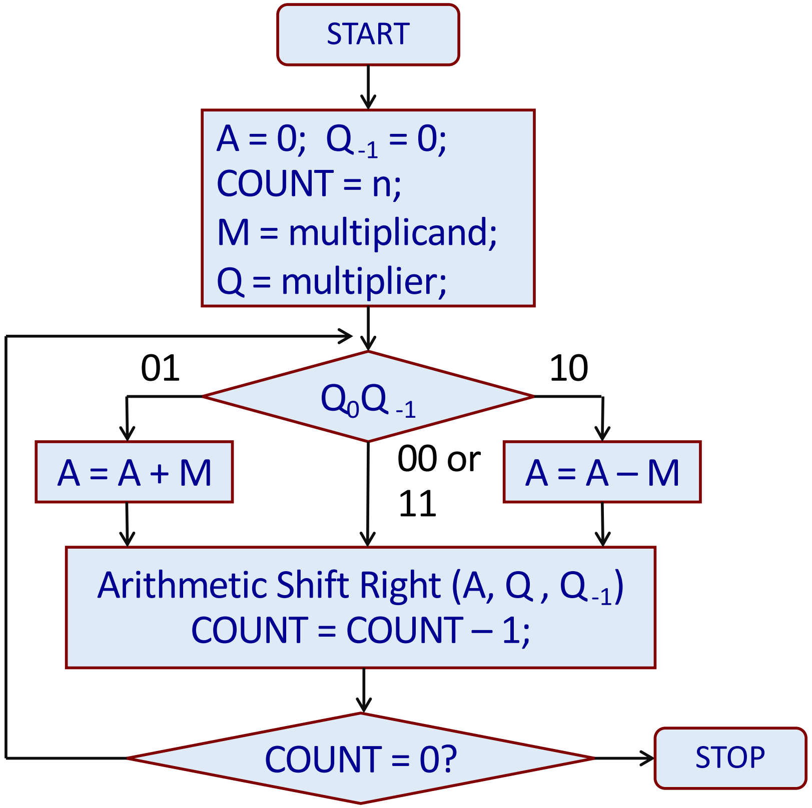
- In the conventional shift-and-add multiplication as discussed, for *n*-bit multiplication, we iterate *n* times.
  - Add either 0 or the multiplicand to the 2*n*-bit partial product (defending on the next bit of the multiplier).
  - Shift the 2*n*-bit partial product to the right.
- Essentially we need *n additions* and *n shift operations*.
- Booth's algorithm is an improvement whereby we can avoid the additions whenever consecutive 0's or 1's are detected in the multiplier.
  - Makes the process faster.

Basic Idea Behind Booth's Algorithm

- We inspect two bits of the multiplier (Q_i, Q_{i-1}) at a time.
  - If the bits are same (00 or 11), we only shift the partial product.
  - If the bits are 01, we do an addition and then shift.
  - If the bits are 10, we do a subtraction and then shift.
- Significantly reduces the number of additions / subtractions.
- Inspecting bit qairs as mentioned can also be expressed in terms of *Booth's Encoding*.
  - Use the symbols +1, -1 and 0 to indicate changes w.r.t. Q_i and Q_{i-1}.
  - 01 → +1, 10 → -1, 00 or 11 → 0.
  - For encoding the least significant bit Q_0, we assume $Q_{-1} = 0$.

- Examples of Booth encoding:
    a) 0 1 1 1 0 0 0 0 :: +1 0 0 -1 0 0 0 0
    b) 0 1 1 1 0 1 1 0 :: +1 0 0 -1 +1 0 -1 0
    c) 0 0 0 0 0 1 1 1 :: 0 0 0 0 +1 0 0 -1
    d) 0 1 0 1 0 1 0 1 :: +1 -1 +1 -1 +1 -1 +1 -1

- The last example illustrates the *worst case* for Booth's multiplication (alternating 0's and 1's in multiplier).
    - In the illustrations, we shall show the two multiplier bits explicitly instead of showing the encoded digits.



M: n-bit multiplicand

Q: n-bit multiplier

A: n-bit temporary register

Q_{-1}: 1-bit flip-flop

Skips over consecutive 0's and 1's of the multiplier Q

Example 1: (-10) x (13)

Assume 5-bit numbers.

M: $(1\,0\,1\,1\,0)_2$
-M: $(0\,1\,0\,1\,0)_2$
Q: $(0\,1\,1\,0\,1)_2$

product = -130
= $(1\,1\,0\,1\,1\,1\,1\,1\,1\,0)_2$

A	Q	Q_{-1}		
0 0 0 0 0	0 1 1 0 1	0	Initialization	
0 1 0 1 0	0 1 1 0 1	0	A = A - M	Step 1
0 0 1 0 1	0 0 1 1 0	1	Shift	
1 1 0 1 1	0 0 1 1 0	1	A = A + M	Step 2
1 1 1 0 1	1 0 0 1 1	0	Shift	
0 0 1 1 1	1 0 0 1 1	0	A = A - M	Step 3
0 0 0 1 1	1 1 0 0 1	1	Shift	
0 0 0 0 1	1 1 1 1 0	1	Shift	Step 4
1 0 1 1 1	1 1 1 0 0	1	A = A + M	Step 5
1 1 0 1 1	1 1 1 1 0	0	Shift	

54

Example 2:

(-31) x (28)

Assume 6-bit numbers.

