Computer Arithmetic

Introduction

Multiplication Algorithms

Division Algorithms

Floating-Point Arithmetic Operations

Sign-magnitude number

- A sign-magnitude number Z can be represented as (A_s, A) where A_s is the sign of Z and A is the magnitude of Z.
- The leftmost position, A_s, is the sign bit.
- The sign bit is either positive = 0 or negative = 1

Number	Signed-Magnitude
+3	0 11
+2	0 10
+1	0 01
+0	0 00
-0	1 00
-1	1 01
-2	1 10
-3	1 11

2’s Complement Representation

- **Two's complement** is a method of signifying negative integers in computer science .
- To compute 2′s complement of a number:
  - Find binary representation in 8 bits.
  - Complement the entire positive number and then add one.
  - Eg: +33 is represented as 00100001 and –33 is represented as 11011111.

Addition

Algorithm:

- Add the two numbers, including their sign bits.
- Discard any carry out of the sign bit position.
- All negative numbers must be in the 2’s complement form.
- If the sum obtained is negative, then it is in 2’s complement form.

$$\boxed{\begin{array}{c} AC \leftarrow AC + BR \\ V \leftarrow \text{overflow} \end{array}}$$

Examples

$$\begin{array}{rl} +6 & 00000110 \\ +13 & 00001101 \\ \hline +19 & 00010011 \end{array} \qquad\qquad \begin{array}{rl} -6 & 11111010 \\ +13 & 00001101 \\ \hline +7 & 00000111 \end{array}$$

Subtraction

Algorithm:

- Take the 2’s complement form of the subtrahend(including the sign bit).
- Add it to the minuend (including the sign bit).
- A carry out of the sign bit position is discarded.

Examples

+6	00000110	-6	11111010
<u>-13</u>	<u>11110011</u>	<u>-13</u>	<u>11110011</u>
-7	11111001	-19	11101101

Overflow

- When two numbers of n digits each are added and the sum occupies n + 1 digits.
- Since the width of a register is finite,a flag(overflow) is set.
- For addition of signed numbers, an overflow cannot occur if one is positive and one is negative – both need to have the same sign.
- Detected if the carry into the sign bit position and the carry out of the sign bit position are not equal.

Benefits of 2’s Complement

- Addition and subtraction simplified in the 2’s complement system.
- In 8 bits, -0 has been eliminated, replaced by –128 (10000000),for which there is no corresponding positive integer.

FLOATING POINT ARITHMETIC

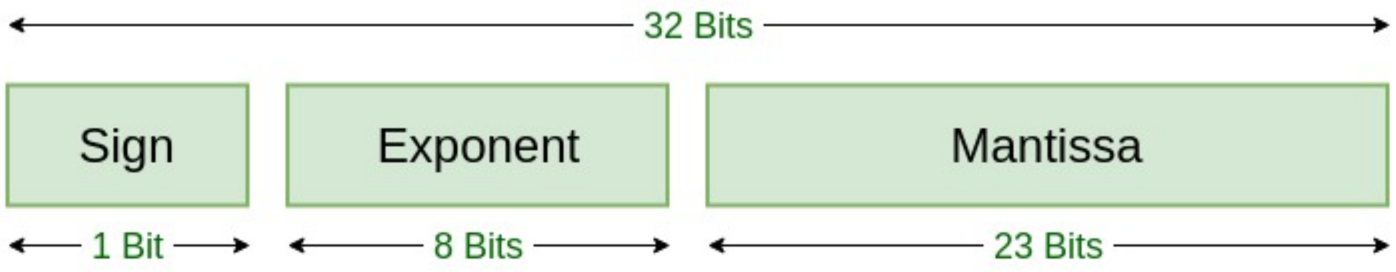
IEEE Standard for Floating point Number

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**. The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

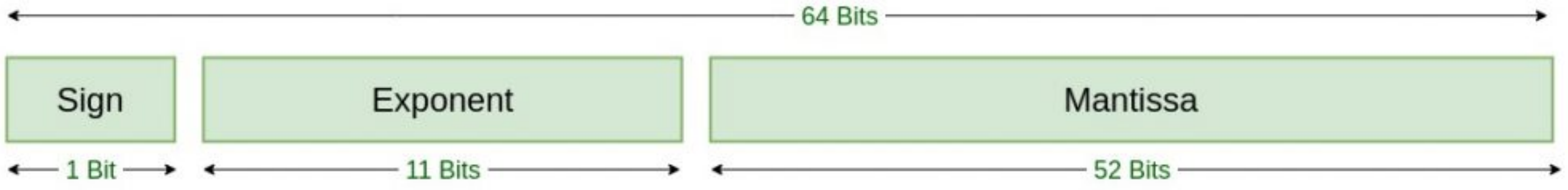
- There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

1. **The Sign of Mantissa –**
   This is as simple as the name. 0 represents a positive number while 1 represents a negative number.
2. **The Biased exponent –**
   The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.
3. **The Normalised Mantissa –**
   The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

- **IEEE 754 numbers are divided into two based on the above three components: single precision and double precision.**
-



Single Precision
IEEE 754 Floating-Point Standard



Double Precision
IEEE 754 Floating-Point Standard

TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

BINARY REPRESENTATION OF FLOATING POINT NUMBERS

Converting decimal fractions into binary representation.

Consider a decimal fraction of the form: 0.d1d2...dn

We want to convert this to a binary fraction of the form:

0.b1b2...bn (using binary digits instead of decimal digits)

COMPUTER REPRESENTATION OF FLOATING POINT NUMBERS

In the CPU, a 32-bit floating point number is represented using IEEE standard format as follows:

```
    S | EXPONENT | MANTISSA
```

where S is one bit, the EXPONENT is 8 bits, and the MANTISSA is 23 bits.

- The ***mantissa*** represents the leading significant bits in the number.

- The ***exponent*** is used to adjust the position of the binary point (as opposed to a "decimal" point)

The mantissa is said to be **normalized** when it is expressed as a value between 1 and 2. I.e., the mantissa would be in the form 1.xxxx.

The leading integer of the binary representation is not stored. Since it is always a 1, it can be easily restored.

The "S" bit is used as a sign bit and indicates whether the value represented is positive or negative

(0 for positive, 1 for negative).

