A.R. Hurson
Department of Computer Science
Missouri University of Science & Technology

Background Module3

- **Arithmetic Logic Unit**
  - It is a functional box designed to perform "basic" arithmetic, logic, and shift operations on the data.
  - Implementation of the basic operations such as logic, program control, and data transfer operations are easier than arithmetic and I/O operations. Therefore, in this section we concentrate on arithmetic operations.

Background Module3

- **Arithmetic Logic Unit**
  - For a simple machine, the ALU should at least be able to perform operations such as:
    - add
    - increment
    - subtract
    - decrement
    - •
    - •
    - •
  - A simple ALU is basically an adder and some control circuits augmented by special circuits to carry out the logic and shift operations .

Background Module3

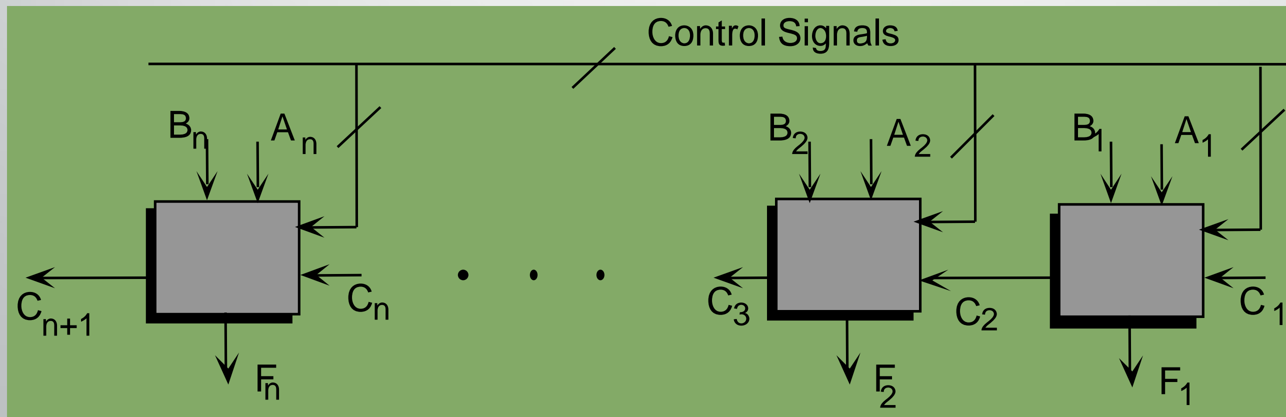
- **Arithmetic Logic Unit**
  - An ALU can be of three types:
    - Serial
    - Parallel
    - Functional (Modular)

Background Module3

- **Arithmetic Logic Unit**
  - Similar to the definition of serial and parallel adders, one can define serial and parallel ALUs.
  - In a serial ALU during each clock pulse one bit of operand(s) participates in the operation.
  - In a parallel ALU operation on all the bits of operand(s) is initiated simultaneously.
  - In a simple word a parallel ALU can be looked at as a cascade of identical units forming a one dimensional array of cells.