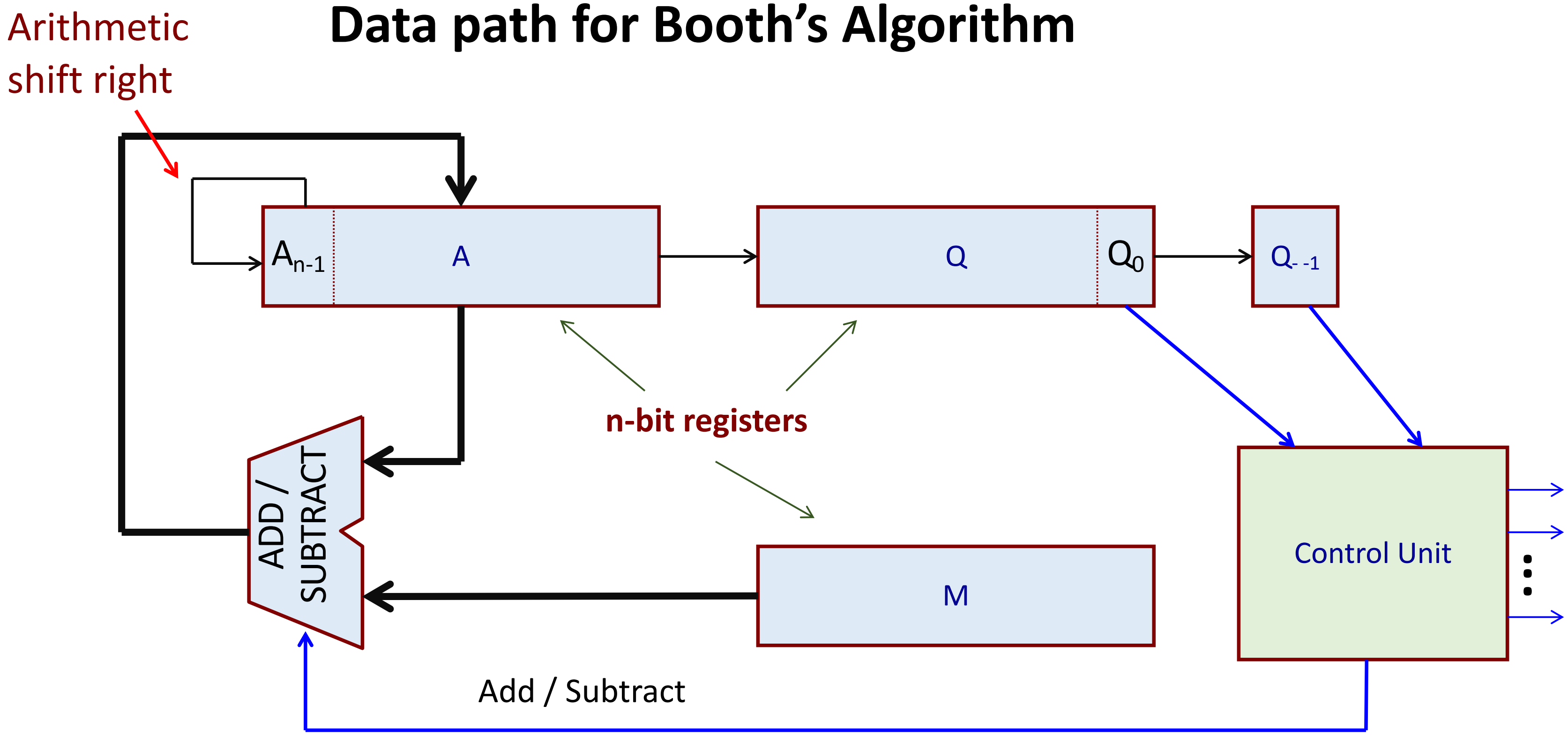
M: $(1\,0\,0\,0\,0\,1)_2$
-M: $(0\,1\,1\,1\,1\,1)_2$
Q: $(0\,1\,1\,1\,0\,0)_2$

product = -868
= $(1\,1\,0\,0\,1\,0\;0\,1\,1\,1\,0\,0)_2$

A	Q	Q_{-1}		
0 0 0 0 0 0	0 1 1 1 0 0	0	Initialization	
0 0 0 0 0 0	0 0 1 1 1 0	0	Shift	Step 1
0 0 0 0 0 0	0 0 0 1 1 1	0	Shift	Step 2
0 1 1 1 1 1	0 0 0 1 1 1	0	A = A - M	Step 3
0 0 1 1 1 1	1 0 0 0 1 1	1	Shift	
0 0 0 1 1 1	1 1 0 0 0 1	1	Shift	Step 4
0 0 0 0 1 1	1 1 1 0 0 0	1	Shift	Step 5
1 0 0 1 0 0	1 1 1 0 0 0	1	A = A + M	Step 6
$\boxed{1\;1\;0\;0\;1\;0}$	$\boxed{0\;1\;1\;1\;0\;0}$	0	Shift	

55

Data path for Booth's Algorithm



56

Design of Fast Multiplier

a) Bit-pair Recoding of Booth's Multiplication

- A technique that halves the maximum number of summands; derived directly from the Booth's algorithm.
- If we group the Booth-coded multiplier digits in pairs, we observe:
  - (+1, -1): (+1, -1) * M = 2 * M – M = M
  - (0, +1): (0, +1) * M = M
- We need a single addition instead of a pair of addition & subtraction.
  - Other similar rules can be framed.
  - Shown on next slide.

Original Booth-coded pair	Equivalent Recoded pair
(+1, 0)	(0, +2)
(-1, +1)	(0, -1)
(0, 0)	(0, 0)
(0, 1)	(0, 1)
(+1, 1)	--
(+1, -1)	(0, +1)
(-1, 0)	(0, -2)

- Every equivalent recoded pair has at least one 0.
- Worst-case number of additions or subtractions is 50% of the number of multiplier bits.
- Reduces the worst-case time required for multiplication.

