UNIT - II
ARITHMETIC

2.1 INTRODUCTION

Data is manipulated by using the arithmetic instructions in digital computers to give solution for the computation problems. The addition, subtraction, multiplication and division are the four basic arithmetic operations. **Arithmetic processing unit** is responsible for executing these operations and it is located in central processing unit.

The arithmetic instructions are performed on binary or decimal data**. Fixed-point numbers** are used to represent integers or fractions. These numbers can be signed or unsigned negative numbers. A wide range of arithmetic operations can be derived from the basic operations.

Signed and Unsigned Numbers:

Signed numbers:

These numbers require an arithmetic sign. The most significant bit of a binary number is used to represent the sign bit. If the sign bit is equal to zero, the signed binary number is positive; otherwise, it is negative. The remaining bits represent the actual number. The negative numbers may be represented either in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point

- Binary numbers signed magnitude
- Signed な's complement
- Signed |-|'s complement

Unsigned binary numbers:

These are positive numbers and thus do not require an arithmetic sign. An m-bit unsigned number represents all numbers in the range 0 to 2^m ¦ 1. For example, the range of 16-bit unsigned binary numbers is from 0 to 65,53510 in decimal and from 0000 to FFFF16 in hexadecimal.

Signed Magnitude Representation:

The most significant bit (MSB) represents the *sign*. A 1 in the MSB bit position denotes a negative number and 0 denotes a positive number. The remaining *n* · な bits are preserved and represent the magnitude of the number.

Examples:

Number	Signed Magnitude Representation
+3	0011
-3	1011
0	0000
-0	1011
5	0101
-5	1101

One's Complement Representation:

)n one|s complement, positive numbers remain unchanged as before with the sign- magnitude numbers. Negative numbers are represented by taking the one|s complement (inversion, negation) of the unsigned positive number. Since positive numbers always start with a 0, the complement will always start with a 1 to indicate a negative number.

The one|s complement of a negative binary number is the complement of its positive counterpart, so to take the one|s complement of a binary number.

Number	One's complement Representation
00001000 (+8)	11110111
10001000(-8)	01110111
00001100(+12)	11110011
10001100(-12)	01110011

Two's Complement Representation:

)n two|s complement, the positive numbers are exactly the same as before for unsigned binary numbers. A negative number, is represented by a binary number, which when added to its corresponding positive equivalent results in zero.

2.4 Arithmetic

)n two|s complement form, a negative number is the |-|s complement of its positive number with the subtraction of two numbers being A - B = A + ϕ |-|s complement of B ɨ using much the same process as before as basically, two|s complement is adding な to one|s complement of the number.

The main difference between 12 s complement and 22 s complement is that 12 s complement has two representations of 0 (+0): 00000000, and (-0): 11111111. In 22 s complement, there is only one representation for zero: 00000000 (0).

+0: 00000000

|-|s complement of -0:

-0: 00000000 (Signed magnitude representation)

ななななななな ϕな|s complement representationɨ

ななななななな + な= どどどどどどどど ϕ|-|s complement representationɨ

These shows in |-|s complement representation both +ど and -0 takes same value. This solves the **double-zero problem**, which existed in the な|s complement.

Example 2.1: Convert 2_{10} and -2_{10} to 32 bit binary numbers.

+2= **o**000 0000 0000 0010 (16 bits)

= 0000 0000 0000 0000 0000 0000 0000 0010 (32 bits)

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word.

2=0000 0000 00000010

-|-=な|s complement of |- +な

1111 1111 1111 11どな ϕな|s complement of |-ɨ + 1

= **1**111 1111 1111 1110 (16 bits)

= 1111 1111 1111 1111 1111 1111 1111 1110 (32 bits)

To convert to 32 bit number copy the digit in the MSB of the 16 bit number for 16 times and fill the left half.

2.2 FIXED POINT ARITHMETIC

A fixed-point number representation is a real data type for a number that has a fixed number of digits after the radix point or decimal point.

This is a common method of integer representation is sign and magnitude representation. One bit is used for denoting the sign and the remaining bits denote the magnitude. With 7 bits reserved for the magnitude, the largest and smallest numbers represented are +127 and –127. Fixed-point numbers are useful for representing fractional values, usually in base 2 or base 10, when the executing processor has no floating point unit (FPU) or if fixed-point provides improved performance or accuracy for the application at hand. Most low-cost embedded microprocessors and microcontrollers do not have an FPU.

A value of a fixed-point data type is essentially an integer that is scaled by a specific factor. The scaling factor is usually a power of 10 (for human convenience) or a power of 2 (for computational efficiency). However, other scaling factors may be used occasionally, e.g. a time value in hours may be represented as a fixed-point type with a scale factor of 1/3600 to obtain values with one-second accuracy. The maximum value of a fixed-point type is the largest value that can be represented in the underlying integer type, multiplied by the scaling factor; and similarly for the minimum value.

Example:

The value 1.23 can be represented as 1230 in a fixed-point data type with scaling factor of 1/1000.

Precision loss and overflow

- The fixed point operations can produce results that have more bits than the operands there is possibility for information loss.
- In order to fit the result into the same number of bits as the operands, the answer must be rounded or truncated.
- Fractional bits lost below this value represent a precision loss which is common in fractional multiplication.
- If any integer bits are lost, however, the value will be radically inaccurate.
- Some operations, like divide, often have built-in result limiting so that any positive overflow results in the largest possible number that can be represented by the current format.

- Likewise, negative overflow results in the largest negative number represented by the current format. This built in limiting is often referred to as **saturation.**
- Some processors support a hardware overflow flag that can generate an exception on the occurrence of an overflow, but it is usually too late to salvage the proper result at this point.

2.2.1 Addition and Subtraction

In addition, the digits are added bit by bit from right to left, with carries passed to the next digit to the left. Subtraction operation is also done using addition: The appropriate operand is simply negated before being added.

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

Operation	Add Magnitude	Subtract Magnitude		
		When A>B	When A<B	When A=B
(+A) + (+B)	+(A + B)			
(+A) + (- B)		+(A - B)	- (B - A)	+(A - B)
(- A) + (+B)		- (A - B)	+(B - A)	+(A - B)
(- A) + (- B)	- (A + B)			
(+A) - (+B)		+(A - B)	- (B - A)	+(A - B)
(+A) - (- B)	+(A + B)			
(- A) - (+B)	- (A + B)			
(- A) - (- B)		- (A - B)	+(B - A)	+(A - B)

Fig 2.1: Addition and Subtraction operation

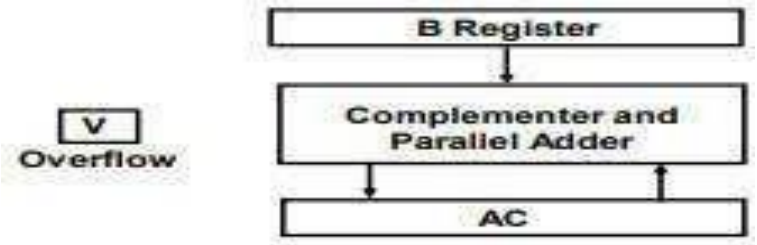
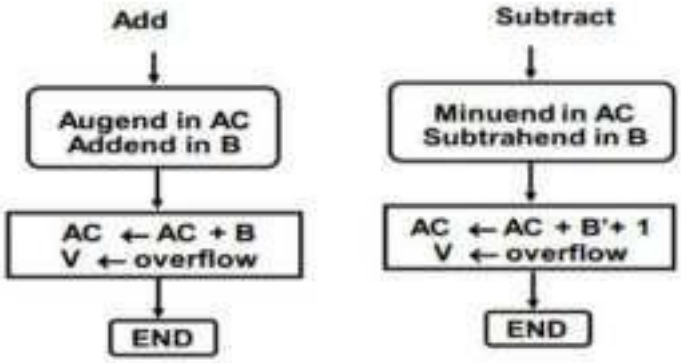


Fig 2.2: Hardware for addition / subtraction



a) Addition **b) Subtraction**

Fig 2.2: Addition and subtraction algorithm

Steps for addition:

- Place the addend in register B and augend in AC.
- Add the contents in B and AC and place the result in AC.
- V register will hold the overflow bits (if any).

