Unit-1, Module-3
Computer Arithmetic and ALU

Question 1: How subtraction is performed if integers are represented in 2′s complement format. Explain with examples considering 8-bit representation. Frame examples to show all possible combination of positive and negative numbers. Also indicate overflow and no overflow cases.

Solution 1–

The rules for subtracting two 8 bits numbers (*a* from *b*, say i.e., *b*-*a*) in 2′s complement format are disused below.

There are four cases (namely, (i) *a,b* are both positive, (ii) *a,b* are both negative, (iii) *a* is negative and *b* is positive, (iv) *a* is positive and *b* is negative) and we discuss the rules for each combination with examples.

Case (i) *a,b* are both positive

1) First, we need to convert *a* into its negative equivalent in 2's complement format. This is achieved by changing all the 1's to 0's and 0's to 1's in *a* and adding one to the resultant.
2) Next, *b* is added to the result obtained in Step-1 (i.e., 2'complement representation of -*a*).
3) If there is an overflow bit generated in last step, we discard it. This gives us the final answer.

   Note: There is no question of overflow in this case as one positive number is being subtracted from another positive number.

   Example: Let *a* be 1 and *b* be 127:

```
   0111 1111   ( 127₁₀)
   1111 1111   (  -1₁₀)
 1 0111 1110     Neglect the overflow.
```

So answer is 0111 1110(126_{10})

Case (ii) *a,b* are both negative (represented in 2's complement format)

1) As we need to perform $-b - (-a) = -b + a = b - a$, we need to convert *a* into its positive equivalent (in 2's complement format). This is achieved by changing all the 1's to 0's

and 0's to 1's in *a* and adding one to the resultant.

2) Next, *b* is added to the result obtained in Step-1 (i.e., 2'complement representation of (*-a)=a*).

3) If there is an overflow bit generated in last step, we discard it. This gives us the final answer.

   Note: There is no question of overflow because in the final operation, one positive number is being subtracted from another positive number.

Example: Let *a* be -39 and *b* be -92:

```
                              1010 0100   ( -92₁₀)
                              0010 0111   (-(-39₁₀))
                             ____________
                              1100 1011
The answer is 1100 1011  ( - 53₁₀)
```

Case (iii) *a* is negative and *b* is positive

1) As we need to perform *b – (-a) = b + a*, we need to convert *a* into its positive equivalent (in 2's complement format). This is achieved by changing all the 1's to 0's and 0's to 1's in *a* and adding one to the resultant.

2) Next, *b* is added to the result obtained in Step-1 (i.e., 2'complement representation of (*-a)=a*).

3) If there is an overflow bit generated in last step, we discard it. This gives us the final answer.

   Note: In this case there may be an overflow as effectively we are adding two positive numbers. In the final answer if the MSB is 1, there is an overflow. It may be noted that MSB being 1 in 2's complement representation implies negative number, which cannot be the case on adding two positive numbers.

Example: Let *a* be -39 and *b* be 92

```
                              0101 1100 ( 92₁₀)
                              0010 0111   (-(-39₁₀))
                             ____________
                              1100 0011
The answer is 1100 0011 (-61₁₀); as the MSB 1, the answer is negative,
which cannot be the case of adding two positive numbers. So there is
overflow.
```

Example: Let *a* be -39 and *b* be 1

```
                              0000 0001 ( 1₁₀)
                              0010 0111   (-(-39₁₀))
                             ____________
                              0010 1000
The answer is 0010 1000 (40₁₀); as the MSB 0, the answer is positive. So
there is no overflow.
```

Case (iv) *a* is positive and *b* is negative

4) As we need to perform *–b – (+a) = -b -a*, we need to convert *a* into its negative equivalent (in 2's complement format). This is achieved by changing all the 1's to 0's and 0's to 1's in *a* and adding one to the resultant.
5) Next, *b* is added to the result obtained in Step-1 (i.e., 2'complement representation of (*-a)=a*).
6) If there is an overflow bit generated in last step, we discard it. This gives us the final answer.

   Note: In this case there may be an overflow as effectively we are adding two negative numbers. In the final answer if the MSB is 0, there is an overflow. It may be noted that MSB being 0 in 2's complement representation implies positive number, which cannot be the case on adding two negative numbers.

