UNIT- II

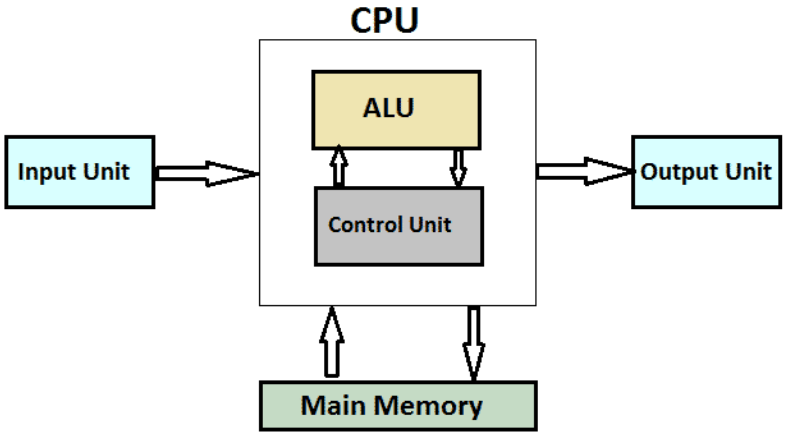
ARITHMETIC OPERATIONS

ALU - Addition and subtraction – Multiplication – Division – Floating Point operations – Subword parallelism.

ALU

Arithmetic And Logic Unit

- ✓ An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations.
- ✓ It represents the fundamental building block of the central processing unit (CPU) of a computer.
- ✓ Modern CPUs contain very powerful and complex ALUs.
- ✓ Some processors contain more than one AU - for example, one for fixed-point operations and another for floating-point operations.
- ✓ In personal computers floating point operations are sometimes done by a floating point unit on a separate chip called a numeric coprocessor.



- ✓ Typically, the ALU has direct input and output access to the processor controller, main memory (random access memory or RAM in a personal computer), and input/output devices.
- ✓ Inputs and outputs flow along an electronic path that is called a bus.
- ✓ The input consists of an instruction word (sometimes called a machine instruction word) that contains an operation code (sometimes called an "op code"), one or more operands, and sometimes a format code.

- ✓ The operation code tells the ALU what operation to perform and the operands are used in the operation.
- ✓ For example, two operands might be added together or compared logically.
- ✓ The format may be combined with the op code and tells, for example, whether this is a fixed-point or a floating-point instruction.
- ✓ The output consists of a result that is placed in a storage register and settings that indicate whether the operation was performed successfully.

- ✓ The ALU unit consists of two subsections namely,
  - Arithmetic Section
  - Logic Section

Arithmetic Section

- ✓ Function of arithmetic section is to perform arithmetic operations like addition, subtraction, multiplication, and division.
- ✓ All complex operations are done by making repetitive use of the above operations.

Logic Section

- ✓ Function of logic section is to perform logic operations such as comparing, selecting, matching, and merging of data.

ADDITION AND SUBTRACTION

Addition

- ✓ Digits are added bit by bit from right to left , with carries passed to the next digit to the left , just as you would do by hand.

Subtraction

- ✓ Subtraction uses addition: the appropriate operand is simply negated before being added.

Binary Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Overflow

- ✓ Overflow occurs when there are insufficient bits in a binary number representation to portray the result of an arithmetic operation.

Half Adder

- ✓ With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates.
- ✓ Let us first take a look at the addition of single bits.

  0+0 = 0

  0+1 = 1

  1+0 = 1

  1+1 = 10

- ✓ These are the least possible single-bit combinations. But the result for 1+1 is 10.
- ✓ Though this problem can be solved with the help of an EXOR Gate, if you do care about the output, the sum result must be re-written as a 2-bit output.
- ✓ Thus the above equations can be written as

  0+0 = 00

  0+1 = 01

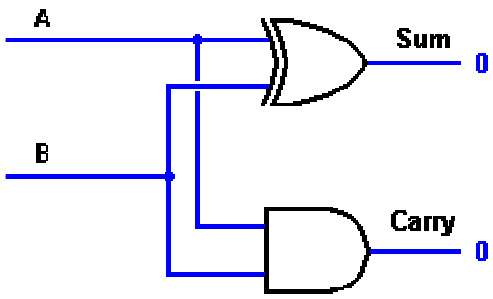
  1+0 = 01

  1+1 = 10

- ✓ Here the output '1'of '10' becomes the carry-out. The result is shown in a truth-table below. 'SUM' is the normal output and 'CARRY' is the carry-out.

INPUTS		OUTPUTS	
A	**B**	**SUM**	**CARRY**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- ✓ From the equation it is clear that this 1-bit adder can be easily implemented with the help of EXOR Gate for the output 'SUM' and an AND Gate for the carry.
- ✓ Take a look at the implementation below.



- ✓ For complex addition, there may be cases when you have to add two 8-bit bytes together. This can be done only with the help of full-adder logic.

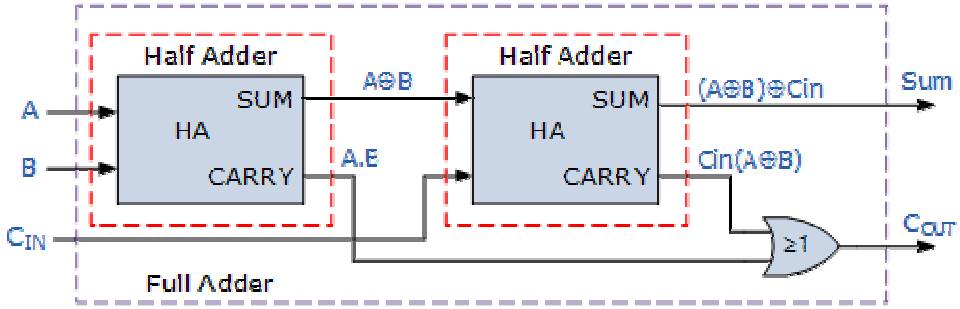
Full Adder

- ✓ This type of adder is a little more difficult to implement than a half-adder.
- ✓ The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs.
- ✓ The first two inputs are A and B and the third input is an input carry designated as C_{in}.
- ✓ When a full adder logic is designed we will be able to string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

- ✓ The output carry is designated as COUT and the normal output is designated as S. Take a look at the truth-table.

INPUTS		OUTPUTS		
A	**B**	**CIN**	**COUT**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- ✓ From the above truth-table, the full adder logic can be implemented.
- ✓ We can see that the output S is an EX-OR between the input A and the half-adder SUM output with B and C_{in} inputs.
- ✓ We must also note that the C_{out} will only be true if any of the two inputs out of the three are HIGH.
- ✓ Thus, we can implement a full adder circuit with the help of two half adder circuits.



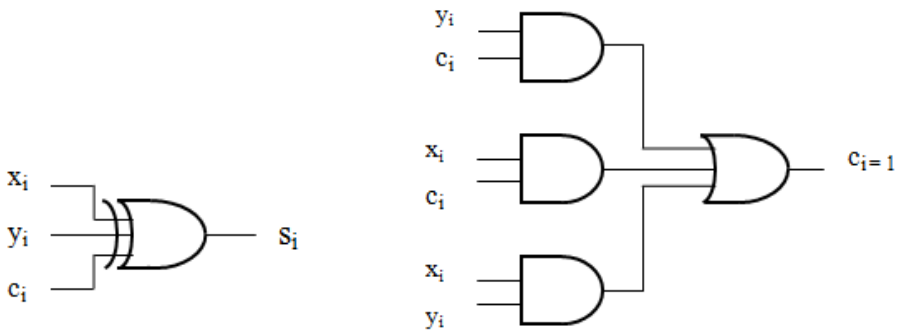
- ✓ The first will half adder will be used to add A and B to produce a partial Sum.
- ✓ The second half adder logic can be used to add C_{in} to the Sum produced by the first half adder to get the final S output.
- ✓ If any of the half adder logic produces a carry, there will be an output carry.
- ✓ Thus, C_{out} will be an OR function of the half-adder Carry outputs.