Example: (+13) X (-22) in 6-bits.

```
Original:  Multiplier --   1   0   1   0   1   0
Booth:     Multiplier --  -1  +1  -1  +1  -1   0
Recoded:   Multiplier --   0  -1   0  -1   0  -2
```

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

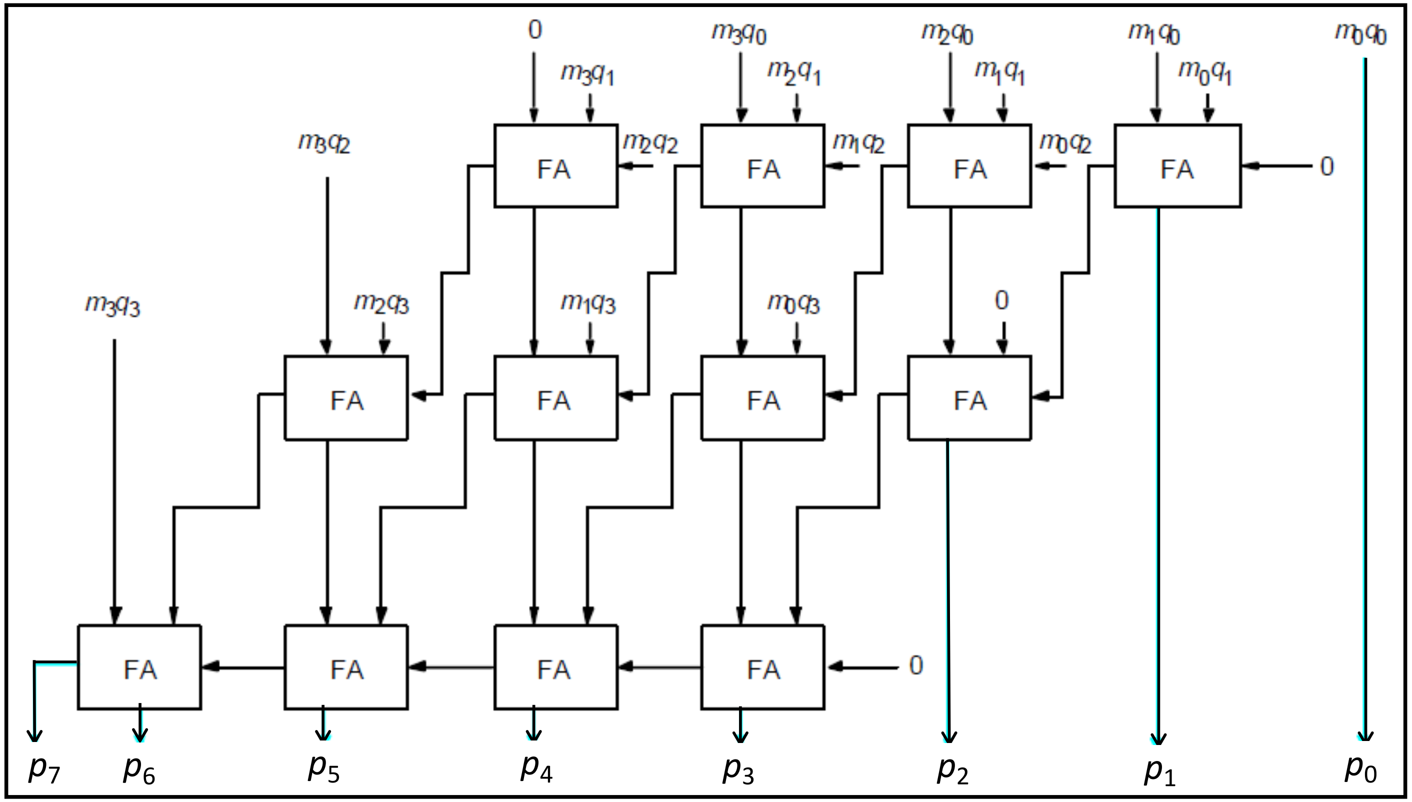
- M = 0 0 1 1 0 1 (+13)
- -1 * M = 1 1 0 0 1 1
- -2 * M = 1 0 0 1 1 0

b) Carry Save Multiplier

- We have seen earlier how carry save adders (CSA) can be used to add several numbers with carry propagation only in the last stage.
- The partial products can be generated in parallel using n^2 AND gates.
- The n partial products can then be added using a CSA tree.
- Instead of letting the carries ripple through during addition, we *save* them and feed it to the next row, at the correct weight positions.

			m_3	m_2	m_1	m_0
			q_3	q_2	q_1	q_0
			m_3q_0	m_2q_0	m_1q_0	m_0q_0
		m_3q_1	m_2q_1	m_1q_1	m_0q_1	
	m_3q_2	m_2q_2	m_1q_2	m_0q_2		
m_3q_3	m_2q_3	m_1q_3	m_0q_3			

