# Explain the procedure for Addition and Subtraction with signed-magnitude data with the help of flowchart.

* + The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate.

If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.

* + For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.
  + The magnitudes are added with a microoperation EA  A + B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.
  + The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
  + 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.
  + 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A A' +1.
  + However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
  + In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.
  + The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.
  + Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.
  + It consists of registers A and B and sign flip-flops As and Bs.
  + Subtraction is done by adding A to the 2's complement of B.
  + The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.
  + The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.
  + The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

A < B



Subtract Operation

Addition Operation

Minuend in A

Augend in A

= 0

= 1

= 1

= 0

As ⊕ Bs

As ⊕ Bs

EA ← A+ B’ + 1

EA ← A+ B

As ≠ Bs

As ≠ Bs

As = Bs

= 0

= 1

E

A ≥ B

≠ 0

= 0

A

END

AS ← 0

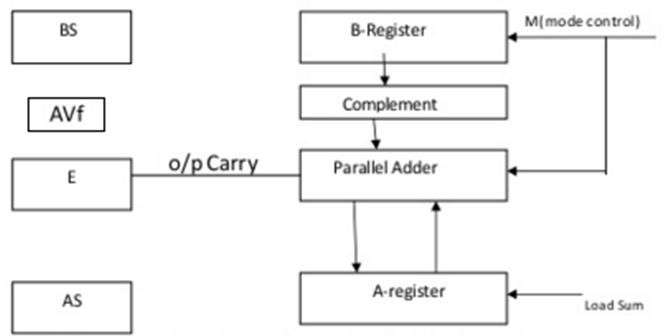
A ← A + 1

As ← As’

A ← A’

AVF ← E

**Figure 7.1: Flowchart for add and subtract operations.**



**Figure 7.2: Hardware for signed-magnitude addition and subtraction**

# Explain the Booth’s algorithm with the help of flowchart.

(Sum ’14,Win ’14, win’15)

* + Booth algorithm gives a procedure for multiplying binary integers in signed- 2’s

complement representation.

* + It operates on the fact that strings of 0’s in the multiplier require no addition but just shifting, and a string of 1’s in the multiplier from bit weight 2k to weight 2m can be treated as 2k+1 – 2m.
  + For example, the binary number 001110 (+14) has a string 1’s from 23 to 21 (k=3, m=1). The number can be represented as 2k+1 – 2m. = 24 – 21 = 16 – 2 = 14. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 24 – M X 21.
  + Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.



Multiply

Multiplicand in BR Multiplier in QR

AC ← 0

Qn + 1 ← 0



= 10

= 01

QnQn+1

= 00

ashr (AC & QR)

≠ 0

= 0

SC

END

AC ← AC + BR

AC ← AC + BR’ + 1

**Figure 7.3: Booth algorithm for multiplication of signed-2's complement numbers**

* + As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.
  + Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the

first least significant 1 in a string of 1’s in the multiplier.

1. The multiplicand is added to the partial product upon encountering the first 0 in a

string of 0’s in the multiplier.

1. The partial product does not change when multiplier bit is identical to the previous multiplier bit.
   * The algorithm works for positive or negative multipliers in 2’s complement

representation.

* + This is because a negative multiplier ends with a string of 1’s and the last operation will be a subtraction of the appropriate weight.
  + The two bits of the multiplier in Qn and Qn+1 are inspected.
  + If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
  + If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
  + When the two bits are equal, the partial product does not change.

# Explain with proper block diagram the Multiplication Operation on two floating point numbers.

* + The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents.
  + No comparison of exponents or alignment of mantissas is necessary.
  + The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product.
  + The double-precision answer is used in fixed-point numbers to increase the accuracy of the product.
  + In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained.
  + Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product.
  + The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.
   * The flowchart for floating-point multiplication is shown in Figure 7.4. The two operands are checked to determine if they contain a zero.
   * If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated.
   * If neither of the operands is equal to zero, the process continues with the exponent addition.
   * The exponent of the multiplier is in q and the adder is between exponents a and b.
   * It is necessary to transfer the exponents from q to a, add the two exponents, and transfer the sum into a.