Steps for subtraction:

- Place the minuend in AC and subtrahend in B.
- Add the contents of AC and its complemented B. Place the result in AC.
- V register will hold the overflow bits (if any).

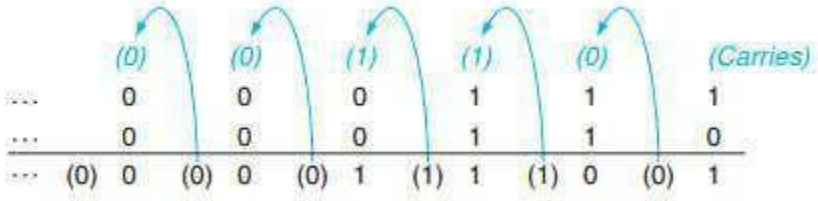


Fig 2.3: Manipulating carry

The figure 2.3 shows binary addition with carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry. If there is a carry at this bit, it will be stored in the overflow register.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means borrow occurred from the sign bit.

Operation	Operand A	Operand B	Result indicating overflow
A+B	>=0	>=0	<0
A+B	<0	<0	>=0
A-B	>=0	<0	<0
A-B	<0	>=0	>=0

Example 2.2: Add 6 and 7.

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten} \end{array}$$

Example 2.3: Subtract 6 from 7.

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ - & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

Example 2.4: Subtract ‖ from ‖ through ‖s complement.

$$\begin{array}{ll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\ + & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten} \end{array}$$

The MIPS instructions for addition and subtraction are given in the following table:

Instruction	Example	Operation
Add	Add $s1, $s2, $s3	S1=s2+s3Overflow detected
Subtract	Sub $s1, $s2, $s3	S1=s2-s3Overflow detected
Add Immediate	Addi $s1, $s2, 100	S1=s2+100Overflow detected
Add unsigned	Addu $s1, $s2, $s3	S1=s2+s3Overflow undetected
Subtract unsigned	Subu $s1, $s2, $s3	S1=s2-s3Overflow undetected
Add immediate unsigned	Addiu $s1, $s2, 100	S1=s2+100Overflow undetected

2.2.2 Multiplication

Multiplication is seen as repeated addition. The first operand is called the multiplicand and the second the multiplier. The final result is called the product. The number of digits in the product is larger than the number in either the multiplicand or the multiplier. The length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n + m bits long. The steps in multiplication are:

- Place a copy of the in the proper place if the multiplier digit is a 1
- Place 0 in the proper place if the digit is 0.

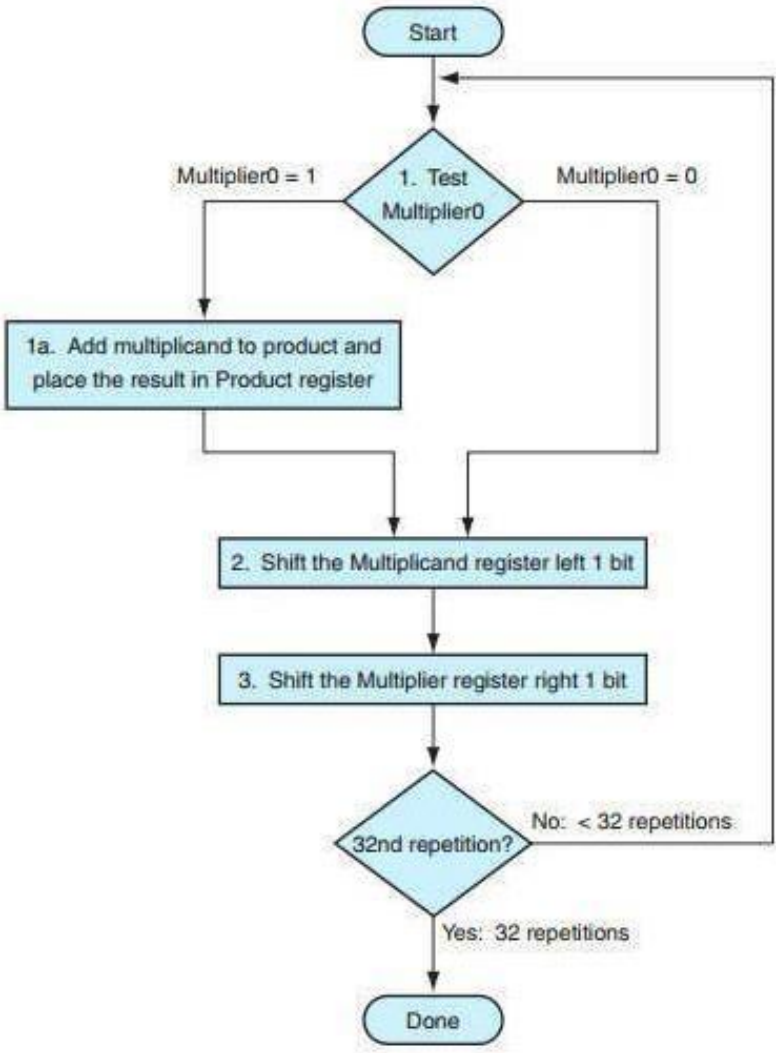


Fig 2.4: Basic multiplication algorithm

Booth's Algorithm:

Booth algorithm gives a procedure for multiplying binary integers in signed- に┤s complement represe ntation.)t operates on the fact that strings of ど┤s in the multiplier require no addition but just shifting, and a string of な┤s in the multiplier from bit weight に^{k} to weight 2^{m} can be treated as $2^{k+1}-2^{m}$.

For example, the binary number 001110 (+14) has a string な┤s from にぬ to にな ゅk=ぬ, m=1). The number can be represented as $2^{k+1}-2^{m} = 24 - 21 = 16 - 2 = 14$. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^{4} - M \times 2^{1}$. Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

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 な in a string of な┤s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 in a string of ど┤s in the multiplier.
3. 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 に┤s complement representation. This is because a negative multiplier ends with a string of な┤s and the last operation will be a subtraction of the appropriate weight. The two bits of the multiplier in on 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.)f the two bits are equal to どな, it means that the first ど in a string of ど┤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.

