UNIT-III

COMPUTER ARTHIMETIC

INTRODUCTION

Arithmetic Instructions in digital computers manipulate data to produce results necessary for the of activity solution of computational problems. These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer. The four basic arithmatic operations are addition,subtracton,multiplication,division. From these basic operations, it is possible to formulate other arithmatic functions and solve scientific problems by means of numerical analysis methods . An arithmetic processor is the part of a pro cessor unit that execute arithmatic operations .

The data type assumed to residein prosser registers during the execution of an arithmetic instructions is specified in the definition of the instruction . An arithmetic instructions may specify binary or decimal data, and in each case the data may be in fixed point or floating-point form . Fixed-point numbers may represent integer or fractions. Negative numbers may be in signed – magnitude or signed – complement representation. The arithmetic processor is very simple if only a binary fixed-point add instructons is included. It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed-point and floating – point representation.

The four basic arithmetic operations are

- ADDITION
- SUBTRACTION
- MULTIPLICATION
- DIVISION

ADDITION AND SUBTRACTION

There are three ways of representing negative fixed point binary numbers :

- Signed Magnitude
- **Sig ed s o ple e t**
- **Sig ed s o ple e t**

Most o puter use the sig ed s o pli e t representation when performing arithmetic operations with integers . For floating point operations , most computers use the signed magnitude representation.

Arithmetic Addition and Subtraction in fixed point representation

- Simply add the two numbers and discard any leftmost carry out bit.

- **Negati e u ers are i s o ple e t**
  - Including the result!

  A-B=A+(-B)

  B i s o ple e t for

Example

+7	0000 0111	-7	1111 1001
+11	0000 1011	+11	0000 1011
+18	0001 0010	+4	0000 0100

+7	0000 0111	-7	1111 1001
-11	1111 0101	-11	1111 0101
-4	1111 1100	-18	1110 1110

s o ple e t Co td.,

- **The s o ple e t a e for ed lea i g all least sig ifi a t s a d the first u ha ged, a d the repla i g s s a d s s i all** other higher significant bits.

- Identify the rightmost 1 and complement all the bits on its left.

- **E . The s o ple e t of** 1101**1**00 = 0010**1**00
- **E . The s o ple e t of** 110111**1** = 001000**1**

MULTIPLICATION ALGORITHMS

Multiplication of two fixed point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift adds operations. This process is best illustrated with a numerical example . the process consist of looking at successive bits of the multiplier , least significant bit first . If the multiplier bit is a 1 . the multiplicand is copied **do ; other ise , zero s are** copied down . the number copied down in successive lines are shifted one position to the left from the previous number . Finally ,the numbers are added their sum forms are product.

EXAMPLE

23	***10111***	***Multiplicand***
19	***x 10011***	***Multiplier***
	10111	
	10111	
	00000	***+***
	00000	
	10111	
437	***110110101***	***Product***

Booths Alogritham for Multiplication

- Decide which operand will be the **multiplier** and which will be the **multiplicand**
- Convert both operands to **two's complement** representation using X bits
  - X must be at least one more bit than is required for the binary representation of the numerically larger operand
- Begin with a product that consists of the multiplier with an additional X leading zero bits

Example

- In the week by week, there is an example of multiplying **2 x (-5)**
- For our example, let's reverse the operation, and multiply (**-5) x 2**
  - The numerically larger operand (5) would require 3 bits to represent in binary (101). So we must use AT LEAST 4 bits to represent the operands, to allow for the sign bit.
- Let's use 5-bit 2's complement:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)

Beginning Product

- The multiplier is:

  11011

- Add 5 leading zeros to the **multiplier** to get the **beginning product**:

  00000 11011

Step 1 for each pass

- Use the **LSB** (least significant bit) and the **previous LSB** to determine the arithmetic action.
  - If it is the FIRST pass, use `0` as the previous LSB.

- Possible arithmetic actions:
  - `00` → no arithmetic operation
  - `01` → add multiplicand to left half of product
  - `10` → subtract multiplicand from left half of product
  - `11` → no arithmetic operation

Step 2 for each pass

- Perform an **arithmetic right shift** (ASR) on the entire product.

- NOTE: For X-bit operands, Booth's algorithm requires X passes.

Example

- Let's continue with our example of multiplying (**-5) x 2**
- Remember:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)

- And we added 5 leading zeros to the **multiplier** to get the **beginning product**:

00000 11011

Example continued

- Initial Product and previous LSB

  00000 11011 0

(Note: Since this is the first pass, we use 0 for the previous LSB)

- Pass 1, Step 1: Examine the last 2 bits

  00000 11011 0

The last two bits are 10, so we need to:

subtract the **multiplicand** from left half of product

Example: Pass 1 continued

- Pass 1, Step 1: Arithmetic action

```
(1)  00000     (left half of product)
    -00010     (mulitplicand)
     11110     (uses a phantom borrow)
```

- Place result into **left half** of product

```
11110 11011  0
```

Example: Pass 1 continued

- Pass 1, Step 2: ASR (arithmetic shift right)
  - Before ASR

    `11110 11011 0`
  - After ASR

    `11111 01101 1`

  (left-most bit was 1, so a 1 was shifted in on the left)
- Pass 1 is complete.

