JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”

DEPARTMENT

OF

ELECTRONICS AND COMMUNICATION ENGINEERING

LECTURE NOTES

EC8552 – COMPUTER ARCHITECTURE AND ORGANIZATION

(Regulation 2017)

Year/Semester: III / V ECE

2020 – 2021

Prepared by

Mr. N. Prabhakaran

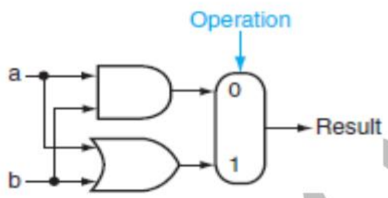
Associate Professor / IT

UNIT – II ARITHMETIC FOR COMPUTERS

Addition and Subtraction – Multiplication – Division – Floating Point Representation – Floating Point Operations – Subword Parallelism

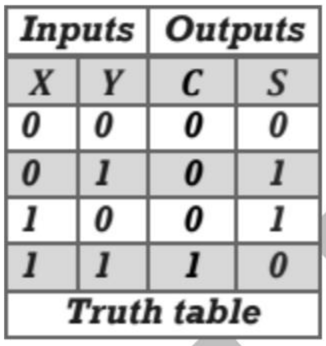
ALU:

- Arithmetic Logic Unit (ALU). Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.
- The arithmetic logic unit (ALU) is the brawn of the computer.



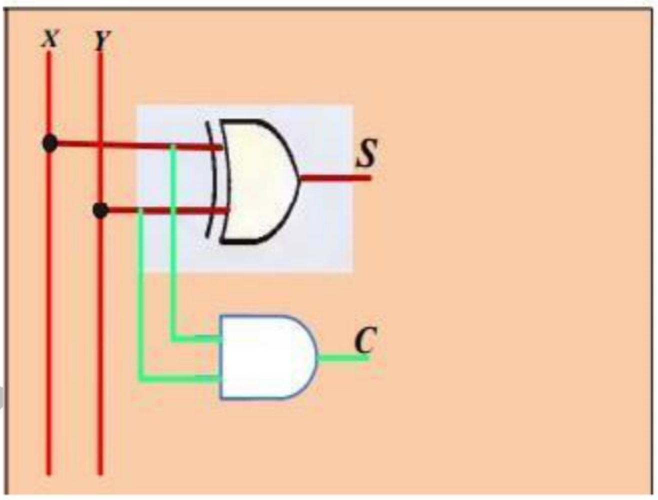
Half Adder

Half Adder: is a combinational circuit that performs the addition of two bits, this circuit needs two binary inputs and two binary output s.



Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table



Where *S* is the sum and *C* is the carry.

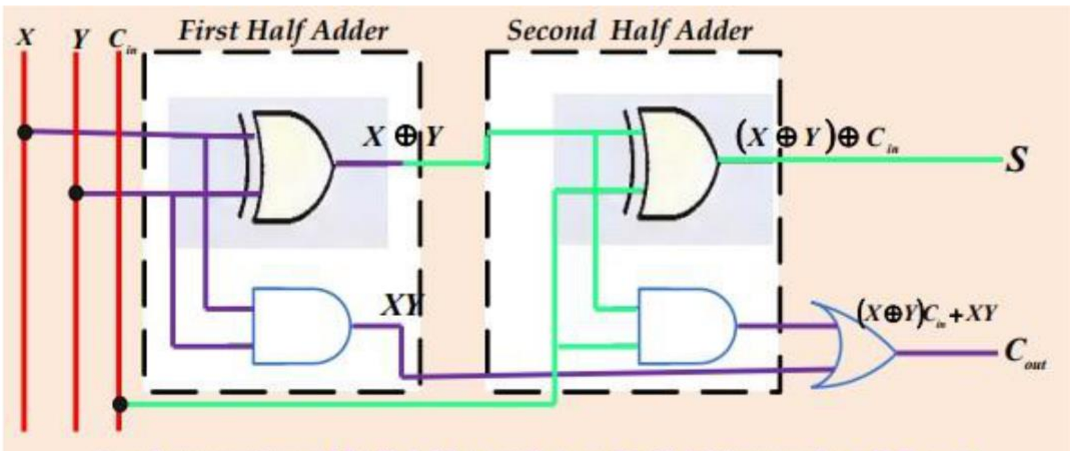
$$\begin{cases} S = X \oplus Y \\ C = XY \end{cases}$$

2} (Using XOR and AND Gates)

Full Adder

- Full Adder is a combinational circuit that performs the addition of three bits (two significant bits and previous carry).
- It consists of three inputs and two outputs, two inputs are the bits to be added, the third input represents the carry form the previous position.
- The full adder is usually a component in a cascade of adders, which add 8, 16, etc, binary numbers.
- An adder must have two inputs for the operands and a single-bit output for the sum and the second output to pass on the carry, called CarryOut.

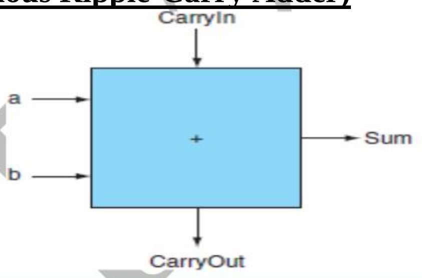
- The CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn.

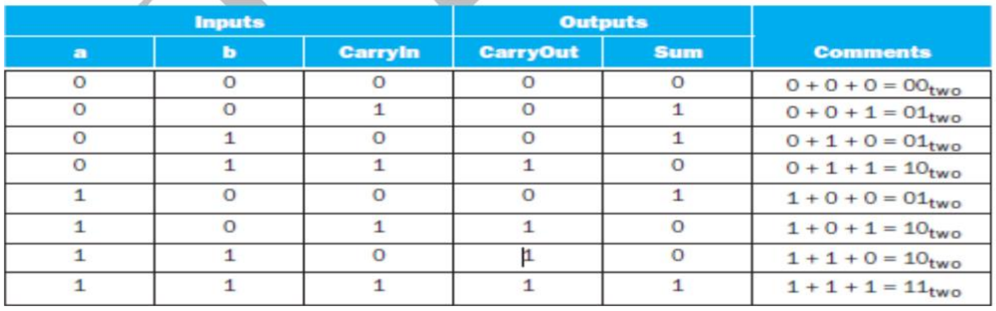


Implementation of Full Adder with two Half Adders and an OR gate

$$\begin{Bmatrix} S = C_{in} \oplus (X \oplus Y) \\ C_{out=} C_{in} \cdot (X \oplus Y) + XY \end{Bmatrix}$$

Binary Adder (Asynchronous Ripple-Carry Adder)

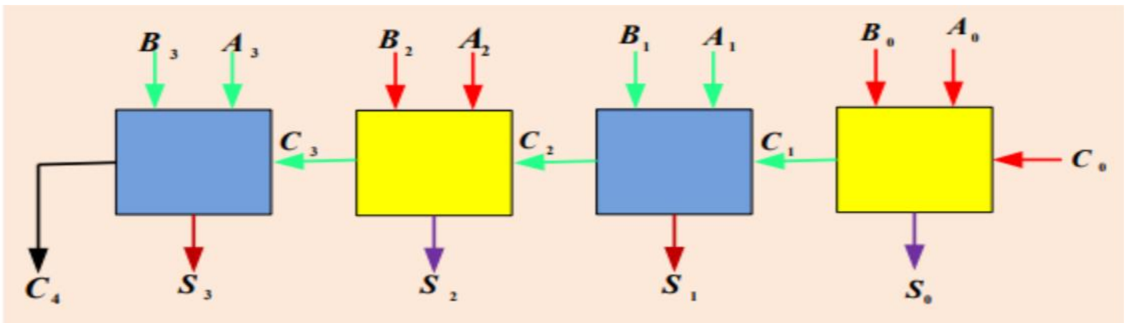




Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0+0+0=00_{two}$
0	0	1	0	1	$0+0+1=01_{two}$
0	1	0	0	1	$0+1+0=01_{two}$
0	1	1	1	0	$0+1+1=10_{two}$
1	0	0	0	1	$1+0+0=01_{two}$
1	0	1	1	0	$1+0+1=10_{two}$
1	1	0	1	0	$1+1+0=10_{two}$
1	1	1	1	1	$1+1+1=11_{two}$

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- A binary adder can be constructed with full adders connected in¬ cascade with the output carry form each full adder connected to the input carry of the next full adder in the chain.
- The four-bit adder is a typical example of a standard component .It can¬ be used in many application involving arithmetic operations.
- The input carry to the adder is and it ripples through the full adders to the output carry