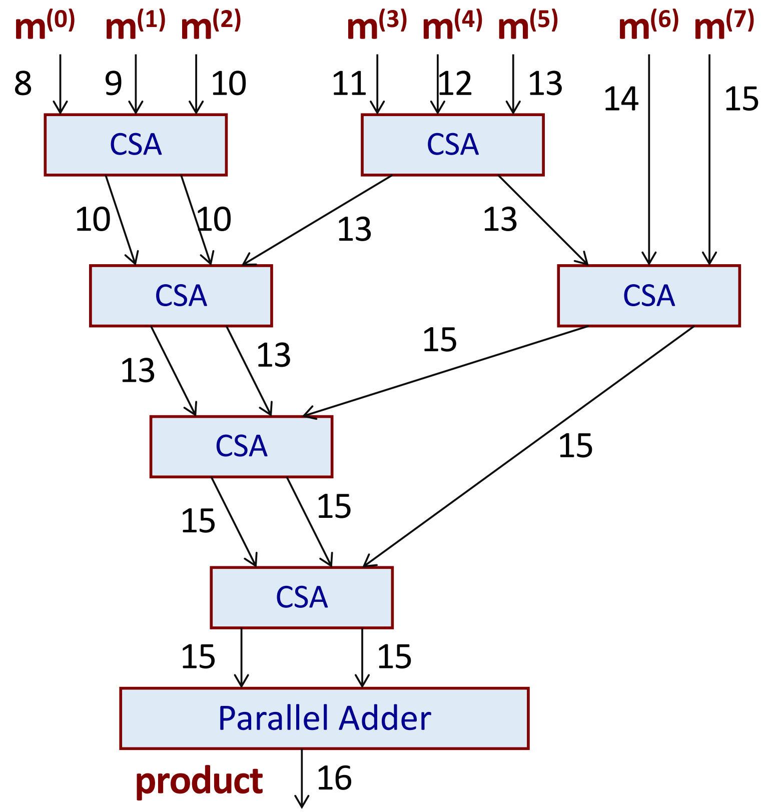
4 x 4 Carry Save Multiplier



- **Wallace Tree Multiplier**
  - A Wallace tree is a circuit that reduces the problem of summing n n-bit numbers to the problem of summing two $\Theta(n)$-bit numbers.
  - It uses $n/3$ (floor of) carry-save adders in parallel to convert the sum of n numbers to the sum of $2n/3$ (ceiling of) numbers.
  - It then recursively constructs a Wallace tree on the $2n/3$ (ceiling of) resulting numbers.
  - The set of numbers is progressively reduced until there are only two numbers left.
  - By performing many carry-save additions in parallel, Wallace trees allow two n-bit numbers to be multiplied in $\Theta(\log_2 n)$ time using a circuit of size $\Theta(n^2)$.

62

- The figure shows a Wallace tree that adds 8 partial products $m^{(0)}$, $m^{(1)}$, …, $m^{(7)}$.
- The partial product $m^{(i)}$ consists of $(n + i)$ bits.
- Each line represents an entire number – the label of an edge indicates the number of bits.
- The carry-lookahead adder at the bottom adds two $(2n\text{-}1)$-bit numbers to give the $2n$-bit product.



Division

64

Introduction

- Division is more complex than multiplication.
- *Example*: Typical values in Pentium-3 processor →
  - Not easy to construct high-speed dividers.
- The ratios have not changed much in later processors.

Instruction	Latency	Cycles / Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Floating-point Add	3	1
Floating-point Multiply	5	2
Floating-point Divide	38	38

- **Latency**:
  - Minimum delay after which the first result is obtained, starting from the time when the first set of inputs is applied.
- **Cycles/Issue**:
  - Whenever a new set of inputs is applied to a functional unit (e.g. adder), it is called an *issue*.
  - Pipelined implementation of arithmetic unit reduces the number of clock cycles between successive issues.
  - For non-pipelined arithmetic units (e.g. divider), the number of clock cycles between successive issues is much higher.
    - Next input can be applied only after the previous operation is complete.

The Process of Integer Division

- In integer division, a *divisor M* and a *dividend D* are given.
- The objective is to find a third number *Q*, called the *quotient*, such that

  D = Q x M + R

  where *R* is the *remainder* such that $0 \le R < M$.
- The relationship $D = Q \times M$ suggests that there is a close correspondence between division and multiplication.
  - Dividend, quotient and divisor correspond to product, multiplicand and multiplier, respectively.
  - Similar algorithms and circuits can be used for multiplication and division.

- One of the simplest division methods is the sequential digit-by-digit algorithm similar to that used in pencil-and-paper methods.

```
                         0 1 1 0           Quotient Q = Q0Q1Q2Q3
Divisor M        1 1 0 | 1 0 0 1 0 1       Dividend D = R0
                         1 1 0             Q0.M         (Does not go; Q0 = 0)
                        ------------
                         1 0 0 1 0 1       R1
                       -   1 1 0           Q1.2^-1.M    (Does go; Q1 = 1)
                        -------------
                           0 1 1 0 1       R2
                       -     1 1 0         Q2.2^-2.M    (Does go; Q2 = 1)
                        -------------
                             0 0 0 1       R3
                               1 1 0       Q3.2^-3.M    (Does not go; Q3 = 0)
                        -------------
                               0 0 1       R4 = Remainder R
```

$D = 37 = (1\,0\,0\,1\,0\,1)_2$

$M = 6 = (1\,1\,0)_2$

Quotient $Q = 6$

Remainder $R = 1$

- In the example, the quotient $Q = Q_0Q_1Q_2\ldots$ is computed one bit at a time.
  - At each step i, the divisor shifted i bits to the right (i.e. $2^{-i}.M$) is compared with the current partial remainder R_i.
  - The quotient bit Q_i is set to 0 (1) if $2^{-i}.M$ is greater than (less than) R_i .
  - The new partial remainder R_{i+1} is computed as:

$$R_{i+1} = R_i - Q_i.2^{-i}.M$$

- **Machine implementation:**
  - For hardware implementation, it is more convenient to shift the partial remainder to the left relative to a fixed divisor; thus

    $R_{i+1} = 2R_i - Q_i.M$ (*instead of* $R_{i+1} = R_i - Q_i.2^{-i}.M$)
  - The final partial remainder is the required remainder shifted to the left, so that $R = 2^{-3}.R_4$ (see next slide).