If a number is smaller than 1, normalizing the mantissa will produce a negative exponent.

But 127 is added to all exponents in the floating point representation, allowing all exponents to be represented by a positive number.

Example 1. Represent the decimal value 2.5 in 32-bit floating point format.

2.5 = 10.1b

In normalized form, this is: $1.01 * 2^1$

The mantissa: M = 01000000000000000000000
(23 bits without the leading 1)

The exponent: E = 1 + 127 = 128 = 10000000b
The sign: S = 0 (the value stored is positive)

So, 2.5 = 01000000001000000000000000000000

Example 2: Represent the number -0.00010011b in floating point form.

0.00010011b = 1.0011 * 2^{-4}

Mantissa: M = 00110000000000000000000 (23 bits with the integral 1 not represented)

Exponent: E = -4 + 127 = 01111011b

S = 1 (as the number is negative)

Result: 1 01111011 00110000000000000000000

Exercise 1: represent -0.75 in floating point format.

Exercise 2: represent 4.9 in floating point format.

MULTIPLICATION ALGORITHM

- Multiplication is binary operation involving multiplicand and multiplier.
- Multiplication of signed binary representation is done manually by a process of successive shift and addition operations.

EXAMPLE

```
23   1 0 1 1 1   Multiplicand
19       × 1 0 0 1 1   Multiplier
---  -----------
     1 0 1 1 1
                      1 0 1 1 1
    0 0 0 0 0       +
            0 0 0 0 0
         1 0 1 1 1
         ---------------
    437 1 1 0 1 1 0 1 0 1       Product
```

- Sign of the product is determined from the sign of the multiplicand and multiplier.
- If alike, the product is positive else it's negative.

HARDWARE IMPLEMENTATION OF SIGNED-MAGNITUDE DATA

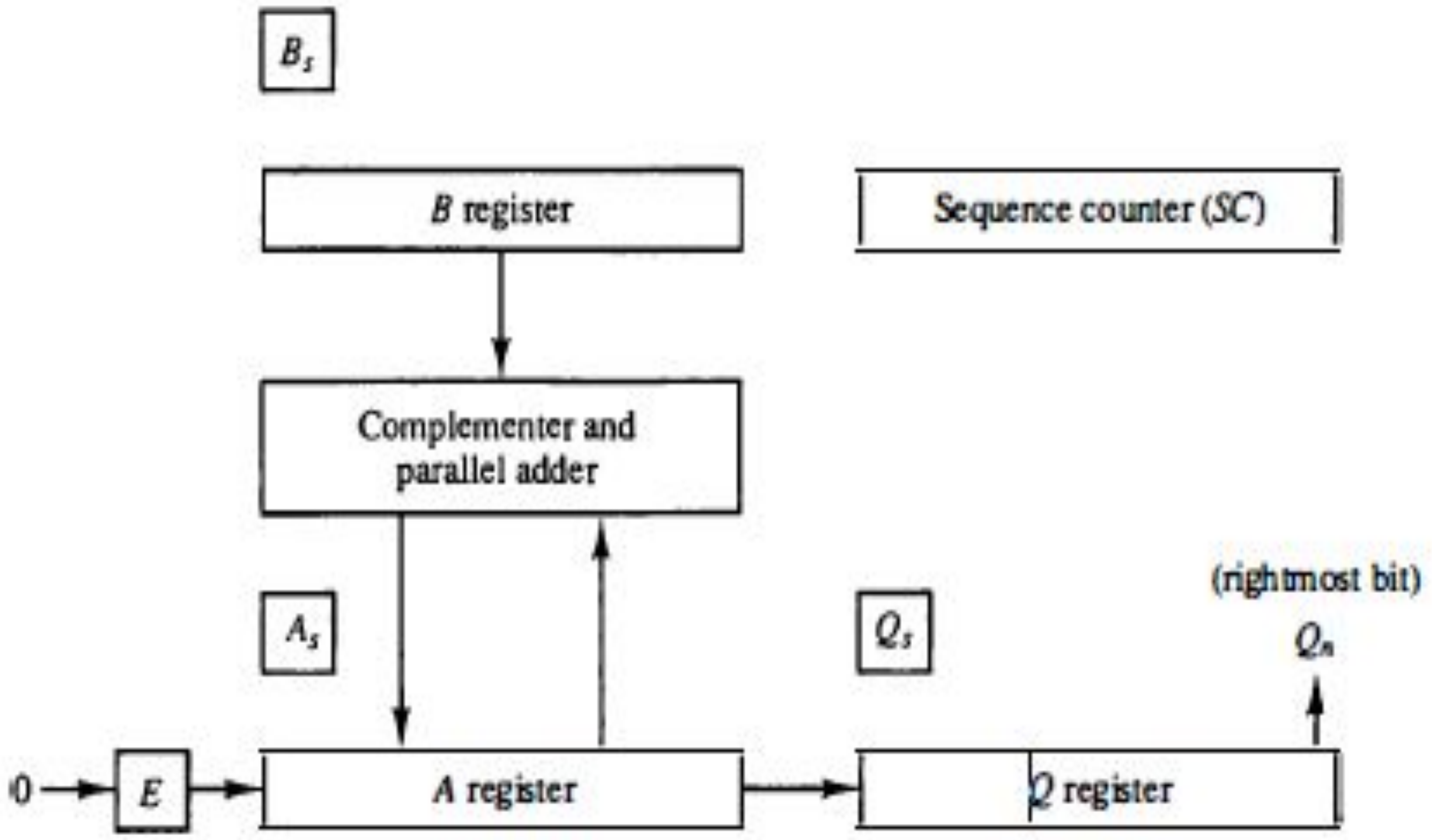
- It's done with the help of an adder for the summation of only two binary numbers and successively accumulates the partial products in a register.
- Secondly, instead of shifting the multiplicand to the left, partial products is shifted to the right.
- Thirdly, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product.