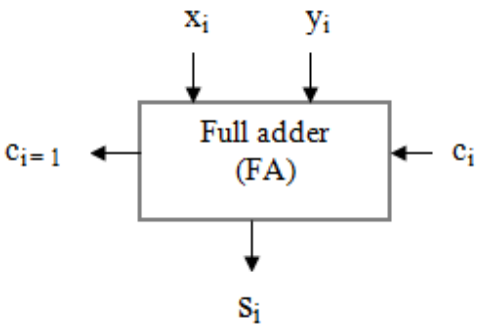
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i\overline{y}_i\overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

✓ Take a look at the implementation of the full adder circuit shown below.

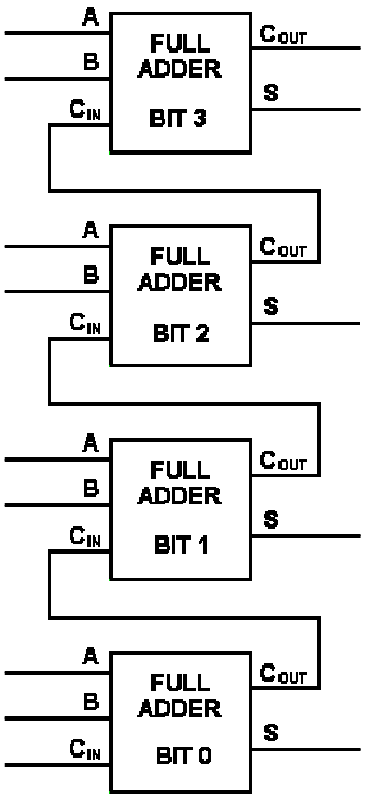




✓ In a computer, for a multi-bit operation, each bit must be represented by a full adder and must be added simultaneously.

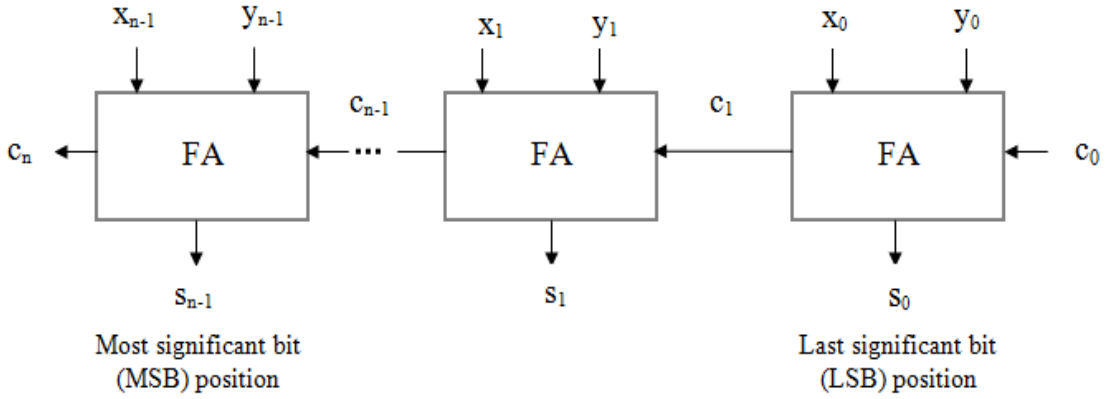
✓ Thus, to add two 8-bit numbers, you will need 8 full adders which can be formed by cascading two of the 4-bit blocks.

✓ The addition of two 4-bit numbers is shown below.



Multi-Bit Addition using Full Adder

✓ N bit adder can be shown as following,



✓ Here n number of binary bits are sent to the adders simultaneously.

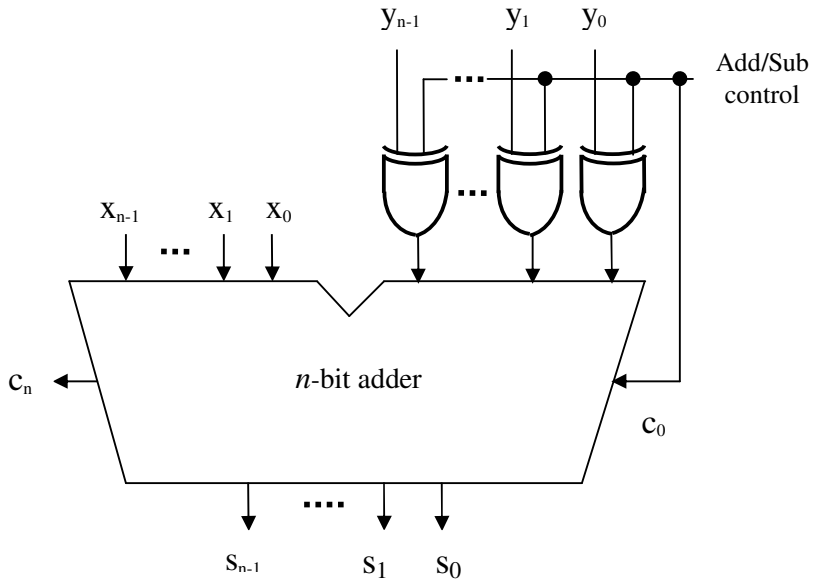
✓ The LSB bit Full Adder, first produce the sum0, and carry1.

✓ The c can be input to the second Full Adder.

✓ This process continued upto all Full Adders and final sum is generated.

ADDITION / SUBTRACTION CIRCUIT

- In order to perform X – Y
- We find the 2's complement of Y and add it to X.
- The above circuit is used to perform either addition or subtraction.
- That can be determined by Add/Sub input control signal.



When add/sub =0

- Then the addition operation is performed.
- Here the Y value is unchanged.
- The carry C_0 value also 0.
- So the X value added with actual Y value and result is produced.

When add/sub =1

- The Y value is 1's complemented by this control line.
- Then C_0 value is 1 which is added.
- Then the 2's complement is got.
- The actual value of X is added with 2's complement number of Y.
- Then the result is X –Y.

……..

Topic 3: Multiplication

MULTIPLICATION

- ✓ At each step we multiply the multiplicand by a single digit from the multiplier.
- ✓ In binary, we multiply by either 1 or 0 (much simpler than decimal)
- ✓ Keep the running sum instead of storing and adding all the partial products at the end.

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

- ✓ The first operand is called the multiplicand and the second the multiplier.
- ✓ The final result is called the product.
- ✓ The length of the product is addition of n-bit multiplicand And m-bit multiplier,
- ✓ That is n+m bits are required to represent all possible products.

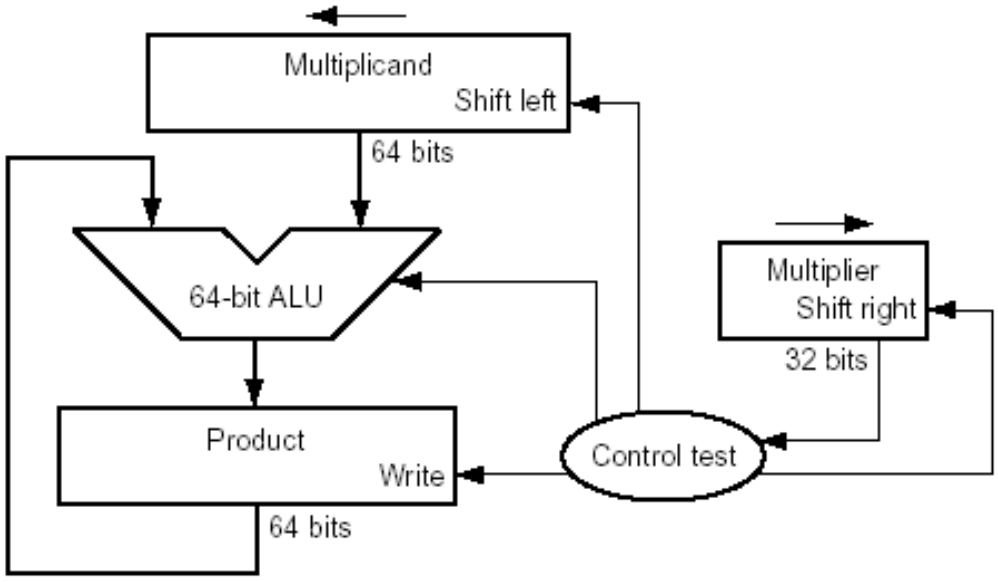
Two choices

- ✓ Due to binary numbers, we have only two choices for each step of the multiplication.