Divisor M			Quotient Q
1 1 0	1 0 0 1 0 1	Dividend = $2R_0$	
Do not subtract	1 1 0	$Q_0.M$	0
	---------------		
	1 0 0 1 0 1	R_1	
	1 0 0 1 0 1 0	$2R_1$	
	1 1 0	$Q_1.M$	0 1
	---------------		
	0 1 1 0 1 0	R_2	
	0 1 1 0 1 0 0	$2R_2$	
	1 1 0	$Q_2.M$	0 1 1
	---------------		
	0 0 0 1 0 0	R_3	
	0 0 0 1 0 0 0	$2R_3$	
	1 1 0	$Q_3.M$	0 1 1 0
	---------------		
	0 0 1 0 0 0	$R_4 = 2^3.R$	

$D = 37 = (1\,0\,0\,1\,0\,1)_2$

$M = 6 = (1\,1\,0)_2$

Quotient $Q = 6$

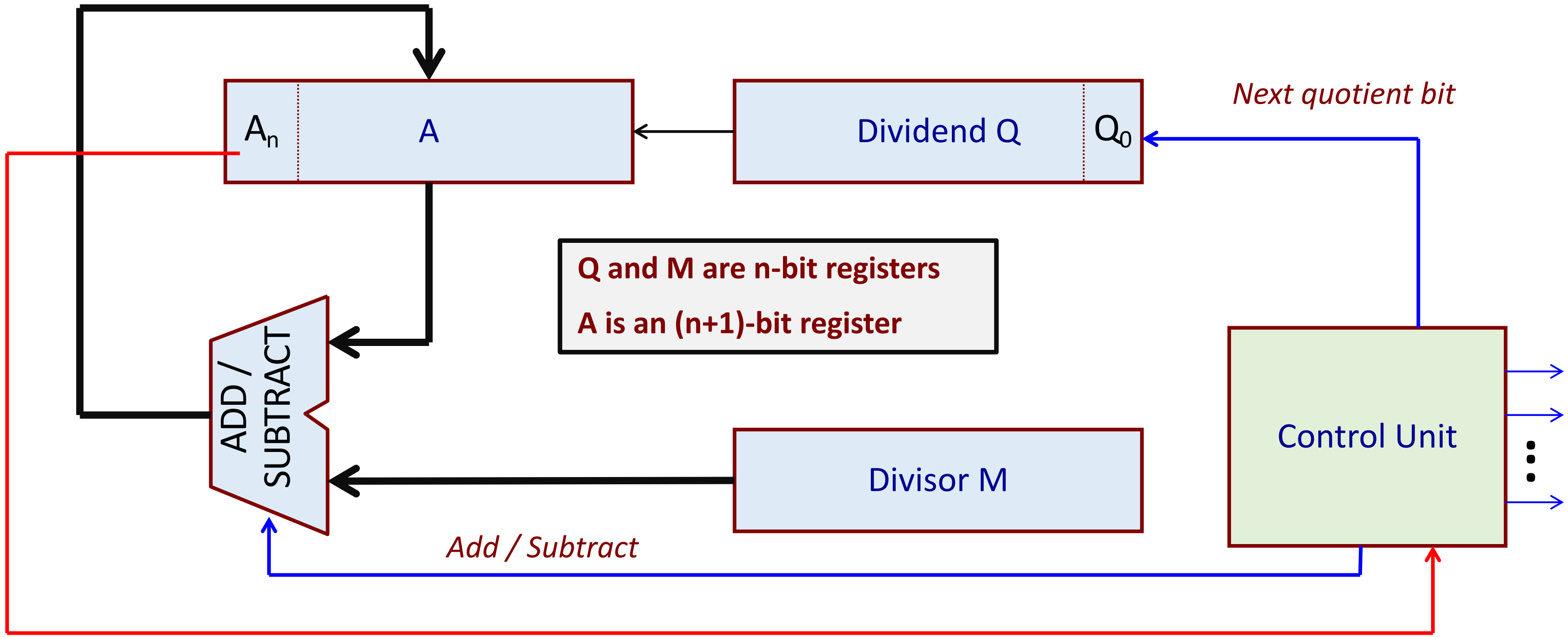
Remainder $R = 1$

71

Two alternatives approaches for division

- We shall discuss two approaches:
  a) Restoring division
  b) Non-restoring division

(a) Restoring Division: The Data Path



Q and M are n-bit registers

A is an (n+1)-bit register

Basic Steps (Restoring Division)

Repeat the following steps *n* times:

a) Shift the dividend left one bit at a time into register *A*.

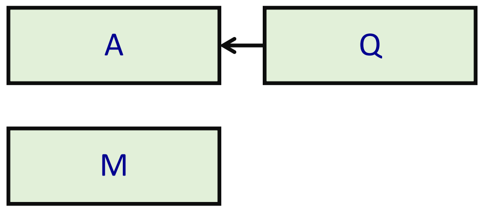
b) Subtract the divisor *M* from this register *A* (*trial subtraction*).

