UNIT II ARITHMETIC OPERATIONS

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

CHAPTER 3
ARITHMETIC FOR COMPUTERS

3.1 Introduction

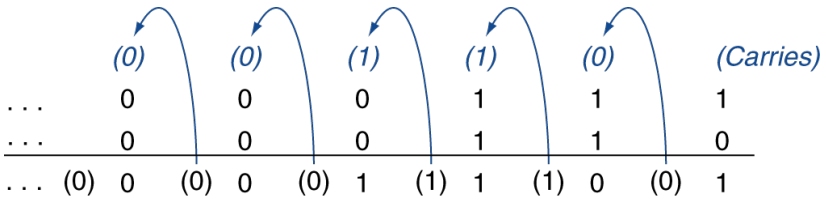
- Computer words are composed of bits; thus words can be represented as binary numbers.
  - How are negative numbers represented?
  - What is the largest number that can be represented in a computer word?
  - What happens if an operation creates a number bigger than can be represented?
  - What about fractions and real numbers?

3.2 Addition and Subtraction

- Addition
  - Digits are added bit by bit from right to left
  - carries passed to the next digit to the left
- Subtraction
  - Uses addition
  - Appropriate operand is simply negated before added

- Binary Addition and Subtraction

Example : Adding $7_{10} + 6_{10}$ in binary



Example: Subtracting $7_{10} - 6_{10}$ in binary

- Can be done directly
- Via addition using the two's complement representation of -6

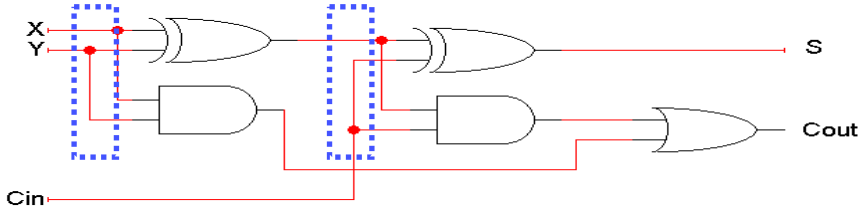
+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

Full Adder

- A full adder circuit takes three bits of input, and produces a two-bit output consisting of a sum and a carry out.

$S = X \oplus Y \oplus Cin$

$Cout = (X \oplus Y)\, Cin + XY$



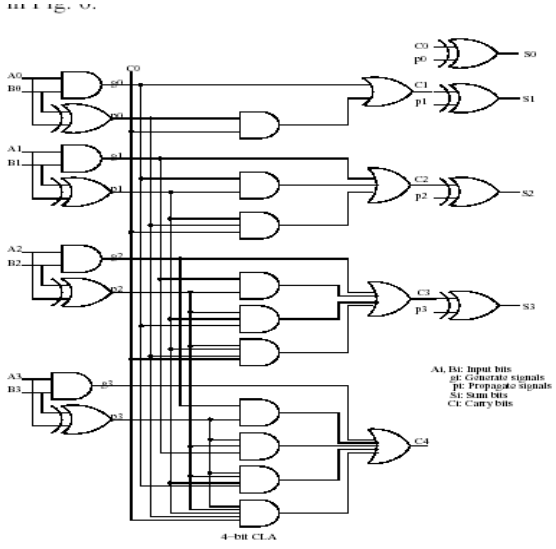
Carry look ahead adder.

- The CLA is used in most ALU designs
- It is faster compared to ripple carry logic adders or full adders especially when adding a large number of bits.
- The Carry Look Ahead Adder is able to generate carries before the sum is produced using the propagate and generate logic to make addition much faster.

$$G_i = A_i.B_i \qquad Pi = (A_i \oplus B_i)$$

$$C_1 = G_0 + P_0.C_0$$
$$C_2 = G_1 + P_1.C_1 = G_1 + P1.G_0 + P_1.P_0.C_0$$
$$C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0$$
$$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3P_2.P_1.G_0 + P_3P_2.P_1.P_0.C_0$$

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i.$$

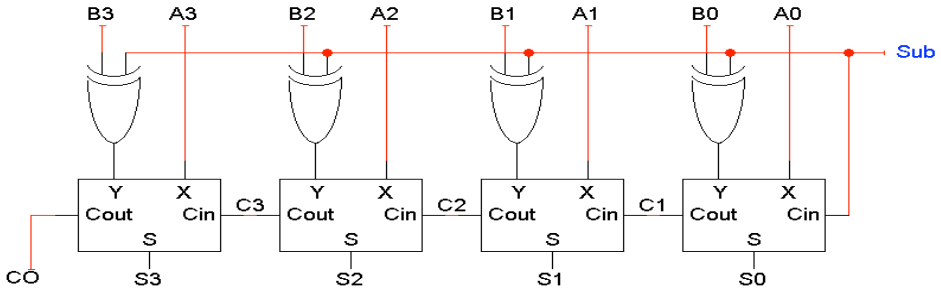


The block diagram of n-bit two's complement adder – sub tractor with an example.

- .XOR gates let us selectively complement the B input.

$X \oplus 0 = X$ $X \oplus 1 = X'$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be A + B + 0, or just A + B.
- When Sub = 1, the XOR gates output B3' B2' B1' B0' and the carry in is 1. Thus, the adder output will be a two's complement subtraction, A - B.