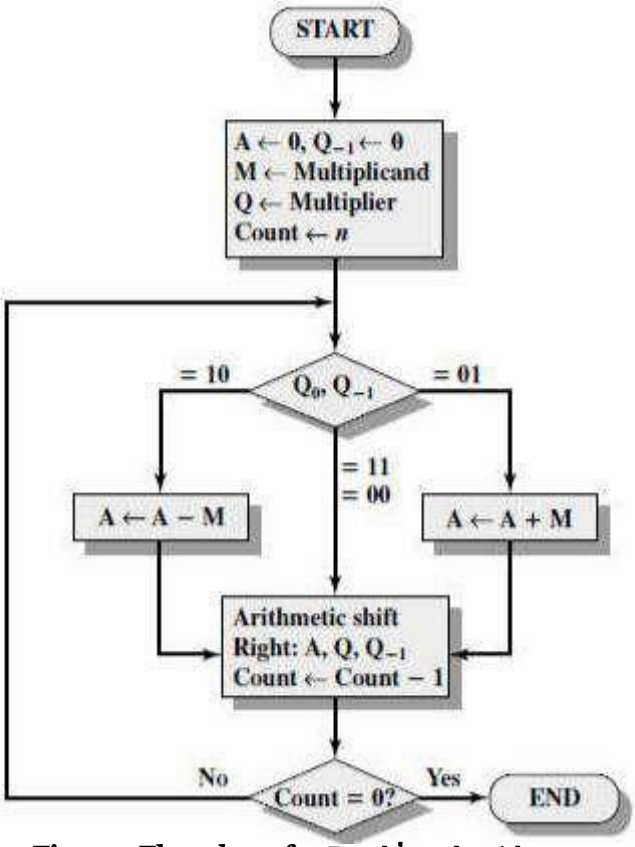


Fig 2.5: Flowchart for Booth's algorithm

Example 2.5: Multiply 7 and 3 using 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	First cycle
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	Third cycle
0001	0101	0	0111	Shift	Fourth cycle

The product is available in AQ.

Example 2.6 : Multiply -5 and -7 using Booth's algorithm

A	Q	Q-1	M
0000	1001	0	4
0101	1001	0	
0010	1100	1	3
1101	1100	1	
1110	1110	0	2
1111	0111	0	1
0010	0011	1	0

The product is available in AQ

2.2.3 Division

Division is repeated subtraction. The two operands (dividend and divisor) and the result (quotient) of divide are accompanied by a second result called the remainder. The following are the terminologies:

- Dividend: A number being divided.
- Divisor: A number that the dividend is divided by.
- Quotient: The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.
- Remainder: The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend

Dividend = Quotient * Divisor + Remainder

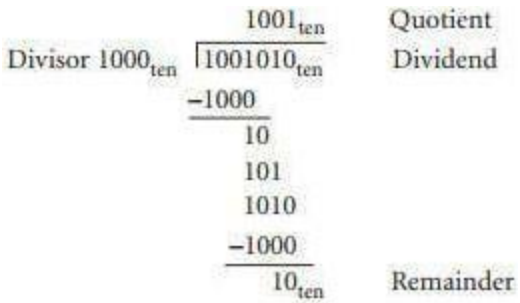


Fig 2.6: Division Terminologies

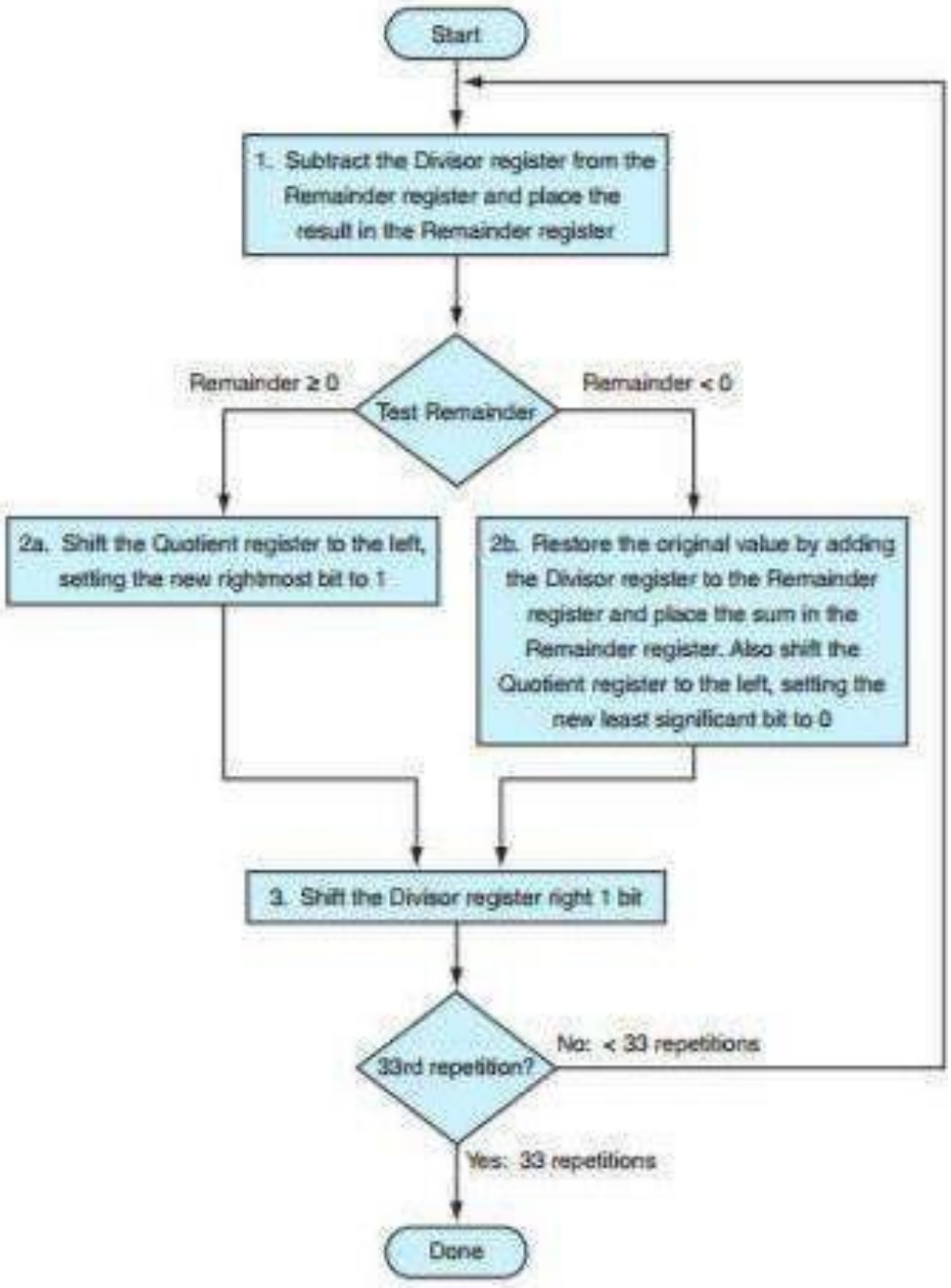


Fig 2.7: Basic division operation

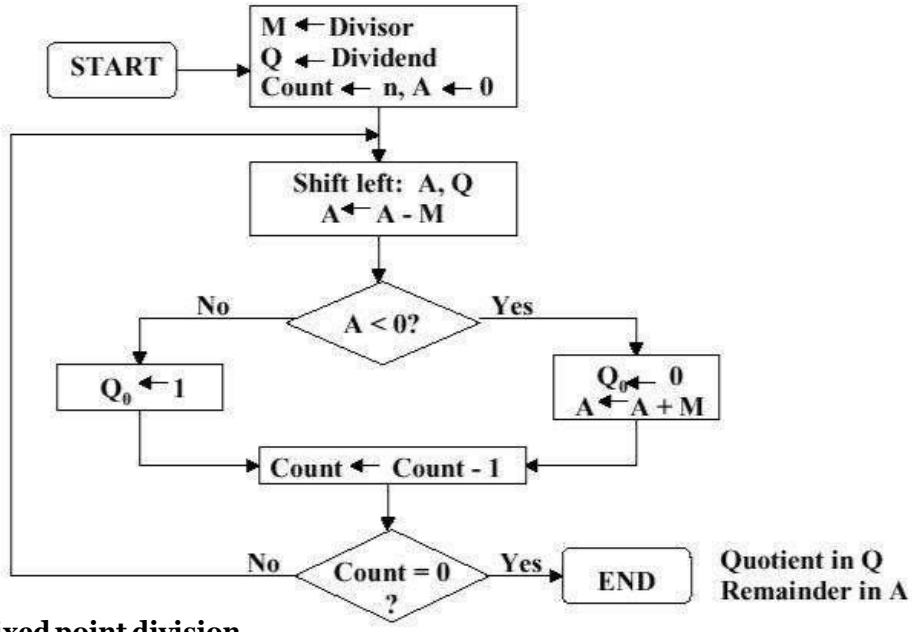


Fig 2.8: Fixed point division

Example 2.7: Divide -7 by 3

A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101		A = A - M	
0000	1110	A = A + M	
0001	1100	Shift	2
1110		A = A - M	
0001	1100	A = A + M	
0011	1000	Shift	3
0000		A = A - M	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		A = A - M	
0001	0010	A = A + M	

Quotient=0010Remainder=0001

Example 2.8: Divide -7 by -3

A	Q	M = 1101	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Subtract	
1111	0010	Restore	
1110	0100	Shift	2
0001		Subtract	
1110	0100	Restore	
1100	1000	Shift	3
1111		Subtract	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Subtract	
1111	0010	Restore	

Example 2.9: Divide 7 by 3

A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101		Subtract	
0000	1110	Restore	
0001	1100	Shift	2
1110		Subtract	
0001	1100	Restore	
0011	1000	Shift	3
0000		Subtract	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		Subtract	
0001	0010	Restore	

Example 2.10: Divide -7 by 3

A	Q	M = 0011	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Add	
1111	0010	Restore	
1110	0100	Shift	2
0001		Add	
1110	0100	Restore	
1100	1000	Shift	3
1111		Add	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Add	
1111	0010	Restore	

MIPS instructions for multiplication and division

Category	Example	Description
Multiply	mult $s2, $s3	Hi, lo=s2 * s3 64 bit signed product in Hi, Lo
Multiply unsigned	multu $s2, $s3	Hi, lo=s2 * s3 64 bit signed product in Hi, Lo
Divide	div $s2, $s3	Lo=s2/s3 (Quotient) Hi=s2 mod s3 (Remainder)
Divide unsigned	divu $s2, $s3	Lo=s2/s3 (unsigned Quotient) Hi=s2 mod s3 (Remainder)
Move from Hi	mfhi $s1	S1=Hi Used to get a copy of Hi
Move from Lo	mflo $s1	S1=lo Used to get a copy of Lo

2.3 FLOATING POINT ARITHMETIC

To represent the fractional binary numbers (IEEE 754 floating point format), it is necessary to consider floating point. If the point is assumed to the right of the sign bit, we can represent the fractional binary numbers as given below:

$$B = (b_0 * 2^0 + b_{-1} * 2^{-1} + b_{-2} * 2^{-2} + \ldots + b_{-(p-1)} * 2^{-(p-1)})$$

With this fractional number system, we can represent the fractional numbers in the following range,

$$-1 < F < 1 - 2^{-(p-1)}$$

The binary point is said to be float and the numbers are called **floating point numbers**. The position of binary point in floating point numbers is variable and hence numbers must be represented in the specific manner is referred to as floating point representation. The floating point representation has three fields. They are:

- **Sign:** Sign bit is the first bit of the binary representation. ┼な┼ implies negative number and ┼ど┼ implies positive number.

  Example: 11000001110100000000000000000001. This is negative number since it starts with 1.

- **Exponent:** It starts from bit next to the sign bit of the binary representation. The exponent field is needed to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, to express an exponent of zero, 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200"127), or ばぬ. The exponents of ┼なにば ゅall どsよ and +なにぱ ゅall なs) are reserved for special numbers.

  Double precision has an 11-bit exponent field, with a bias of 1023. **Example:** For 8 bit conversion: $8 = 2^{3-1}-1=3$. Bias=3.

  For 32 bit conversion: $32=2^{8-1}-1= 127$. Bias=127.

- **Significant digits or Mantissa:** It is calculated from the remaining 23 bits of the binary representation.)t consists of ┼な┼ and a fractional part. This represents the

Precision bits of the number. It is composed of an implicit leading bit (left of the radix point) and the fraction bits (to the right of the radix point). To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways.

Example: 50 can be represented as

1. 0.050×10^3
2. $.5000 \times 10^3$

- 5.000×10^1
- 50.00×10^0
- $5000. \times 10^{-2}$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. In normalized form, 50 is represented as 5.000×101.

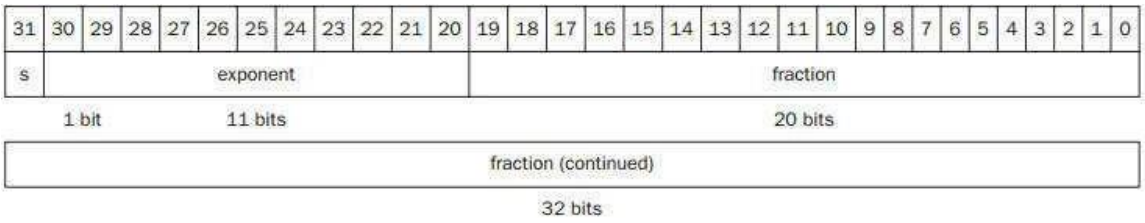


Fig 2.9: Parts of floating point number

Conversion of Decimal number to floating point:

- **Sign bit:** 1 implies negative number and 0 implies positive number.
- **Exponent:** To find the exponent value for binary representation, express the number by the nearest smaller or equal to 2^k number. The bias is determined by $2^{k-1}-1$, where k is the number of bits in exponent field. Add the bias with k value to express the exponent in binary form.
- **Mantissa:** Move the binary point so that there is only one bit from the left. Adjust the exponent of 2 so that the value does not change. This is normalizing the number. Now, consider the fractional part and represented as 23 bits by adding zeros.

Example 2.11. Find the decimal equivalent of the floating point number:

01000001110100000000000000000000

Sign=0

Exponent:

$10000011=131_{10}$

131-127=4

Exponent= 2^4=16

Mantissa:

Remaining 23 bits: 10100000000000000000000

=1*(1/2) + 0*(1/4) + 1*(1/8ょ + ど*ゅな/なはょ +…… = ど.はにの Decimal number= Sign * Exponent * Mantissa

=-1 * 16 *0.625 = -26

Example 2.11: Find the floating point equivalent of -17.

Sign=1 (-ve number)

Exponent:

Bias for 32 bit = 127 (2^{8-1} -1 = 127) 127 + 4 = 131=10000011_2

Mantissa:

17 = 10001_2=1.0001 x 2^4

Fractional part=00010000000000000000000 -17 =1 10000011

$00010000000000000000000_2$

Terminologies:

- **Overflow:** A situation in which a positive exponent becomes too large to fit in the exponent field.
- **Underflow:** A situation in which a negative exponent becomes too large to fit in the exponent field.
- **Double precision:** A floating point value represented in two 32-bit words.