- Repeat this process for bit count, here bit count is 5 so we need to repeat the process up to 5 times to get result.

Final Product

- We have completed 5 passes on the 5-bit operands, so we are done.

- Dropping the previous LSB, the resulting **final product** is:

  11111 10110

Verification

- To confirm we have the correct answer, convert the 2's complement **final product** back to decimal.
- Final product: **11111 10110**
- Decimal value: **-10**

  which is the CORRECT product of:

  (-5) x 2

DIVISION ALGORITHM

Division of two fixed-point binary numbers is signed – magnitude representation is done with paper and pencil by a process of successive compare , shift, subtract operations . Binary divisionsis simpiler than decimal division because the quotient degit are either 0 or 1 and there is no need to estimate how many time the dividend or partial remainders fits into the divisor . The division is simpler then decimal division because the quotient degit are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainders fits into the divisor . the division processs is illustrated by a numerical example :-

11010

Divisor

B= 1 0 0 0 1) *0111000000* *dividend = A*

01110 *5 bit of A<B, quetiont has 5 bits*

011100 *6 bits of A>B*

-10001 *Shift right B and subtract ; enter 1 In Q*

-010110 *7 bit of remainder >B*

--10001 *Shift right B and subtract ; enter 1 In Q*

--001010 *Remainder<B; enter 0 in Q ; Shift right B*

---010100 *Remainder > B*

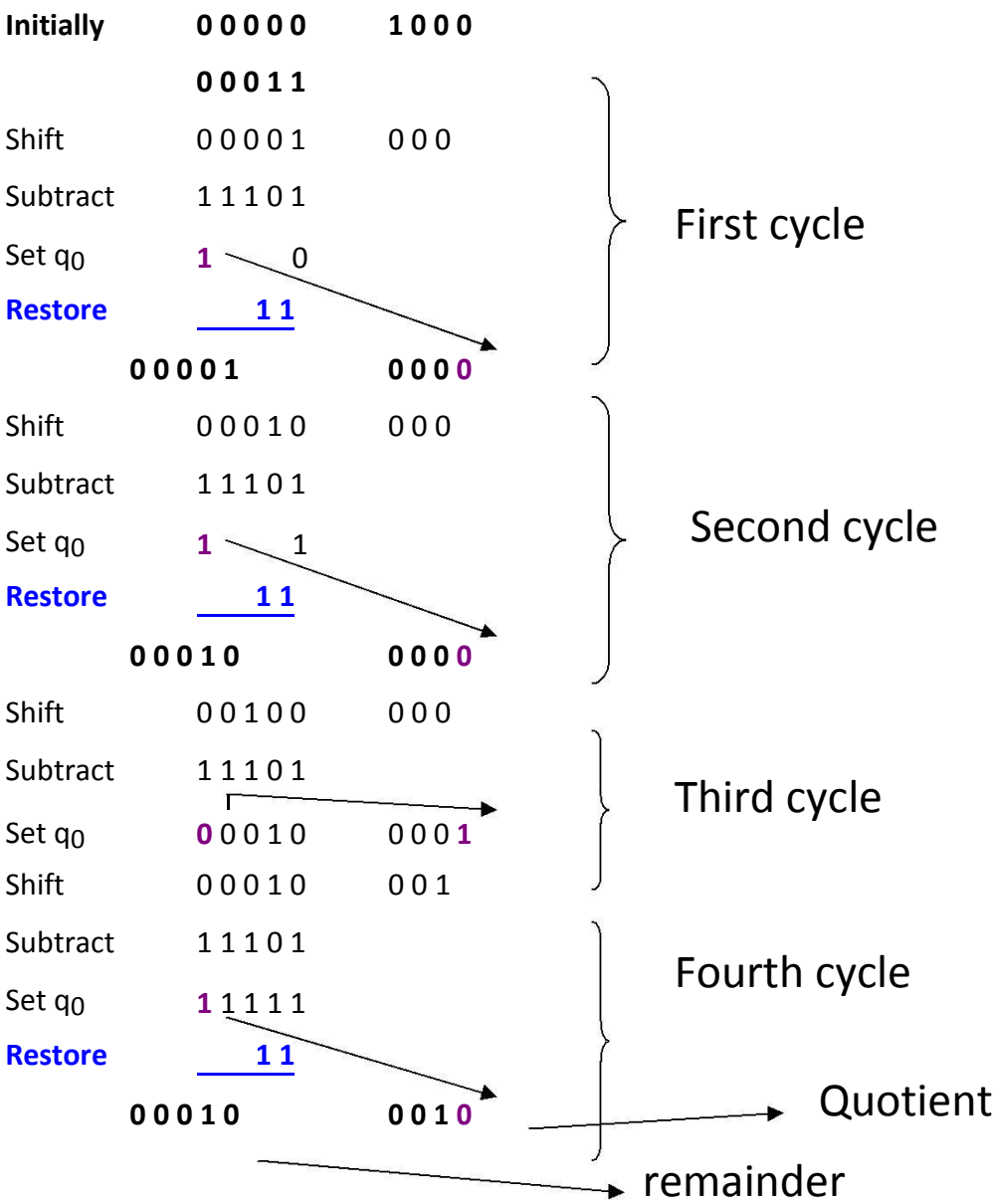
----10001 *Shift right B and subtract; enter 1 in Q*