- Overflow
  - Occurs when the result from an operation cannot be represented with the available hardware

- When Overflow cannot occur in Addition?
  - When adding operands with different signs, **overflow cannot occur.**
  - Reason→ Sum must be no longer than one of the operands
  - Assume case of 32-bit word
  - Example: -10+4=-6
  - Operands fit in 32-bits
  - Sum also fit in 32-bits(Reason: Sum is no larger than operand)

- When Overflow cannot occur in Subtraction?
  - Opposite Principle
  - When the signs of the operands are the same, overflow cannot occur.
  - Example: x - y= x + (-y)
  - Subtracting by negating the second operand and adding
  - No overflow occurs

- When overflow occurs?
  - Adding or subtracting two 32- bit numbers yield a result that needs 33-bits to be fully expressed.
  - Lack of a 33rd bit means the overflow occurs.
  - Sign bit is set with the value of the result instead of the proper sign of the result.
  - Overflow occurs adding two positive numbers and the sum is negative (i.e a carry out occurred in the sign bit)
  - In Subtraction subtracting a negative number from positive number and getting a negative result
  - Subtracting a positive number from negative number and getting a positive result.(i.e borrow occurred from the sign bit)

Overflow Conditions for Addition and Subtraction

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

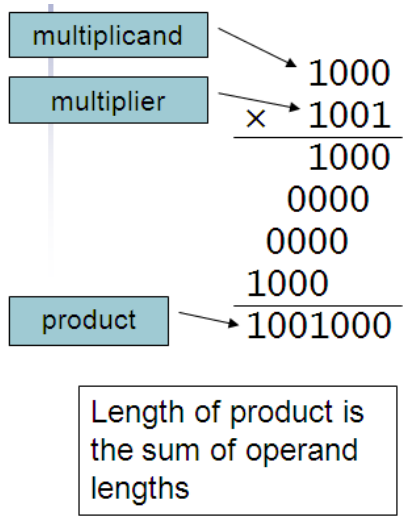
Arithmetic for multimedia

- Same operation executed for multiple data items
- Uses a fixed length register and partitions the carry chain to allow utilizing the same functional unit for multiple operations
- E.g. a 64 bit adder can be utilized for two 32-bit add operations simultaneously

Saturating operations

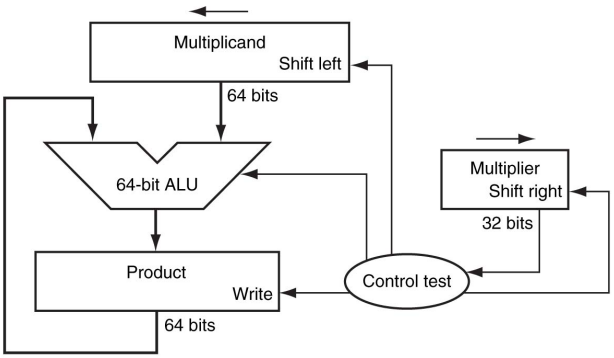
- With saturation arithmetic the result of an operation is bounded in a range between a minimum and a maximum value. For example, with saturation arithmetic in the range [0, 255]
- On overflow, result is largest representable value
- E.g., clipping in audio(Saturation refers to pushing a device beyond its normal limits), saturation in video

3.3 Multiplication



Sequential Version of the Multiplication Algorithm and Hardware

- Hardware resemble the paper- and-pencil method
- Move the multiplicand left one digit each step to add with intermediate products

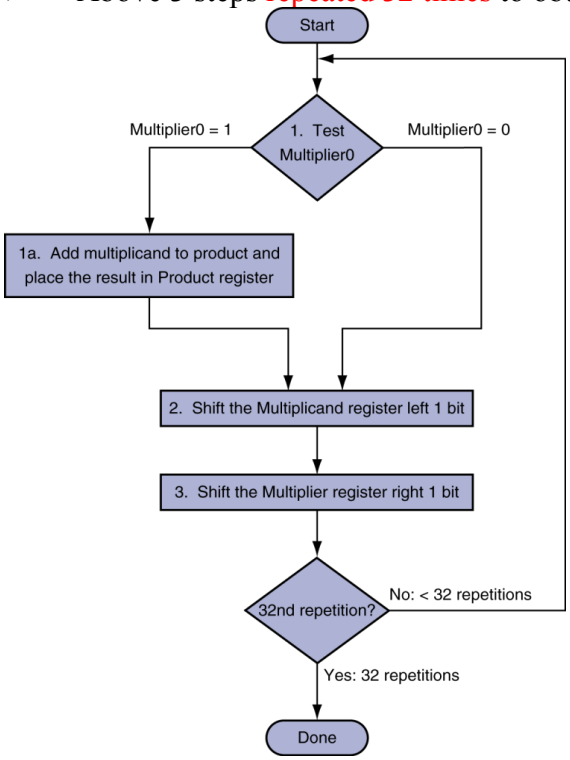


- Over 32 steps, a 32-bit multiplicand move 32-bits to left
- Hence 64-bit Multiplicand register initialized with 32-bit multiplicand in right half and zero in the left half