- **Single precision:** A floating point value represented in a single 32-bit word.

	Sign	Exponent	Fraction
Single Precision	1 [31]	8 [30–23]	23 [22–00]
Double Precision	1 [63]	11 [62–52]	52 [51–00]

Fig 2.10: Floating point formats

Example 2.12: The IEEE-754 32-bit floating-point representation pattern is 0 10000000 110 0000 0000 0000 0000 0000. What is the number?

Sign bit S = 0 (positive number)

Exponent E = 10000000_2 = 128_{10} (in normalized form)

Fraction is 1.11_2 (with an implicit leading 1) = $1 + 1\times2^{-1} + 1\times2^{-2} = 1.75_{10}$

The number is $+1.75 \times 2^{(128-127)} = +3.5_{10}$

Example 2.13: Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 100 0000 0000 0000 0000 0000. Find the decimal number.

Sign bit S = 1 (negative number)

E = 0111 1110_2 = 126_{10} (in normalized form)

Fraction is 1.1_2 (with an implicit leading 1) = $1 + 2^{-1} = 1.5_{10}$

The number is -1.5 × 2^ (126-127) = -0.75D

Example 2.14: Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 000 0000 0000 0000 0000 0001. What is the decimal number?

Sign bit S = 1 (negative number) E = 0111 1110_2 = 126_{10} (in normalized form) Fraction is 1.000 0000 0000 0000 0000 0001B (with an implicit leading 1) = $1 + 2^{-23}$

The number is - $(1 + 2^{-23}) \times 2^{(126-127)}$ = -0.50000005960464477539062 5

Example 2.15: Express 85.125 in single and double precision.

85 = 1010101

0.125 = 001

85.125 = 1010101.001

=1.010101001 x 2^6

Sign = 0

1. **Single precision:**

Biased exponent 127+6=133

133 = 10000101

Normalized mantisa = 010101001

The IEEE 754 Single precision = 0 10000101 01010100100000000000000

2. **Double precision:**

Biased exponent 1023+6=1029

1029 = 10000000101

Normalized mantisa = 010101001

The IEEE 754 Double precision=

0 10000000101 0101010010000000000000000000000000000000000000000000000

2.3.1 Floating point addition and subtraction

Floating-point numbers are coded as sign/magnitude, reversing the sign-bit inverses the sign. Consequently the same operator performs as well addition or subtraction according to the two operands signs. The steps in floating point addition are:

- Rewrite the smaller number such that its exponent matches with the exponent of the larger number.
- Add the mantissas
- Renormalize the mantissa by shifting mantissa and adjusting the exponent.
- Check for overflow/underflow of the exponent after normalization.
- If the mantissa does not fit in the space reserved for it, it has to be rounded off.