Background Module3

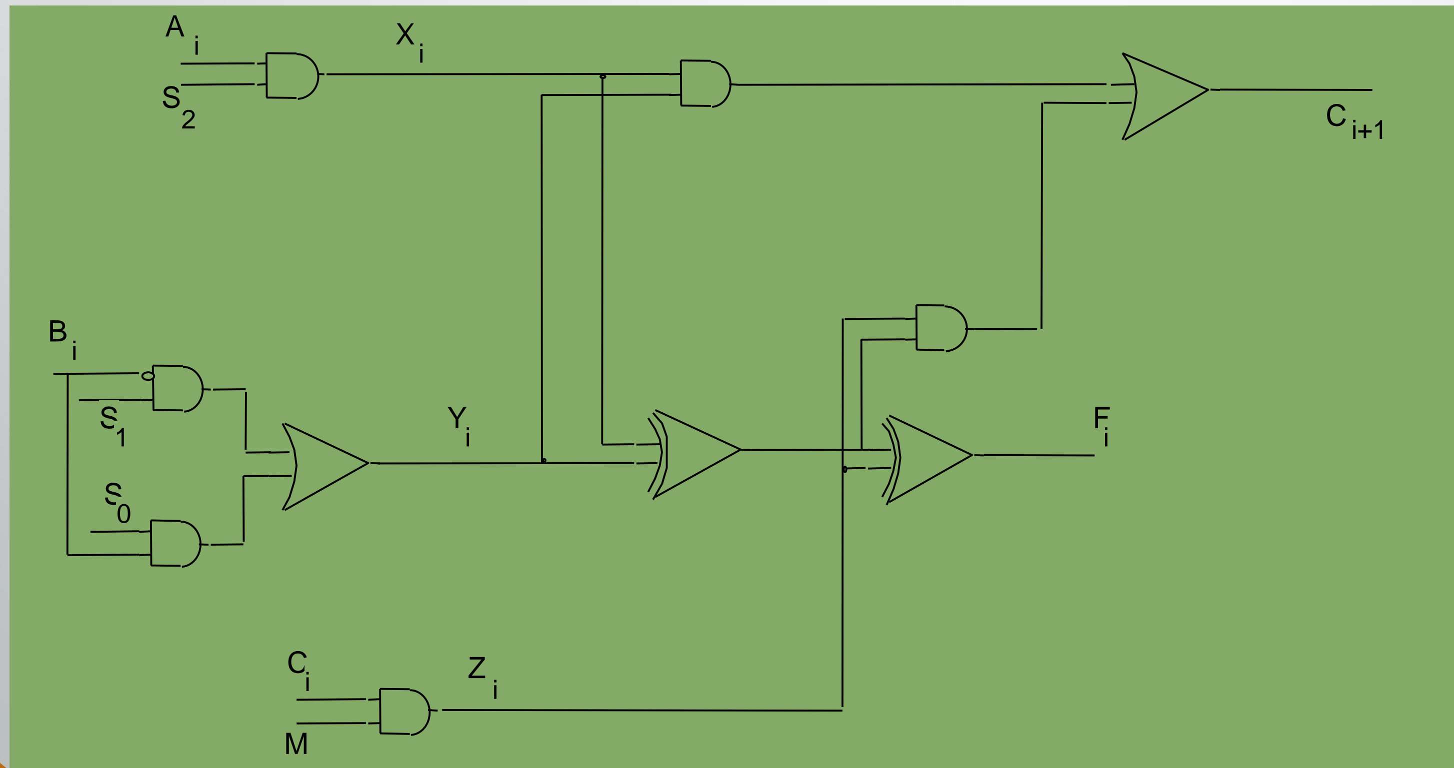
- **Arithmetic Logic Unit —** Parallel ALU
  - ALU operation is determined by the control signals.
  - In a very simple form, the bit-pattern of the control signals is determined by the operation code.
  - In a parallel ALU, one needs to determine the design of a unit and then replicate it.



Background Module3

- **Arithmetic Logic Unit —** Parallel ALU (Example)
  - S_2, S_1, S_0, and M are the control signals.
  - A_i, B_i, and C_i are the operand bits and carry-in, respectively.
  - F_i and C_{i+1} are the result bit and carry-out, respectively.

Background Module3



Background Module3

- **Arithmetic Logic Unit ─** Parallel ALU (Example)
  - The function of each stage can be defined as:

$$F_i = X_i \oplus Y_i \oplus Z_i$$

$$C_{i+1} = X_iY_i + (X_i \oplus Y_i)Z_i = X_iY_i + X_iZ_i + Y_iZ_i$$

  - By appropriate setting of the control signals one can initiate a variety of the operations.

Background Module3

- **Arithmetic Logic Unit –** Parallel ALU (Example)

$M = 0$
$S_2 = 1$
$S_1 = 1 \quad \Rightarrow F \leftarrow$
$S_0 = 1$

$M = 1$
$S_2 = 1$
$S_1 = 1 \quad \Rightarrow F \leftarrow A - 1$
$S_0 = 1$
$C_1 = 0$