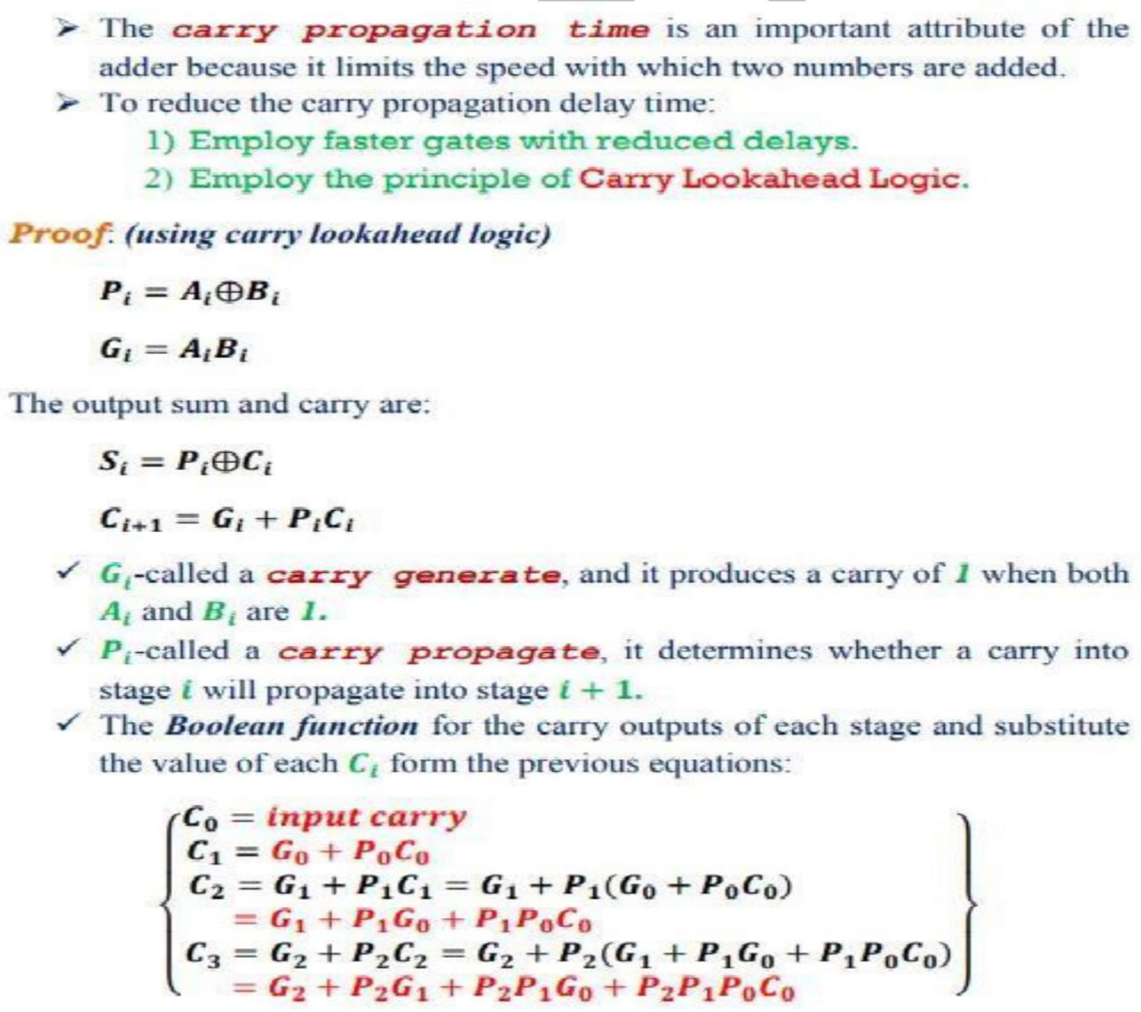
bit binary adder requires full adders



Four-bit Adder (Ripple Carry Adder)

CARRY-LOOK AHEAD ADDER

- Fast adder circuit must speed up the generation of carry signals. Carry look ahead logic uses the concepts of generating and propagating carries. Where Si is the sum and Ci+1 is the carry out.



➢ The ***carry propagation time*** is an important attribute of the adder because it limits the speed with which two numbers are added.

➢ To reduce the carry propagation delay time:
  1) Employ faster gates with reduced delays.
  2) Employ the principle of Carry Lookahead Logic.

Proof: ***(using carry lookahead logic)***

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The output sum and carry are:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

✓ G_i-called a ***carry generate***, and it produces a carry of ***1*** when both A_i and B_i are ***1***.

✓ P_i-called a ***carry propagate***, it determines whether a carry into stage i will propagate into stage $i+1$.

✓ The ***Boolean function*** for the carry outputs of each stage and substitute the value of each C_i form the previous equations:

$$\left\{\begin{array}{l} C_0 = input\ carry \\ C_1 = G_0 + P_0 C_0 \\ C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) \\ \quad = G_1 + P_1 G_0 + P_1 P_0 C_0 \\ C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ \quad = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{array}\right\}$$

- The three Boolean functions C_1, C_2 and C_3 are implemented in the ***carry lookahead generator.***

The two level-circuit for the output carry C_4 is not shown, it can be easily derived by the equation.

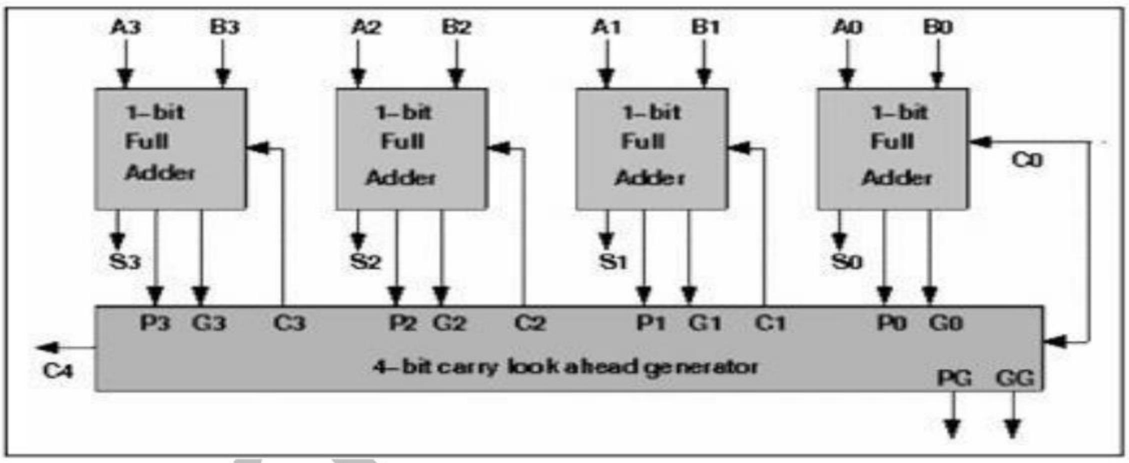
- C_3 does not have to wait for C_2 and C_1 to propagate, in fact C_3 is propagated at the same time as C_1 and C_2.

Adv:

1. Circuit is simplicity
2. Structure is slightly faster
3. Easy to understand
4. To eliminate inter stage carry delay.

Dadv:

1. Carry look-ahead is expensive

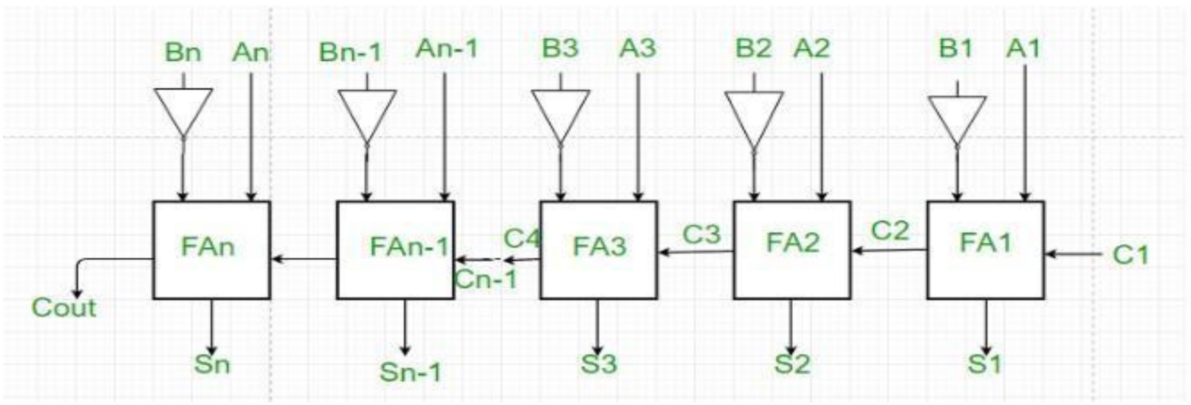


Carry Propagation Delay

The sum and carry output of any stage cannot be produced until the input carry occurs. This leads to a time delay in the addition process.

Parallel Subtractor

- A Parallel Subtractor is a digital circuit capable of finding the arithmetic difference of two binary numbers that is greater than one bit in length by operating on corresponding pairs of bits in parallel.
- The parallel subtractor can be designed in several ways including combination of half and full subtractors, all full subtractors or all full adders with subtrahend complement input.



Advantages of parallel Adder/Subtractor

- The parallel adder/subtractor performs the addition operation faster as compared to serial adder/subtractor.
- Time required for addition does not depend on the number of bits.
- The output is in parallel form i.e all the bits are added/subtracted at the same time.
- It is less costly.

Disadvantages of parallel Adder/Subtractor

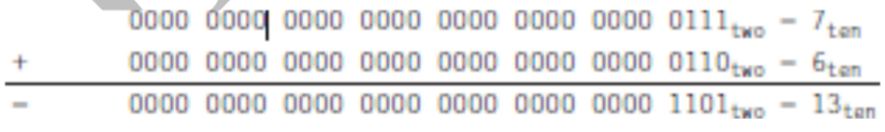
- Each adder has to wait for the carry which is to be generated from the previous adder in chain.
- The propagation delay(delay associated with the travelling of carry bit) is found to increase with the increase in the number of bits to be added.

ADDITION AND SUBTRACTION:

- Digits are added bit by bit from right to left, with carries passed to the next digit to the left.
- Subtraction uses addition. The appropriate operand is simply negated before being added.

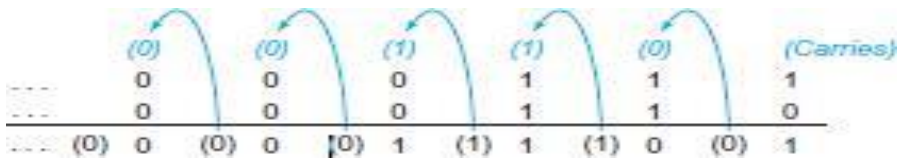
Binary addition:

Let's try adding 6_{ten} to 7_{ten} in binary.



```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
------------------------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 1101two = 13ten
```

The following figure shows the sums and carries. The carries are shown in parentheses.



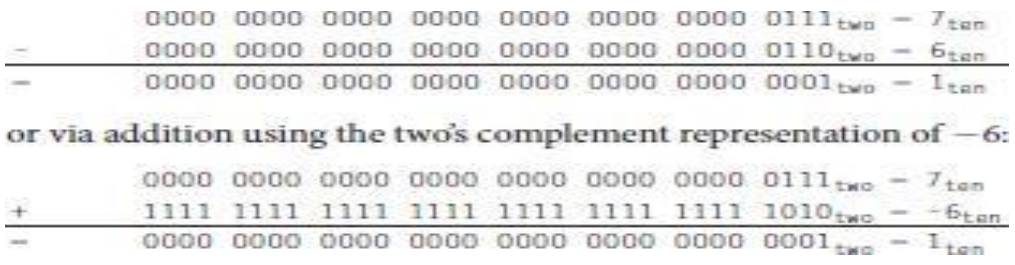
```
          (0)      (0)      (1)      (1)      (0)      (Carries)
. . .      0        0        0        1        1        1
. . .      0        0        0        1        1        0
. . . (0)  0   (0)  0   (0)  1   (1)  1   (1)  0   (0)  1
```

- Binary addition, showing 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.