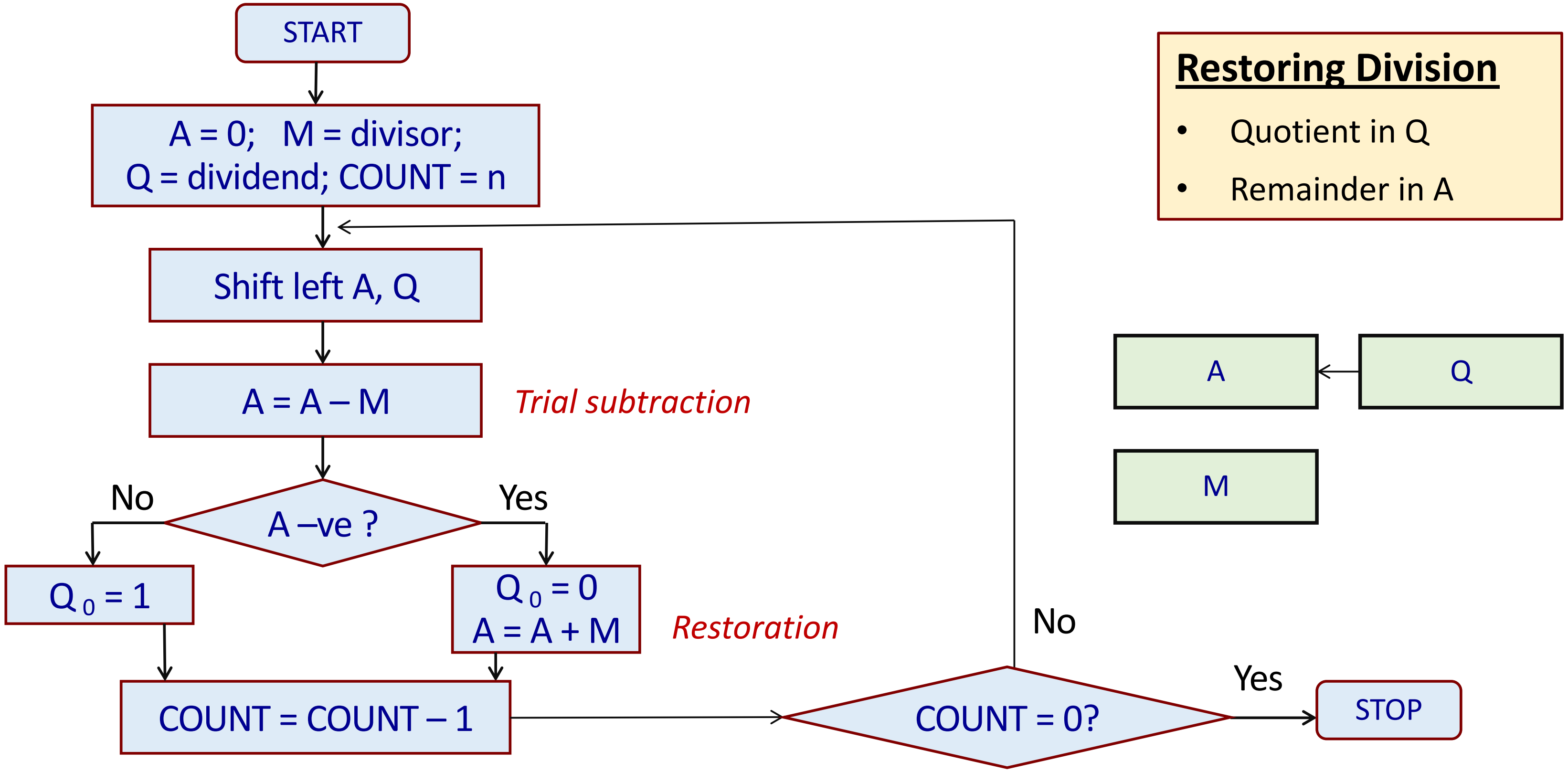
c) If the result is negative (*i.e. not going*):

- Add the divisor *M* back into the register *A* (*i.e. restoring back*).
- Record 0 as the next quotient bit.

d) If the result is positive:

- Do not restore the intermediate result.
- Record 1 as the next quotient bit.





Restoring Division

- Quotient in Q
- Remainder in A

- **Analysis**:
  - For n-bit divisor and n-bit dividend, we iterate n times.
  - Number of trial subtractions: n
  - Number of restoring additions: $n/2$ on the average
    - Best case: 0
    - Worst case: n

A Simple Example: 8/3 for 4-bit representation (n=4)

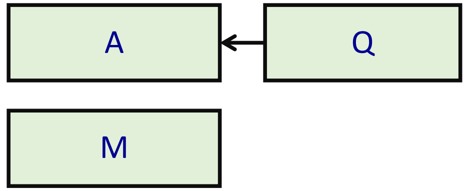
```
Initially:   0 0 0 0 0    1 0 0 0

Shift:       0 0 0 0 1    0 0 0 –
Subtract:      0 0 1 1
Set Q0:      1 1 1 1 0
Restore:       0 0 1 1
             0 0 0 0 1    0 0 0 0
Shift:       0 0 0 1 0    0 0 0 –
Subtract:      0 0 1 1
Set Q0:      1 1 1 1 1
Restore:       0 0 1 1
             0 0 0 1 0    0 0 0 0
```

```
Shift:       0 0 1 0 0    0 0 0 –
Subtract:      0 0 1 1
Set Q0:      0 0 0 0 1
             0 0 0 0 0    0 0 0 1
Shift:       0 0 0 1 0    0 0 1 –
Subtract:      0 0 1 1
Set Q0:      1 1 1 1 1
Restore:       0 0 1 1
             0 0 0 1 0    0 0 1 0
```