Choice 1: Just place a copy of the multiplicand (1 x multiplicand) in the proper place if the multiplier digit is a 1, or

Choice 2: Place 0 (0 x multiplicand) in the proper place if the digit is 0.

- ✓ Since binary numbers always use 0 and 1, thus always offers these two choices.
- ✓ The following diagram describes how multiplication operation can be done.



✓ The components are,

- 32 bit Multiplier Register
- 64 bit Multiplicand Register
- 64 bit Product Register
- 64 bit ALU
- Control Test Mechanism

Initial Values :

- Multiplier Register contains actual Value of Multiplier.
- Multiplicand Register contains

  Multiplicand in right half,

  and zero in left half.
- Product Register contains is 0.

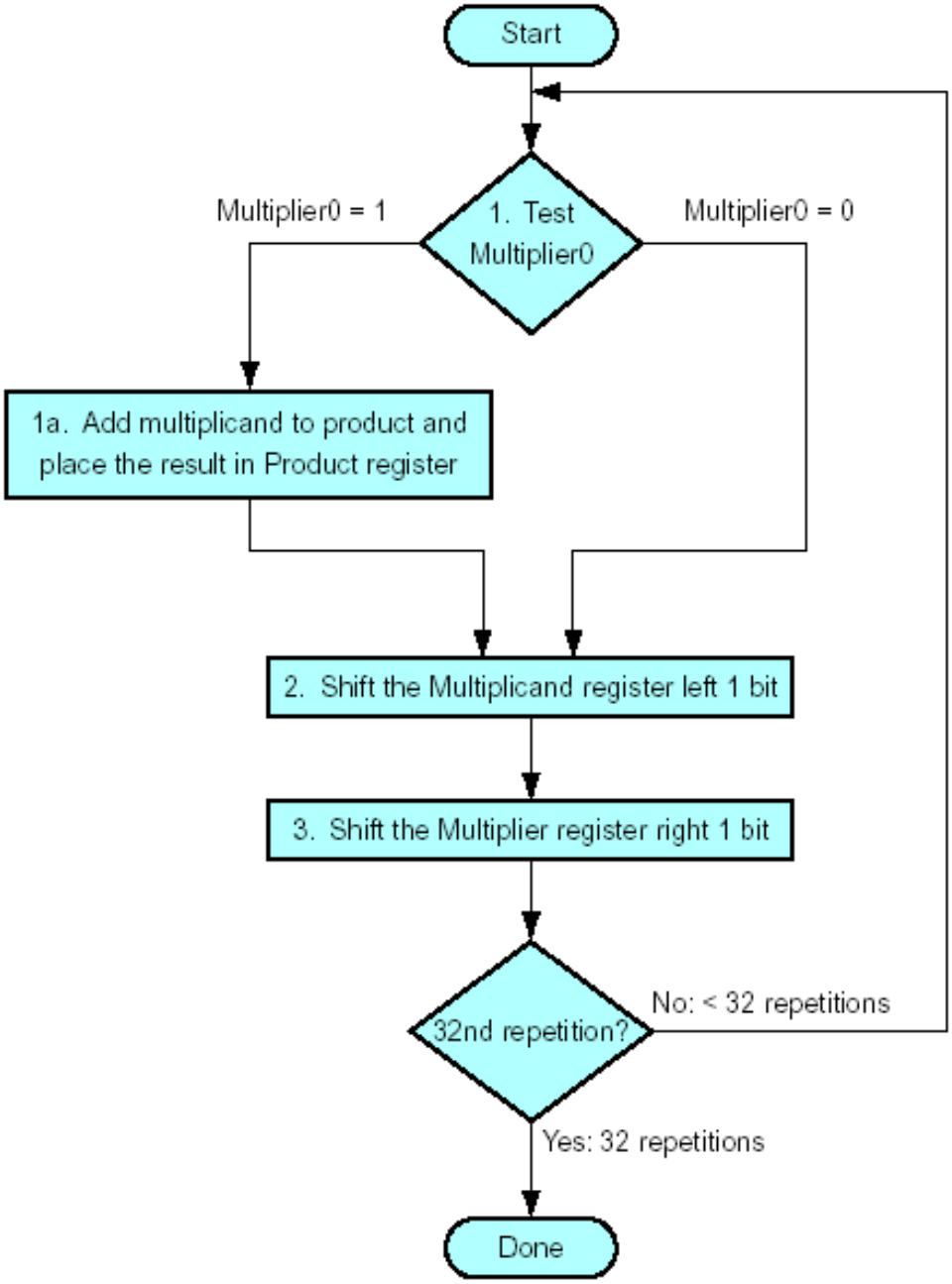
✓ Let (0010 x 0011) (2 x 3)

Multiplier Register	Multiplicand Register	Product Register
0011	0000 0010	0000 0000

Steps:

✓ The 32 bit Multiplicand starts in the right half.

- ✓ It is shifted left in one bit on each step.
- ✓ The multiplier shifted opposite direction at each step.
- ✓ Control decides when to shift Multiplier and Multiplicand.
- ✓ And also decides when to write new values in Product Register.



- ✓ The above flowchart describes three basic steps.

1. The LSB of Multiplier determines whether Multiplicand is added to the Product register or not.
2. Right shift in the Multiplier
3. Left shift in the Multiplicand.

✓ Example, Multiply 2ten x 3ten , or 0010_{two} x 0011_{two}

✓ The following table describes the each step of the multiplication operation.

Iteration		Steps	Multiplier	Multiplicand	Product
0		Initial Values	0011	0000 0010	0000 0000
1	i	LSB of multiplier is 1, Prod = Prod + Mcand	0011	0000 0010	0000 0010
	ii	Right Shift in Multiplier	0001	0000 0010	0000 0010
	iii	Left Shift in Mcand	0001	0000 0100	0000 0010
2	i	LSB of multiplier is 1, Prod = Prod + Mcand	0001	0000 0100	0000 0110
	ii	Right Shift in Multiplier	0000	0000 0100	0000 0110
	iii	Left Shift in Mcand	0000	0000 1000	0000 0110
3	i	LSB of multiplier is 0, No operation	0000	0000 1000	0000 0110
	ii	Right Shift in Multiplier	0000	0000 1000	0000 0110
	iii	Left Shift in Mcand	0000	0001 0000	0000 0110
4	i	LSB of multiplier is 0, No operation	0000	0001 0000	0000 0110
	ii	Right Shift in Multiplier	0000	0001 0000	0000 0110
	iii	Left Shift in Mcand	0000	0010 0000	0000 0110

Signed Multiplication

✓ So far, we have dealt with positive numbers.

✓ The easiest way to understand how to deal with signed numbers is,

to first convert the multiplier and multiplicand to positive numbers and then remember the original signs.

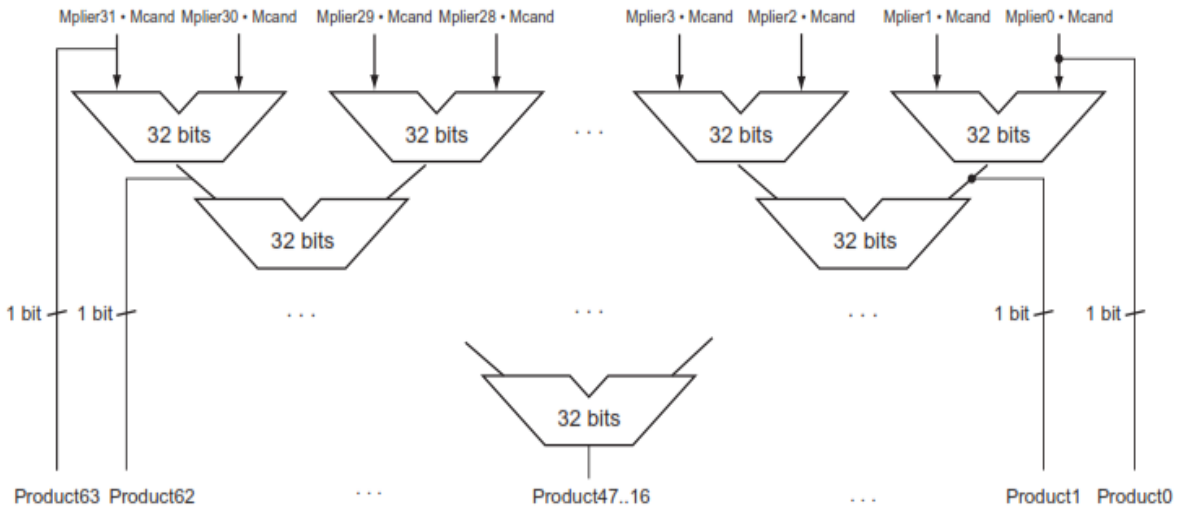
✓ The algorithms should then be run for 31 iterations, leaving the signs out of the calculation.