----000110 *remainder <B; enter 1 in Q*

-----00110 *final remainder*

EXAMPLE OF BINARY DIVISION

A restoring-division example



Algorithms for Division

The restoring-division algorithm:

S1: DO n times

Shift A and Q left one binary position.

Subtract M from A, placing the answer back in A.

If the sign of A is 1, set q_0 to 0 and add M back to A **(restore A)**; otherwise, set q_0 to 1.

5) Algorithms for Division

The non-restoring division algorithm:

S1: Do n times

If the sign of A is 0, shift A and Q left one binary position and subtract M from A;
otherwise, shift A and Q left and add M to A.

S2: If the sign of A is 1, add M to A.

Floating Point Representation

- *A floating point is always interpreted to represent a number in the following form:*
- $m.r^{e}$
- r is base (radix)
- *e* is exponent
- ***O l the a tissa a d the e po e t e are ph sicall*** *represented in the register(including their sign)*
- A floating-point binary number is said to be <u>normalized</u> if the **ost sig ifi a t digit of the a tissa is o zero.**

Floating Point Representation (Contd.,)

- Normalized numbers provide the maximum possible precision for the floating-point number.

- A zero cannot be normalized because it does not have a nonzero digit.

  It is usuall represe ted i the floati g poi t all s i the a tissa a d exponent.

- Decimal number +6132.789 is represented in floating point with a fraction and an exponent as follows:

  Mantissa: +.6132789 Exponent:+04

Floating Point Representation (Contd.,)

- Binary number +1001.11 is represented with an 8-bit fraction and a six bit exponent as follows:

  Mantissa: 01001110 Exponent: 0000100

- The fraction has a zero in the leftmost position to denote positive.
- The binary point of the fraction follows the sign bit but is not shown in the register.
- The floating point number is equivalent to
- $m \times 2^{e} = + (.10001110)_2 \times 2^{+4}$.

Floating Point Representation (Contd.,)

- Arithmetic operations with floating-point numbers are more complicated than arithmetic with fixed point numbers and their execution takes longer and requires more complex hardware.

- However, floating point representation is a must for scientific computations because of the scaling problems involved with

  fixed-point computations.

Two representations

Single precision and double precision or 32 bit representation and 64 bit representation.

- The basic format allows representation in single and double precision
- 1. Basic: single (32 bits) and double (64 bits)
- single: Sign(1),Exponent(8),mantissa(23)
- double: Sign (1) Exponent(11) mantissa(52)

Floating point addition and subtraction

- BASIC ALGORITHM
- Subtract exponents (d = Ex - Ey).

Align mantissa

- Shift right d positions the mantissa of the operand with the smallest exponent.
- Select as exponent of the result the largest exponent.
- Add (Subtract) mantissa and produce sign of result.

- Normalization of result. Three situations can occur:

  (a) The result already normalized: no action is
  needed 1.10011111 0.00101011

  1.11001010

Effective operation addition: there might be an overflow of the significand. The normalization consists in

- Shift right the significand one position

  Increment by one the exponent

1.1001111 0.0110110

10.0000101

1.00000101

- Effective operation subtraction: the result might have leading zeros. Normalize:
  Shift left the mantissa by a number of positions corresponding to the number of leading zeros.

  Decrement the exponent by the number of leading zeros. 1.1001111 1.1001010

  0.0000101 NORM
  1.0100000
- 5. Round:

  According to the specified mode. Might require an addition. If overflow occurs, normalize by a right shift and increment the exponent.
- 6. Determine exception flags and special values : exponent overflow (special value infinity), exponent underflow (special value gradual under-flow), inexact, and the special value zero.

Floating point multiplication

x and y - normalized operands represented by (Sx,Mx,Ex) and (Sy,My,Ey)

1. Multiply mantissas add exponents, and determine sign of the result.
2. Normalize result and update exponent
3. Round
4. Determine exception flags and special values

Floating point Division

x and y - normalized operands represented by (Sx,Mx,Ex) and (Sy,My,Ey)

1. divide mantissas and subtract exponents, and determine sign of the result
2. Normalize result and update exponent
3. Round
4. Determine exception flags and special values