Remainder 00010 = 2   Quotient 0010 = 2

(b) Non-Restoring Division



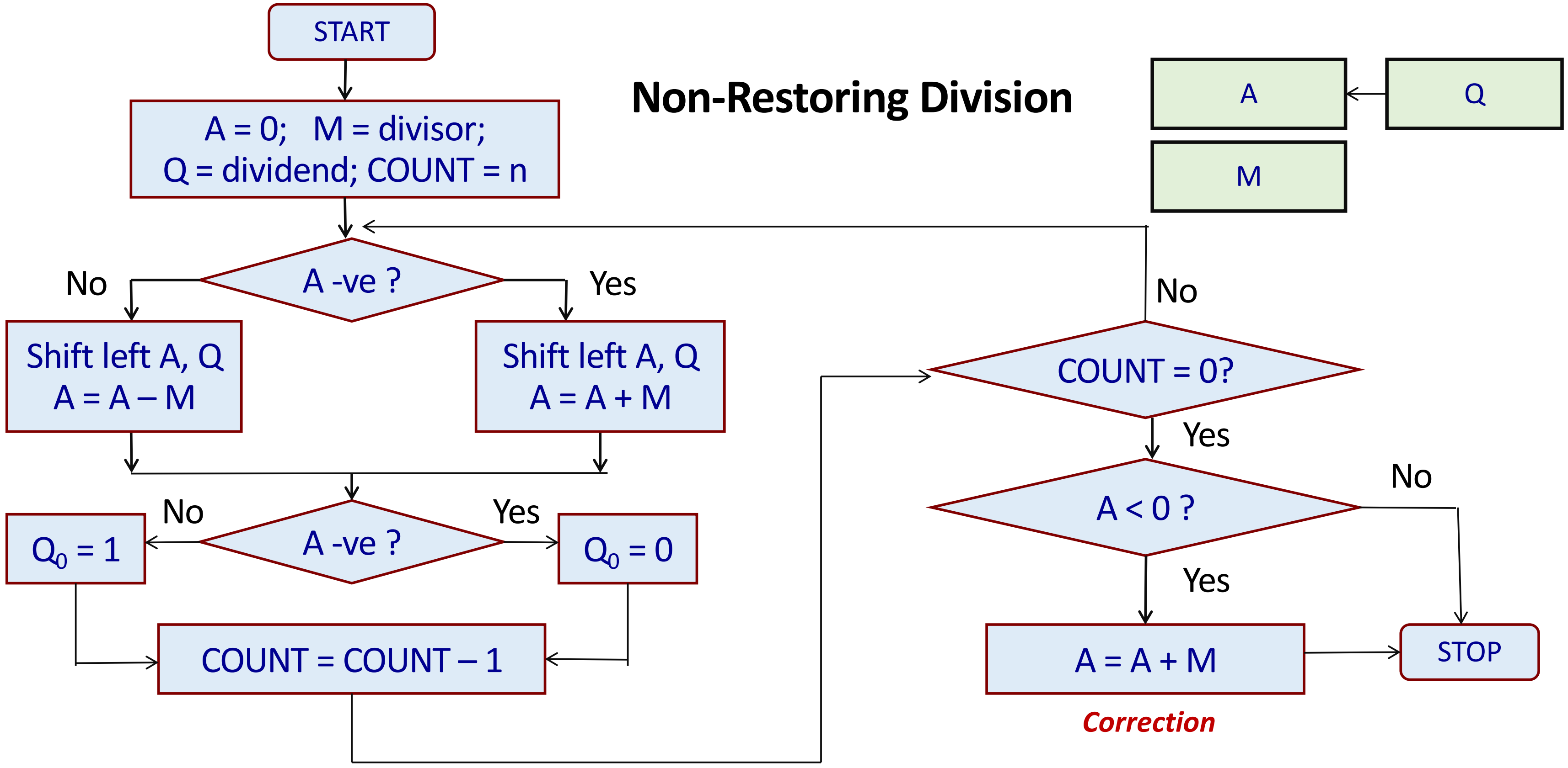
- The performance of restoring division algorithm can be improved by exploiting the following observation.
- In restoring division, what we do actually is:
  - If A is positive, we shift it left and subtract M.
    - That is, we compute $2A - M$.
  - If A is negative, we restore is by doing $A + M$, shift it left, and then subtract M.
    - That is, we compute $2(A + M) - M = 2A + M$.
- We can accordingly modify the basic division algorithm by eliminating the restoring step → *NON-RESTORING DIVISION*.

Shift left means multiplying by 2.

- **Basic steps in non-restoring division:**
  a) Start by initializing register *A* to 0, and repeat steps (b)-(d) *n* times.
  b) If the value in register *A* is positive,
     - Shift *A* and *Q* left by one bit position.
     - Subtract *M* from *A*.
  c) If the value in register *A* is negative,
     - Shift *A* and *Q* left by one bit position.
     - Add *M* to *A*.
  d) If *A* is positive, set $Q_0 = 1$; else, set $Q_0 = 0$.
  e) If *A* is negative, add *M* to *A* as a final corrective step.