- ✓ As we learned in grammar school, we need negate the product only if the original signs are different.
- ✓ Hence, the shifting steps would need to extend the sign of the product for signed numbers.
- ✓ When the algorithm completes, the lower word would have the 32-bit product.

Faster Multiplication

- ✓ Moore's Law has provided so much more in resources that hardware designers can
- ✓ now build much faster multiplication hardware.
- ✓ Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits.
- ✓ Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier:

  one input is the multiplicand AND ed with a multiplier bit, and the other is the output of a prior adder.
- ✓ A straightforward approach would be to connect the outputs of adders on the
- ✓ right to the inputs of adders on the left , making a stack of adders 32 high.
- ✓ An alternative way to organize these 32 additions is in a parallel tree, in following figure,



- ✓ Instead of using single 32 bit adder in 32 times, the hardware uses 32 adders and then organizes them into minimize delay.

BOOTH'S ALGORITHM

- ✓ Booth algorithm gives a procedure for multiplying binary integers in signed 2's complement representation in efficient way, i.e., less number of additions/subtractions required.
- ✓ It operates on the fact that strings of 0's in the multiplier require no addition.

Normal Multiplication

```
                        0  1  0  1  1  0  1
                        0  0  1  1  1  1  0
                      ---------------------------
                        0  0  0  0  0  0  0  0
                     0  1  0  1  1  0  1
                  0  1  0  1  1  0  1
               0  1  0  1  1  0  1
            0  1  0  1  1  0  1
         0  0  0  0  0  0  0
      0  0  0  0  0  0  0
   ------------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

- ✓ Booth algorithm requires examination of the multiplier bits and shifting of the partial product.
- ✓ Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged

Booth Multiplier Recoding Table

Multiplier		Version of multiplicand Selected by bit i
Bit i	Bit i-1	
0	0	0 x M
0	1	+1x M
1	0	-1 x M
1	1	0 x M

Booth multiplication

Original multiplier : 0 0 1 1 1 1 0

Recoded multiplier : 0 +1 0 0 0 -1 0

```
                        0  1  0  1  1  0  1        Multiplicand
                        0 +1  0  0  0 -1  0        Recoded multiplier
                      ----------------------------
                        0  0  0  0  0  0  0
                     1  0  1  0  0  1  1
                  0  0  0  0  0  0  0
               0  0  0  0  0  0  0
            0  0  0  0  0  0  0
         0  1  0  1  1  0  1
      0  0  0  0  0  0  0
   ------------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0           Product
```

```
                        0  1  0  1  1  0  1

 1's comp:              1  0  1  0  0  1  0
                                        1     +
                       ----------------------------
 2's comp:              1  0  1  0  0  1  1
```

Worst case Multiplier : 0 1 0 1 0 1 0 1 0 1

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1

Ordinary case Multiplier:

```
1  1  0  0  0  1  0  0  0
0 -1  0  0 +1 -1  0  0  0
```

Good case Multiplier : 1 1 1 1 0 0 0 0

0 0 0 -1 0 0 0 0

Topic 4: Division

DIVISION

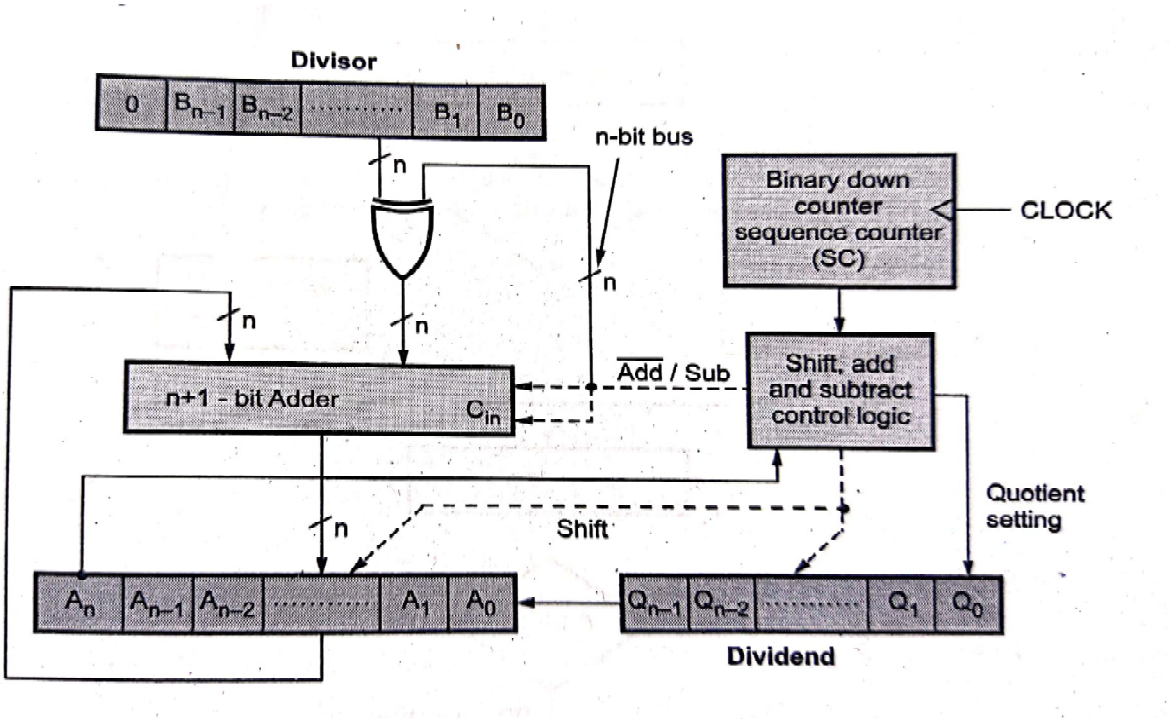
- ✓ The reciprocal operation of multiply is divide.
- ✓ An operation that is even less frequent and even more quirky.
- ✓ It even offers the opportunity to perform a mathematically invalid operation:
  - dividing by 0.
- ✓ The example is dividing 1001010, by 1000.

$$
\begin{array}{rl}
 & 1001_{ten} \quad \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} \quad \text{Dividend} \\
 & \underline{-1000} \\
 & \quad 10 \\
 & \quad 101 \\
 & \quad 1010 \\
 & \underline{-1000} \\
 & \quad 10_{ten} \quad \text{Remainder}
\end{array}
$$

- ✓ Divide's two operands, called the dividend and divisor, and the result, called the quotient, are accompanied by a second result, called the remainder.
- ✓ Here is another way to express the relationship between the components:

Dividend = Quotient x Divisor + Remainder

- ✓ where the remainder is smaller than the divisor.
- ✓ Infrequently, programs use the divide instruction just to get the remainder, ignoring the quotient.
- ✓ The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt.
- ✓ Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend:
- ✓ it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.
- ✓ Let's assume that both the dividend and the divisor are positive.
- ✓ And hence the quotient and the remainder are nonnegative.



✓ **A Division Algorithm and Hardware**

The following figure shows hardware of Division. The components are,

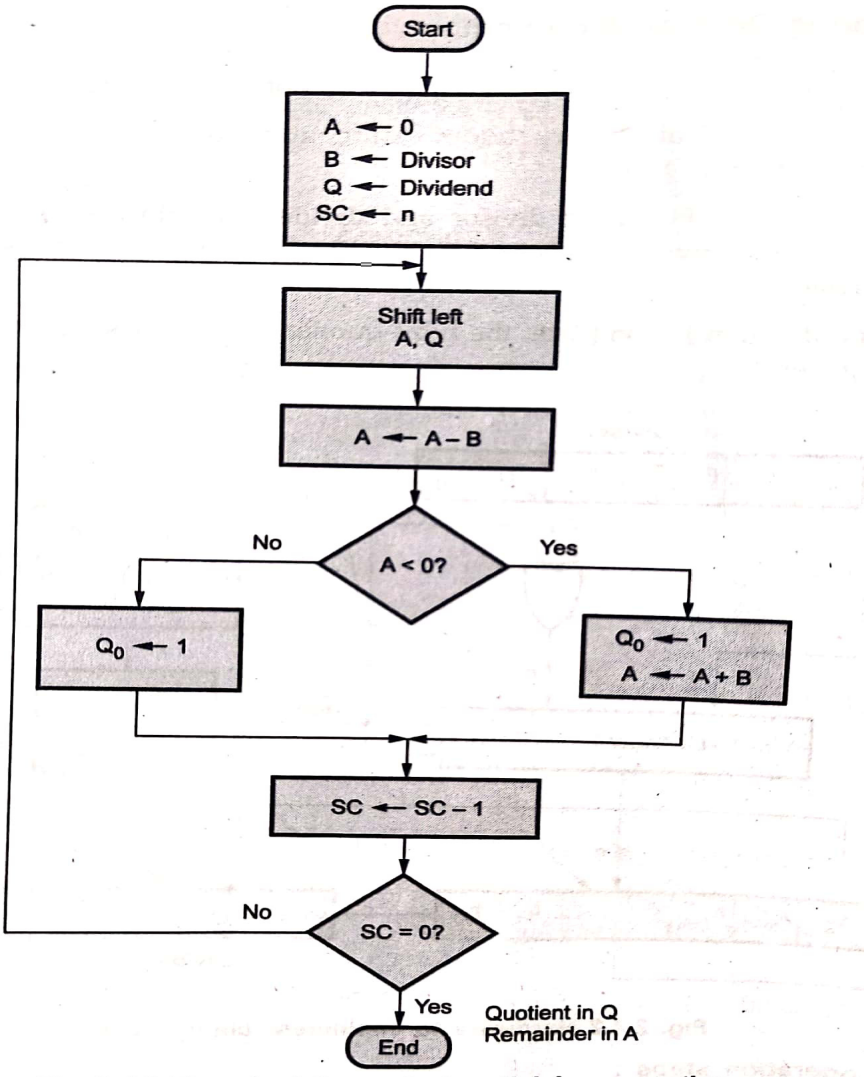
1. Divisor Register 32 bit plus one sign bit.
2. ALU 64 bit plus one sign bit
3. A Register 32 bit plus one sign bit
4. Q Register 32 bit

✓ A register initialized into zero.

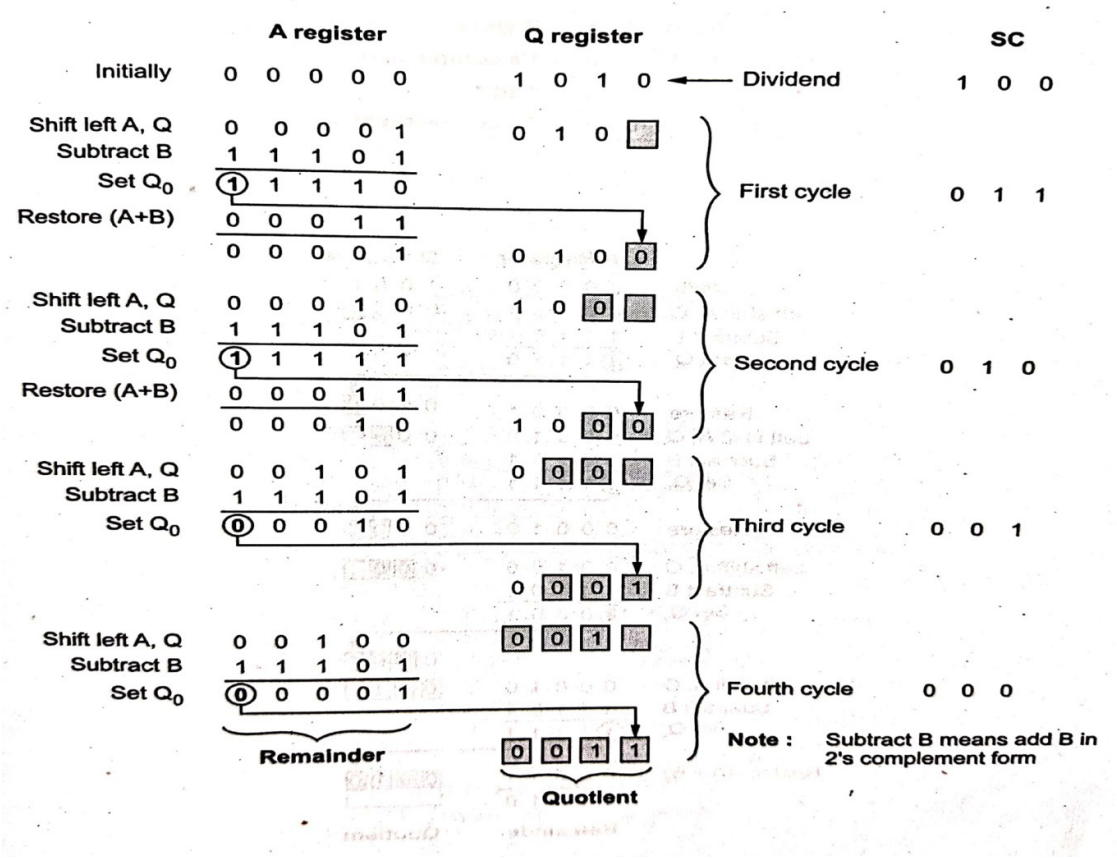
✓ Divisor content stored into B register

✓ Dividend content stored into Q register.

✓ Following figure shows steps of the division algorithm.



- ✓ A & Q register are placed horizontally from A & Q.
- ✓ Shift the A,Q register from left.
- ✓ Subtract divisor from A register and placed into A register.
- ✓ Depending upon the sign bit of A register, set the values into Q_0, and deciding whether apply restore or not
- ✓ Repeat the steps until all the bits of evaluated.
- ✓ **The above circuit is called Restoring Division.**

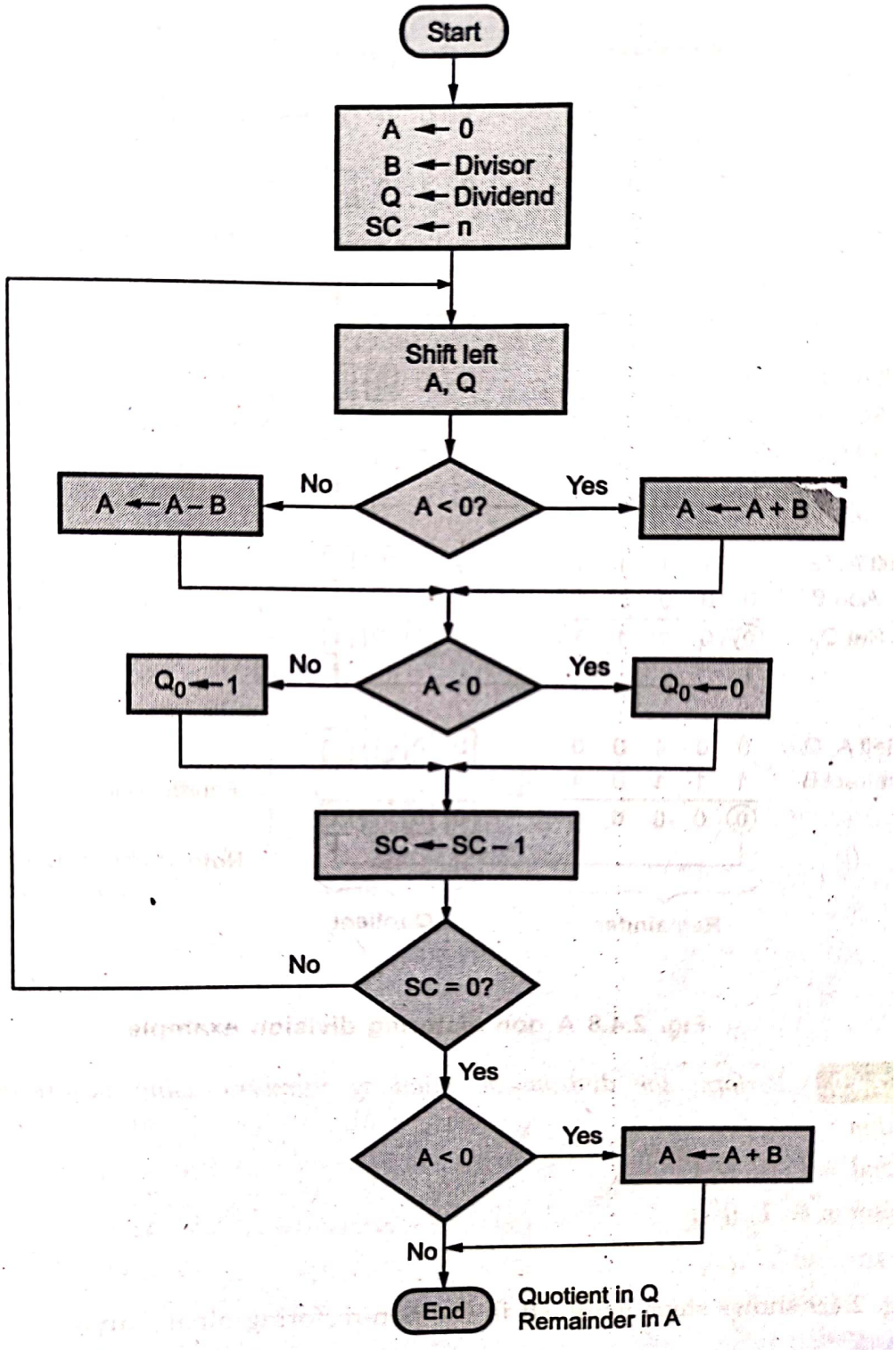


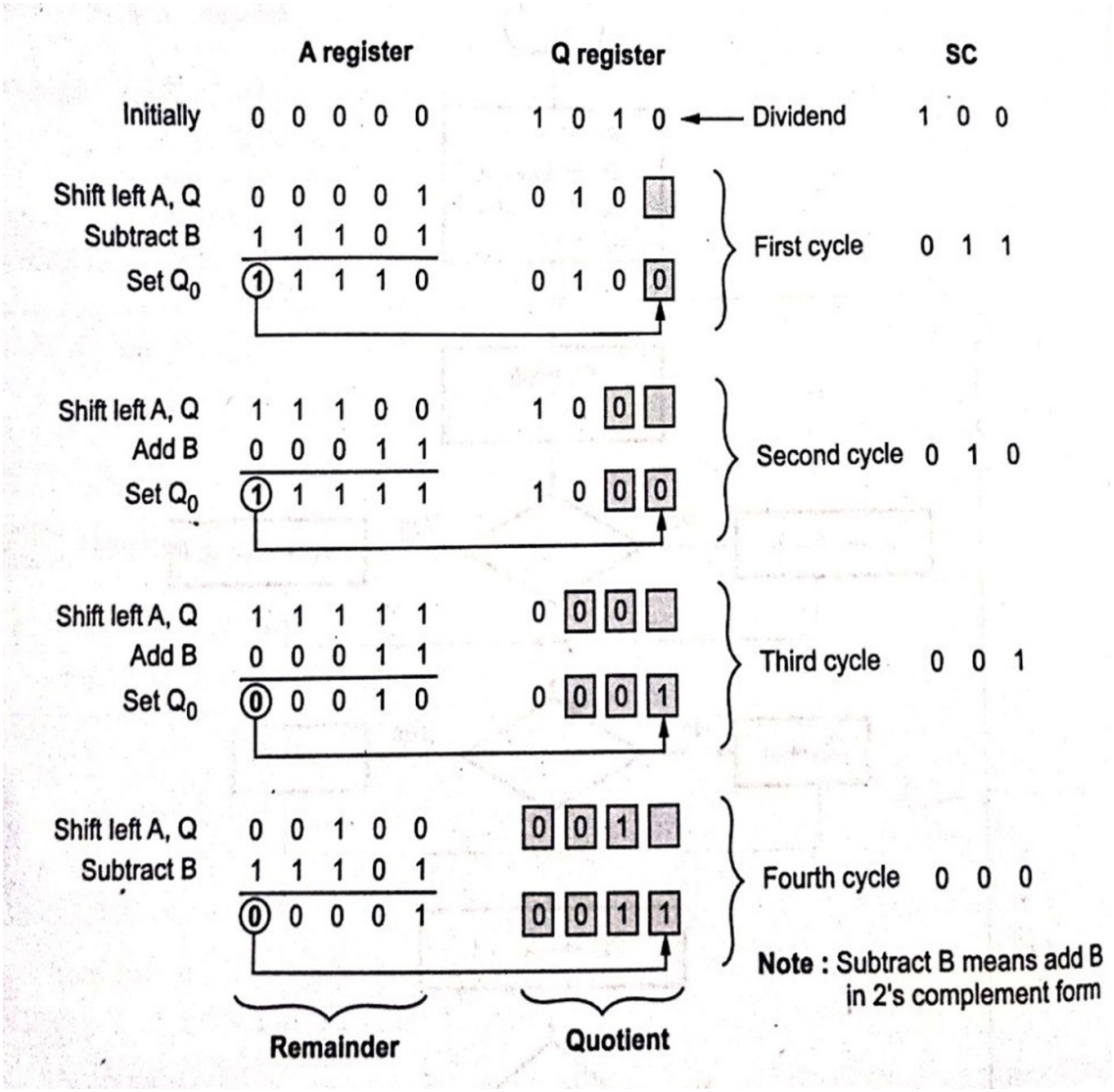
Restoring algorithm always produce delayed output, By applying some changes in the above algorithm, we van get Non restoring division.

Non Restore Division

- ✓ Same circuit used in Restoring division can be applied in Non Restoring division also.

  The flow chart will be,


Start
A ← 0
B ← Divisor
Q ← Dividend
SC ← n
Shift left
A, Q
A < 0?
No
Yes
A ← A – B