Background Module3

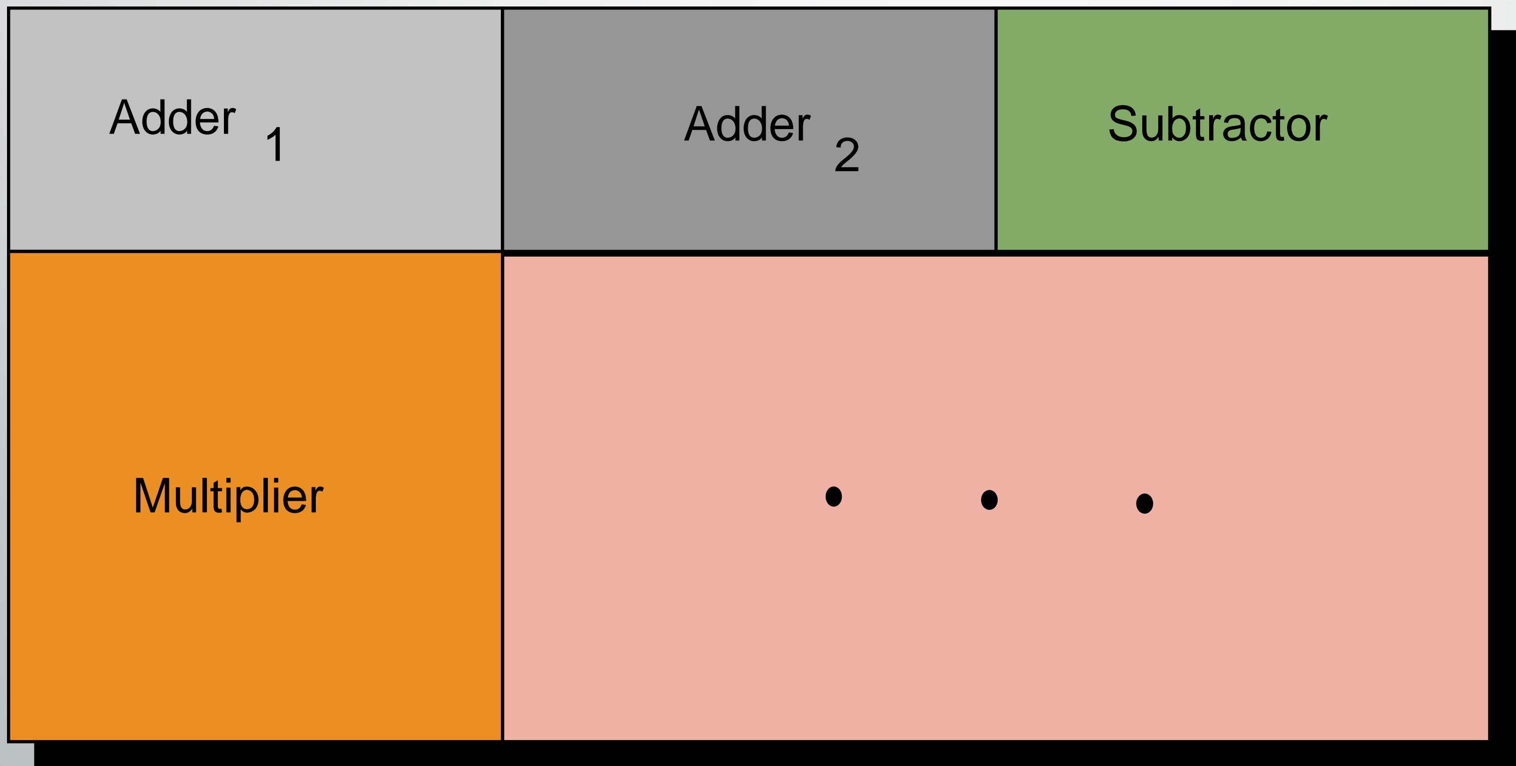
- **Arithmetic Logic Unit**
  - As discussed before a parallel ALU offers a higher speed relative to a serial ALU.
  - How can one improve the performance (speed) of ALU further? In other words; is it possible to build (design) an ALU faster than a parallel ALU?

Background Module3

- **Arithmetic Logic Unit –** Functional (modular) ALU
  - In this model, ALU is a collection of independent units each tailored (specialized) for a specific operation. As a result, execution of independent operations can be overlapped.
  - This approach allows an additional degree of concurrency relative to a parallel ALU, since it allows several operations to be performed on data simultaneously.
  - However, this speed improvement comes at the expense of extra overhead which is needed to detect data independent operations.

Background Module3

- **Arithmetic Logic Unit —** Functional (modular) ALU



Background Module3

- **Arithmetic Logic Unit –** Question
  - Is it possible to improve the performance of an ALU further?
  - Naturally, we can improve performance (physical speed) by taking advantage of the advances in technology.
  - How can we improve the logical speed of the ALU further?
  - In a functional ALU, is it possible to devise algorithms which allow one to improve the performance of the basic operations?
  - If this is a valid direction, then the question of "How to design a fast ALU?" will change to "How to design a fast adder, a fast multiplier, ...?"

Background Module3

- **Arithmetic Logic Unit —** Arithmetic Algorithm
  - The goal is to develop a set of algorithms which allow the execution of basic arithmetic operations on data values presented in different formats and notations (i.e., fixed point, floating point, signed numbers).
  - In general fixed point operations are easier and faster than floating point operations.
  - Signed numbers could be in signed magnitude or signed complement format.