HARDWARE FOR MULTIPLY OPERATION

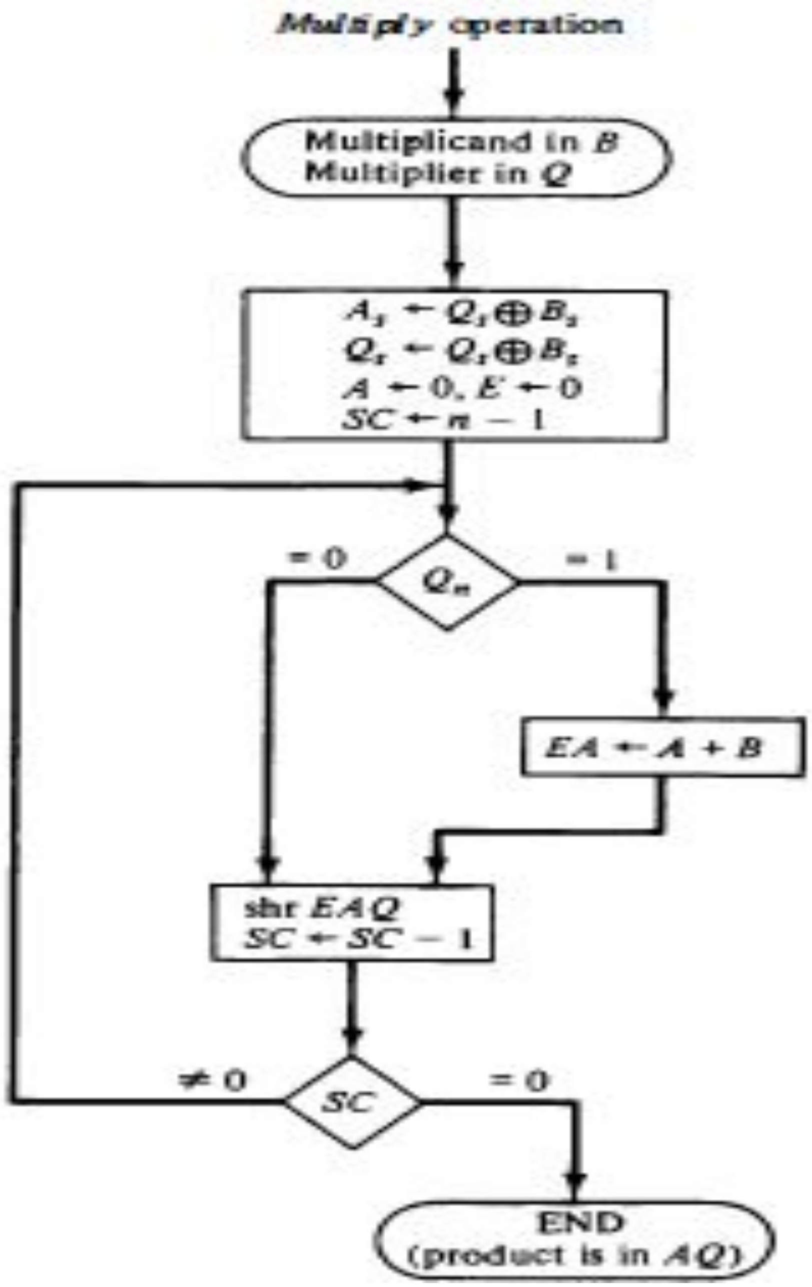
Steps:

- The multiplier is stored in the Q register and its sign in QS
- Set the sequence counter to a number equal to the number of bits in the multiplier
- Initially, multiplicand is in register B and multiplier in Q. Sum of A and B forms a partial product. This is transferred to the EA register. Both partial product and multiplier shift to the right.
- The corresponding signs are in Bs and Qs respectively.
- Signs are compared, and both A and Q are set to correspond to the sign of the product.
- Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits in the multiplier.
- LSB of the multiplier in Qn is tested, if a 1, the multiplicand in B is added to the present partial product in A. if it's 0 nothing is done.
- Register EAQ is then shifted once to the right to form the new partial product. SC is decremented by 1.
- SC value checked, if not equal to zero, the process is repeated and partial product is formed. If equal to zero, the process stops.

Hardware for multiply operation



Flowchart for multiply operation



EXAMPLE

Ques: multiply 23 *19 using binary multiplier or EAQ Multiplier.

Sol: 23= 10111(Multiplicand)

  19=10011(Multiplier)

B=10111

E= 0

A=no. of bits in Q=>   A=00000

Q= 10011

SC= 101

Example Contd..

23*19

B=23=10111 (N BIT)

Q=19=10011 (N BIT)

RESULT = 2 N+1= N + N+1(E)= AQ REGIS

E= 0

A=00000

SC= 101=5

Qn is the LSB of Q register

Numerical example for Binary Multiplier

Multiplicand B=10111	E	A	Q	SC
1) initialization	0	00000	10011	101
2) Qn=1, EA->A+B		10111		
	0	10111	10011	
2) ii) shr EAQ	0	01011	11001	100
3) Qn=1, EA-> A+B		10111		
	1	00 010	11001	
Shr EAQ	0	10001	01100	011
4) Qn= 0, shr EAQ	0	01000	10110	010
5) Qn=0, shr EAQ	0	00100	01011	001
6) Qn=1, EA->A+B		10111		
	0	110 11		
Shr EAQ	0	01101	10101	000

- E A Q
- 0 10111 10011= 01011110011

Shr EAQ = 0 01011 11001

Ans 23*19= AQ=0110110101

exercise

- Perform 45*17 multiplication using binary multiplier.

- 45= 6 bit =B
- Q=6 bit
- A= 6 bit
- SC= 6

Booth’s Algorithm

Booth’s Algorithm

- Works on signed-2’s complement represented binary numbers
- Includes examination of multiplier bits and shifting of partial sum
- But works on one very important property of 2-complement representation

Booth’s Algorithm Continued …

- Algorithm is based on the following property

  6 = -2 + 8

  0110 = - 0010 + 0100

- A string of 1’s can be replaced by a initial subtract for the first 1 encountered and finally addition for the last 1 in the series.
- Works well for signed numbers too.