A ← A + B
A < 0
No
Yes
Q_0 ← 1
Q_0 ← 0
SC ← SC – 1
SC = 0?
No
Yes
A < 0
Yes
A ← A + B
No
End
Quotient in Q
Remainder in A



If the sign bit of A is 1, then restore the A register by adding divisor to get correct remainder.

Topic 5: Floating Point Operations

FLOATING POINT OPERATIONS

- ✓ Programming languages support numbers with fractions, which are called reals in mathematics. Here are some examples of reals:

$3.14159265\ldots_{ten}$ (pi)

$2.71828\ldots_{ten}$ *(e)*

0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$ (seconds in a nanosecond)

$3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$ (seconds in a typical century)

Scientific Notation

✓ A notation that renders numbers with a single digit to the left of the decimal point.

Normalized Number

✓ A number in scientific notation that has no leading 0s is called a normalized number.

Why it is called Floating Point Numbers?

✓ because it represents numbers in which the binary point is not fixed, as it is for integers.

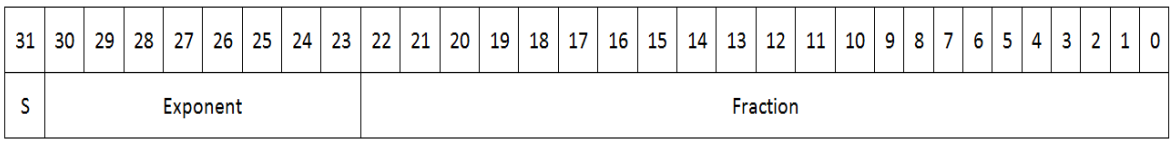
Advantages of Scientific Notation

1) It simplifies exchange of data that includes floating-point numbers
2) It simplifies the floating-point arithmetic algorithms
3) It increases the accuracy of the numbers.

32 bit Floating-Point Representation

✓ Floating-point numbers are usually a multiple of the size of a word.

✓ The representation of a MIPS floating-point number is shown below,



1 Bit 8 Bits 23 Bits

✓ where

s is the sign of the floating-point number (1 meaning negative),

exponent is the value of the 8-bit exponent field

fraction is the 23-bit fraction number.

✓ In general, 32 bit floating-point numbers are of the form,

$$(-1)^{S} \times (1+\text{Fraction}) \times 2^{E-127}$$

F involves the value in the fraction field

E involves the value in the exponent field.

S represents sign bit.

Over flow

A situation in which a positive exponent becomes too large to fit in the exponent field.

Underflow (floating- point)

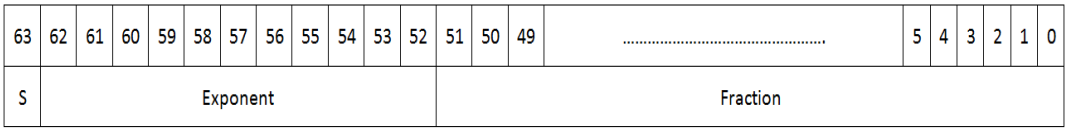
A situation in which a negative exponent becomes too large to fit in the exponent field.

Single Precision

A floating-point value represented in a single 32- bit word.

Double Precision

- ✓ A floating-point value represented in two 32-bit words.
- ✓ The representation of a double precision floating-point number takes two MIPS words, as shown below,



1 Bit 11 Bits 52 Bits

where

s is still the sign of the number,

exponent is the value of the 11-bit exponent field, and

fraction is the 52-bit number in the fraction field.

Advantage

- ✓ its primary advantage is its greater precision because of the much larger fraction.

FLOATING POINT ARITHMETIC

Add/Subtract rule

1.Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponent.

2.Set the exponent of the result equal to the larger exponent.