Example: Let *a* be 39 and *b* be -92:

```
  1010 0100    ( -92₁₀)
  1101 1001   (-39₁₀)
1 0111 1101     Neglect the overflow.
```

The answer is 0111 1101 ($+125_{10}$); as the MSB 0, the answer is positive, which cannot be the case of adding two negative numbers. So there is overflow.

Example: Let *a* be 39 and *b* be -1

```
 1111 1111 ( -1₁₀)
 1101 1001 (-39₁₀)
11101 1000   Neglect the overflow.
```

The answer is 1101 1000 (-40_{10}); as the MSB 1, the answer is negative. So there is no overflow.

Question 2: Write the steps of Booth's Algorithm for 2's complement multiplication. Show the execution steps to perform the following multiplications:

a. 6*4,
b. 6*(-4),
c. (-6)*4,
d. (-6)*(-4)

Assume 4 bit representation for the multiplicand and the multiplier.

Solution 2:-

Booth's algorithm basically works by repeatedly adding the multiplicand with itself based on the multiplicand.

To elaborate we consider 3 Binary variables namely, *A*, *M* and *Q*, where

- *A* (size is 4 bits) is initialized to 0 and it holds 4 MSB bits of the final answer of the multiplication
- *M* (size is 4 bits) is loaded with multiplicand
- *Q* (size is 5 bits), where the 4 MSB bits are initialized with multiplier and LSB bit is initialized to 0. The 4 MSB bits of *Q* holds the 4 LSB bits of the final answer of the multiplication

The steps of Booth's algorithm are as follows:

1. Determine the two least significant (rightmost) bits of *Q*.

   (1) If they are 01, update the value of *A*=*A* + *M*. Ignore any overflow.
   (2) If they are 10, update the value of *A*=*A* - *M*. Ignore any overflow.
   (3) If they are 00, do nothing.
   (4) If they are 11, do nothing.

2. Arithmetically, right shift values in *AQ* taken conjointly.
3. Repeat steps 1 and 2, 4 times.
4. Drop the least significant (rightmost) bit from *Q* and then *AQ* taken conjointly gives the final answer of the multiplication.

a) 6 * 4

Initial values of variables

- A = 0000
- M = 0110
- Q = 01000

Perform the loop four times :

1. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00100
2. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00010
3. A = 1010 (A=A-M). The last two bits of Q are 10.
   - Arithmetic right shift AQ. So, A=1101 and Q=00001
4. A = 0011 (A=A+M, and discard carry). The last two bits of Q are 01.
   - Arithmetic right shift AQ. So, A=0001 and Q=10000
   - The final answer is 0001 1000 (AQ, after discarding the LSB of Q), which is 24.

b) 6 * (-4)

Initial values of variables

- A = 0000
- M = 0110
- Q = 11000

Perform the loop four times:

1. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=01100
2. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00110
3. A = 1010 (A=A-M). The last two bits of Q are 10.
   - Arithmetic right shift AQ. So, A=1101 and Q=00011
4. A = 1101 (remains unchanged). The last two bits of Q are 11.
   - Arithmetic right shift AQ. So, A=1110 and Q=10001
   - The final answer is 1110 1000 (AQ, after discarding the LSB of Q), which is -24.

c) (-6) * 4

Initial values of variables

- A = 0000
- M = 1010
- Q = 01000

Perform the loop four times:

1. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00100
2. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00010
3. A = 0110 (A=A-M, and discard carry). The last two bits of Q are 10.
   - Arithmetic right shift AQ. So, A=0011 and Q=00001
4. A = 1101 (A=A+M, and discard carry). The last two bits of Q are 01.
   - Arithmetic right shift AQ. So, A=1110 and Q=10000
   - The final answer is 1110 1000 (AQ, after discarding the LSB of Q), which is -24.

d) (-6) * (-4)

Initial values of variables

- A = 0000
- M = 1010
- Q = 11000

Perform the loop four times:

1. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=01100
2. A = 0000 (remains unchanged). The last two bits of Q are 00.
   - Arithmetic right shift AQ. So, A=0000 and Q=00110
3. A = 0110 (A=A-M, and discard carry). The last two bits of Q are 10.
   - Arithmetic right shift AQ. So, A=0011 and Q=00011
4. A = 0011 (remains unchanged). The last two bits of Q are 11.
   - Arithmetic right shift AQ. So, A=0001 and Q=10001
   - The final answer is 0001 1000 (AQ, after discarding the LSB of Q), which is 24.