Multiplicand in BR

Multiplier in QR

= 0

BR

≠ 0

QR

≠ 0

= 0

shl AQ

a ← a - 1

A1

= 1

END

(product is in AC)

Multiply mantissa

a ← a - bias

a ← a + b

a ← q

AC ← 0

= 0

**Figure 7.4: Multiplication of floating-point numbers**

* + Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias.
  + The correct biased exponent for the product is obtained by subtracting the bias number from the sum.
  + The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and Q.
  + Overflow cannot occur during multiplication, so there is no need to check for it.
  + The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized.
  + If it is a 0, the mantissa in AQ is shifted left and the exponent decremented.
  + Note that only one normalization shift is necessary. The multiplier and multiplicand were originally normalized and contained fractions. The smallest normalized operand is 0.1, so the smallest possible product is 0.01.
  + Therefore, only one leading zero may occur.
  + Although the low-order half of the mantissa is in Q, we do not use it for the floating- point product. Only the value in the AC is taken as the product.

# Multiply the (-9) with (-13) using Booth’s algorithm. Give each

**step.** (Sum’14)

* + A numerical example of booth algorithm is shown for n=5. It shows the step-by-step multiplication of (-9) X (-13) = +117.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 9:  1’s complement of 9:  2’s complement of 9: | | | 01001  10110  + 1 | | | 13: 01101  1’s complement of 13: 10010  + 1  2’s complement of 13: 10011 (-13) | |
| 10111 (-9) | | |
| **AC** | **QR(-13)** | **Qn+1** | | **M(BR)(-9)** | **SC** | | **Comments** |
| 00000 | 10011 | 0 | | 10111 | 5 | | Initial value |
| 01001 | 10011 | 0 | | 10111 | 4 | | Subtraction: AC=AC+BR’+1 |
| 00100 | 11001 | 1 | | 10111 | Arithmetic Shift Right |
| 00010 | 01100 | 1 | | 10111 | 3 | | Arithmetic Shift Right |
| 11001 | 01100 | 1 | | 10111 | 2 | | Subtraction: AC=AC+BR’+1 |
| 11100 | 10110 | 0 | | 10111 | Arithmetic Shift Right |
| 11110 | 01011 | 0 | | 10111 | 1 | | Arithmetic Shift Right |
| 00111 | 01011 | 0 | | 10111 | 0 | | Subtraction: AC=AC+BR’+1 |
| **00011** | **10101** | 1 | | 10111 | Arithmetic Shift Right |

Answer: -9 X -13 =117 => 001110101

# Multiply the (7) with (3) using Booth’s algorithm. Give each

**step.**

7: 0111 3: 0011

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **AC** | **QR(3)** | **Qn+1** | **M(BR)(7)** | **SC** | **Comments** |
| 0000 | 0011 | 0 | 0111 | 4 | Initial value |
| 1001 | 0011 | 0 | 0111 | 3 | Subtraction: AC=AC+BR’+1 |
| 1100 | 1001 | 1 | 0111 | Arithmetic Shift Right |
| 1110 | 0100 | 1 | 0111 | 2 | Arithmetic Shift Right |
| 0101 | 0100 | 1 | 0111 | 1 | Addition: AC=AC+BR |
| 0010 | 1010 | 0 | 0111 | Arithmetic Shift Right |
| **0001** | **0101** | 0 | 0111 | 0 | Arithmetic Shift Right |