Booth’s Algorithm Continued …

- Given this property now while finding the product the following rules are applied when looking at the bits in the multiplier
  - Subtract the multiplicand from partial product on encountering “01” (First least significant 1)
  - Add the multiplicand to partial product on encountering “10” (First 0 following a 1)
  - No change if the bits are same
- No matter what the case we still need to shift the partial product

BOOTH MULTIPLICATION ALGORITHM

- It involves multiplying binary integers in a signed 2’s complement representation.
- It operates on the fact that strings of 0’s in the multiplier requires no addition but just shifting and string of 1’s in a multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} – 2^m$.
- As in all multiplication schemes, this algorithm requires examination of the multiplier bits and shifting of partial product.
- Prior to the shifting, the multiplicand may be added to, subtracted from the partial product, or left unchanged according to the following rules:-
  - **The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier.**
  - **The multiplicand is added to the partial product upon encountering the first 0. in a string of 0’sin the multiplier.**
  - **The partial product does not change when the multiplier bit is identical to the previous multiplier bit.**

Hardware

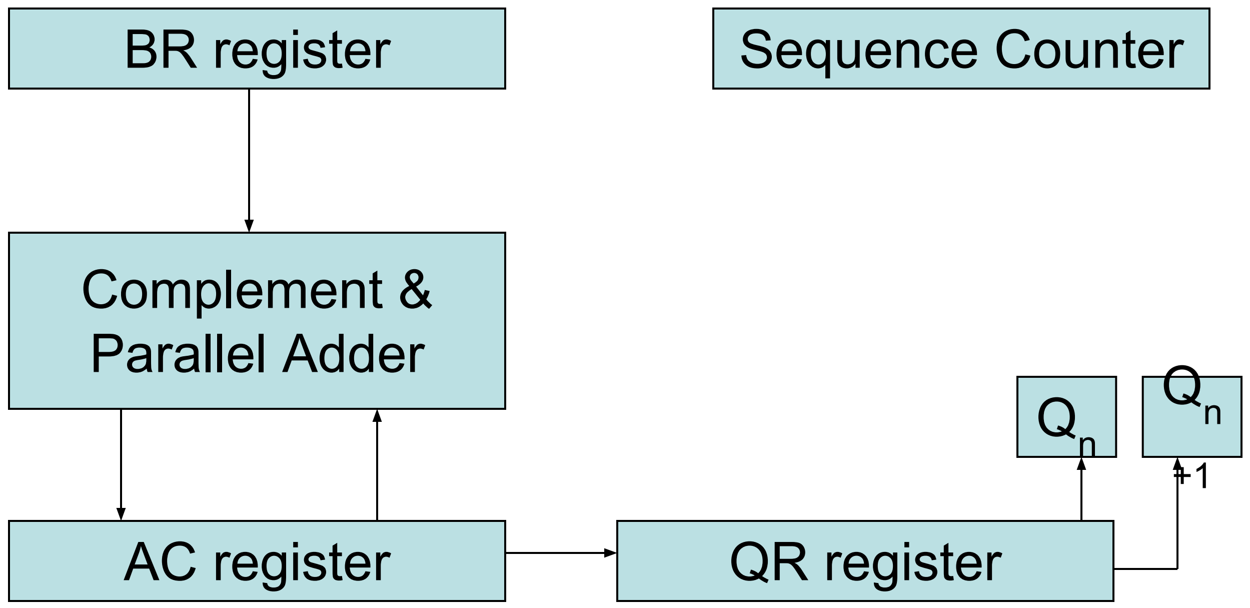
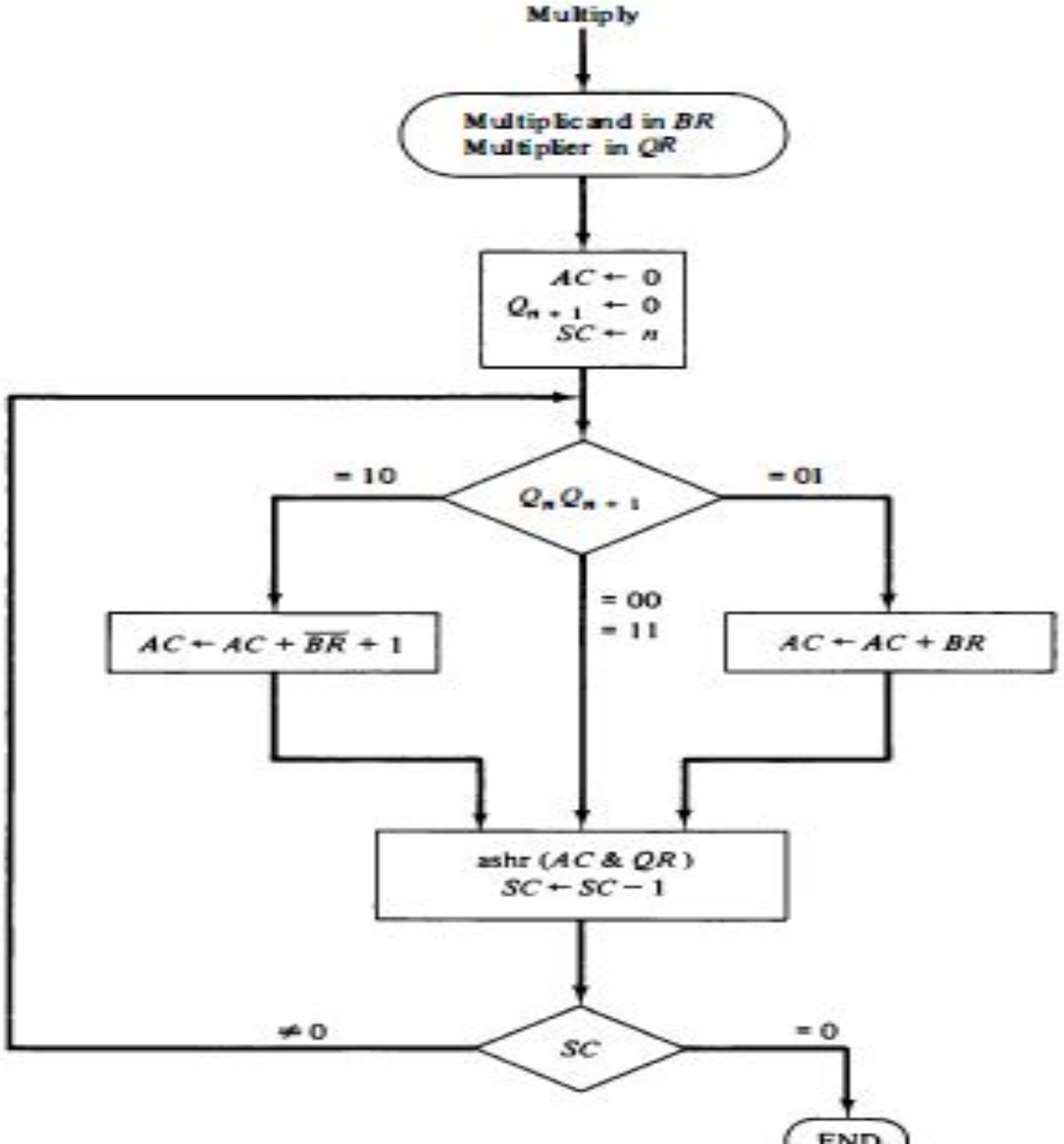