79
79

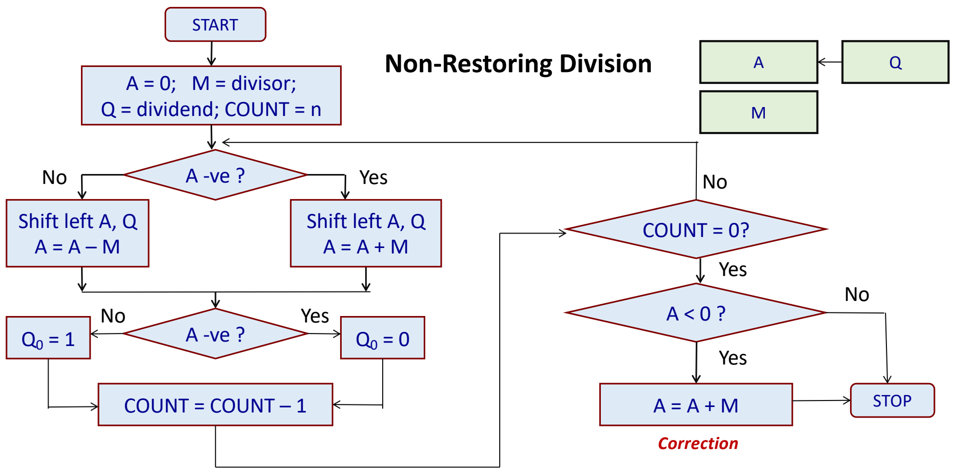
Non-Restoring Division

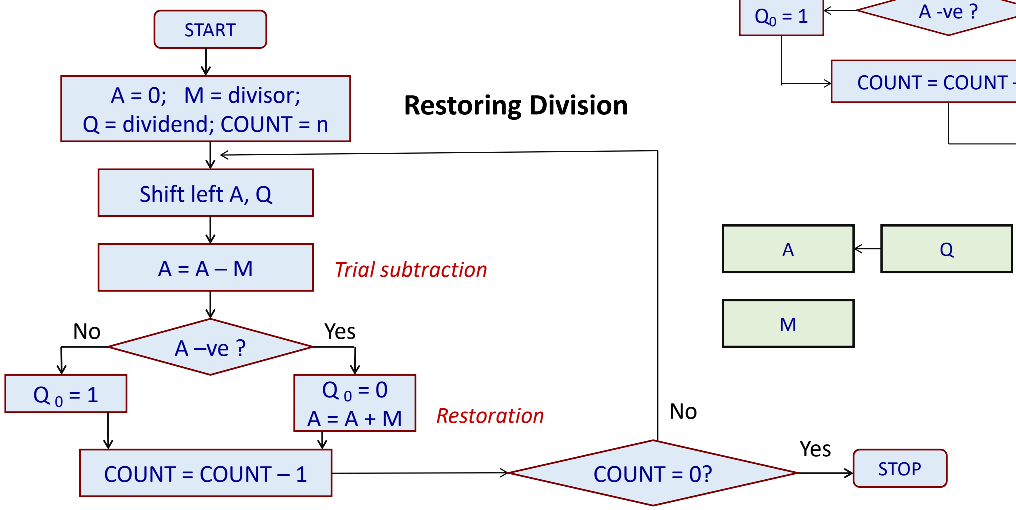


80

Restoring Division:

- If A is positive, we shift it left and subtract M. That is, we compute $2A - M$.
- If A is negative, we restore is by doing $A + M$, shift it left, and then subtract M. That is, we compute $2(A + M) - M = 2A + M$.





A Simple Example: 8/3 for n=4

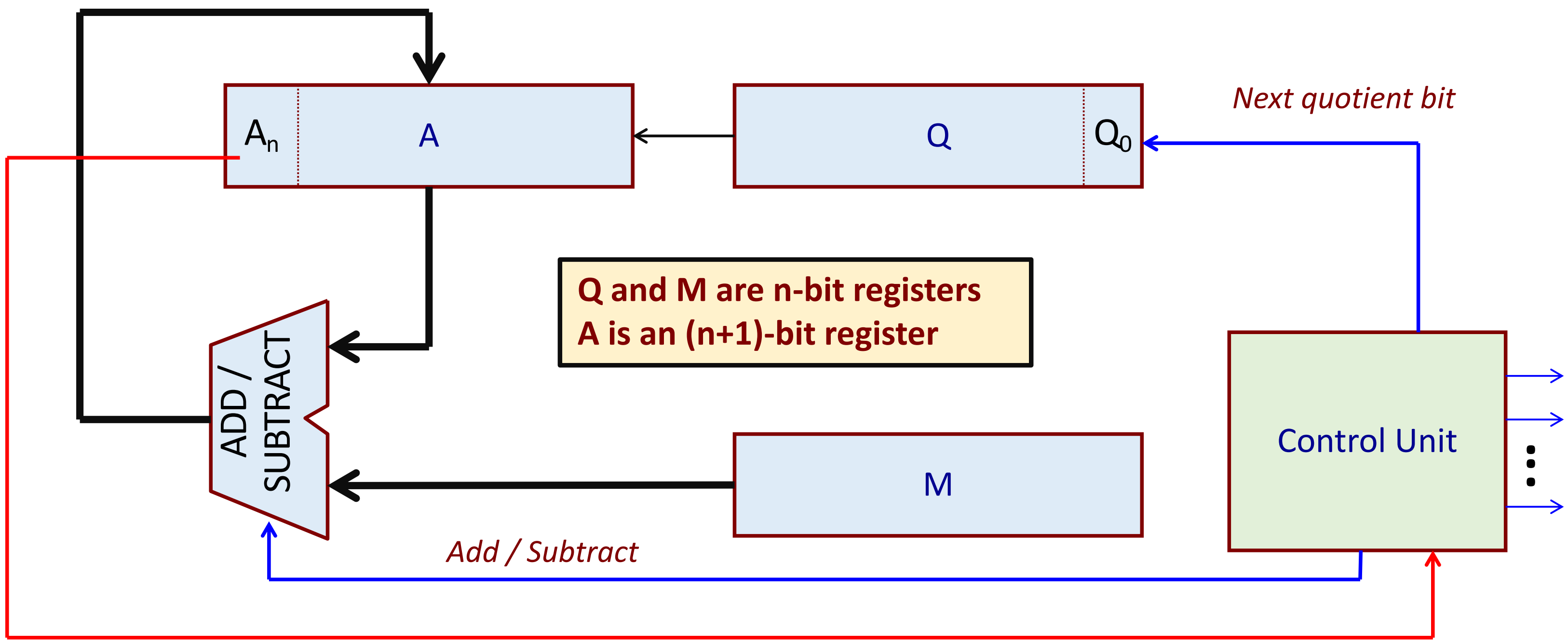
Initially:	0 0 0 0 0	1 0 0 0
Shift:	0 0 0 0 1	0 0 0 –
Subtract: –	0 0 1 1	
Set Q_0:	1 1 1 1 0	0 0 0 0
Shift:	1 1 1 0 0	0 0 0 –
Add:	0 0 1 1	
Set Q_0:	1 1 1 1 1	0 0 0 0
Shift:	1 1 1 1 0	0 0 0 –
Add:	0 0 1 1	
Set Q_0:	0 0 0 0 1	0 0 0 1

Shift:	0 0 0 1 0	0 0 1 –
Subtract: –	0 0 1 1	
Set Q_0:	1 1 1 1 1	0 0 1 0
Correction Add:		
	1 1 1 1 1	
	0 0 0 1 1	
	0 0 0 1 0	

Quotient
0010 = 2

Remainder
00010 = 2

Data Path for Non-Restoring Division



High Speed Dividers

- Some of the methods used to increase the speed of multiplication can also be modified to speed up division.
  - High-speed addition and subtraction.
  - High-speed shifting.
  - Combinational array divider (implementing restoring division).
- The main difficulty is that it is very difficult to implement division in a pipeline to improve the performance.
  - Unlike multiplication, where carry-save Wallace tree multipliers can be used for pipeline implementation.