First Multiplication Algorithm using the previous hardware

3 basic steps for each bit

- LSB bit of multiplier (Multiplier$_0$) determines whether multiplicand is added to the product register
- Left shift move the intermediate operands to left
- Shift right of multiplier register give us the next bit of the multiplier to examine for the next iteration
- Above 3 steps repeated 32 times to obtain the product



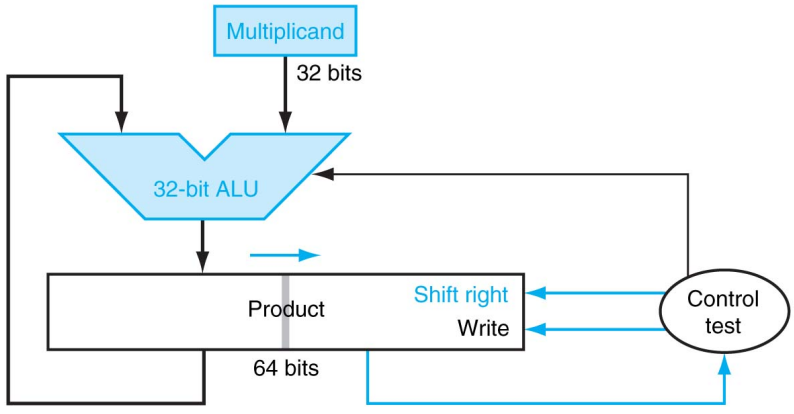
- Multiply 2_{10} x 3_{10} or 0010_2 x 0011_2

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

Refined Version of Multiplication Hardware

- Previous Algorithm easily refined to take 1 clock cycle per step
- Speed up by performing operations in parallel
- If the multiplier bit is 1:
  - Multiplier and Multiplicand are shifted while the Multiplicand is added to the product
- An ALU addition operation can be very time consuming when done repeatedly.
- Computers can shift bits faster than adding bits
- Hardware is optimized to halve the width of the adder and registers by noticing the unused portions of registers and adders

Revised hardware
A Multiply Algorithm

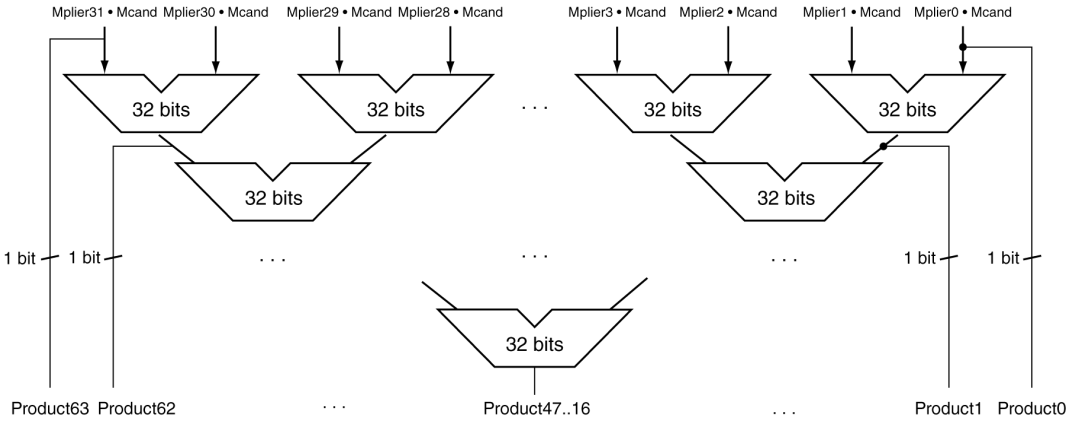


Faster Multiplication

- Faster multiplication Hardware
- Providing one 32-bit adder for each bit of the multiplier
- One input is the multiplicand ANDed with the multiplier bit
- Other is the output of a prior adder
- Rather than using single 32-bit adder 31 times
- This unroll the loop to use 31 adders organized to minimize the delay

Faster Multiplier

- Uses multiple adders
- Can be pipelined
- Several multiplication performed in parallel

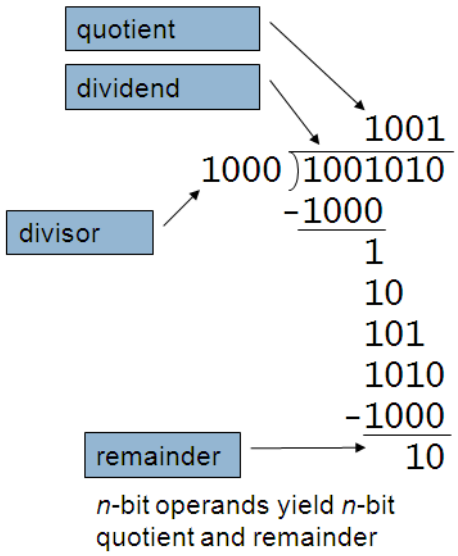


Booth's algorithm for multiplication of signed two's complement numbers.

- Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.
- Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).
  - A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1) bits with zeros.
  - S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.
  - P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.
- Determine the two least significant (rightmost) bits of P.
  - If they are 01, find the value of P + A. Ignore any overflow.
  - If they are 10, find the value of P + S. Ignore any overflow.
  - If they are 00, do nothing. Use P directly in the next step.
  - If they are 11, do nothing. Use P directly in the next step.
- Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.
- Repeat steps 2 and 3 until they have been done y times.
- Drop the least significant (rightmost) bit from P. This is the product of m and r.