Figure Hardware for Booth algorithm

Flow Chart for Booth Algorithm



BOOTH

- 1. –ve no 2’s complement
- 2. no. represent 5 bit no
- 3. carry bit will be lost
- 4. after performing ashr lsb lost ,
- Lost bit will be store in qn+1

Example for BOOTH algorithm

Ques: multiply (-9)*(-13) using booth algorithm.

Solution:

+9= 01001

-9=10110+1=10111=BR(MULTIPLICAND)

+13=01101

-13=10010+1=10011=QR(MULTIPLIER)

Ex contd..

Q_n=lsb of QR

$Q_{n+1}= 0$

BR= 10111(-9)multiplicand

BR’+1=01001

AC=00000

QR= 10011(-13) multiplier

SC=101=5=total no. of steps to reach result

Result will be store in AC ,QR

Ex of Multiplication using Booth Algorithm

Q_n	Q n+1	BR= 10111 BR'+1= 01001	AC	QR	Q n+1	SC
		initial	00000	10011	0	101
1	0	Subtract AC-BR	01001			
			01001	10011		
		ashr	00100	11001	1	100
1	1	Ashr	00010	01100	1	011
0	1	ADD (AC+BR)	10111			
			11001	01100		
		Ashr	11100	10110	0	010
0	0	Ashr	11110	01011	0	001
1	0	subtract	01001			
			00111	01011		
		ashr	00011	10101	1	000

RULE for BOOTH Algo

- +A*+B= RESULT +VE
- 2. (–A)* (B)=-VE RESULT 2’S COMPLEMENT
- 3. (A)*(-B)= -VE NO. Result 2’s complement
- 4. (-A)*(-B)= =+VE

Performance

- Faster than the general paper – pencil algorithm
- As this algorithm reduces the number of actually additions and subtraction performed
- But encounters the worst case scenario when the multiplier bit pattern consists of alternating 1’s & 0’s. In this case the performance is bad than that of the paper-pencil algorithm

Example

- A = 2, B = -6, compute A*B
- A = 0010, B = 1010 (worst case)

```
      0010      Multiplicand
     *1010      Multiplier
     ----------
       0000      00 – No OP
          -1010        11 – Subtract Multiplier
        +1010          01 – Add Multiplier
       -1010           10 – Subtract Multiplier
----------------------
1111 0100
```

Array Multiplier

ARRAY MULTIPLIER

- ARRAY MULTIPLIER
- Using a combinational circuit, it's possible to obtain the multiplication of two binary numbers using one micro-operation.
- And the time it takes is equivalent to the time for the signals to propagate through the gates that form the multiplication array.
- This is made possible by the development of the integrated circuits.
- EXAMPLE
- 2–bit by 2–bit array multiplier.

```
•                 b1 b0
•                 a1 a0
•            -------
•                   a0b1 a0b0
•         a1b1    a1b0
• c3          c2         c1       c0
```

- NB. AND gates and HA are used to achieve this results.

2 BIT BY 2BIT MULTIPLIER A*B= A1A0 * B1B0

a1 a0

- B1b0

---

boa1 b0a0

- 2 BIT BY 2BIT MULTIPLIER
- A*B= A1A0 * B1B0

- 3BIT BY 2 BIT ARRAY MULTIPLIER
- A2A1A0 * B1B0

- 2BIT BY 3BIT ARRAY MULTIPLIER

Elementary Multiplication Algorithm

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

- Consist of successive adding and shifting partial product
- Multiple adding and shifting operations needed in serial multiplier
- Can we do it one operation?

2 × 2 multiplication

$$\begin{array}{cccc}
 & & b_1 & b_0 \\
 & & a_1 & a_0 \\
\cline{3-4}
c_2 \nwarrow & c_1 \nwarrow & a_0b_1 & a_0b_0 \\
 & a_1b_1 & a_1b_0 & \\
\hline
p_3 & p_2 & p_1 & p_0
\end{array}$$