Background Module3

Arithmetic Logic Unit – Comparison of 2-Unsigned Numbers (Direct Comparison)

- Two binary bits (a, b) are equal iff:

$$x = (a \wedge b) \vee (\overline{a} \wedge \overline{b}) = 1$$

- Consequently two registers of n-bits are equal iff:

  $x = x_{n-1} \wedge x_{n-2} \wedge \ldots \wedge x_0 = 1$ where $x_i = A_i \odot B_i$ $0 \le i \le n-1$

- In other words

$$A = B \Rightarrow x = \bigwedge_{i=0}^{n-1} x_i = 1$$

Background Module3

- **Arithmetic Logic Unit** – Comparison of 2-Unsigned Numbers (Direct Comparison)
  - Similarly

$$A > B \Rightarrow A_{n-1}\overline{B}_{n-1} + x_{n-1}A_{n-2}\overline{B}_{n-2} + \dots + x_{n-1}x_{n-2}\dots x_1A_0\overline{B}_0 = 1$$

$$A < B \Rightarrow A_{n-1}B_{n-1} + x_{n-1}A_{n-2}B_{n-2} + \dots + x_{n-1}x_{n-2}\dots x_1A_0B_0 = 1$$

Background Module3

- **Arithmetic Logic Unit –** Comparison of 2-Unsigned Numbers (Direct Comparison)
  - **Example:**

A= 1 1 0 0 1 1 1

B= 1 1 0 1 0 0 0

$$\begin{array}{ccccccc} x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}$$

$$x_6 \wedge x_5 \wedge x_4 \wedge x_3 \wedge x_2 \wedge x_1 \wedge x_0 = 1 \wedge 1 \wedge 1 \wedge 0 \wedge 0 \wedge 0 \wedge 0 = 0 \Rightarrow A \neq B$$

Background Module3

$x6 \wedge x5 \wedge x4 \wedge x3 \wedge x2 \wedge x1 \wedge x0 = 1 \wedge 1 \wedge 1 \wedge 0 \wedge 0 \wedge 0 \wedge 0 = 0$

$\Rightarrow A \neq B$

$$A_6 \overline{B_6} + x_6 A_5 \overline{B_5} + x_6 x_5 A_4 \overline{B_4} + x_6 x_5 x_4 A_3 \overline{B_3} + x_6 x_5 x_4 x_3 A_2 \overline{B_2} + x_6 x_5 x_4 x_3 x_2 A_1 \overline{B_1} + x_6 x_5 x_4 x_3 x_2 x_1 A_0 \overline{B_0} = 0+0+0+0+0+0+0 = 0$$

$\Rightarrow A \ngtr B$

$$B_6 \overline{A_6} + x_6 B_5 \overline{A_5} + x_6 x_5 B_4 \overline{A_4} + x_6 x_5 x_4 B_3 \overline{A_3} + x_6 x_5 x_4 x_3 B_2 \overline{A_2} + x_6 x_5 x_4 x_3 x_2 B_1 \overline{A_1} + x_6 x_5 x_4 x_3 x_2 x_1 B_0 \overline{A_0} = 0+0+0+1+0+0+0 = 1$$

$\Rightarrow A < B$

Background Module3

- **Arithmetic Logic Unit** – Comparison of 2-Unsigned Numbers (Direct Subtraction)
  - Assume *E* is the borrow-out of a parallel subtractor then:

    if $A \geq B$ then $E = 0$

    if $A < B$ then $E = 1$

    if $A = B$ then $E = 0$ and result all 0s
  - In other words, to check the relative magnitude of two unsigned numbers say, *A* and *B*. Perform A-B and then check the borrow-out. The relative magnitude of *A* and *B* is determined according to the above relationships.

Background Module3

- **Arithmetic Logic Unit –** Comparison of 2-Unsigned Numbers (2^s Complement Addition)
  - Assume E is the carry-out of a parallel adder then:

    if $A < B$ then $E = 0$

    if $A \geq B$ then $E = 1$

    if $A = B$ then $E = 1$ and result all 0s

Background Module3

- **Arithmetic Logic Unit** – Comparison of 2-Unsigned Numbers (2^s Complement Addition)
  - **Proof:**

$$A - B = A + \tilde{B} = A + 2^n - B = 2^n + A - B$$

  - if $A \geq B \Rightarrow A - B \geq 0 \;\Rightarrow 2^n + A - B \geq 2^n \;\Rightarrow E = 1$
  - if $A < B \Rightarrow A - B < 0 \;\Rightarrow 2^n + A - B < 2^n \;\Rightarrow E = 0$
  - if $A = B \Rightarrow A - B = 0 \;\Rightarrow 2^n + A - B = 2^n \;\Rightarrow E = 1$ and result all 0^s
  - To determine the relative magnitude of two unsigned numbers, perform $A + \tilde{B}$ and then check the carry-out.