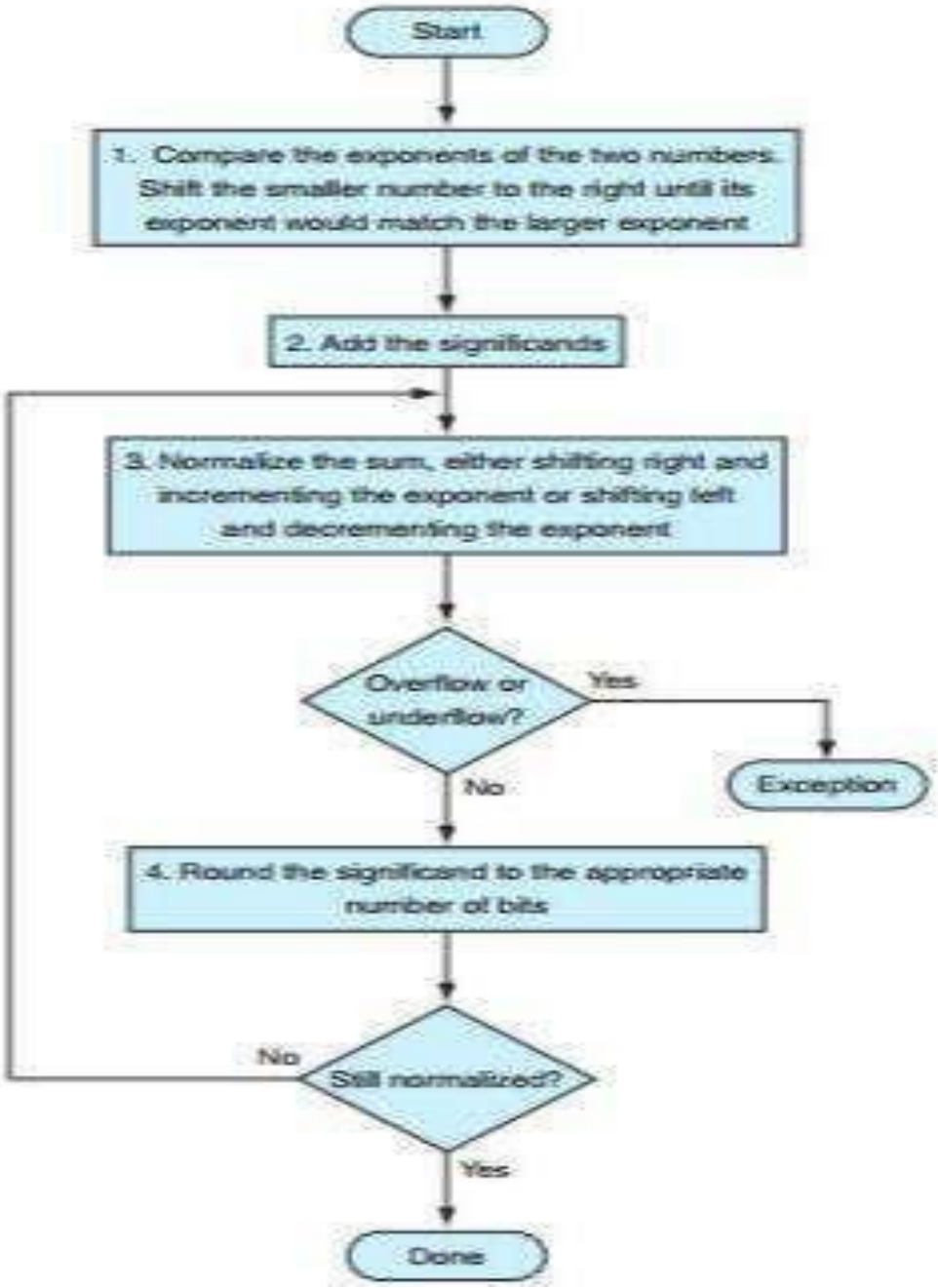


Fig 2.11: Flowchart for floating point addition / subtraction

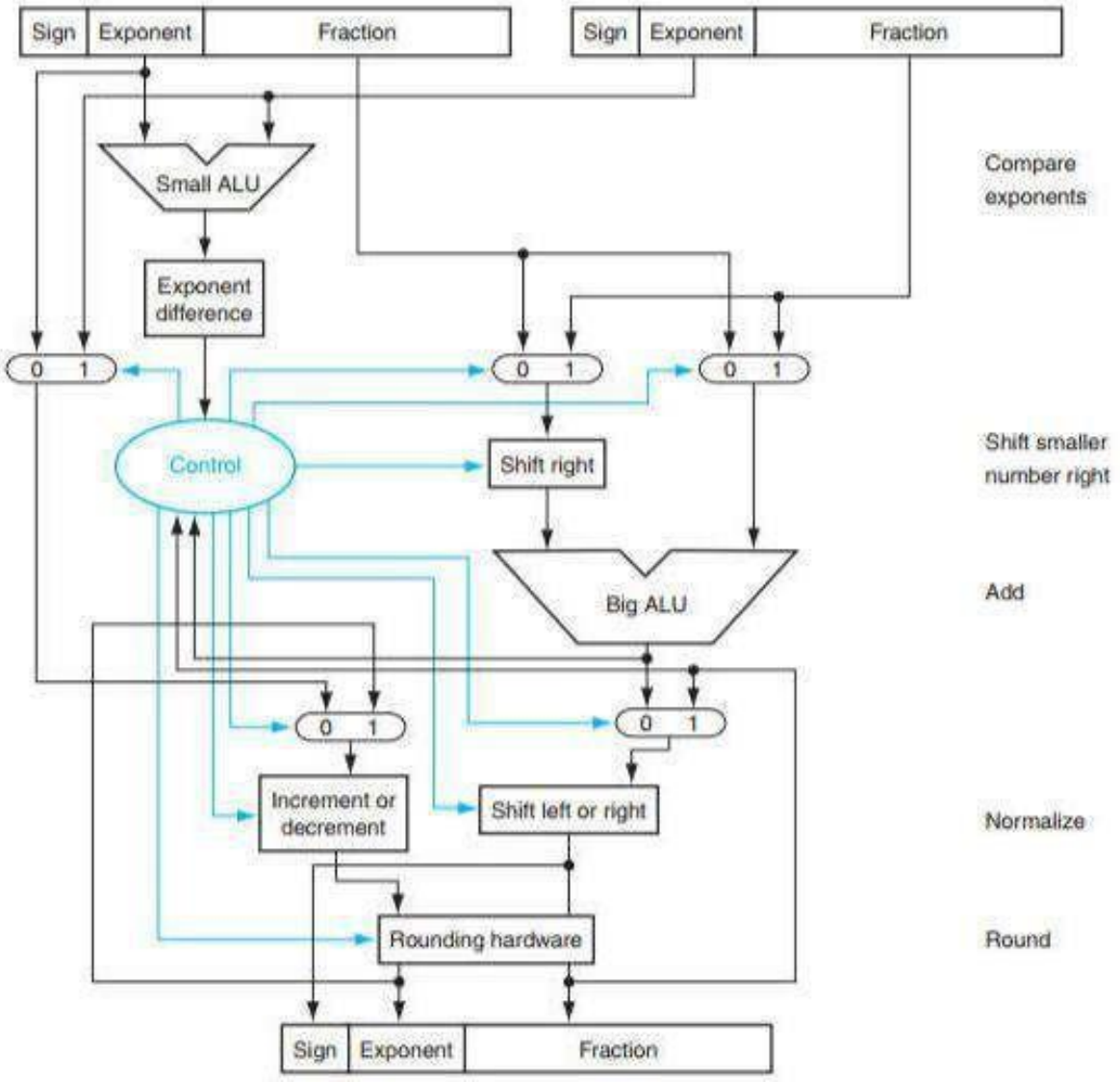


Fig 2.12: Hardware for floating point addition

The addition operation proceeds as the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significant of the smaller number, and the significant of the larger number. The smaller significant is shifted right, and then the significant are added together using the big ALU.

The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the final result.

Example 2.16: Add 0.5 + (-0.4375)

0.5 = 0.1 × 20 = 1.000 × 2-1 (normalized)

-0.4375 = -0.0111 × 20 = -1.110 × 2-2 (normalized)

Step 1: Rewrite the smaller number such that its exponent matches with the exponent of the larger number.

$-1.110 \times 2^{-2} = -0.1110 \times 2^{-1}$

Step 2: Add the mantissas

$$\begin{array}{l} 1.000 \times 2^{-1} + \\ \underline{-0.1110 \times 2^{-1}} \\ 0.001 \times 2^{-1} \end{array}$$

Step 3: Renormalize the mantissa by shifting mantissa and adjusting the exponent. s$0.001 \times 2^{-1} = 1.000 \times 2^{-4}$

-126 <= -4 <= 127 (-4 is within the range of -126 and 127). No overflow or underflow

Step 4: The sum fits in 4 bits so rounding is not required

Example 2.17: Express the following numbers in IEEE 754 format and find their sum:

2345.125 and 0.75. Single precision format of 2345.125:

0	10001010	00100101001001000000000

Single precision format of 0.75:

0	01111110	10000000000000000000000

Exponent of 2345.125 > exponent of 0.75 10001010-01111110=00000110 = $(12)_{10}$
Shift 0.75 to 12 positions right: 0.00000000000011000000000 Add:
1. 00100101001001000000000 (1 is added before . since this is a positive number)
+ 0.00000000000011000000000 (0 is added before . since it is a negative number)

1. 00100101001111000000000

The sum is normalized. There is no underflow. The final sum is

0	10001010	00100101001111000000000

The result is +ve hence 0 is filled in the sign field. The exponent value of 2345.125 is copied in the exponent field of the result, since the 0.75 is adjusted to the exponent of 2345.125.

Example 2.18: Subtract -1.00000000000000010011010x2^{-1} from

1.00000000101100010001101x2^{-6} .

+1.00000000101100010001101x2^{-6}

-1.00000000000000010011010x2^{-1}

Change the +1.00000000101100010001101x2^{-6} into power of 2^{-6}.

0.00000100000000101100010001101x2^{-1}

To perform subtraction take 2's complement of -1.00000000000000010011010x2^{-1} which is 1 0.11111111111111101100110 x 2^{-1} (Here first 1 is the overflow bit).

Now add both numbers

0 0.00000100000000101100010001101x2^{-1}

1 0.11111111111111101100110 x 2^{-1}1

1.00001000000001000101010 01101x2^{-1}

2.3.2 Floating point multiplication

The following are the steps in floating point multiplication:

- Add the exponents
- Multiply the significant digits
- Normalize the product
- Round-off the product (if necessary)