Binary subtraction:

- Subtracting 6_{ten} from 7_{ten} can be done directly



```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
– 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

or via addition using the two's complement representation of −6:

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 1111 1111 1111 1111 1111 1111 1111 1010two = –6ten
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

When overflow cannot occur in addition and subtraction?

Case: 1

- When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10+4=-6.
- Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

Case: 2

- When the signs of the operands are the same, overflow cannot occur. To see this, remember that c – a= c + (-a) because we subtract by negating the second operand and then add.
- Therefore, when we subtract operands of the same sign we end up by adding operands of different signs.

When overflow can occur in addition and subtraction?

Case: 1

- Overflow occurs when adding two positive numbers and the sum is negative

Case: 2

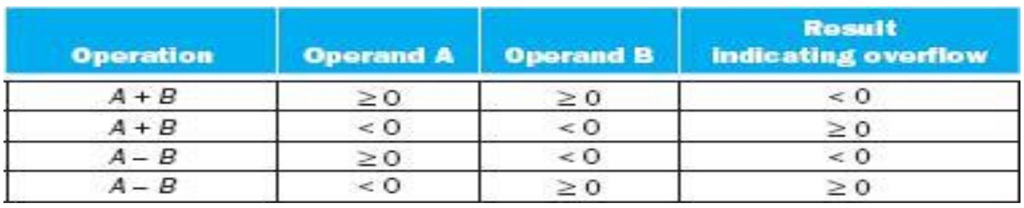
- Overflow occurs when adding two negative numbers and the sum is positive. This spurious sum means a carry out occurred into the sign bit.

Case: 3

- Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result.

Case: 4

- When we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a 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

- Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.
- Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned (subu) do not cause exceptions on overflow.

Exception:

- Exception also called interrupt on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt:

- An exception that comes from outside of the processor.

EPC:

- MIPS include a register called the Exception Program Counter (EPC) to contain the address of the instruction that caused the exception.
- The instruction move from system control (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the off ending instruction via a jump register instruction.

MULTIPLICATION

- The first operand is called the multiplicand and the second the multiplier. The final result is called the product.
- If we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n+ m bits long.
- That is, n+ m bits are required to represent all possible products.
- For example, **Multiplying** 1000_{ten} by 1001_{ten}: **Multiplicand** 1000_{ten} **Multiplier** 1001_{ten}

$$
\begin{array}{lr}
 & 1000_{ten} \times 1001_{ten} \\
\hline
 & 1000 \\
 & 0000 \\
 & 0000 \\
 & 1000 \\
\hline
\textbf{Product} & 1001000_{ten}
\end{array}
$$

Case: 1

- Just place a copy of the multiplicand (1 x multiplicand) in the proper place if the multiplier digit is a 1.

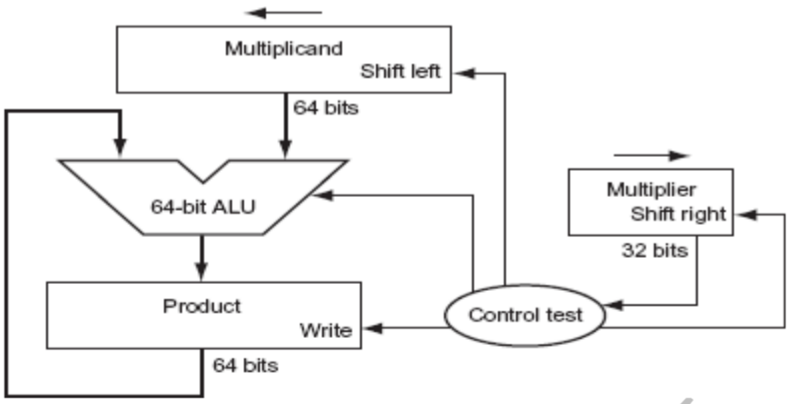
Case: 2

- Place 0 (0 x multiplicand) in the proper place if the digit is 0.

FIRST VERSION OF THE MULTIPLICATION HARDWARE

- The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
- The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step.
- The multiplier is shifted in the opposite direction at each step.
- The algorithm starts with the product initialized to 0.

- Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.



<u>Step: 1</u>

- The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.
- If the least significant bit of the multiplier is 1, add the multiplicand to the product.

<u>Step: 2</u>

- If not, go to the next step. Shift left the multiplicand register by 1 bit.

Step: 3

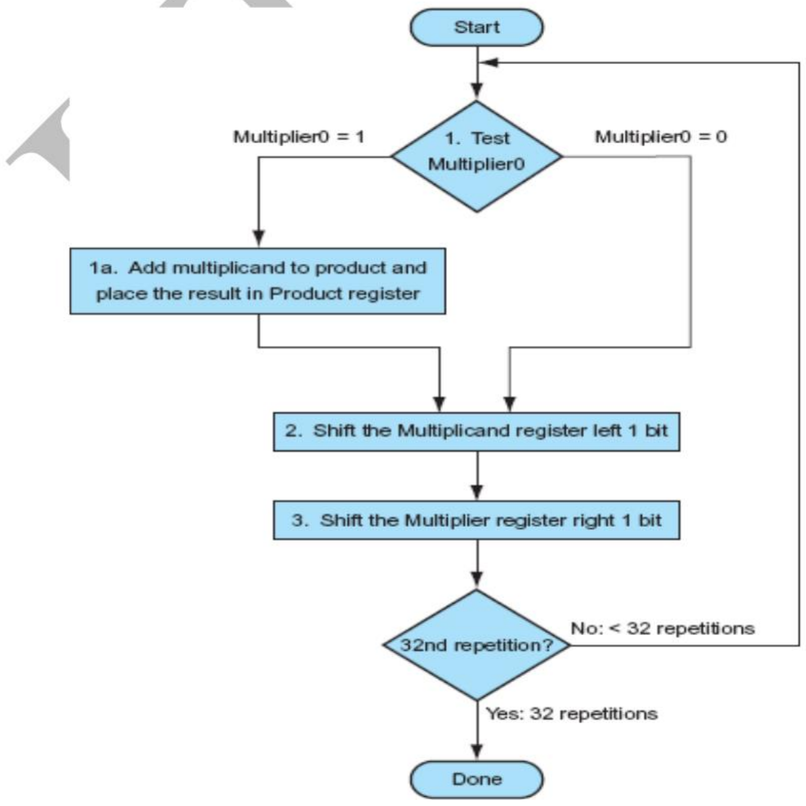
- Then shift right the multiplier register by 1 bit. These three steps are repeated 32 times to obtain the product.
- If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.

Example:

Using 4-bit numbers to save space, multiply 2ten x 3ten, or 0010two x 0011two.

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

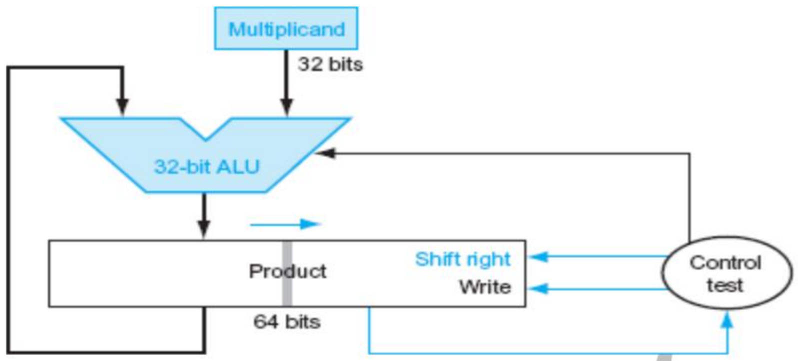
Flowchart:


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Refined version of the multiplication hardware:

- Comparing with the first algorithm the Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits.
- Now the product is shifted right. The separate Multiplier register also disappeared.

The multiplier is placed instead in the right half of the Product register.

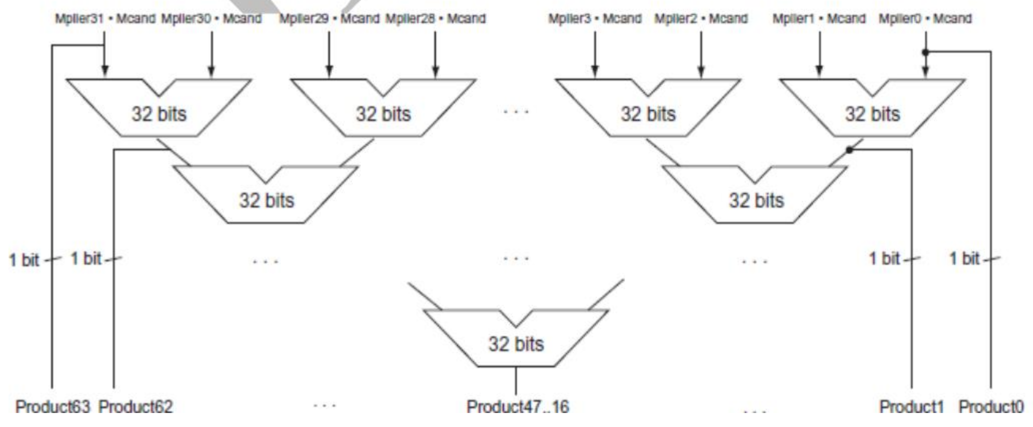


Signed Multiplication

- 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.

Faster Multiplication

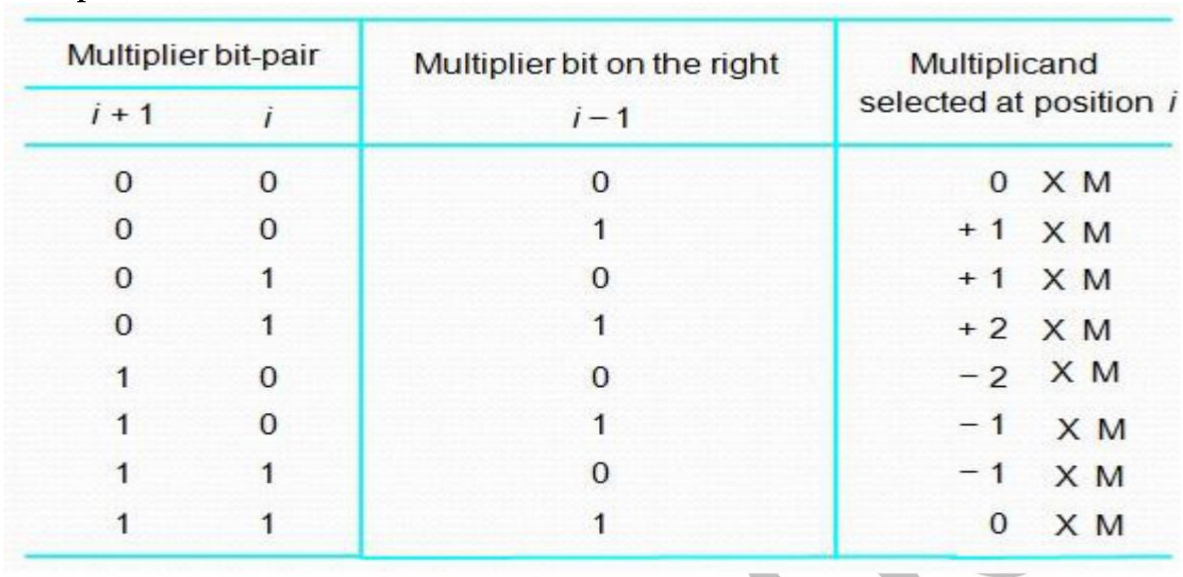
- Hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by the 32 multiplier bits.
- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier:
- One input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- To connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high.
- Rather than use a single 32-bit adder 31 times, this hardware “unrolls the loop” to use 31adders and then organizes them to minimize delay.



BOOTH’S BIT-PAIR RECODING OF THE MULTIPLIER.
A=+13 (Multiplicand) AND B= -6 (Multiplier)
Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).

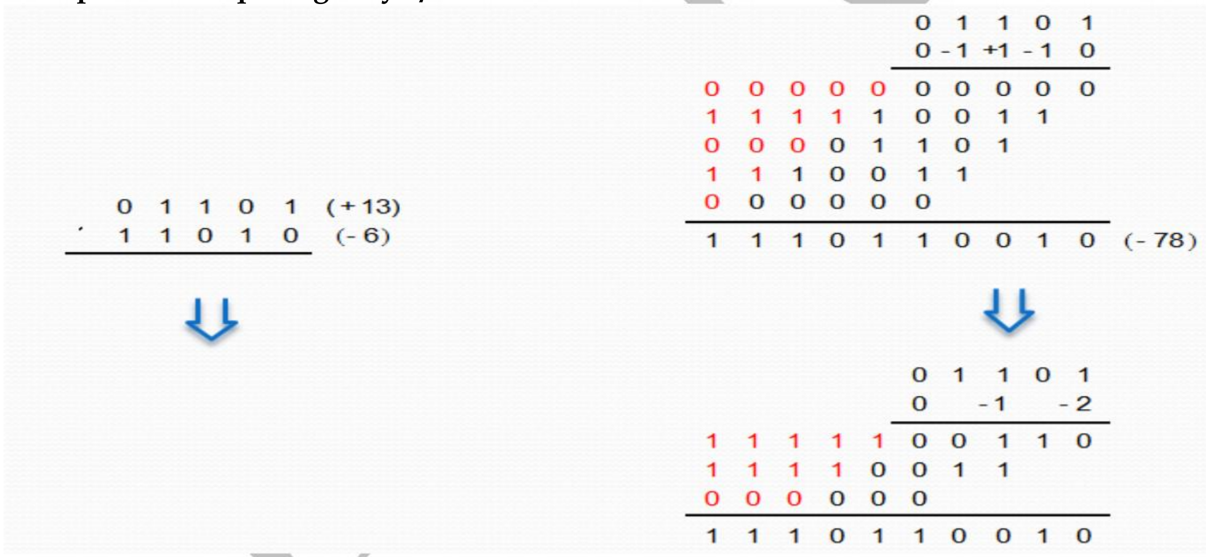
Example

Multiplicand Selection Decisions



Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	− 2 X M
1	0	1	− 1 X M
1	1	0	− 1 X M
1	1	1	0 X M

Multiplication requiring only n/2 summands