Question 3: Write the steps of Binary division. Show the execution steps to perform the following divisions:

a) 9*4,
b) 9*(-4),
c) (-9)*4,
d) (-9)*(-9)

Assume 4 bit representation for the dividend and the divisor.

Solution 3:-

Booth's division algorithm basically works by repeatedly subtracting the divisor from the dividend and left shifting the dividend. If the result is positive, 1 is placed at the corresponding position of the quotient. Else, 0 is placed in the quotient.

To elaborate we consider 3 Binary variables namely, *A*, *M* and *Q*, where

- *A* (size is 4 bits) is initialized to 0 and after computation finally it holds the Quotient
- *M* (size is 4 bits) is loaded with divisor
- *Q* (size is 4 bits) is initialized with dividend. Finally, it holds the remainder.

The steps of Binary division algorithm are as follows:

1. Left shift *AQ* taken conjointly. Subtract *M* form *A;* i.e., *A=A-M.* (2's complement subtraction)

    a) If *A* is positive (i.e., MSB of *A* is 0), then fill the vacated LSB of *Q* with 1.
    b) If *A* is negative (i.e., MSB of *A* is 1), then restore the value of *A* i.e., *A-A+M. F*ill the vacated LSB of *Q* with 0.

2. Repeat steps 1, 4 times.
3. *Q* holds the reminder and then *A* holds the Quotient.

a) 9 / 4

Initial values of variables

- $A = 0000$
- $M = 0100$
- $Q = 1001$

Perform the loop four times:

1. *AQ* is left shifted leading to *A*=0001 and *Q*=001-.
   A = 1101 (*A=A-M,* 0001-0100=1101). As *A* is negative it is restored. So *A*=0001. The vacated LSB of *Q* is filled with 0. So, finally, *A*=0001 and *Q*=0010.

2. *AQ* is left shifted leading to *A*=0010 and *Q*=010-.
   A = 1110 (*A=A-M,* 0010-0100=1110). As *A* is negative it is restored. So *A*=0010. The vacated LSB of *Q* is filled with 0. So, finally, *A*=0010 and *Q*=0100.

3. *AQ* is left shifted leading to *A*=0100 and *Q*=100-.
   A = 0000 (*A=A-M,* 0100-0100=0000). *A* is positive. The vacated LSB of *Q* is filled with 1. So, finally, *A*=0000 and *Q*=1001.

4. *AQ* is left shifted leading to *A*=0001 and *Q*=001-.
   A = 1101 (*A=A-M,* 0001-0100=1101). As *A* is negative it is restored. So *A*=0001. The vacated LSB of *Q* is filled with 0. So, finally, *A*=0001 and *Q*=0010.

Final Answer: Quotient= 2 (value in *Q*) and Remainder =1 (value in *A*)

For the signed numbers, in case of division, we consider the divisor and the dividend in their positive form and determine the quotient and reminder. After that we apply the following rule:

- Dividend and remainder have the same sign
- Quotient is negative if signs disagree

So,

b) 9/(-4), Quotient: -2, and Reminder =+1
c) (-9)/4, Quotient: -2, and Reminder =-1
d) (-9)/(-9) Quotient: 2, and Reminder =-1

Question 4. Consider the following floating point expressions (addition and subtraction)

1) $5.566 * 10^2 + 7.777 * 10^2$
2) $3.344 * 10^1 + 8.877 * 10^{-2}$
3) $7.744 * 10^{-3} – 6.666 * 10^{-3}$
4) $8.844 * 10^{-3} – 2.233 * 10^{-1}$

Evaluate these expressions and discuss the general procedure for floating-point addition and subtraction.

Assume 4 digits for the significand and 1 digit for the exponent

Solution 4:

The major difference in addition (and subtraction) of fixed point numbers compared to floating point numbers is the handling of exponents by proper alignment of the significands. There are three basic phases of the algorithm for addition and subtraction:

1. Align the significands.
2. Add or subtract the significands.
3. Normalize the result.

Note: It is assumed that none of the operands is zero

Step 1: Significand alignment. The first step is to adjust the numbers so that the two exponents are equal. This may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left (decreasing its exponent). However, it may be noted that both these shifting operations may result in the loss of digits (i.e., precision), it is the smaller number that is generally shifted; any digits that are lost are therefore of relatively small significance. So, the smaller number is shifted right and its exponent is increased till the exponents of both the numbers become same. It may be noted that in this process if the significand of the smaller number becomes 0, then the other number is reported as the result. Thus, if two numbers have exponents that differ greatly, the lesser number is lost.