3.Perform addition/subtraction on the mantissa and determine the sign of the result.

4.Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissa is needed.

Multiply Rule

1. Add the exponent and subtract 127.
2. Multiply the mantissa and determine the sigh of the result.
3. Normalize the result value, if necessary.

Divide Rule

1. Subtract the exponent and add 127.
2. Divide the mantissa and determine the sign of the result.
3. Normalize the result value, if necessary.

FLOATING POINT ADDITION

- ✓ Let's add the two numbers,

$$9.999 \times 10^{1} \quad + \quad 1.610 \times 10^{-1}$$

- ✓ Assume that we can store only four digit of the significant and two decimal of the exponent.

Step 1: Align the smaller exponent number into bigger exponent number.

So that both the exponent values are same.

1.610×10^{-1} that can be written as 0.01610×10^{1}

But we can represent only four digits, 0.016×10^{1}

Step 2: Add the significant,

$$\begin{array}{r} 9.999 \\ 0.016 \\ \hline 10.015 \end{array}$$

The sum is 10.015×10^1

Step 3: Normalize the scientific notation.

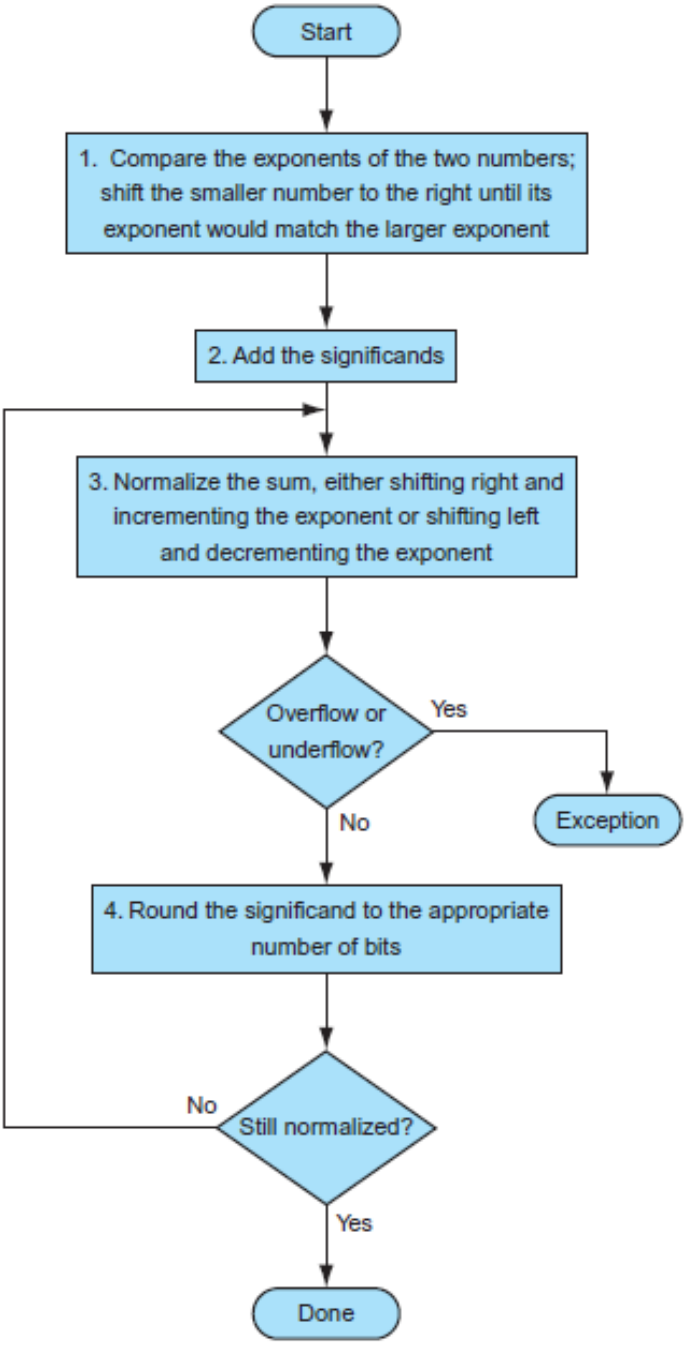
10.015×10^1 it becomes, 1.0015×10^2

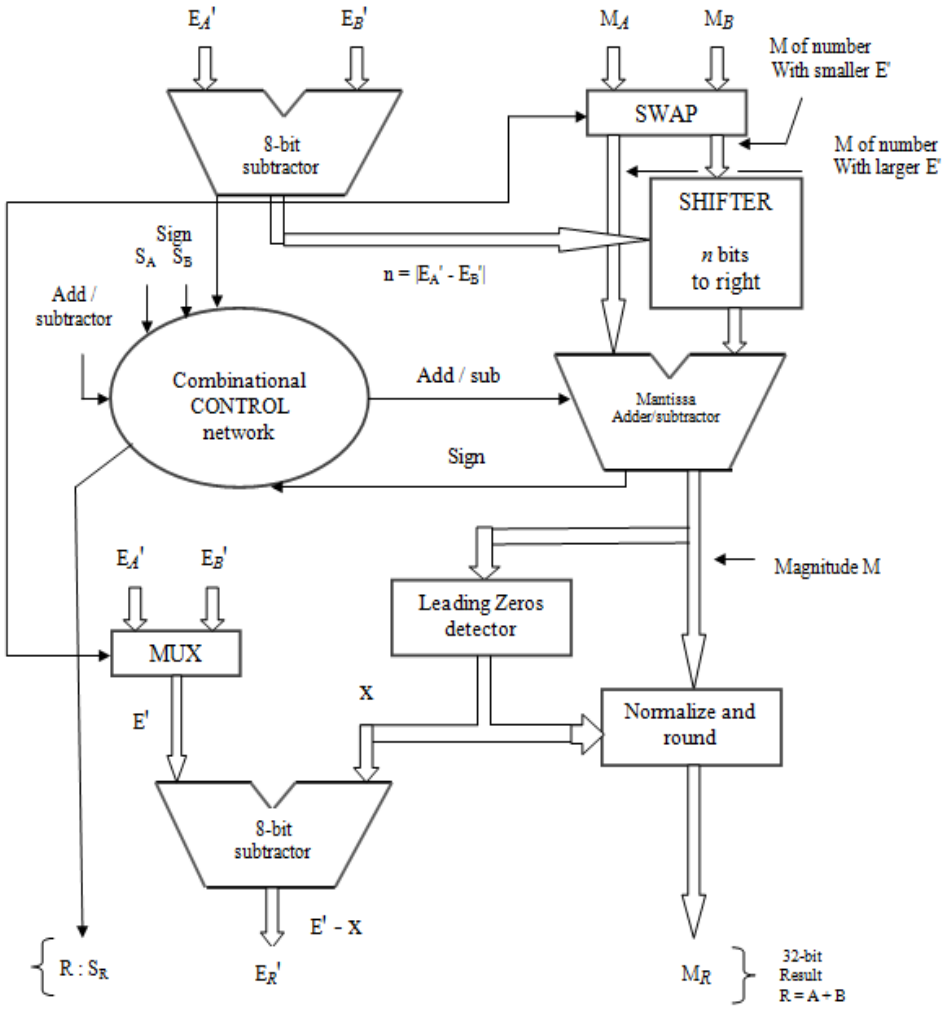
- ✓ Whenever the exponent increased or decreased, we must check overflow or underflow.
- ✓ That is make sure that the exponent still fits in the field.

Step 4: For representing only four digits, round the number,

1.0015×10^2 it becomes, 1.002×10^2

- ✓ The following figure shows that algorithm for binary floating point addition.


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done



IMPLEMENTING FLOATING-POINT OPERATIONS

- First compare exponents to determine how far to shift the mantissa of the number with the smaller exponent.
- The shift-count value n
  - → is determined by 8 bit subtractor &
  - → is sent to SHIFTER unit.
- In step 1, sign is sent to SWAP network (Figure 9.26).
  - If sign=0, then $E_A > E_B$ and mantissas M_A & M_B are sent straight through SWAP network.
  - If sign=1, then $E_A < E_B$ and the mantissas are swapped before they are sent to SHIFTER.
- In step 2, 2:! MUX is used. The exponent of result E is tentatively determined as E_A if $E_A > E_B$ or E_B if $E_A < E_B$
- In step 3, CONTROL logic
  - → determines whether mantissas are to be added or subtracted.
  - → determines sign of the result.
- In step 4, result of step 3 is normalized. The number of leading zeros in M determines number of bit shifts(X) to be applied to M.

FLOATING POINT MULTIPLICATION

- ✓ Let's multiply the two numbers,

$$1.110 \times 10^{10} \quad \mathbf{x} \quad 9.200 \times 10^{-5}$$

- ✓ Assume that we can store only four digit of the significant and two decimal of the exponent.

Step 1: Adding the exponent.

New exponent = 10 + (-5) = 5

Step 2 : Multiplication of the significant.