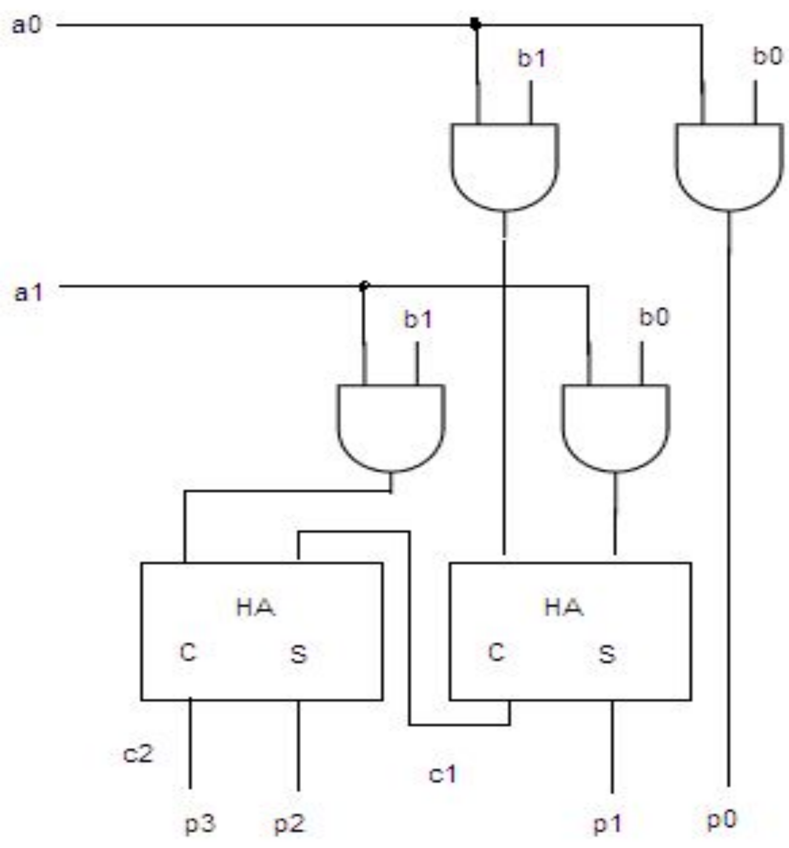
$p_0 = a_0$ AND b_0

$p_1 = (a_0$ AND $b_1)$ XOR $(a_1$ AND $b_0)$

$p_2 = c_1$ XOR $(a_1$ AND $b_1)$

$p_3 = c_2$

2 × 2 simple combinatorial multiplier



- 2 × 2 multiplier can be made with combinatorial circuit.
- Multiplication of two numbers can be performed in one microoperation.

Figure 10.9 2-bit by 2-bit array multiplier.

Division Algorithms: Signed- Magnitude Data

Division Algorithms: Outline

- Division: Shift – subtract Methodology
  - 3 ways to represent signed number
    - Signed Magnitude
    - 1’s Compliment
    - 2’s Compliment
  - Dividend = divisor * quotient + remainder
- Division By Hand: Long Division
- Hardware Algorithm: Restoring Method

Division Algorithm: Long Division

1. Determine Quotient Sign bit
2. Compare divisor with partial remainder
3. Subtract divisor from partial remainder if partial remainder > divisor
4. Shift divisor to the right
5. Repeat 2–4 until visit least significant bit of partial remainder
6. Combine the sign result from number 1 with the quotient and remainder

Signed-Magnitude

Dividend = -221 = 111011101

Divisor = 3 = 011

1. $Q_s = \text{Divisor}_s \text{ XOR } \text{Dividend}_s = 1$

   $\text{Remainder}_s = \text{Dividend}_s = 1$

2. – 4.

```
            1  0  0  1  0  0  1
          ______________________
11   /    1  1  0  1  1  1  0  1
          1  1
     __________________________
             0  0  1  1
                   1  1
     __________________________
                      0  1  0  1
                            1  1
     __________________________
                            1  0
```

Quotient = 11001001 = -73

Remainder = 110 = -2

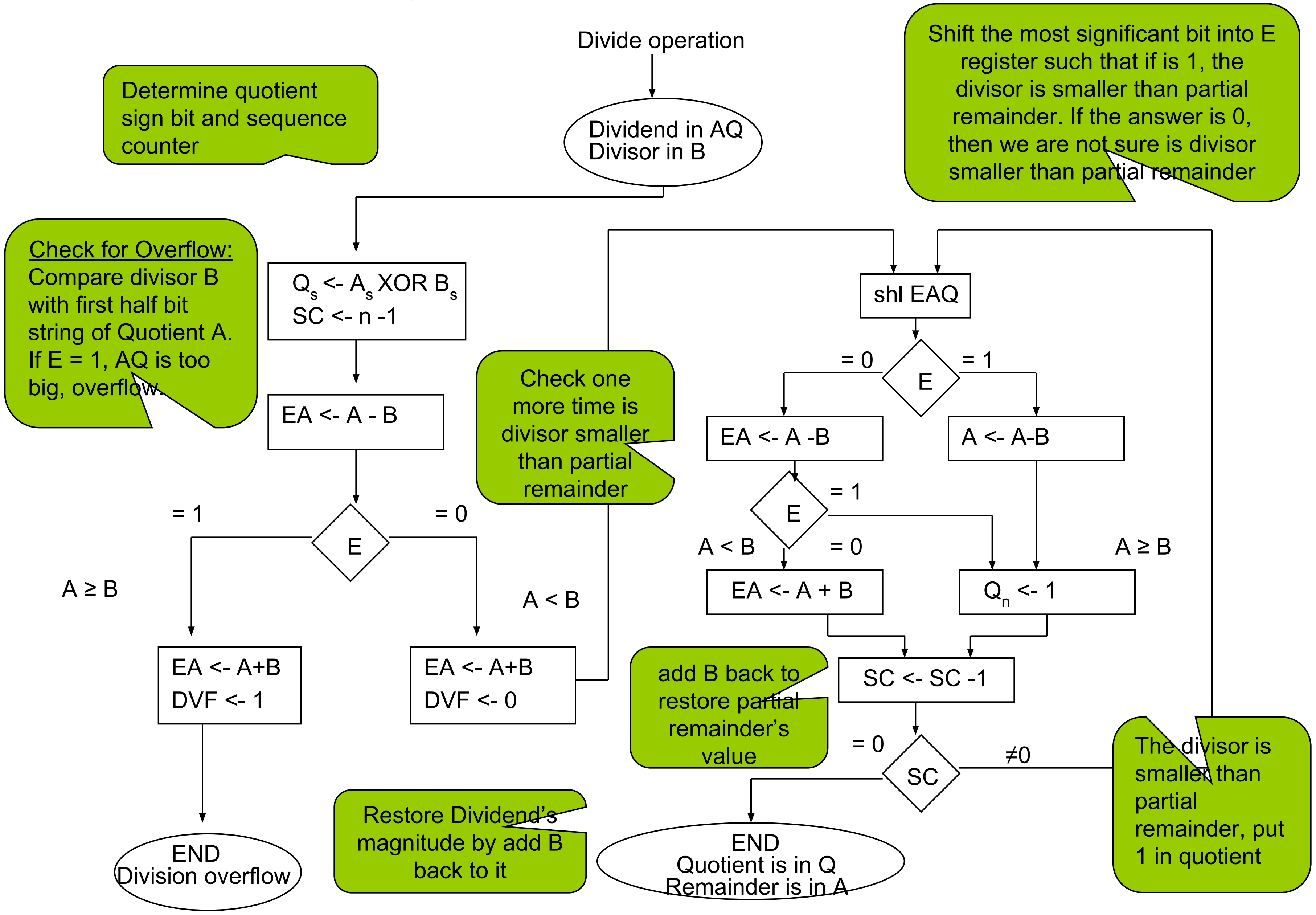
- 221 = -73 * 3 + -2

Division Algorithm: Restoring Method

- Division Overflow
  - Constraint by register size
  - Dividend = A Q; Divisor = B
    - A > B overflow
  - DVF: divide-overflow flip-flop
- Divide by Zero
  - Take care when check overflow