Background Module3

- **Arithmetic Logic Unit** – Comparison of 2-Unsigned Numbers

(1s Complement Addition)

  - Assume E is the carry-out of a parallel adder then:

if $A \leq B$ then $E = 0$

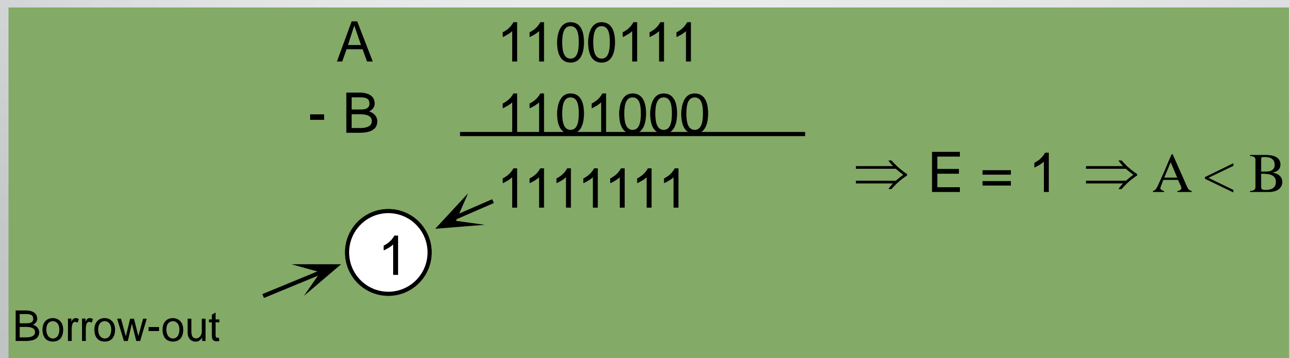
if $A = B$ then $E=0$ and result all 1s

if $A > B$ then $E = 1$

  - To determine the relative magnitude of two unsigned numbers, perform $A + \overline{B}$ and then check the carry-out.

Background Module3

- **Arithmetic Logic Unit –** Comparison of 2-Unsigned Numbers (Direct Subtraction)
  - Example



Background Module3

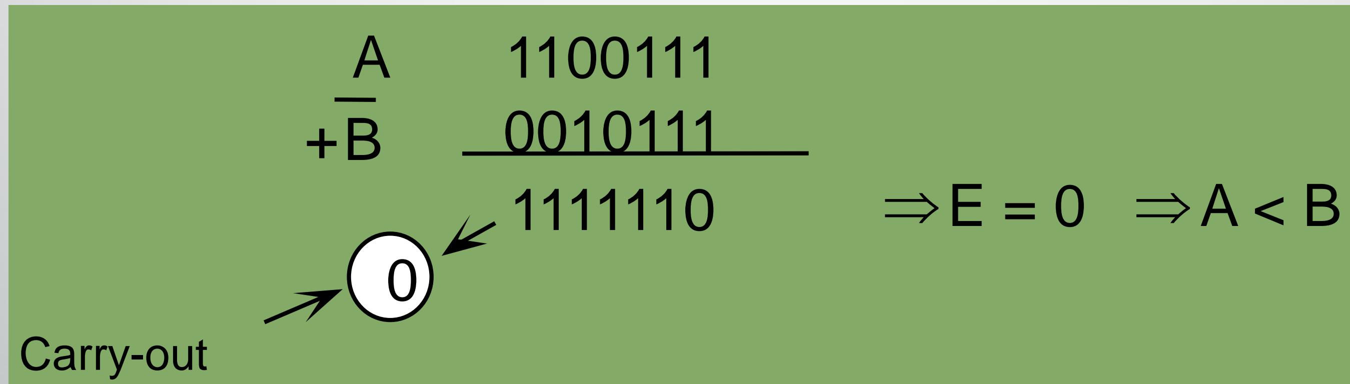
- **Arithmetic Logic Unit –** Comparison of 2-Unsigned Numbers (2s Complement Addition)
  - Example

$$
\begin{array}{rl}
A & 1100111 \\
+\ \tilde{B} & \underline{0011000} \\
 & 1111111 \quad \Rightarrow E = 0 \quad \Rightarrow A < B
\end{array}
$$

Carry-out: 0

Background Module3

- **Arithmetic Logic Unit –** Comparison of 2-Unsigned Numbers (1s Complement Addition)
  - Example



Background Module3

- Questions
  - Prove the aforementioned formulas for the 1^s complement addition operation.
  - How can one determine the relative magnitude of signed numbers?