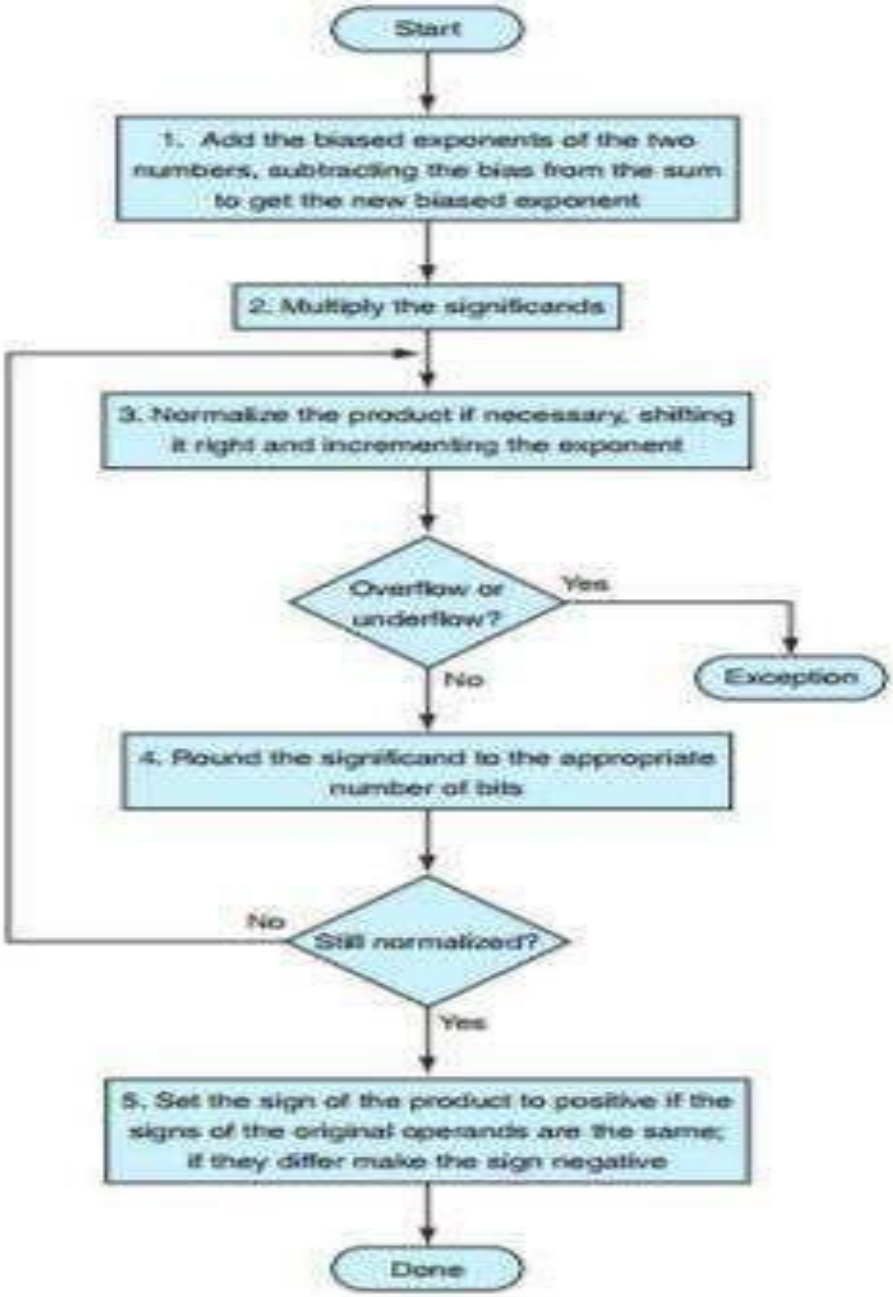


Fig 2.13: Flowchart for Floating point multiplication

Example 2.19: Multiply 1.110×10^{10} by 9.200×10^{-5}. Express the product in 3 decimal places.

1. Add the exponents

Exponent of the product=10-5=5

- Multiply the significant digits 1.110 x 9.200=10.212000
- Normalize the product

$10.212 \times 10^{5} = 1.0212 \times 10^{6}$

4. Round-off

$1.0212 \times 10^{6} = 1.021 \times 10^{6}$

Example 2.20: Perform binary multiplication on 0.5 and -0.4375.

$0.5 = 1.000 \times 2^{-1}$

$0.4375 = -1.110 \times 2^{-2}$1.

Add the exponents

Exponent of the product=-1+-2=-3

- Multiply the significant digits 1.000 x -1.110=-1.110
- Normalize the product

-1.110×10^{-3} is already normalized.

Example 2.21: Multiply -1.110 1000 0100 0000 10101 0001 x 2^{-4} and 1.100 0000 0001 0000 0000 0000 x 2^{-2}.

1. Add the exponents

Exponent of the product=-4 + -2=-6 2. Multiply the significant digits

-1.110 1000 0100 0000 10101 0001 x 1.100 0000 0001 0000 0000 0000

= 10.101110001111101111110011001010000100000000000

3. Normalize the product $1.010111000111110111111001100101000010000000000000 \times 2^{-5}$

4. Round-off (Only 23 fraction bits)

$1.01011100011111011111100 \times 2^{-5}$

2.3.3 MIPS floating point instructions

MIPS provide several instructions for floating point numbers for performing the following operations:

- Arithmetic
- Data movement (memory and registers)
- Conditional jumps

Floating Point (FP) instructions work with a different bank of registers. Registers are named f0 to $f31. MIPS floating-point registers are used in pairs for double precision numbers and referred using even numbers. Single precision numbers end with .s and double precision numbers end with .d.

Category	Example	Description
FP add single	add.s $f2, $f4, $f6	f2=f4 + f6
FP subtract single	sub.s $f2, $f4, $f6	f2=f4 - f6
FP multiply single	mul.s $f2, $f4, $f6	f2=f4 * f6
FP divide single	div.s $f2, $f4, $f6	f2=f4 / f6
FP add double	add.d $f2, $f4, $f6	f2=f4 + f6
FP subtract double	sub.d $f2, $f4, $f6	f2=f4 - f6
FP multiply double	mul.d $f2, $f4, $f6	f2=f4 * f6
FP divide double	div.d $f2, $f4, $f6	f2=f4 -/f6
Load wordcopr,1	Lwcl $f1, 100 ($s2)	F1=memory[s2+100]32 bit data to FP register
Store word copr,1	Swcl $f1, 100 ($s2)	Memory[s2+100]=f132 bit data to memory
Branch on FP true	Bclt 25	If(cond==1) goto PC+4+100PC relative branch if cond is true
Branch on FP false	Bclt 25	If(cond==0) goto PC+4+100PC relative branch if cond is false

FP compare single (eq, ne, li, le, gt, ge)	C.lt.s $f2, $f4	If(f2 < f4) Cond=1; else cond=0
FP compare double (eq, ne, li, le, gt, ge)	C.lt.d $f2, $f4	If(f2 < f4) Cond=1; else cond=0

2.4 HIGH PERFROMANCE ARTHMETIC

The performance improvement in arithmetic operations like addition, multiplication and division will increase the overall computational speed of the machine.

2.4.1 High performance adders

The high performance adders takes an extra input namely the transit time.

The transmit time of a logical unit is used as a time base in comparing the operating speeds of different methods, and the number of individual logical units required is used in the comparison of costs.

The two multi-bit numbers being added together will be designated as A and B, with individual bits being A1, A2, B1, etc. The third input will be C. Outputs will be S (sum) R (carry), and T (transmit). The two multi bit numbers being added together will be designated asA and B, with individual bits being A1, A2, B1, etc. The third input will be C. Outputs will be S (sum) R (carry), and T (transmit).

The time required to perform an addition in conventional adder is dependent on the time required for a carry originating in the first stage to ripple through all intervening stages

to the S or R output of the final stage. Using the transit time of a logical block as a unit of time, this amounts to two levels to generate the carry in the first stage, plus two levels per stage for transit through each intervening stage, plus two levels to form the sum in the final stage, which gives a total of two times the number of stages.

$C_n = R_{n-1}$
$C_n = D_{n-1} \,||\, T_{n-1}\, R_{n-2}$
$C_n = D_{n-1} \,||\, T_{n-1}\, D_{n-2} \,||\, T_{n-1} T_{n2}\, R_{n-3}$

By allowing n to have successive values starting with one and omitting all terms containing a a resulting negative subscript, it may be seen that each stage of the adder will

require one OR stage with n inputs and n AND circuits having one through n inputs, where N is the position number of the particular stage under consideration.