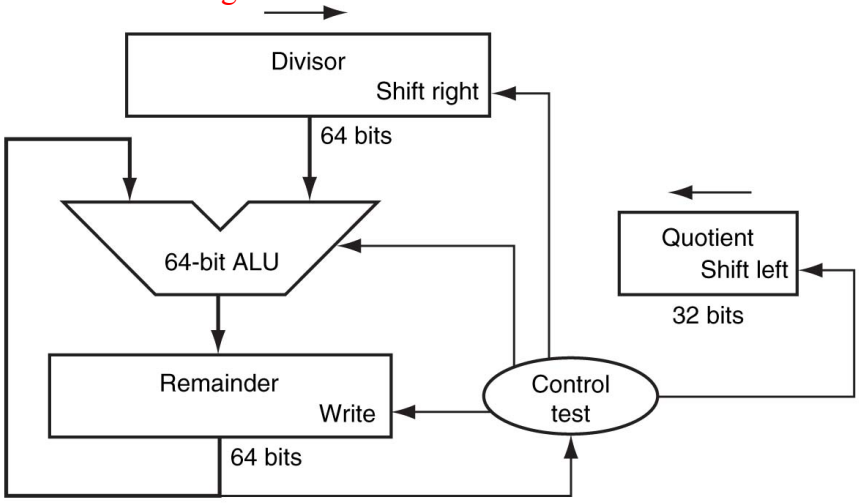
3.4 Division

- Reciprocal operation of multiply is divide
- Less Frequent
- More unusual
- Perform mathematical invalid operation :dividing by 0
- Relationship between the Components



A Division Algorithm and Hardware

- 32-bit Quotient set to 0
- Divisor is placed in the left half of the 64-bit Divisor Register
- Each iteration move the divisor to the right one digit
- Remainder register initialized with dividend

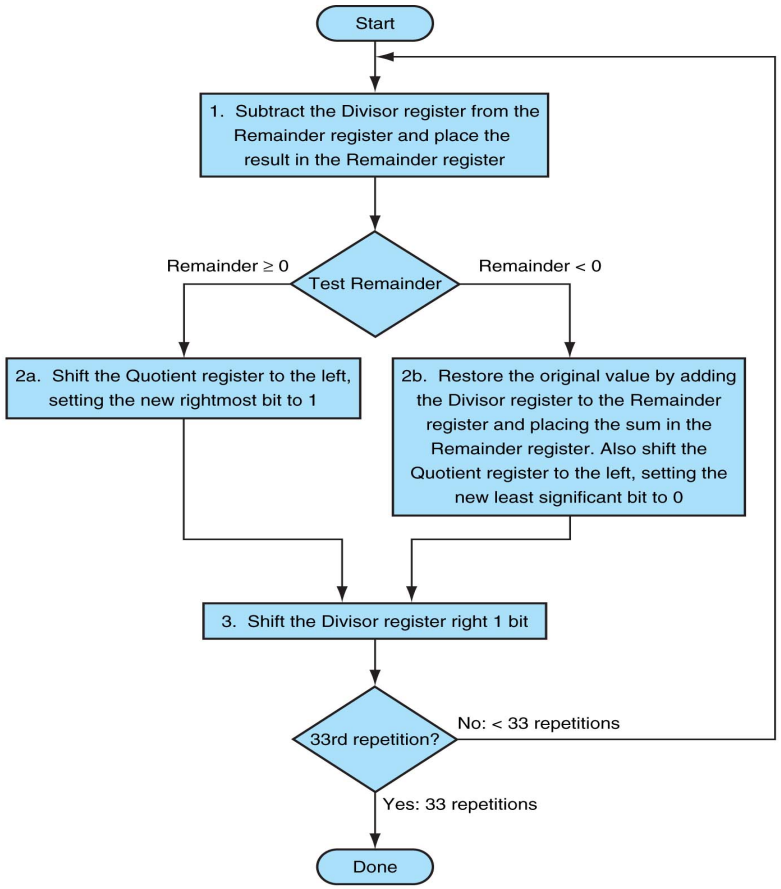


A division Algorithm using the Hardware

3 steps for division algorithm

To check whether the divisor is smaller than dividend

- Subtract the divisor
- Check the result
  - If the result is positive, divisor is smaller or equal to dividend generate 1 in quotient
  - Result is negative restore the original value by adding divisor back to remainder and generate 0 in quotient
- Divisor is shifted right and iterated again



- Remainder is positive, the divisor did go into the dividend
- Step 2a generates a 1 in the quotient.
- A negative remainder after step 1 means that the divisor did not go into the dividend
- Step 2b generates a 0 in the quotient and adds the divisor to the remainder
- Final shift right, in step 3, aligns the divisor properly relative to the dividend for the next iteration.
- Steps are repeated 33 times.