Background Module3

- **Arithmetic Logic Unit –** Addition Algorithm (2s Complement Numbers)
  - Add numbers including the sign bits. If there is a carry-out of the last digit, then disregard it.

Background Module3

- **Arithmetic Logic Unit –** Addition Algorithm (1s Complement Numbers)
  - Add numbers including the sign bits. If there is a carry-out then increment the result by 1 (wrap around the carry).

Background Module3

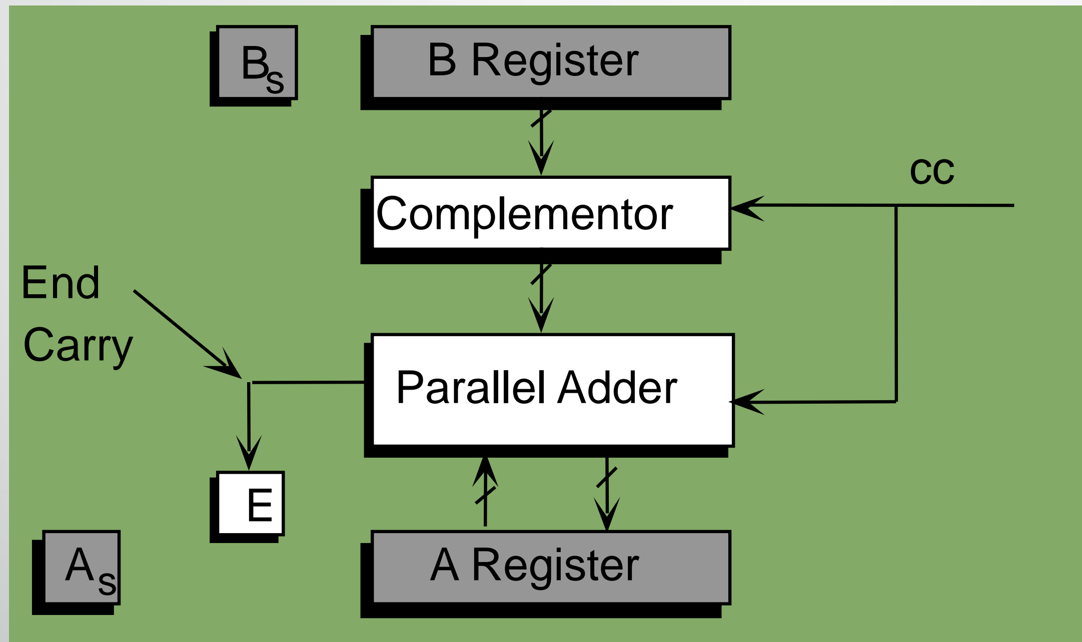
- **Arithmetic Logic Unit –** Addition Algorithm (Signed magnitude numbers)
  - Addition (Subtraction):
    - When the signs of numbers (say A and B) are identical (different) add the two magnitudes. Sign of the result is the same as A.
    - When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller magnitude from the larger one. Sign of the result is the same as the sign of the larger magnitude.
    - If result zero then set the sign of the result to 0.

Background Module3

- **Arithmetic Logic Unit –** Addition Algorithm (Signed magnitude numbers)
  - How to reduce hardware requirements of the proposed algorithm?
    - Subtraction can be converted into addition.
    - Magnitudes can be compared using an adder.

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)



- cc = 0 if (addition and $A_s = B_s$) or (subtraction and $A_s \neq B_s$)
- cc = 1 if (addition and $A_s \neq B_s$) or (subtraction and $A_s = B_s$)

Background Module3

- **Arithmetic Logic Unit –** Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A (+29)          0011101
+B (+14)   ⇒   +  0001110          ⇒ Same signs and addition

                   011101
                   001110           Add the magnitudes.
            E
            0 ←    101011           E = 0