```
   0 1 1 0 1   (+13)
 ´ 1 1 0 1 0   (-6)
```

```
                0  1  1  0  1
                0 -1 +1 -1  0
 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  1  1
 0  0  0  0  1  1  0  1
 1  1  1  0  0  1  1
 0  0  0  0  0  0
 1  1  1  0  1  1  0  0  1  0   (-78)
```

```
                0  1  1  0  1
                0    -1    -2
 1  1  1  1  1  0  0  1  1  0
 1  1  1  1  0  0  1  1
 0  0  0  0  0  0
 1  1  1  0  1  1  0  0  1  0
```

BOOTH'S MULTIPLICATION ALGORITHM WITH SUITABLE EXAMPLE Booth's Algorithm Principle:

- Performs additions and subtractions of the Multiplicand, based on the value of the multiplier bits.
- The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed.

The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the multiplier.

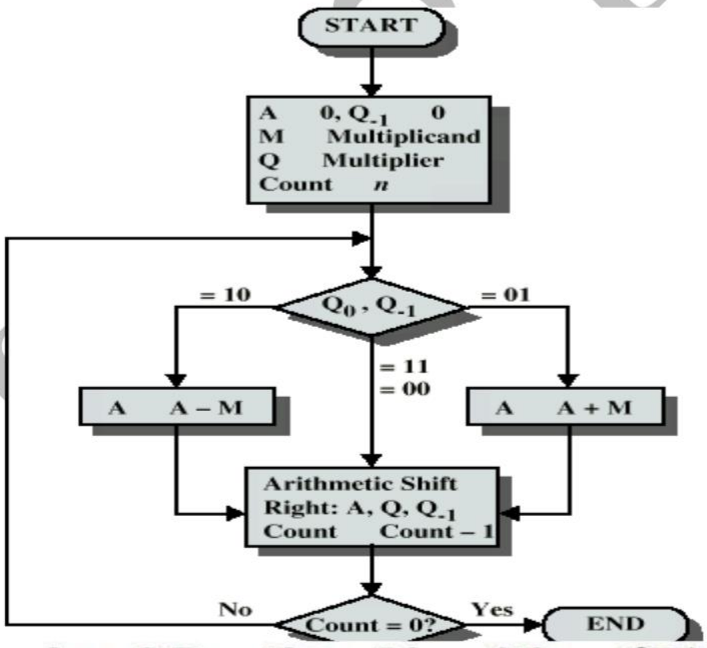
- If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated operations and shifts are performed on partial results, the result register is the accumulator (A).
- Booth‘s algorithm gives a procedure for multiplying signed binary integer. It is based on the fact that strings of 0‘s in the multiplier require no addition but only shifting and a string of 1‘s in the multiplier require both operations.

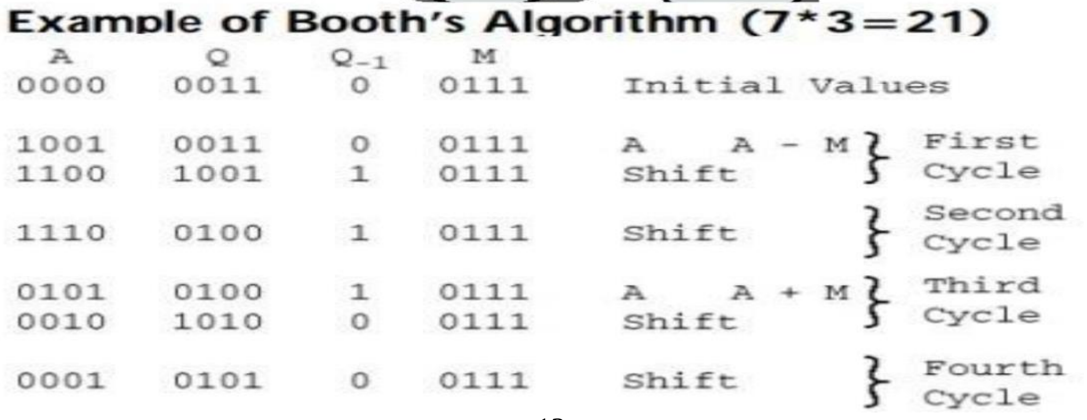
Algorithm

The Q0 bit of the register Q and Q-1 is examined:

- If two bits are the same (11 or 00), then all of the bits of the A, Q and Q_1 registers are shifted to the right 1 bit. This shift is called arithmetic shift right.
- If two bits differ i.e., whether 01, then the multiplicand is adder or 10, then the multiplicand is subtracted from the register A. after that, right shift occurs in the register A, Q and Q_1.

Flowchart of Booth's Algorithm for 2's complement multiplication





Example of Booth's Algorithm (7*3=21)

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	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

DIVISION ALGORITHM AND HARDWARE:

Dividend:

- A number being divided is called dividend.

Divisor:

- A number that the dividend is divided by is called divisor.

Quotient:

- It is called the primary result of a division.
- A number that when multiplied by the divisor and added to the remainder produces the dividend is known as quotient.

Remainder:

- It is the secondary result of a division.
- A number that when added to the product of the quotient and the divisor produces the dividend is known as remainder.

The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

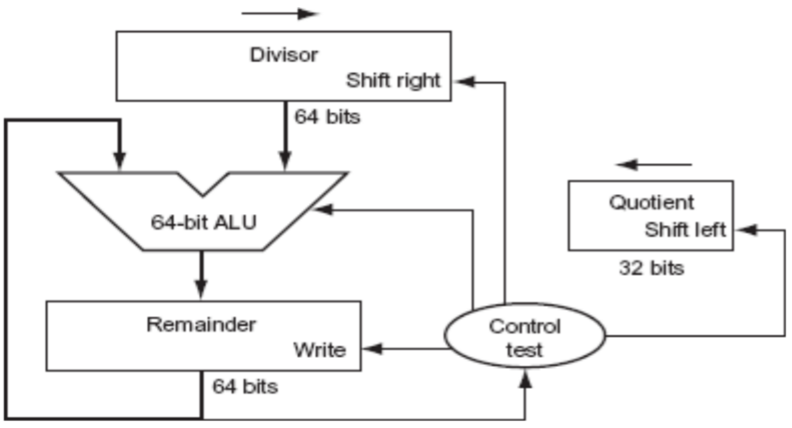
```
                   1001ten        Quotient
Divisor 1000ten ⌐1001010ten       Dividend
               −1000
                  10
                  101
                  1010
                 −1000
                    10ten         Remainder
```

- 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:

$$\boxed{\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}}$$

Division Hardware:

- The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits.
- The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration.
- The remainder is initialized with the dividend.
- Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.



Divide algorithm:

Step: 1

- It must first subtract the divisor register from the Remainder register and place the result in the Remainder register.

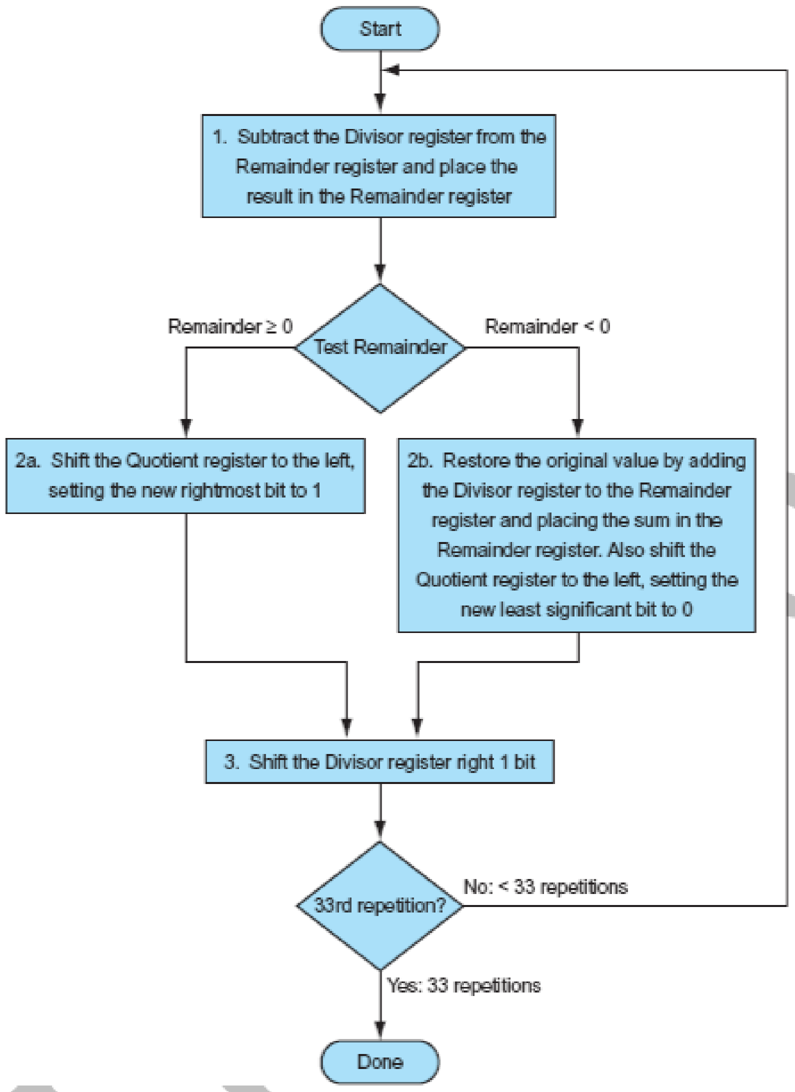
Step: 2

- Next we performed the comparison in the set on less than instruction.
- If the result is positive, the divisor was smaller or equal to the dividend, so shift the Quotient register to the left, setting the new rightmost bit to 1.
- If the result is negative, the next step Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register.
- Also shift the Quotient register to the left, setting the new least significant bit to 0