Step 2: Addition/Subtraction. Next, the two significands are added or subtracted. Note that there is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. If this results in exponent overflow then range of exponent is to be increased.

Step 4: Normalization. Normalization consists of shifting significand digits left and increasing the exponent until the most significant digit is non-zero.

a) $5.566 * 10^2 + 7.777 * 10^2$

Step 1: As the exponents are equal we can just add the significands i.e., Step 2
Step 2: 5.566 + 7.777= 13.343
There is significand overflow, so we shift significand right, drop the 5^{th} digit and increment the exponent by 1 i.e., $1.334 * 10^3$
Step 3: The result is already normalized, so the sum is $1.334 * 10^3$.

b) $3.344 * 10^1 + 8.877 * 10^{-2}$

Step 1: As the exponents are not equal we right shift by 3 digits the significant of the smaller number
$8.877 * 10^{-2} = 0.887 * 10^{-1} = 0.088 * 10^0 = 0.008 * 10^1$;
Step 2: Now the exponents are equal we just add the significands
(3.344 + 0.008) = 3.352
There is no significand overflow
Step 3: The result is already normalized, so the sum is $3.352 * 10^1$

c) $7.744 * 10^{-3} – 6.666 * 10^{-3}$

Solution:
Step 1: As the exponents are equal we can just subtract the significands i.e., Step 2
Step 2: $(7.744 – 6.666)^{3} = 1.078$
There is no significand overflow.
Step 3: The result is already normalized, so the result out of subtraction is
$1.078 * 10^{-3}$

d) $8.844 * 10^{-3} – 2.233 * 10^{-1}$

Step 1: As the exponents are not equal we right shift by 2 digits the significant of the smaller number
$8.844 * 10^{-3} = 0.884 * 10^{-2} = 0.088 * 10^{-1}$
Step 2: Now the exponents are equal we can just subtract those signed significands
(0.088 – 2.233) =-2.145
There is no significand overflow
Step 3: The result is already normalized, so the answer is $-2.145 * 10^{-1}$

Question 5. What are the basic operations performed by an ALU in a computer?

Solution 5.

ALU stands for Arithmetic and Logic unit and as the name suggests it performs the following operations

Arithmetic Operations: It provides various computational capabilities to the computer in terms of arithmetic manipulation that includes various operations like addition, subtraction etc. Examples:

- Add - Find sum of two operands
- Subtract – Find difference of two operands
- Multiply - Find product of two operands
- Divide- Find quotient of two operands
- Absolute - Replace operand by its absolute value
- Negate - Change sign of operand
- Increment - Add 1 to operand
- Decrement - Subtract 1 from operand

Logical Operations: They perform various bit-style operations/manipulations on non-numeric data. Examples:

- AND - Perform logical AND
- OR - Perform logical OR
- NOT (complement) - Perform logical NOT
- Exclusive-OR Perform logical XOR
- Test - Test specified condition and perform accordingly
- Compare - Make logical or arithmetic comparison of two or more operands, set flag(s) based on outcome
- Set - Control Variables Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
- Shift Left (right)- shift operand, introducing constants at end
- Rotate Left (right)- shift operand, with wrap around end

Question 6. Explain the broad working principle of an ALU using its black box architectural view.

Solution 6.

The black box architecture of an ALU is shown in the figure below.

The basic functionalities of the I/Os of the ALU as shown in the figure discussed below:

- Data is given to the ALU from the registers as inputs.
- The control Unit gives the required signal according to the logic or arithmetic operation that is required to be done (and is specified by the corresponding instruction). Also, the control unit controls the movement of data into and out of ALU (via the registers and Flags).
- The input data for the operation is given from the input registers. Registers are temporary storage locations.
- The result of the operation is again stored in registers (output).
- Values of Flags are Binary and are set or reset based on results of the computations. For example, the ALU will set the overflow flag high if the result of an ADD operation generates an overflow i.e., the number of bits in the result of the ADD operation is more than of the register. Some of the main flag bits are N (negative), Z (Zero), C (Carry), O (Overflow).