        A  =  0111011     Final result (+43).
```

Background Module3

- **Arithmetic Logic Unit –** Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A (-14)        101110
+B (-7)  ⇒  +  100111      ⇒ Same signs and addition

                01110
                00111        Add the magnitudes.
      E
      0 ←       10101        E = 0

      A  =  110101      Final result (-21).
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A (-14)          101110
-B (-7)    ⇒   -  100111       ⇒  Same signs and subtract
                  ------

                   01110
                   11001          Add 2s complete of B to A.
                  ------
        E
        1 <-       00111          E = 1, A>B

        A  =      100111          Final result (-7).
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A (+14)          001110
-B (+20)   ⇒   -  010100         ⇒ Same signs and subtract

                  01110
                  01100          Add 2s complete of B to A.
           E
           0 <--  00111     E = 0, A<B, take 2s complement of result

A  =  100110       Final result (-6).
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A (-14)          101110
          ⇒                        ⇒  Same signs and subtract
-B (-14)      +   101110
                  ------

                  01110
                  10010            Add 2s complete of B to A.
                  -----
         E
         1 ←      00000    E = 1, and result zero

         A  =    000000     Final result (+0).
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A   ⇒        111011
+B          + 000111      ⇒ Different signs and addition
            ________

               11011      Take 2's complement of B and add.
               11001
            ________
             E
          1 ← 10100       E = 1 and result ≠ 0 ⇒ A > B

     A = 110100           Final result.
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A          010111
+B          101011     ⇒ Different signs and addition
           -------

             10111
             10101     Take 2s complement of B and add.
            ------
            E
          1 ← 01100    E = 1 and result ≠ 0 ⇒ A > B

     A = 001100        Final result.
```

Background Module3

- **Arithmetic Logic Unit** – Addition Algorithm (Signed magnitude numbers)
  - Example

```
 A     000111
-B     001011      Same signs and subtraction.
      -------

        00111
        10101      Take 2's complement of B and add.
      -------
       E
   0 <- 11100      E = 0 ⇒ B > A

A = 100100         Final result, take 2's complement
                   of result.
```

Background Module3

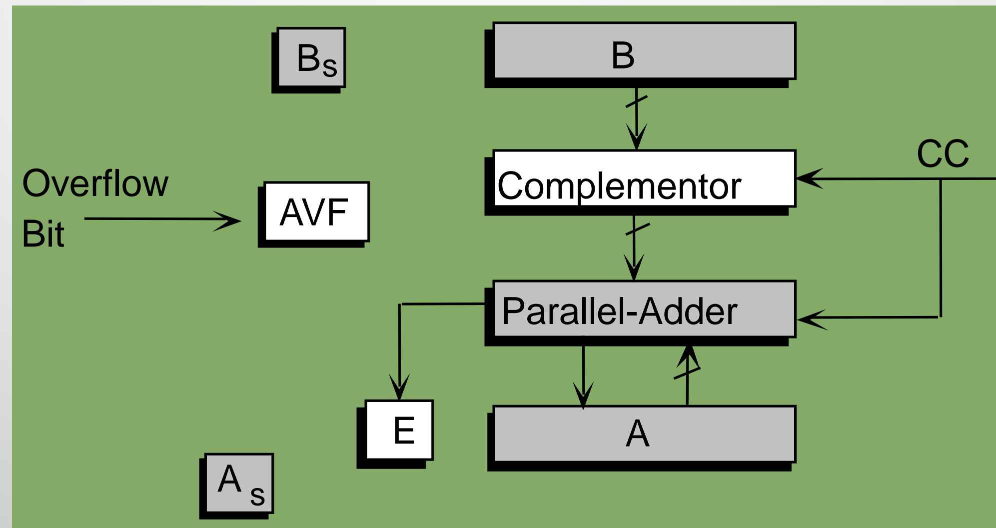
- Questions
  - What is an overflow?
  - How can we detect overflow for:
    - Signed magnitude addition?
    - 1^s complement addition?
    - 2^s complement addition?

Background Module3

- **Arithmetic Logic Unit** — Addition Overflow
  - Overflow occurs when two numbers of n digits each are added and the sum occupies $n+1$ digits.
  - An overflow cannot occur after an addition if one number is positive and the other is negative.
  - In general, overflow can only occur when adding numbers of the same sign or subtracting numbers of different signs.