2.4.2 High performance Multiplication

Multiplication using variable length shift

- The multiplier and the partial product will always be shifted the same amount and at the same time.
- The multiplier is shifted in relation to the decoder, and the partial product with relation to the multiplicand.
- Operation is assumed starting at the low-order end of the multiplier, which means that shifting is to the right.
- If the lowest-order bit of the multiplier is a one, it is treated as though it had been approached by shifting across zeros.

Rules:

- When shifting across zeros (from low order end of multiplier), stop at the first one.

  a) If this one is followed immediately by a zero, add the multiplicand, then shift across all following zeros.

  b) If this one is followed immediately by a second one, subtract the multiplicand, then shift across all following ones.

2. When shifting across ones (from low order end of multiplier), stop at the first zero.

   a) If this zero is followed immediately by a one, subtract the multiplicand, then shift across all following ones.

   b) If this zero is followed immediately by a second zero, add the multiplicand, then shift across all following zeros.

- A shift counter or some equivalent device must be provided to keep track of the number of shifts and to recognize the completion of the multiplication.

- If the high-order bit of the multiplier is a one and is approached by shifting across ones, that shift will be to the first zero beyond the end of the multiplier, and that zero along with the bit in the next higher order position of the register will be decoded to determine whether to add or subtract.
- For this reason, if the multiplier is initially located in the part of the register in which the product is to be developed, it should be so placed that there will be at least two blank positions between the locations of the low-order bit of the partial product and the high-order bit of the multiplier.
- Otherwise the low-order bit of the product will be decoded as part of the multiplier.

Multiplication Using Uniform Shifts

- Multiplication which uses shifts of uniform size and permits predicting the number of cycles that will be required from the size of the multiplier is preferable to a method that requires varying sizes of shifts.
- The most important use of this method is in the application of carry-save adders to multiplication although it can also be used for other applications.

Uniform shifts of two

- Assume that the multiplier is divided into two-bit groups, an extra zero being added to the high-order end, if necessary, to produce an even number of bits.
- Only one addition or subtraction will be made for each group, and, using the position of the low-order bit in the group as a reference, this addition or subtraction will consist of either two times or four times the multiplicand.
- These multiples may be obtained by shifting the position of entry of the multiplicand into the adder one or two positions left from the reference position.
- The last cycle of the multiplication may require special handling.
- Following any addition or subtraction, the resulting partial product will be either correct or larger than it should be by an amount equal to one times the multiplicand.
- Thus, if the high-order pair of bits of the multiplier is 00 or 10, the multiplicand would be multiplied by zero or two and added, which gives a correct partial product.
- If the high-order pair of bits is 01 or 11, the multiplicand is multiplied by two or four,

not one or three, and added. This gives a partial product that is larger than it should be, and the next add cycle must correct for this.

- Following the addition the partial product is shifted left- two positions. This multiplies it by four, which means that it is now larger than it should be by four times the multiplicand.
- This may be corrected during the next addition by subtracting the difference between four and the desired multiplicand multiple.
- Thus, if a pair ends in zero, the resulting partial product will be correct and the following operation will be an addition.
- If a pair ends in a one, the resulting partial product will be too large, and the following operation will be a subtraction.
- It can now be seen that the operation to be performed for any pair of bits of the multiplier may be determined by examining that pair of bits plus the low-order bit of the next higher-order pair.
- If the bit of the higher-order pair is a zero, an addition will result; if it is one, a subtraction will result. If the low-order bit of a pair is considered to have a value of one and the high-order bit a value of two, then the multiple called for by a pair is the numerical value of the pair if that value is even and one greater if it is odd.
- If the operation is an addition, this multiple of the multiplicand is used. If the operation is a subtraction (the low-order bit of the next higher order pair a one), this value is combined with minus four to determine the correct multiple to use.
- The result will be zero or negative, with a negative result meaning subtract instead of add.

Multiplication Using Carry-Save Adders

- When successive additions are required before the final answer is obtained, it is possible to delay the carry propagation beyond one stage until the completion of all of the additions, and then let one carry-propagate cycle suffice for all the additions. Adders used in this manner are called **carry-save adders.**
- A carry-save adder consists of a number of stages, each similar to the full adder. It differs from the ripple-carry adder in that the carry (R) output is not connected directly

to the next-higher-order stage of the same adder, but goes to an intermediate register or other device in the same manner as the sum (S) output.

- A carry-save adder has three inputs which, as far as use is concerned, may be considered identical, and two outputs which are not identical and must be treated in different manners.
- The procedure for adding several binary numbers by using a carry-save adder would be as follows.
- Designate the inputs for the nth bit as A_n, B_n, and C, and the outputs for the same bit as S_n and R, where S_n is the sum output and R. is the carry output.
- In the first cycle enter three of the input numbers into A, B, and C.
- In the second cycle enter the S and R obtained from the previous cycle into A and B and the fourth input number into C.
- In this operation S_n goes into A_n, but R_n goes into B_{n+1}, where B_{n+1}isin the next higher-order bit position than B.
- This is continued until all of the input numbers have been entered into the adder.
- Each add cycle advances all carries one position, add cycles as already described may be continued with zeros being entered into the third input each time until the R outputs of all stages become zero.
- The alternative is to enter S and R into a carry-propagate adder and allow time for one cycle through it.
- This carry-propagate adder may be completely separate from the carry-save unit, or it may be a combined unit with a control line for selecting either carry-save or carry-propagate operation.

SUB WORD PARALLELISM

A sub word is a lower precision unit of data contained within a word. In sub word parallelism, multiple sub words are packed into a word and then process whole words.

With the appropriate sub word boundaries this technique results in parallel processing of sub words. Since the same instruction is applied to all sub words within the word, this is a

form of SIMD(Single Instruction Multiple Data) processing. It is possible to apply sub word parallelism to noncontiguous sub words of different sizes within a word. In practical implementation is simple if sub words are same size and they are contiguous within a word. The data parallel programs that benefit from sub word parallelism tend to process data that are of the same size.

Example: If word size is 64bits and sub words sizes are 8,16 and 32 bits. Hence an instruction operates on eight 8bit sub words, four 16bit sub words, two 32bit sub words or one 64bit sub word in parallel.

Advantages of sub word parallelism

- Sub word parallelism is an efficient and flexible solution for media processing because algorithm exhibit a great deal of data parallelism on lower precision data.
- It is also useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data.
- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors.
- One key advantage of sub word parallelism is that it allows general-purpose processors to exploit wider word sizes even when not processing high-precision data.
- The processor can achieve more sub word parallelism on lower precision data rather than wasting much of the word-oriented data paths and registers.

Support for sub word parallelism

- Data-parallel algorithms with lower precision data map well into sub word-parallel programs.
- The support required for such sub word-parallel computations then mirrors the needs of the data-parallel algorithms.
- To exploit data parallelism, we need sub word parallel compute primitives, which perform the same operation simultaneously on sub words packed into a word.
- These may include basic arithmetic operations like add, subtract, multiply, divide, logical, and other compute operations.

- Data-parallel computations also need

1. Data alignment before or after certain operations for sub words representing fixed-point numbers or fractions
2. Sub word rearrangement within a register so that algorithms can continue parallel processing at full clip
3. A way to expand data into larger containers for more precision in intermediate computations. Similarly, a way to contract it to a fewer number of bits after the computation's completion and before its output.
4. Conditional execution
5. Reduction operations that combine the packed sub words in a register into a single value or a smaller set of values.
6. A way to clip higher precision numbers to fewer bits for storage or transmission.
7. The ability to move data between processor registers and memory, as well as the ability to loop and branch to an arbitrary program location.