```
    1.110
    9.200
  -------
    0 0 0 0
  0 0 0 0
2 2 2 0
9 9 9 0
-------------
1 0 2 1 2 0 0 0
```

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significant:

10.212000

With exponent value as per the step 1,

$$10.212000 \times 10^{5}$$

Step 3: Normalize the answer

$$1.0212000 \times 10^{6}$$

Step 4: We assumed that significant only four digits long.

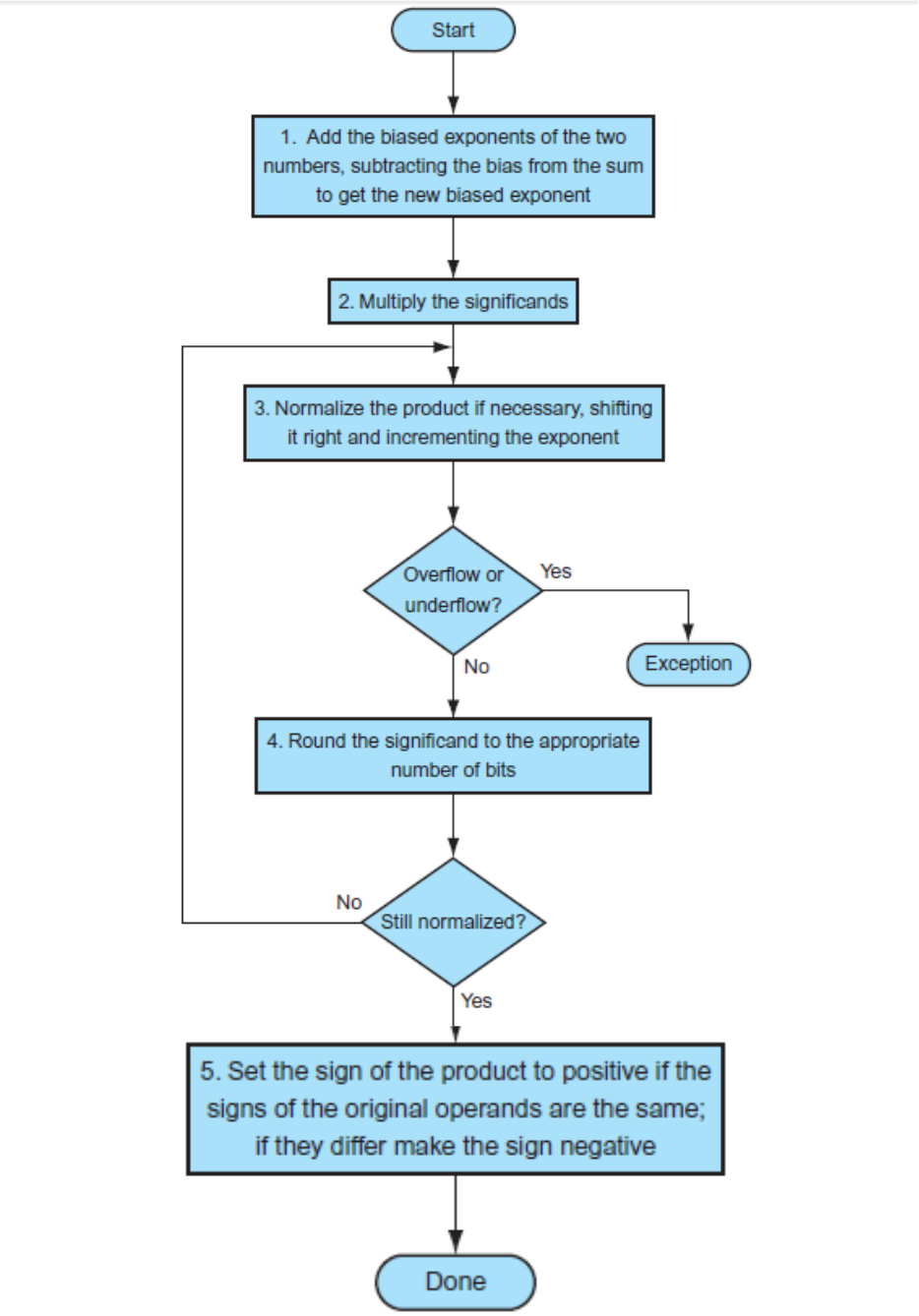
$$1.021 \times 10^{6}$$

Step 5:

The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$$+1.021 \times 10^{6}$$

✓ The following figure shows that algorithm for binary floating point Multiplicatin.



Topic 6: Subword parallelism

SUBWORD PARALLELLISM

- ✓ A subword is a lower precision unit of data contained within a word.
- ✓ In subword parallelism, multiple subwords are packed into a word and then process whole words.
- ✓ This technique results in parallel processing of subwords.
- ✓ Since the same instruction is applied to all subwords within the word, This is a form of SIMD(Single Instruction Multiple Data) processing.
- ✓ It is possible to apply subword parallelism to noncontiguous subwords of different sizes within a word.
- ✓ In practical implementation is simple if subwords are same size and they are contiguous within a word.
- ✓ For example if word size is 64bits and subwords sizes are 8,16 and 32 bits. Hence an instruction operates on eight 8bit subwords, four 16bit subwords, two 32bit subwords or one 64bit subword in parallel.

- Subword 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
  - Example: for 128-bit adder:
    Sixteen 8-bit adds
    Eight 16-bit adds
    Four 32-bit adds