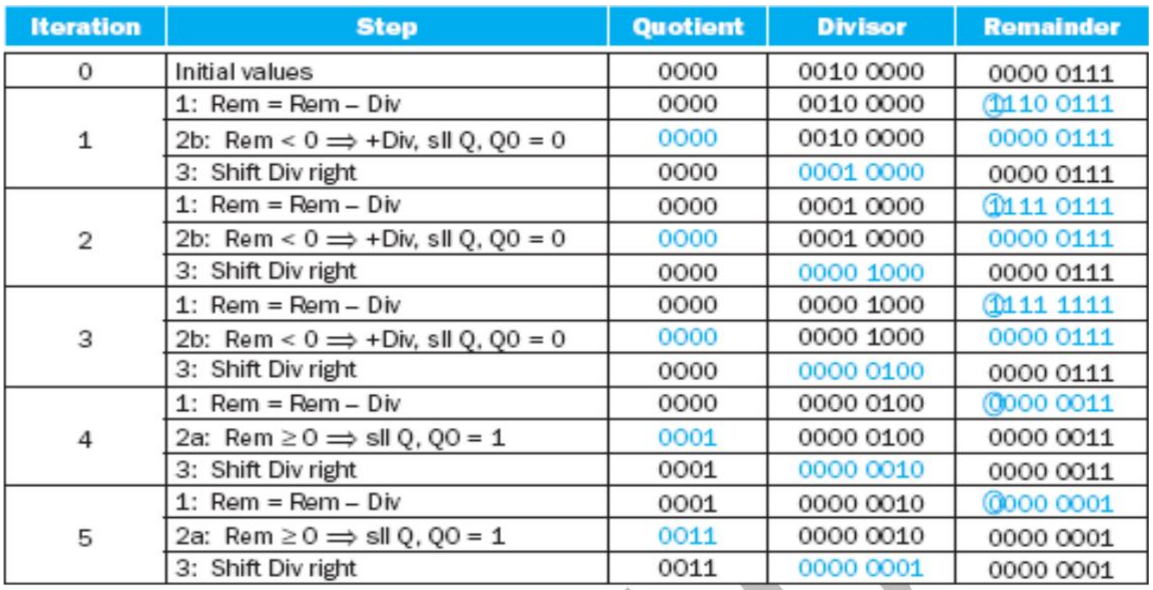
Step: 3

- The divisor is shifted right by 1 bit and then we iterate again.

- The remainder and quotient will be found in their registers after the iterations are complete.


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

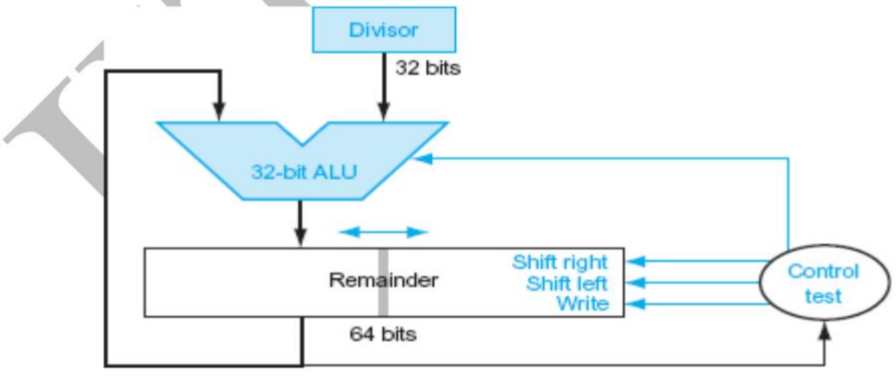
Using a 4-bit version of the algorithm to save pages, let's try dividing 7ten by 2ten, or 0000 0111two by 0010two.



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

<u>An improved version of the division hardware:</u>

- The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.
- Compared to above division hardware, the ALU and Divisor registers are halved and the remainder is shifted left.
- This version also combines the Quotient register with the right half of the Remainder register.



<u>Signed Division</u>

- The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.
- The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{\text{ten}}$ by $\pm 2_{\text{ten}}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3, + \text{Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2) \\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3, \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

Rule:

- The dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

$$-(x \div y) \neq (-x) \div y$$

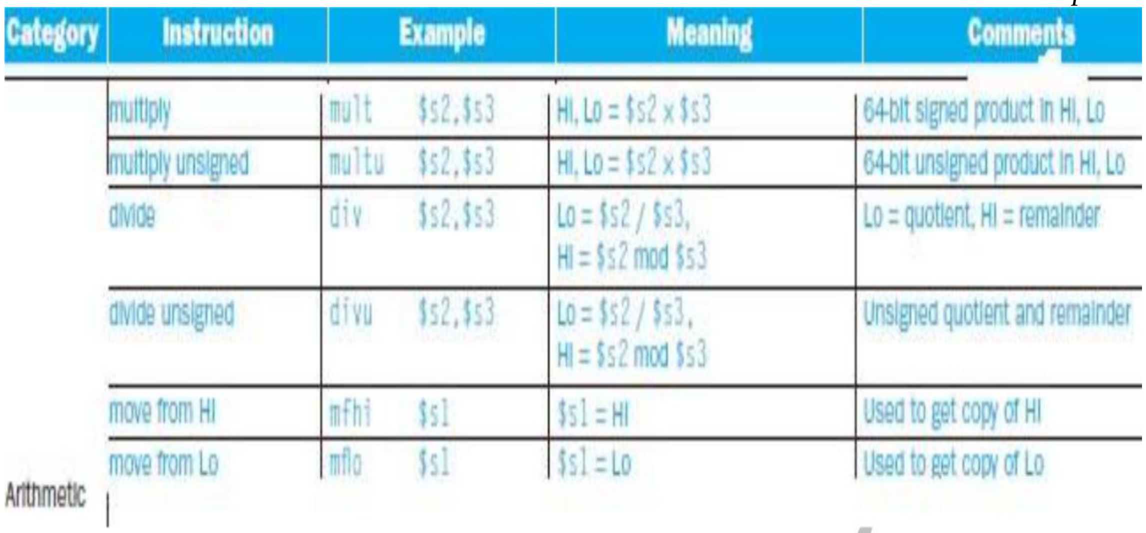
We calculate the other combinations by following the same rule:

$$+7 \div -2\text{: Quotient} = -3, \text{Remainder} = +1$$

$$-7 \div -2\text{: Quotient} = +3, \text{Remainder} = -1$$

Faster Division

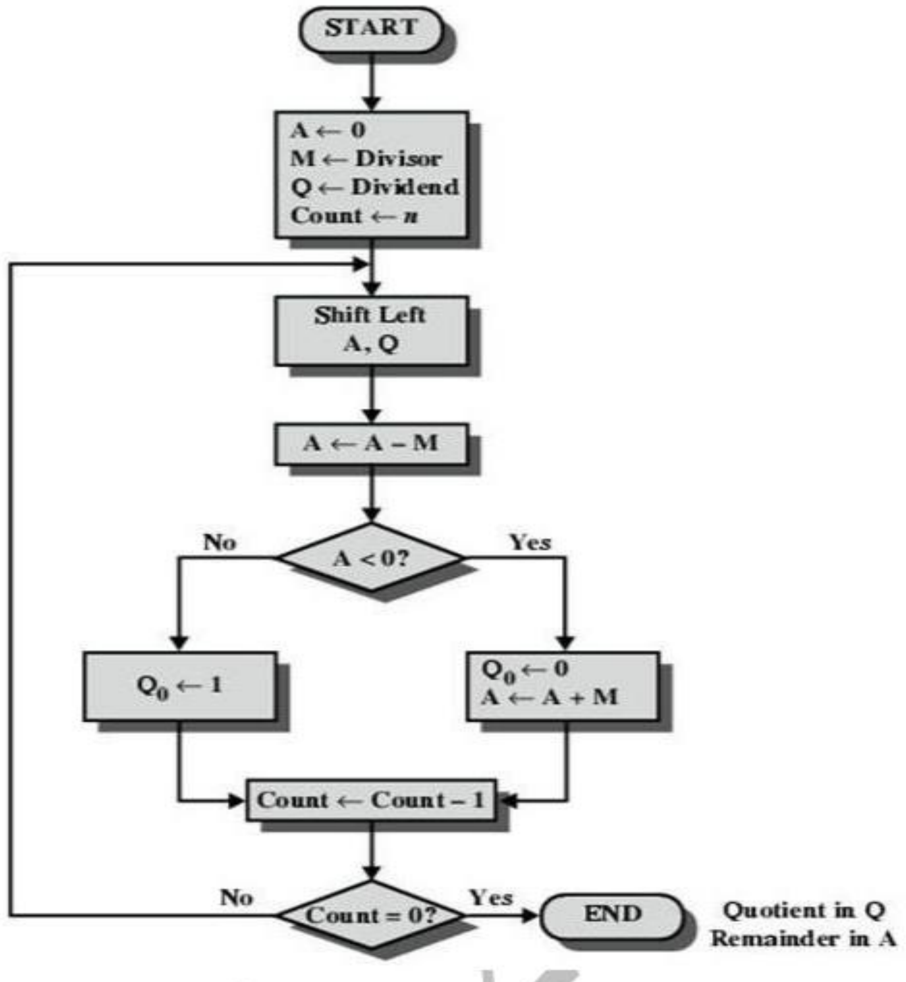
- There are techniques to produce more than one bit of the quotient per step.
- The SRT division technique tries to predict several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder.
- These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.



Category	Instruction	Example	Meaning	Comments
Arithmetic	multiply	mult $s2,$s3	Hi, Lo = $s2 x $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 x $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo

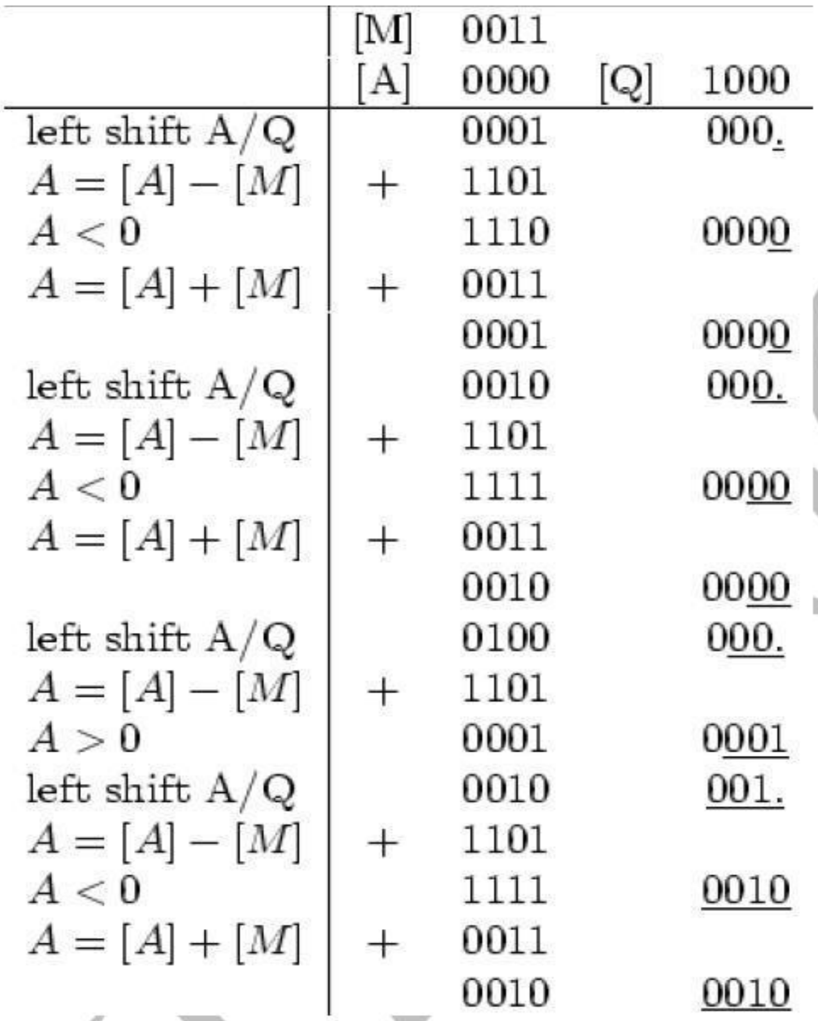
<u>Restoring Division Algorithm</u>

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat fro step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder


START
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left A, Q
A ← A – M
A < 0?
No
Yes
Q0 ← 1
Q0 ← 0
A ← A + M
Count ← Count – 1
Count = 0?
No
Yes
END
Quotient in Q
Remainder in A

Example: 8 divide by 3=2 (2/3)

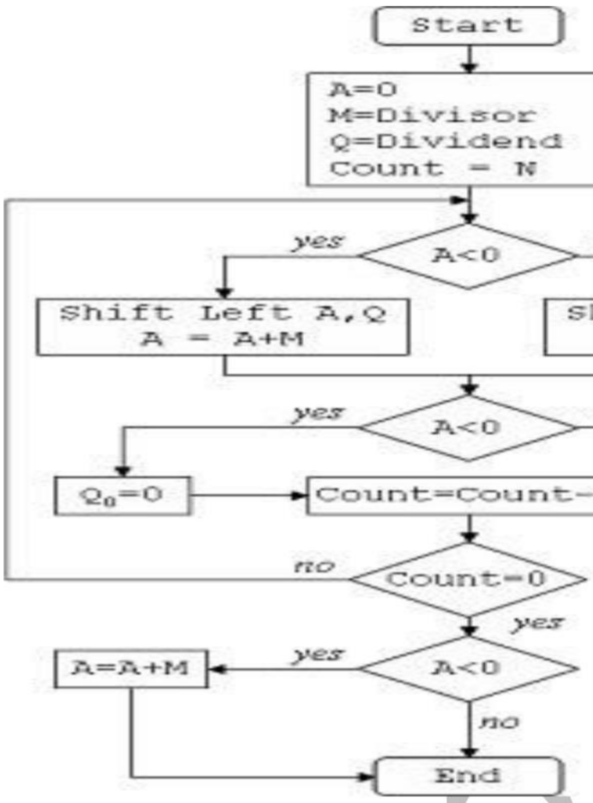
The quotient (0010)2 = 2 is in register Q, and the reminder (0010)2 = 2 is in register A.

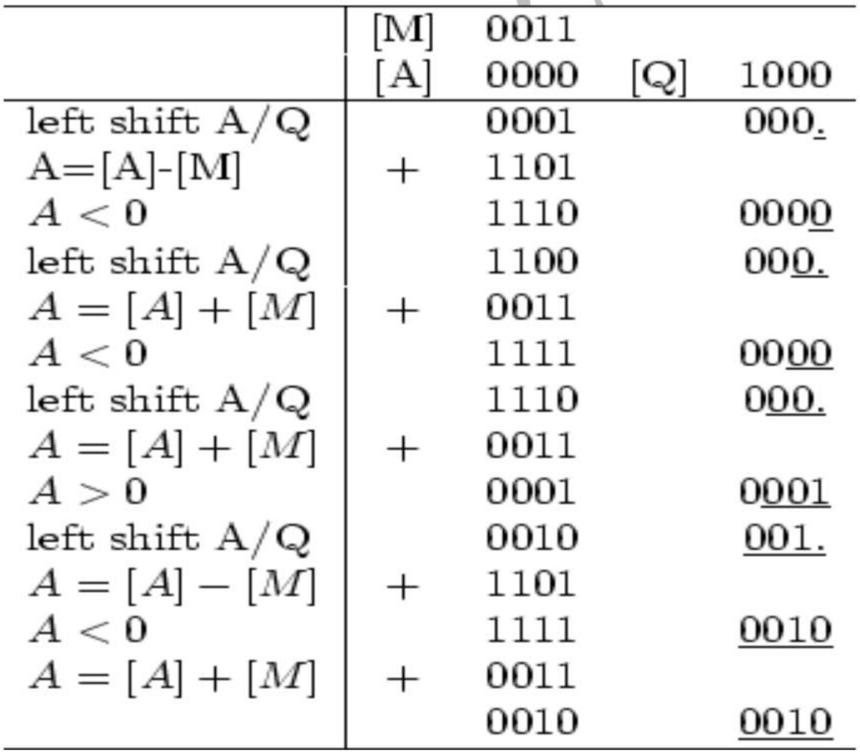


	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1110		0000
$A = [A] + [M]$	+	0011		
		0001		0000
left shift A/Q		0010		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0000
$A = [A] + [M]$	+	0011		
		0010		0000
left shift A/Q		0100		000.
$A = [A] - [M]$	+	1101		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

Non-Restoring Division Algorithm

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder





	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
A=[A]-[M]	+	1101		
$A < 0$		1110		0000
left shift A/Q		1100		000.
$A = [A] + [M]$	+	0011		
$A < 0$		1111		0000
left shift A/Q		1110		000.
$A = [A] + [M]$	+	0011		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

FLOATING POINT

Normalized number:

- A number in floating-point notation that has no leading 0^s is known as normalized number. i.e., a number start with a single nonzero digit.
- For example, $1.0_{ten}\times10^{-9}$ is in normalized scientific notation, but $0.1_{ten}\times10^{-8}$ and $10.0_{ten}\ \times10^{-10}$ are not.

Binary numbers in scientific notation:

- To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. $1.0_{two}\times2^{-1}$

Floating-Point Representation

Floating point:

- Computer arithmetic that represents numbers in which the binary point is not fixed.

Fraction:

- The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.

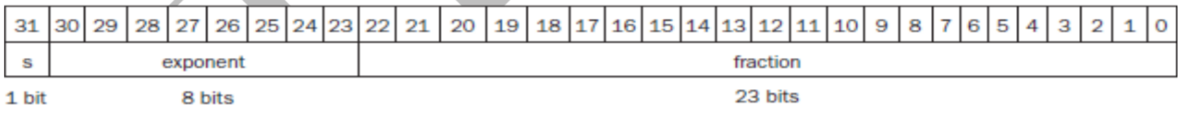
Exponent:

- In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Single precision:

- A floating-point value represented in a single 32-bit word. Floating-point numbers are usually a multiple of the size of a word.
- Where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.
- F involves the value in the fraction field and E involves the value in the exponent field.

Format:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

Overflow:

- A situation in which a positive exponent becomes too large to fit in the exponent field is known as overflow.

Underflow:

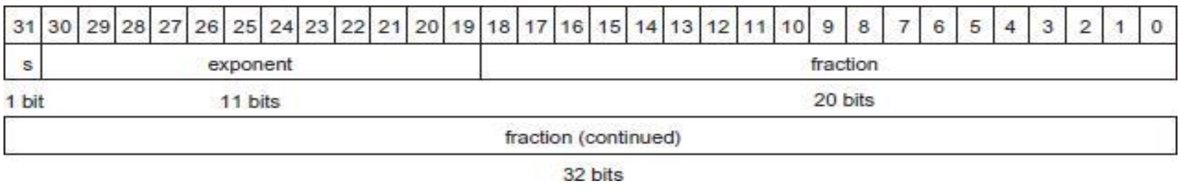
- A situation in which a negative exponent becomes too large to fit in the exponent field is known as underflow.

Double precision:

- One way to reduce chances of underflow or overflow is called double, and operations on doubles are called double precision floating-point arithmetic.
- It has a larger exponent. A floating-point value represented in two 32-bit words.

- 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.

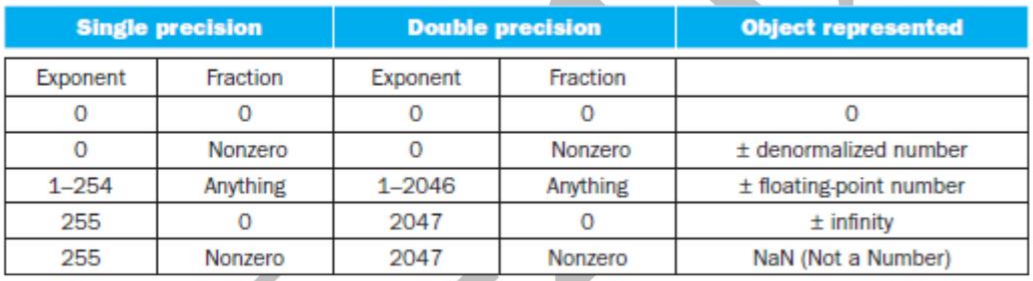
Format:



IEEE 754 Format:

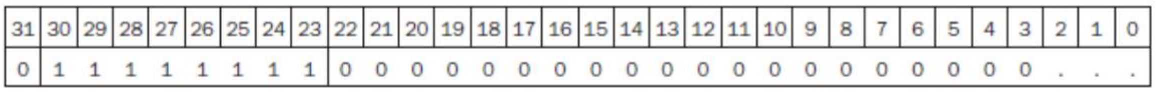
- MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{+308}$ and almost as large as $2.0_{ten} \times 10^{-308}$.
- Although double precision does increase the exponent range.
- Its primary advantage is its greater precision because of the much larger fraction.
- IEEE 754 makes the leading 1-bit of normalized binary numbers implicit.
- Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1+52).

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



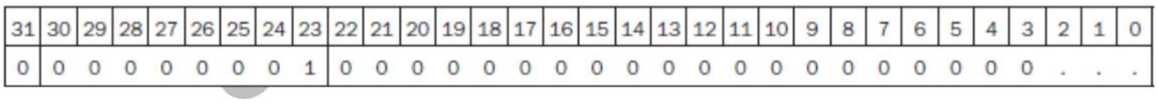
Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

For example, $1.0_{two} \times 2^{-1}$ would be represented as



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

- The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$.
- This convention is called biased notation, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.
- IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1+127_{ten}$, or 126_{ten}=$0111\ 1110_{two}$, and +1 is represented by 1+127, or $128_{ten} = 1000\ 0000_{two}$.
- The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{\text{two}} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{\text{two}} \times 2^{+127}.$$

Example: 1

Floating-Point Representation

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

The number -0.75_{ten} is also

$$-3/4_{\text{ten}} \text{ or } -3/2^2_{\text{ten}}$$

It is also represented by the binary fraction

$$-11_{\text{two}}/2^2_{\text{ten}} \text{ or } -0.11_{\text{two}}$$

In scientific notation, the value is

$$-0.11_{\text{two}} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{\text{two}} \times 2^{-1}$$

The general representation for a single precision number is

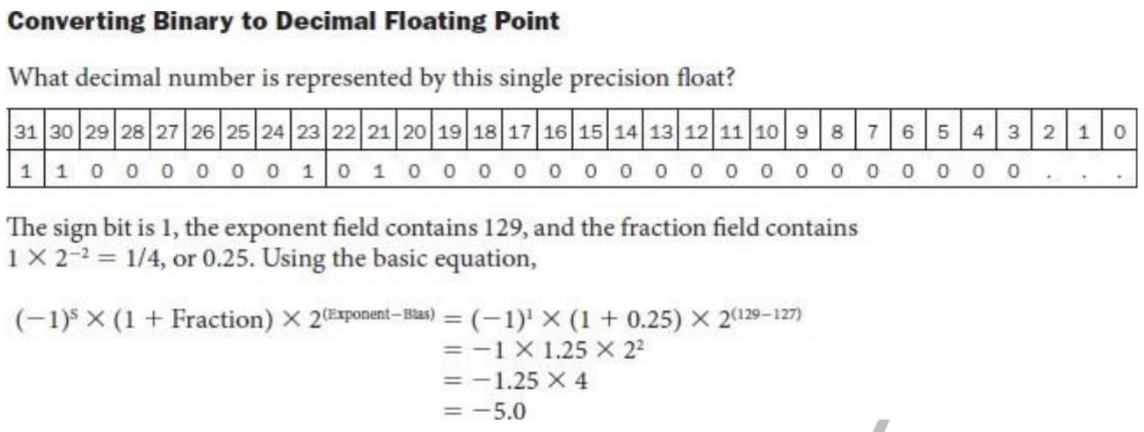
$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{\text{two}} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

Example: 2



Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)}\\ &= -1 \times 1.25 \times 2^2\\ &= -1.25 \times 4\\ &= -5.0\end{aligned}$$

FLOATING-POINT ADDITION

- Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Example: 1

Perform floating point addition for the following numbers.

$$\mathbf{9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}}$$

Step 1:

- Compare the exponent of both the operands.
- If it equal add the two operand (significand) .If it is not equal then increase the smaller exponent.