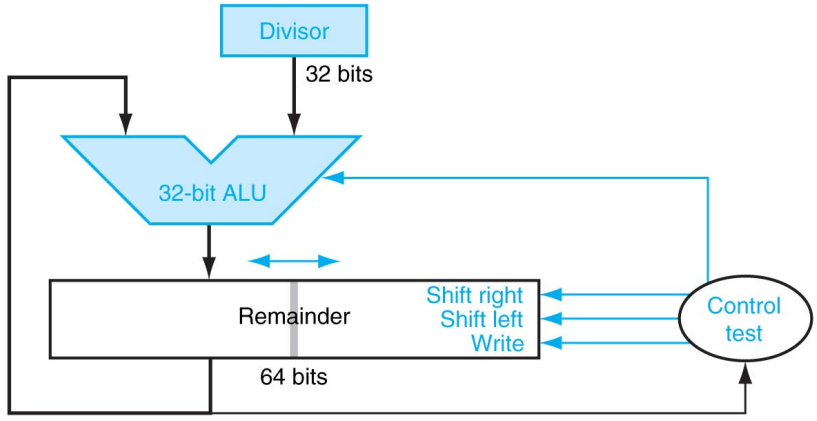
A Divide Algorithm

- Using 4-bit version
  $7_{10}/2_{10}$ or $0000\ 0111_2$ by 0010_2
- Algorithm takes n+1 steps to get quotient and remainder

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

Improved version of the division Algorithm

- Algorithm and hardware refined to be faster and cheaper
- Divisor register, ALU, and Quotient register are all 32 bits wide
- Remainder register left at 64 bits
- ALU and Divisor registers are halved and the remainder is shifted left.
- Combines the Quotient register with the right half of the Remainder register.



Non – restoring division technique.

A variant that skips the restoring step and instead workswith negative residuals

• If P is negative

- (i-a) Shift the register pair (P,A) one bit left
- (ii-a) Add the contents of register B to P

• If P is positive
- (i-b) Shift the register pair (P,A) one bit left
- (ii-b) Subtract the contents of register B from P
- (iii) If P is negative, set the low-order bit of A to 0, otherwise set it to 1

• After n cycles
- The quotient is in A

• If P is positive, it is the remainder, otherwise it has to be restored add B to it to get the remainder.

P	A	Operation
00000	1110	Divide 14 = 1110 by 3 = 11. B register always contains 0011
00001	110	step 1(i-b): shift
+00011		step 1(ii-b): subtract b (add two's complement)
11110	1100	step 1(iii): P is negative, so set quotient bit to 0
11101	100	step 2(i-a): shift
+00011		step 2(ii-a): add b
00000	1001	step 2(iii): P is +ve, so set quotient bit to 1
00001	001	step 3(i-b): shift
+11101		step 3(ii-b): subtract b
11110	0010	step 3(iii): P is -ve, so set quotient bit to 0
11100	010	step 4(i-a): shift
+00011		step 4(ii-a): add b
11111	0100	step 4(iii): P is -ve, set quotient bit to 0
+00011		Remainder is negative, so do final restore step
00010		

- **The quotient is 0100 and the remainder is 00010**

Faster Division

- Use many adders to speed up the multiply but cannot be done for divide
- Reason: Sign of difference is to be known before the next step of the algorithm (Subtraction is conditional on sign of remainder)
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
- Named for its creators (Sweeney, Robertson, and Tocher)
- SRT division is a popular method for division in many microprocessor implementations.
- SRT division is similar to non-restoring division
  - Still require multiple steps

3.5 Floating Point

- Programming Languages support numbers with fractions called reals in mathematics
- Example of reals
- 3.14159265…
- 2.71828…
- 0.00001 or **1.0×10^{-5}**

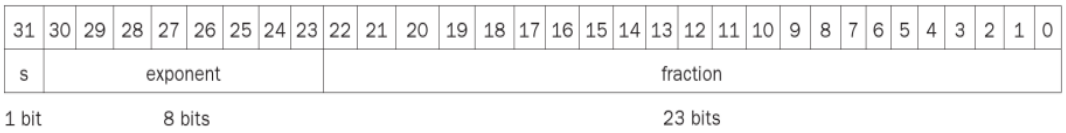
A number in scientific notation has no leading 0s is called a **Normalized number**

- To keep a number in normalized form a base can be increased or decreased by exactly the number of bits
- Number must be shifted to have one nonzero digit to the left of the decimal point
- **Normalized**: **1.0×10^{-9}**
- Not normalized: 0.1×10^{-8}, 10.0×10^{-10}
- Computer arithmetic that supports it is called floating point, because the binary point is not fixed, as it is for integers

- Scientific Notation for reals in normalized form offers three advantages
  - Simplifies exchange of data includes floating point numbers
  - Simplifies floating-point arithmetic algorithms
  - Increases the accuracy of the numbers to be stored in a word