1. Determine Sign for quotient and remainder
2. Check for Overflow
3. Shift dividend to left
4. Check is partial remainder larger than divider
5. If larger, subtract divisor from partial remainder and put 1 for quotient
6. If smaller, do nothing
7. Repeat 3–6 for N times such that N = number of how many bit pattern a register can hold -1
8. Combine sign with quotient and remainder

Division Algorithm: Restoring Method


Divide operation
Dividend in AQ Divisor in B
Determine quotient sign bit and sequence counter
Shift the most significant bit into E register such that if is 1, the divisor is smaller than partial remainder. If the answer is 0, then we are not sure is divisor smaller than partial remainder
Check for Overflow: Compare divisor B with first half bit string of Quotient A. If E = 1, AQ is too big, overflow.
Q_s <- A_s XOR B_s
SC <- n -1
EA <- A - B
E
= 1
= 0
A ≥ B
A < B
EA <- A+B
DVF <- 1
EA <- A+B
DVF <- 0
END Division overflow
Restore Dividend's magnitude by add B back to it
Check one more time is divisor smaller than partial remainder
shl EAQ
E
= 0
= 1
EA <- A -B
A <- A-B
E
= 1
= 0
A < B
A ≥ B
EA <- A + B
Q_n <- 1
add B back to restore partial remainder's value
SC <- SC -1
SC
= 0
≠0
END Quotient is in Q Remainder is in A
The divisor is smaller than partial remainder, put 1 in quotient

Division Algorithm: Restoring Method

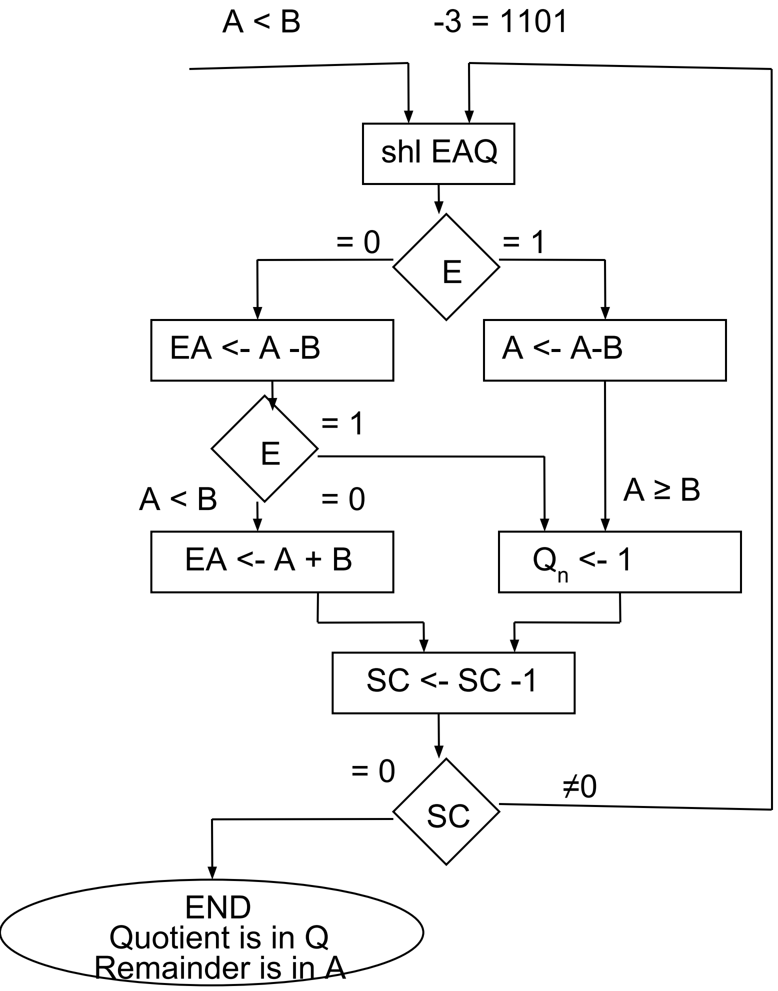
-15 / 3 = -5 …0

15 = 0000 1111

-15 = 1111 0001

3 = 0011

-3 = 1101



1. Qs = 1 XOR 0 = 1…negative

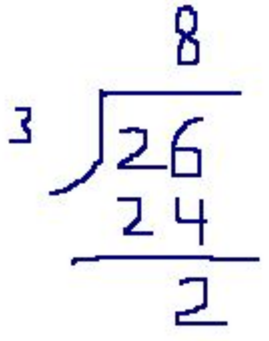
```
     E    A          Q
     0    0 0 0 0    1 1 1 1
          1 1 0 1                  2. Check for Overflow
     -----------------------
     0    1 1 0 1    1 1 1 1       E = 1 not overflow;
          0 0 1 1                  restore value
     -----------------------
     1    0 0 0 0    1 1 1 1
     0    0 0 0 1    1 1 1 0       3&4. Shift to the Left
          1 1 0 1                  Check is partial
     -----------------------       remainder big enough
     0    1 1 1 0    1 1 1 0       6. E =0. Restore original
          0 0 1 1                  partial remainder
     -----------------------
S3   1    0 0 0 1    1 1 1 0       3&4. Shift to the Left
     0    0 0 1 1    1 1 0 0       Check is partial
          1 1 0 1                  remainder big enough
     -----------------------
S2   1    0 0 0 0    1 1 0 1       E = 1; Qn = 1
     0    0 0 0 1    1 0 1 0       3&4. Shift to the Left
          1 1 0 1                  Check is partial
     -----------------------       remainder big enough
     0    1 1 1 0    1 0 1 0       6. E =0. Restore original
          0 0 1 1                  partial remainder
     -----------------------
S1   1    0 0 0 1    1 0 1 0
     0    0 0 1 1    0 1 0 0       3&4. Shift to the Left
          1 1 0 1                  Check is partial
     -----------------------       remainder big enough
     1    0 0 0 0    0 1 0 1
          remainder      quotient
```