- i.e., shift the smaller number to the right until its exponent would match the larger exponent. As per our example

  $\mathbf{1.610_{ten} \times 10^{-1}} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}$
- But we can represent only four decimal digits so, after shifting, the number is really

  $0.016_{ten} \times 10^{1}$

Step 2:

Now add the Significand

$$\begin{array}{r} 9.999 \times 10^{1} \\ 0.016 \times 10^{1} \\ \hline 10.015 \times 10^{1} \end{array}$$

Step 3:

- Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.
- This sum is not in normalized scientific notation, so we need to adjust it:

  $10.015 \times 10^{1} = 1.0015 \text{ X } 10^{2} = 1.0015 \text{ X } 10^{2}$
- Whenever the exponent is increased or decreased, we must check for overflow or underflow. i.e., we must make sure that the exponent still fits in its field.

Step 4:

- Round the significand to the appropriate number of bits.
- If the sum may no longer be normalized and we would need to perform step 3 again.

  $\mathbf{1.002_{ten} \times 10^{2}}$

Example: 2

Perform floating point addition for the following numbers.

0.5_{ten} and -0.4375_{ten}

Solution:

Assuming that we keep 4 bits of precision. 0.5_{ten}

Operand 1: Convert the operands to binary

$0.5 \times 2 = 1.0$

Scientific Notation

$\mathbf{0.1 \times 2^{0}}$

Normalizing the above value

$\mathbf{1.0 \times 2^{-1}}$

Operand 2: Convert the operands to binary -0.4375_{ten}

$0.4375 \times 2 = 0.8750$

$0.8750 \times 2 = 1.7500$

$0.7500 \times 2 = 1.5000$

$1.5000 \times 2 = 1.0000$

Scientific Notation

$\mathbf{0.0111 \times 2^{0}}$

Normalizing the above value

$\mathbf{1.110 \times 2^{-2}}$

Step 1:

The significand of the number with the lesser exponent ($-1.110_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2:

Add the significands

$$\begin{array}{rl} & 1.000 \times 2^{-1} \\ - & 0.111 \times 2^{-1} \text{ [Subtraction]} \\ \hline & 0.001 \times 2^{-1} \end{array}$$

Step 3: Normalize the sum and checking for overflow or underflow

$0.001 \times 2^{-1} = 1.0 \times 2^{-4}$

Step 4: Round the sum

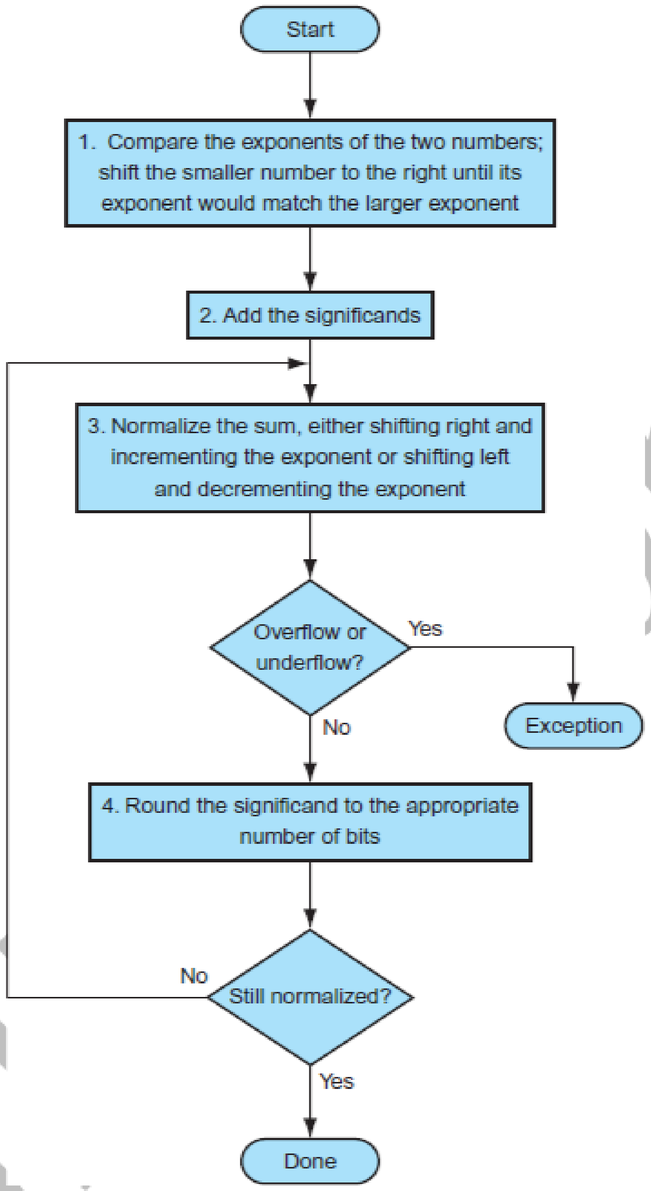
$$\mathbf{1.000 \times 2^{-4}}$$

Then convert the sum to decimal

$\mathbf{1.000 \times 2^{-4} = 0.0001_{two}}$

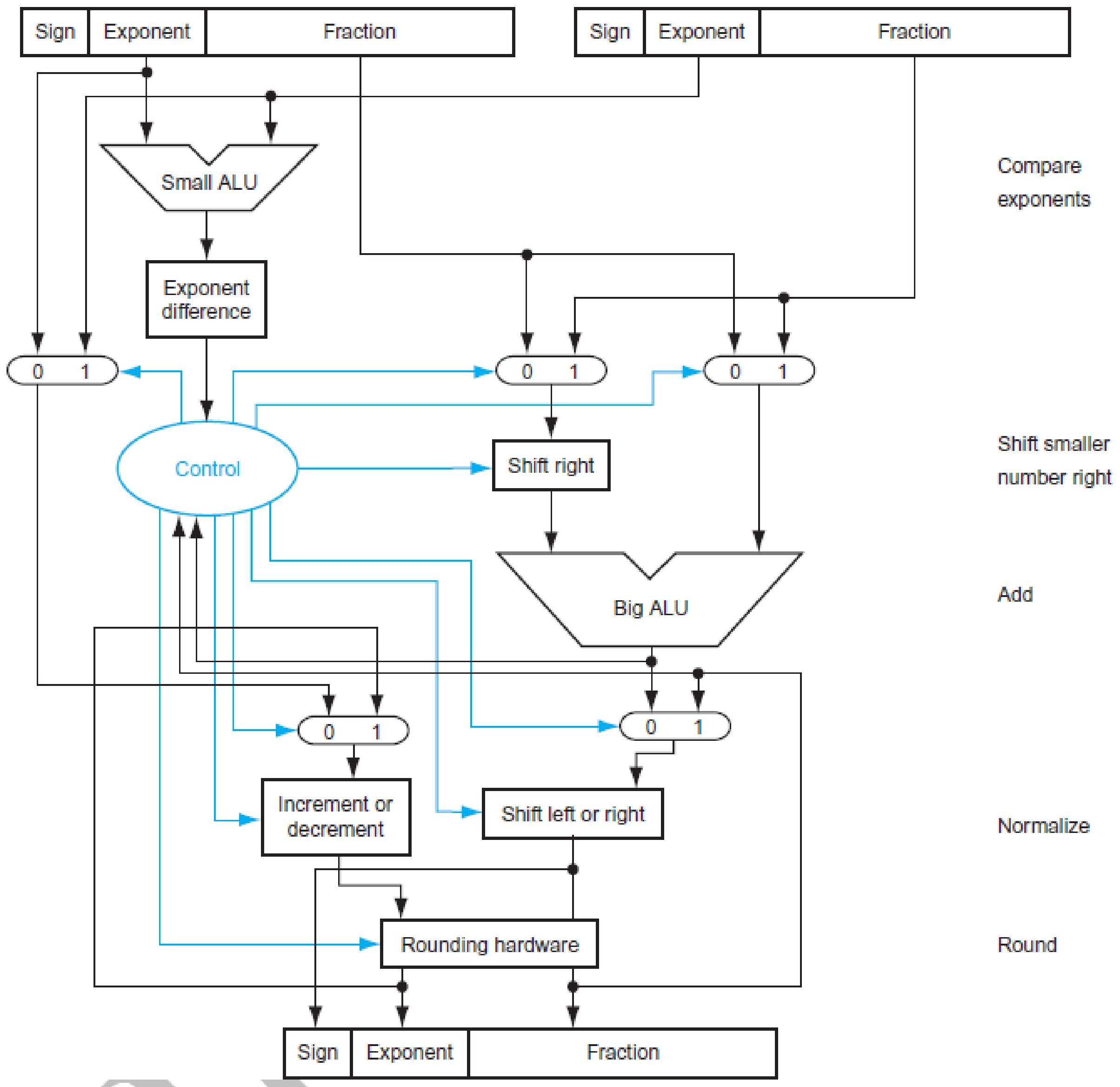
$\mathbf{1 / 2^{4} = 1/16_{ten} = 0.0625_{ten}}$

Flowchart:


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
Still normalized?
No
Yes
Done

- First, 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 significand of the smaller number, and the significand of the larger number.
- The smaller significand is shifted right, and then the significands 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 actual final result.

Block Diagram:



FLOATING-POINT MULTIPLICATION

Example: 1

Multiplying decimal numbers in scientific notation:

$\mathbf{1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}}$

Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1:

- We calculate the exponent of the product by simply adding the exponents of the operands together:

  New exponent = 10 + (-5) = 5

- Let's do this with the biased exponents as well to make sure we obtain the same result:

  10 + 127 = 137, and -5 + 127 = 122, so New exponent = 137 +122=259

- This result is too large for the 8-bit exponent field.
- The problem is with the bias because we are adding the biases as well as the exponents.

New exponent= (10+127)+(-5+127)=(5+2X127)=259

- To get the correct biased sum when we add biased numbers, we must subtract the bias from the sum.
- New exponent=137+122-127=259-127=132= (5+127) and 5 is indeed the exponent we calculated initially.

Step 2:

Next comes the multiplication of the significands:

```
        1.110ten
    X   9.200ten
   ____________
        0000
       0000
      2220
     9990
  _____________
The product is    10212000ten
```

- Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212ten x 105

Step 3:

This product is unnormalized, so we need to normalize it:

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

- After the multiplication, the product can be shifted right one digit and adding 1 to the exponent.
- At this point, we can check for overflow and underflow. Underflow may occur if both operands are small, that is, if both have large negative exponents.

Step 4:

Round of the Product

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

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

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

Example: 2

Multiple the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in the above algorithm

- Binary equivalent of $0.5_{ten} = 1.000 \times 2^{-1}$ and $-0.4375_{ten} = -1.110 \times 2^{-2}$

Step 1 :

Adding the Exponents without

bias -1 + (-2) = -3

Or using the biased representation

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2 : Multiplying the significands

$$
\begin{array}{r}
1.000_{two} \\
\times \quad 1.110_{two} \\
\hline
0000 \\
1000 \\
1000 \\
1000 \\
\hline
1110000_{two}
\end{array}
$$

The Product is $1.110000_{two} \times 2^{-3}$,but we use 4 bits, so it is $1.110_{two} \times 2^{-3}$

Step 3:

Normalize the product, as per our example it is already normalized

one 1.110×2^{-3}

Step 4:

Rounding the product no change

1.110 x 2-3

Step 5:

Since the signs of the original operands differ, make the sign of the product negative.

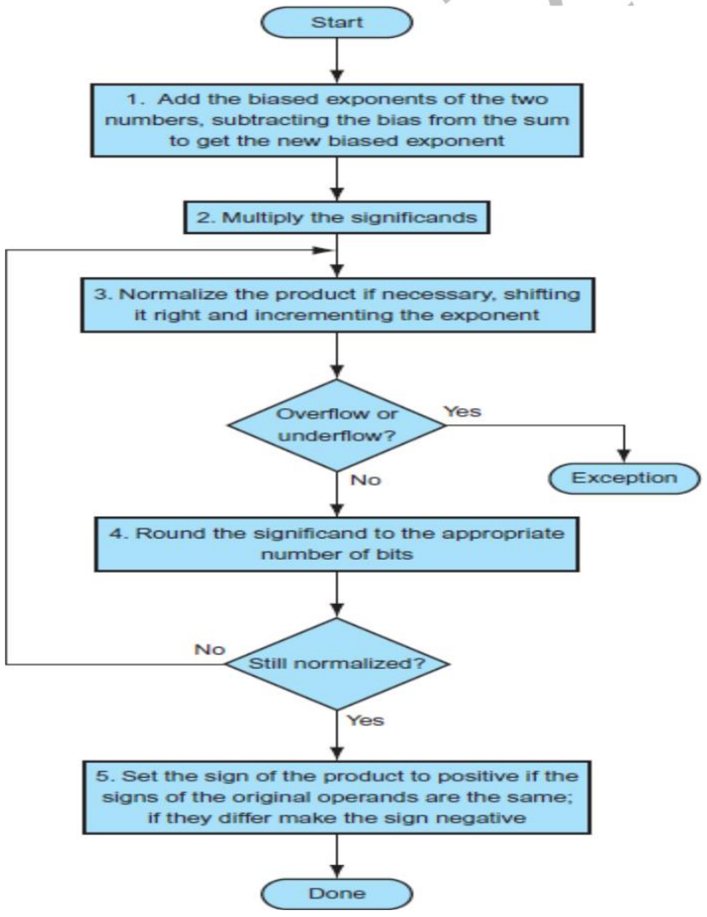
Hence, the product is

-1.110 x 2-3

Converting to decimal to check our results:

-1.110 x 2-3 = - 0.00111 = 1 /8 + 1 / 16 + 1 / 32 = - 0.21875

Flow Chart:


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Guard Bit:

- Extra bits kept on the right during intermediate calculations of floating point numbers is called guard bit and it used to improve rounding accuracy.

Round:

- Method to make the intermediate floating-point result fit the floating-point format.
- The goal is typically to find the nearest number that can be represented in the format.

Sticky Bit:

- A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Subword Parallelism:

- By partitioning the 128-bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bitoperands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.
- The cost of such partitioned adders was small.
- Given that the parallelism occurs within a wide word, the extensions are classified as subword parallelism.
- It is also classified under the more general name of data level parallelism.
- They have been also called vector or SIMD, for single instruction, multiple data.