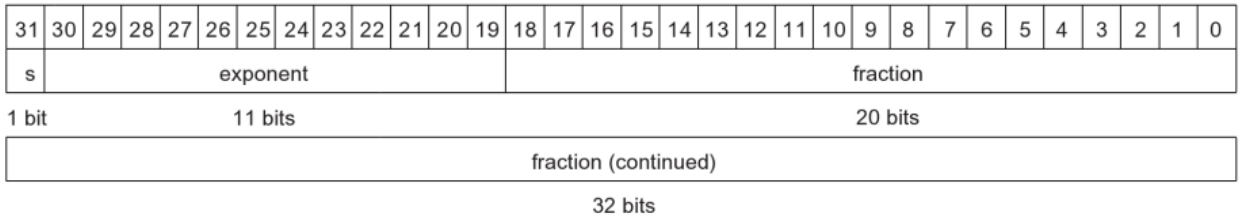
Floating-Point Representation

- Includes Fraction and Exponent
- Representation of ARM(Advanced RISC machine) Floating-point Number
- S →Sign of the floating point number
- E→ Value of the 8-bit exponent field
- F→ 23-bit number



Format of Floating-Point numbers

- Overflow Interrupt occurs in Floating-point Arithmetic
- Overflow→ Positive Exponent is too large to be represented in the Exponent field
- Underflow→ Negative Exponent is too large to be represented in the Exponent field
- To reduce the chances of Underflow and overflow
- Another format is used has a larger exponent
- Double Precision floating-point arithmetic
- Both called IEEE 754 Floating-point Standard invented in 1980



Double Precision floating-point

- S →Sign of the floating point number
- E→ Value of the 11-bit exponent field
- F→ 32-bit number

Representation

- sign, exponent, significand

$$(-1)^{sign} \times significand \times 2^{exponent}$$

- more bits for significand gives more accuracy
- more bits for exponent increases range

Exponent => range, significand => precision

- zero: 0 in the exponent and significand
- +/- infinity: all ones in exponent, 0 in significand
- NaN: all ones in exponent, nonzero significand

IEEE 754 Encoding of Floating-Point Numbers

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)

Floating-Point Addition

- Add numbers in scientific Notation

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

1. Align decimal points

   Shift number with smaller exponent

   $9.999_{10} \times 10^{1} + 0.0610_{10} \times 10^{1}$
2. Add significands

   $9.999_{10} \times 10^{1} + 0.0610_{10} \times 10^{1}$
3. Normalize result & check for over/underflow