Division algorithms--overview

Division with a paper and pencil

We repeatedly

- Comparison
- Shifting
- Subtraction



Division on computers: only deals with 0s and 1s

the same procedures, only easier than the decimals

How it is done in binary—an example

Dividend: 111011 (59)

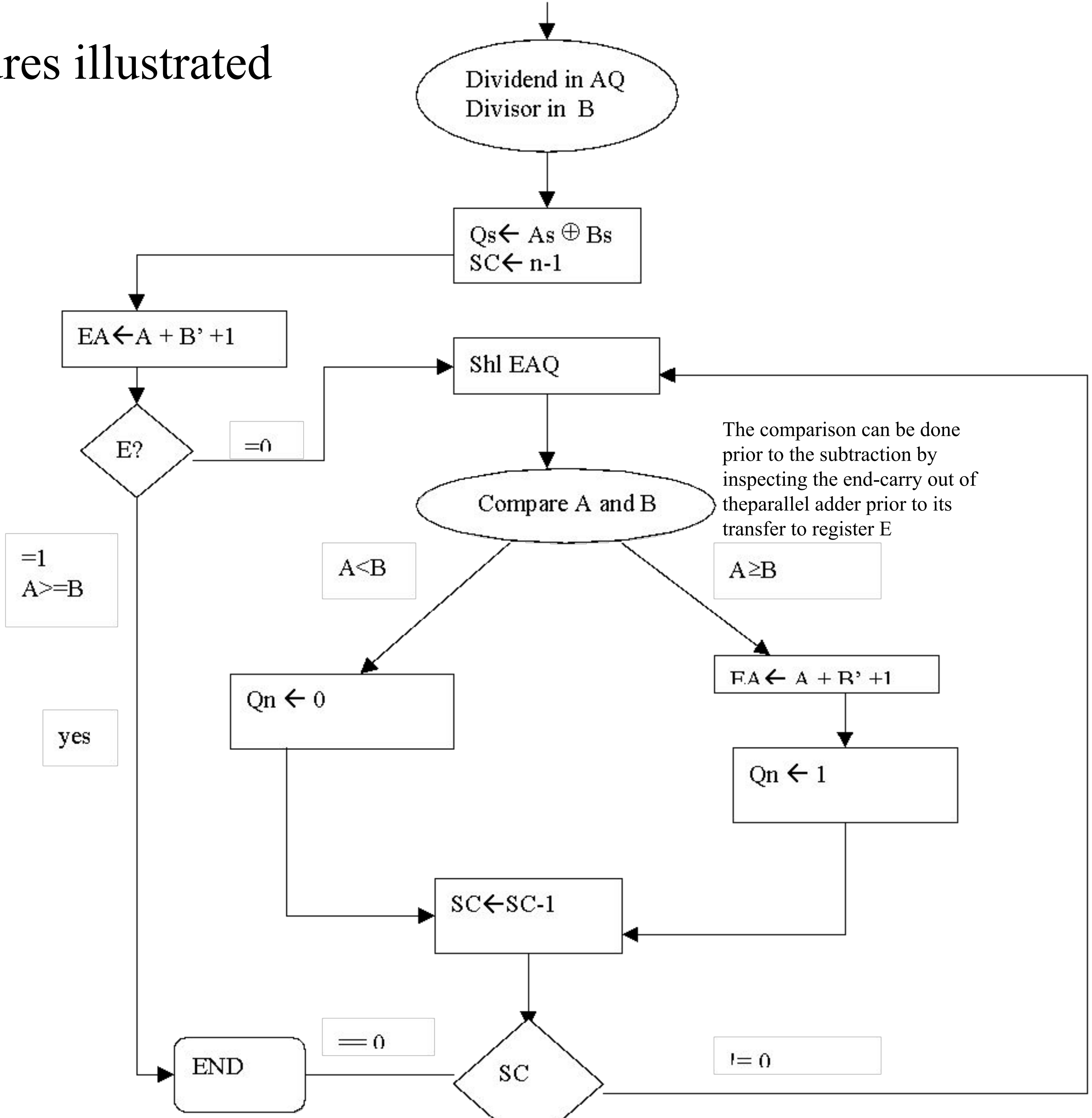
Divisor: 11 (3)

Quotient: 10011 (19)

Remainder: 10 (2)

```
      10011 r 10
    _______
 11)111011
    -11
     ______
      101
      -11
     ______
       101
        11
     ______
        10
```

Procedures illustrated



The comparison can be done prior to the subtraction by inspecting the end-carry out of theparallel adder prior to its transfer to register E

An example

Dividend A: 11011(27)

Divisor B: 101 (5), B’+1=011

	E	A	Q	SC	
Dividend:			011	011	3
Check overflow—no overflow since A<B					
<u>Start division</u>					
Shl EAQ		0	110	110	3
Compare A,B(A>B)		1			
Add B’+1			011		
Q_3=1		-----			
		0 01	110	2	
Shl EAQ		0	011	100	2
Compare A,B(A<B)		0			
Q_2=0					
Shl EAQ		0	111	000	1
Compare A,B (A>B)		1			
Add B’+1			011		
Q_1=1		010	000	0	

Reminder 010 (2)

Quotient:101(5)

Division algorithms

Restoring method
Comparing method
Non-restoring method

They use the same principle, only slightly differs in the implementation.

Restoring method: in dividing magnitudes: First, subtraction to find out if $A \geq B$ or $A<B$, if $A+ B'+1 =1$, then it denotes that $A \geq B$, otherwise, $A<B$. If $A<B$, then we need to add the B back to restore original A before we could do the shl EAQ operation.

Comparing method: No subtraction before determination of A vs B

Non-restoring method: B is not added if the difference is negative, rather, the negative difference is shifted left and then B is added.

Overflow protection

The division operation may result in a quotient with an overflow.

Example: If length (A) = 2 length (B)

$$\begin{tikzcd}[column sep=small] & \text{(A)} \arrow[d] & \text{(B)} \arrow[d] \\ \text{registers} & \text{AQ} & \text{B} \end{tikzcd}$$

Register A holds high-order half bits of the dividend A Check if the number in register A is >= B, □ overflow

Another benefit of this: avoid division by zero.

In implementation, the overflow detection mechanism is set up by a special flip-flop (divide-overflow flip-flop, labled as DVF).