Answer: 7 X 3 =21 => 00010101

# Multiply the (15) with (13) using Booth’s algorithm. Give each

**step.**

15X13=195

15: 01111

13: 01101

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **AC** | **QR(15)** | **Qn+1** | **M(BR)(13)** | **SC** | **Comments** |
| 00000 | 01111 | 0 | 01101 | 5 | Initial value |
| 10011 | 01111 | 0 | 01101 | 4 | Subtraction: AC=AC+BR’+1 |
| 11001 | 10111 | 1 | 01101 | Arithmetic Shift Right |
| 11100 | 11011 | 1 | 01101 | 3 | Arithmetic Shift Right |
| 11110 | 01101 | 1 | 01101 | 2 | Arithmetic Shift Right |
| 11111 | 00110 | 1 | 01101 | 1 | Arithmetic Shift Right |
| 01100 | 00110 | 1 | 01101 | 0 | Addition: AC=AC+BR |
| **00110** | **00011** | 1 | 01101 | Arithmetic Shift Right |

Answer: 15X13=195 => 0011000011

# Multiply the (+15) with (-13) using Booth’s algorithm. Give each

**step.**

|  |  |  |  |
| --- | --- | --- | --- |
| 15: | 01111 | 13: | 01101 |
|  |  | 1’s complement of 13: | 10010 |

+ 1

2’s complement of 13: 10011 (-13)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **AC** | **QR(-13)** | **Qn+1** | **M(BR)(+15)** | **SC** | **Comments** |
| 00000 | 10011 | 0 | 01111 | 5 | Initial value |
| 10001 | 10011 | 0 | 01111 | 4 | Subtraction: AC=AC+BR’+1 |
| 11000 | 11001 | 1 | 01111 | Arithmetic Shift Right |
| 11100 | 01100 | 1 | 01111 | 3 | Arithmetic Shift Right |
| 01011 | 01100 | 1 | 01111 | 2 | Addition: AC=AC+BR |
| 00101 | 10110 | 0 | 01111 | Arithmetic Shift Right |
| 00010 | 11011 | 0 | 01111 | 1 | Arithmetic Shift Right |
| 10011 | 11011 | 0 | 01111 | 0 | Subtraction: AC=AC+BR’+1 |
| **11001** | **11101** | 1 | 01111 | Arithmetic Shift Right |

Answer: (+15) X (-13) = -195 => **1100111101**

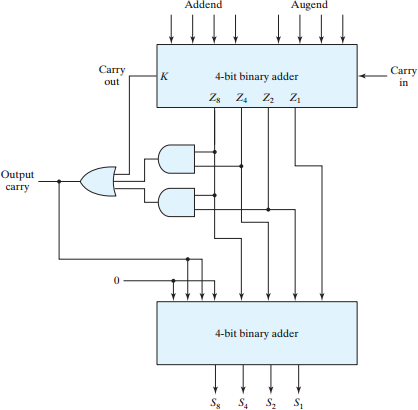
To verify 0011000010

+ 1

+195=> 0011000011

# Explain BCD adder with its block diagram. (Win’14)

* + BCD representation is a class of binary encodings of decimal numbers where each decimal digit is represented by a fixed number of bits.
  + BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit in BCD form.



**Figure 7.5: BCD Adder**

* + Since each input digit does not exceed 9, the output sum cannot be greater than 19(9+9+1). For example: suppose we apply two BCD digits to 4-bit binary adder.
  + The adder will form the sum in binary and produce a result that may range from 0 to 19.
  + In following figure 7.5, these binary numbers are represented by K, Z8, Z4, Z2, and Z1.
  + K is the carry and subscripts under the Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.
  + When binary sum is equal to or less than or equal to 9, corresponding BCD number is identical and therefore no conversion is needed.
  + The condition for correction and output carry can be expressed by the Boolean function:

C= K + Z8Z4 + Z8 Z2

* + When it is greater than 9, we obtain non valid BCD representation, then additional binary 6 to binary sum converts it to correct BCD representation.
  + The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum.
  + When C is equal to 1, binary 0110 is added to the binary sum using bottom 4-bit binary adder. The output carry generated from the bottom binary-adder may be ignored.