   $1.0015_{10} \times 10^{2}$
4. Round and renormalize if necessary

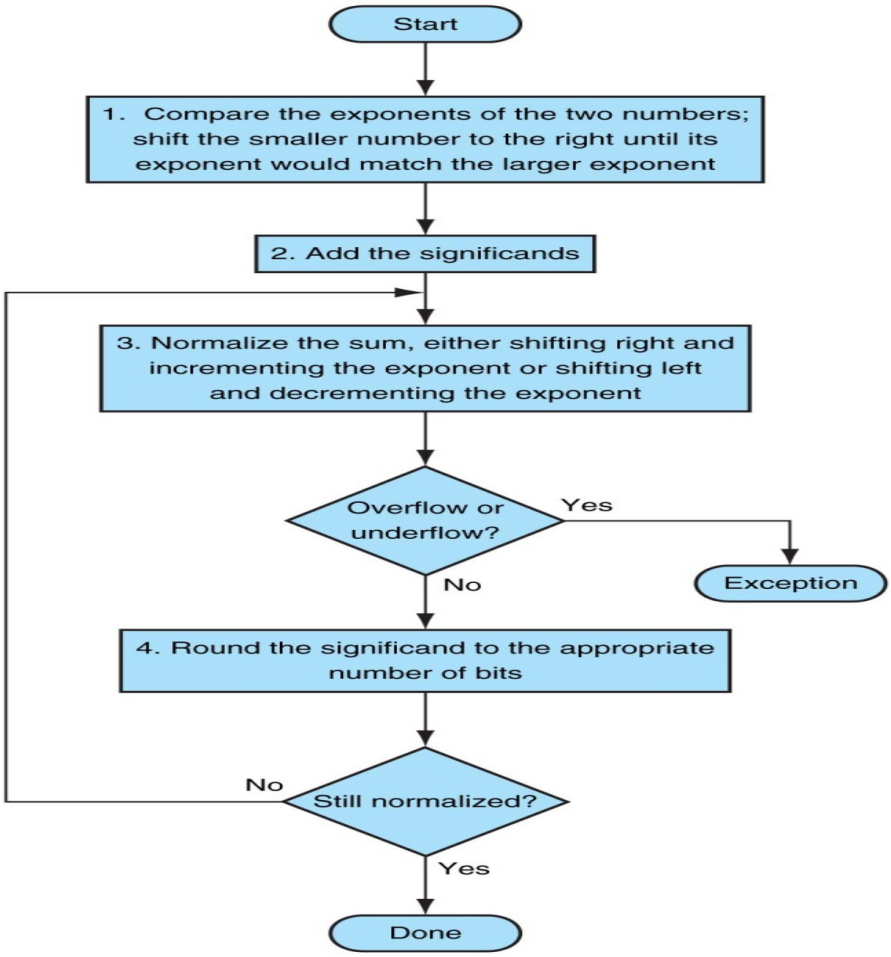
   $1.002_{10} \times 10^{2}$

Steps

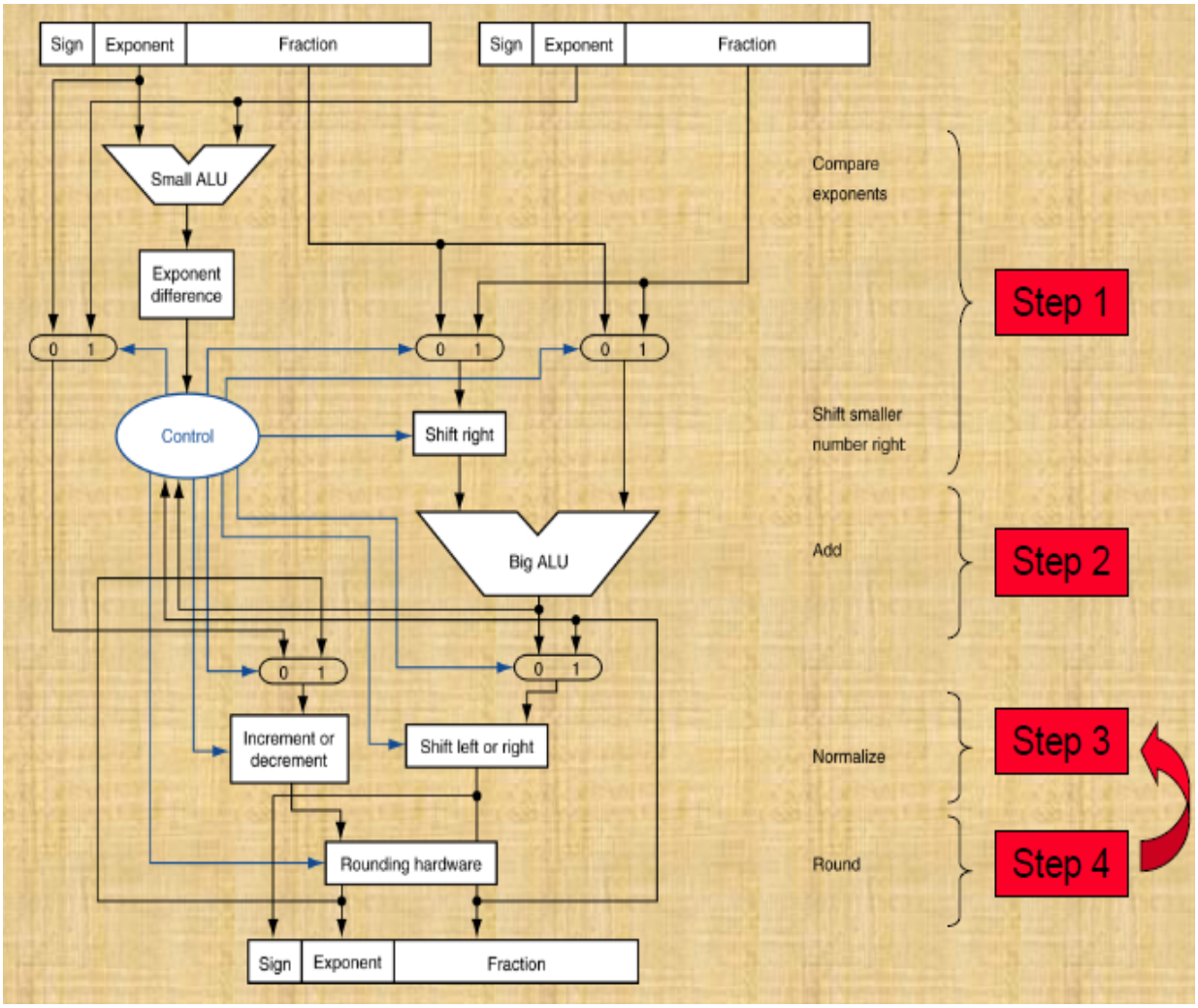
- The first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number
- If sum is not normalized adjust it
- Adjust the exponent
- Whenever the exponent is increased or decreased check for overflow or underflow(Exponent should fits in its field)

Flow chart for Floating-Point Addition

Block diagram for Arithmetic Unit dedicated to Floating-Point Addition



The normal path is to execute steps 3 and 4 once but if rounding causes the sum to be unnormalized step 3 must be repeated



Example

0.5_{10} + -0.4375_{10}

$0.5_{10} = ½_{10} = 0001_2 / 0010_2 = 0.1_2 \times 2^0 = 1.000_2 \times 2^{-1}$

$-0.4375_{10} = -7/16_{10} = -7/2^4{}_{10}$

$= -0.0111_2 = -0.0111_2 \times 2^0 = -1.110 \times 2^{-2}$

Step 1: Take the significand of the lesser exponent shift it right until its exponent matches the larger number