Question 7. Assume that we have an 4 bit ALU. For the operations given below what are the vales of the Flags.

a) 5 + (-5) ; assumed signed arithmetic
b) 2 + (-3); assumed signed arithmetic
c) 8+8; assumed unsigned arithmetic
d) 4+5; assumed unsigned arithmetic
e) 7+1; assumed signed arithmetic

Solution 7.

The details of the flags and the rules to set or reset them are discussed in the table below.

FLAGS	RULES TO SET/RESET	
	SET	RESET
Z (Zero)	If the result of an operation is 0.	If the result of an operation is not equal to 0.
N (Negative) This flag is of importance if the arithmetic is signed	If the result of an operation is negative.	If the result of an operation is not negative.
C (Carry) This flag is of importance if the arithmetic is unsigned	1) If the addition of two numbers results in carry out of the most significant bits. 2) If a subtraction of two numbers requires borrow into the most significant bits that are subtracted.	In all other cases.
O (Overflow) This flag is of importance if the arithmetic is signed	1) If the sum of two numbers positive (with sign bit 0) yields a negative number (with sign bit 1.) 2) If the sum of two negative numbers (with sign bit 1) yields a positive number (with sign bit 0.)	In all other cases.
EP (Even Parity)	If the result of operation has even number of 1's.	In all other cases.

a) 5 + (-5) ; assumed signed arithmetic

5 is represented as 0111 in 2's complement format
-5 is represented as 1001 in 2's complement format

Now,

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

- For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this holds in this case, Z is 1. It may be noted that as we have considered 2's complement arithmetic, we ignore the carry and consider only the 4 Bits as the final answer, which is zero.
- The MSB of the final answer (after ignoring the carry) is 0, indicating that it is positive. So N flag is 0.
- Now as carry is generated the C Flag is set to 1. However, as the arithmetic is signed, the value of C is ignored.
- Since both the numbers are of different signs, O flag is 0.
- As the number of 1s in the answer is zero (even), EP flag is set to 1.

b) 2 + (-3); assumed signed arithmetic

2 is represented as 0010 in 2's complement format
-3 is represented as 1101 in 2's complement format

Now,

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

- For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this does not hold in this case, Z is 0.
- The MSB of the final answer is 1, indicating that it is negative. So N flag is 1.
- Now as no carry is generated the C Flag is reset to 0. However, as the arithmetic is signed, the value of C is ignored.
- Since both the numbers are of different signs, O flag is 0.
- As the number of 1s in the answer is four (even), EP flag is set to 1.

c) 8+8; assumed unsigned arithmetic

8 is represented as 1000 in unsigned format

Now,

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

- For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this holds in this case, Z is 1.
- The MSB of the final answer is 0, indicating that it is positive. So N flag is 0. However, as this is unsigned authentic N is ignored.
- Now as carry is generated the C Flag is set to 1.
- Since both the numbers are negative (i.e., sign bit is 1) but sign bit of the answer is 0, which indicates that answer is positice. So, O flag is 1. However, as this is unsigned authentic N is ignored.
- As the number of 1s in the answer is zero (even), EP flag is set to 1.

d) 4+5; assumed unsigned arithmetic

4 is represented as 0100 in unsigned format
5 is represented as 0101 in unsigned format

Now,

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

- For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this does not hold in this case, Z is 0.
- The MSB of the final answer is 1, indicating that it is negative. So N flag is 1. However, as this is unsigned authentic N is ignored.
- Now as no carry is generated the C Flag is reset to 0.
- Since both the numbers are positive (i.e., sign bit is 0) but sign bit of the answer is 1, which indicates that answer is negative. So, O flag is 1. However, as this is unsigned authentic N is ignored.
- As the number of 1s in the answer is two (even), EP flag is set to 1.

e) 7+1; assumed signed arithmetic

7 is represented as 0111 in 2's complement format
1 is represented as 0001 in 2's complement format

Now,

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

- For computing the Z flag, ALU checks if all the 4 bits of the answer are 0s. As this does not hold in this case, Z is 0.
- The MSB of the final answer is 1, indicating that it is negative. So N flag is 1.
- Now as no carry is generated the C Flag is reset to 0.
- Since both the numbers are positive (i.e., sign bit is 0) but sign bit of the answer is 1, which indicates that answer is negative. So, O flag is 1. As O is 1, the answer is not valid and so is the flag N.
- As the number of 1s in the answer is 1 (odd), EP flag is set to 0.