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

Step 2: Add the significand

$= 1.000_2 \times 2^{-1} + (-0.111 \times 2^{-1}) = 0.001_2 \times 2^{-1}$

Step 3: Normalize the sum, Checking for overflow and Underflow

$1.000_2 \times 2^{-4}$

No overflow or Underflow

Step 4: Round the sum → Sum already fits exactly in 4 bits

Sum is then $1.000_2 \times 2^{-4} = 0.0001000_2 = 0.0001_2$

$= 1/2^4{}_{10} = 1/16_{10} = 0.0625_{10}$

Floating-Point Multiplication

Consider a 4-digit decimal example

$$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$$

1. Add exponents

   New exponent = 10 + –5 = 5

2. Multiply significands

   $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$

3. Normalize result & check for over/underflow

   1.0212×10^{6}

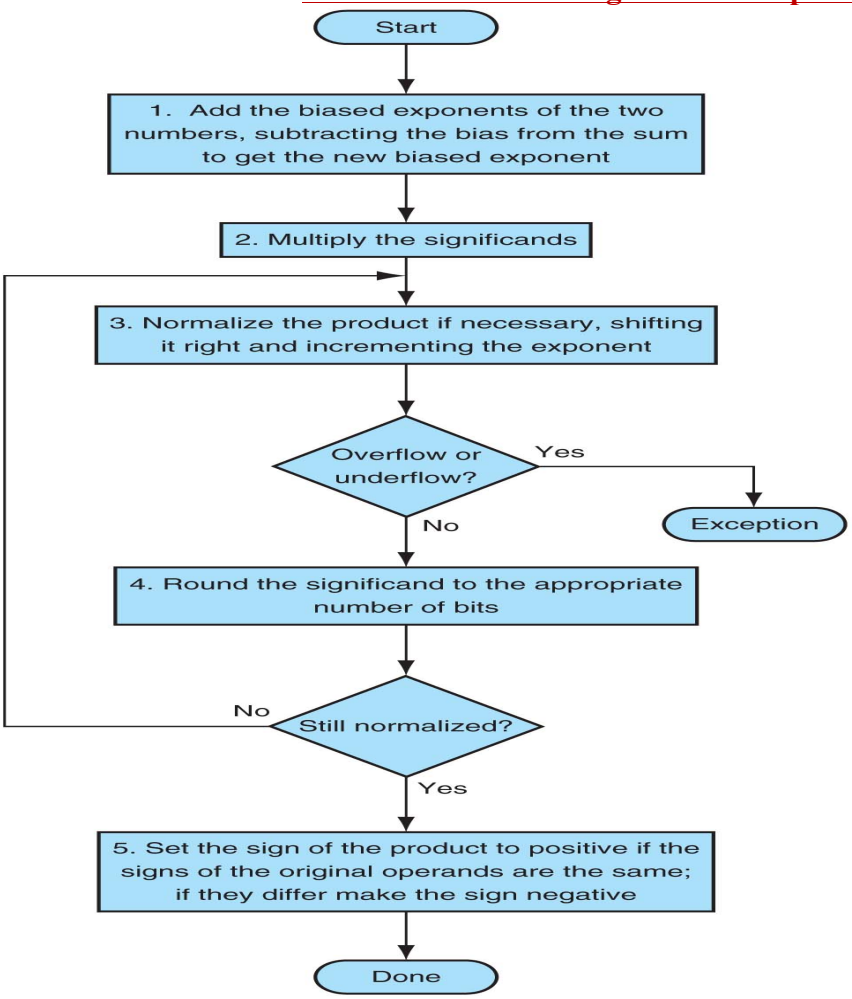
4. Round and renormalize if necessary

   1.021×10^{6}

5. Determine sign of result from signs of operands

   $+1.021 \times 10^{6}$

Flow chart for Floating-Point Multiplication



- Calculating the new exponent of the product by adding the biased exponent
- Multiplication of significands followed by optional normalization step
- Size of exponent checked for overflow and underflow
- Product is rounded
- If Rounding needs further normalization again check for exponent size

- Set the sign bit
  - Set 1 if the sign of the operands were different
  - Set 0 if the sign of the operands were same

0.5_{10} x -0.4375_{10}

$0.5_{10} = 1/2_{10} = 0001_2 / 0010_2 = 0.1_2 \times 2^0 = 1.000_2 \times 2^{-1}$

$-0.4375_{10} = -7/16_{10} = -7/2^4{}_{10} = -0.0111_2 = -0.0111_2 \times 2^0 = -1.110 \times 2^{-2}$

Step: 1 Adding the Exponent without bias → 3
Step 2: Multiplication of significands → Product is $1.110000_2 \times 2^{-3}$
Keep it to four bits $1.110_2 \times 2^{-3}$
Step 3: Check the product whether normalized & Check the exponent for overflow or underflow
Already normalized, No overflow and underflow
Step 4: Rounding the product no change
Step 5: Sign of the original operands differ , make sign of the product negative $-1.110_2 \times 2^{-3}$
Convert to decimal to check the results: $-1.110_2 \times 2^{-3} = -0.001110_2 = -0.00111_2$
$= -7/2^5{}_{10} = -7/32_{10} = -0.21875_{10}$

3.6 Subword parallelism

- Need more processing of multimedia information
  - Images, video, audio, 3D graphics, etc.
- Mismatch between wide data paths and the relatively short data types found in multimedia applications
- Resources were wasted and performance was poor
- IA-64 instruction-set architecture (ISA) includes a rich set of multimedia instructions to accelerate the processing of different forms of multimedia data
- These instructions implement the ISA concept of *subword parallelism* also called *packed parallelism* or *microSIMD parallelism* .
- Data is partitioned into smaller units called *subwords*.
- Subword
  - A small data item contained within a word
- Multiple subwords are packed into one register and then the multiple subwords can be processed simultaneously
- Improves the performance of multimedia applications