Background Module3

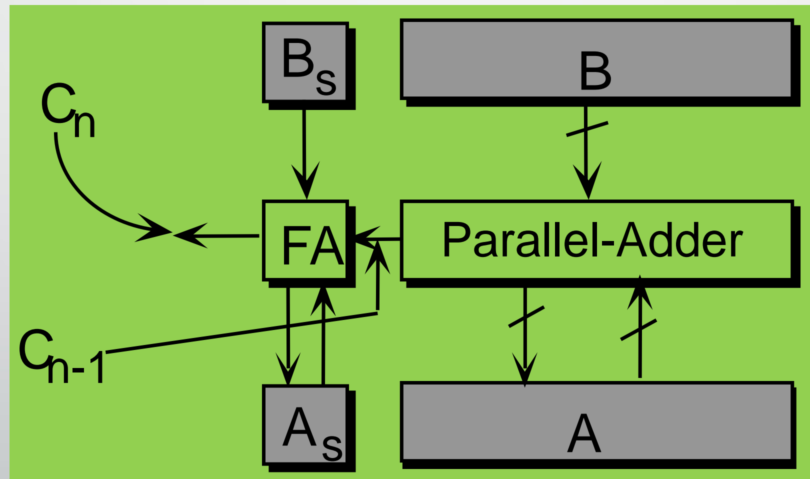
- **Arithmetic Logic Unit –** Addition Overflow (Signed magnitude numbers)
  - Detection of overflow in signed magnitude addition is simple. Carry-out of the magnitude bits represents an overflow:



If (cc = 0 and E = 1) then AVF $\Leftarrow$ 1 otherwise AVF $\Leftarrow$ 0

Background Module3

- **Arithmetic Logic Unit –** Addition Overflow (2s Complement Numbers)
  - Overflow is occurred when adding numbers of the same sign results in a number of a different sign.



  - Overflow occurs if $C_n \oplus C_{n-1} = 1$

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm
  - Multiplication can be performed as a sequence of additions.
  - B * Q $\Rightarrow$ add B, Q times
  - Two general approaches have been adapted for multiplication.
    - Software approach
    - Hardware approach

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm
  - Software approach

$AC \Leftarrow 0;$

For $i = 1$ to Q do;

$AC \Leftarrow (AC) + (B);$

End;

  - Time complexity of this algorithm is 0(m) where m is the magnitude of Q.

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm
  - Software approach
    - Time complexity of the algorithm can be improved if the number of iterations is based on the multiplier length:

$AC \Leftarrow 0;$

$\text{For } i = 1 \text{ to } n \text{ Do};$

$\quad \text{if } Q_1 = 1 \text{ then } C_{out}\, AC \Leftarrow (AC) + (B);$

$AC_n \ldots AC_2 AC_1 Q_n \ldots Q_2 Q_1 \Leftarrow C_{out} AC_n \ldots AC_2 AC_1 Q_n \ldots Q_2;$

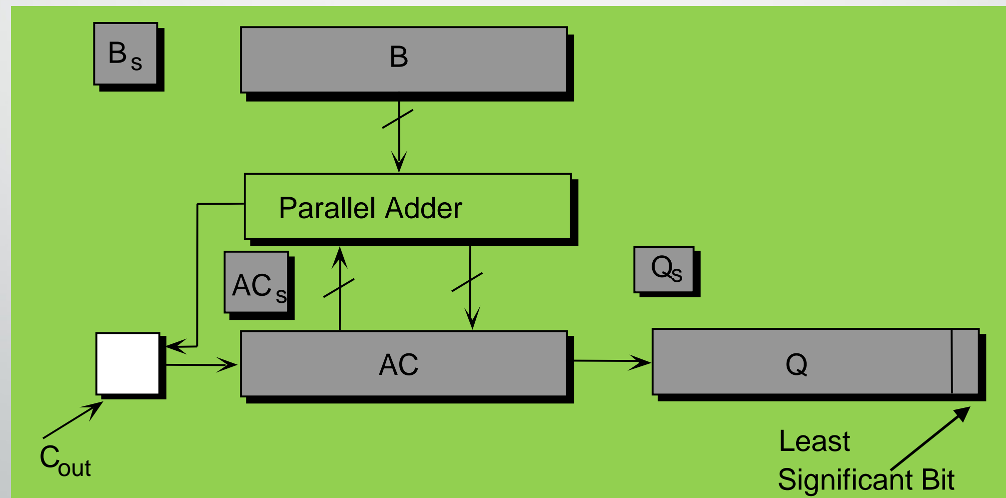
$\text{End};$

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm
  - Hardware Approach
    - The simplest hardware approach is based on the previous algorithm - e.g., add and shift.
      - In each iteration the least-significant bit of multiplier (Q) is checked;
      - If one, then B is added to the accumulator and the contents of accumulator and Q is shifted right one position.
      - If zero, just shift accumulator and Q to the right.

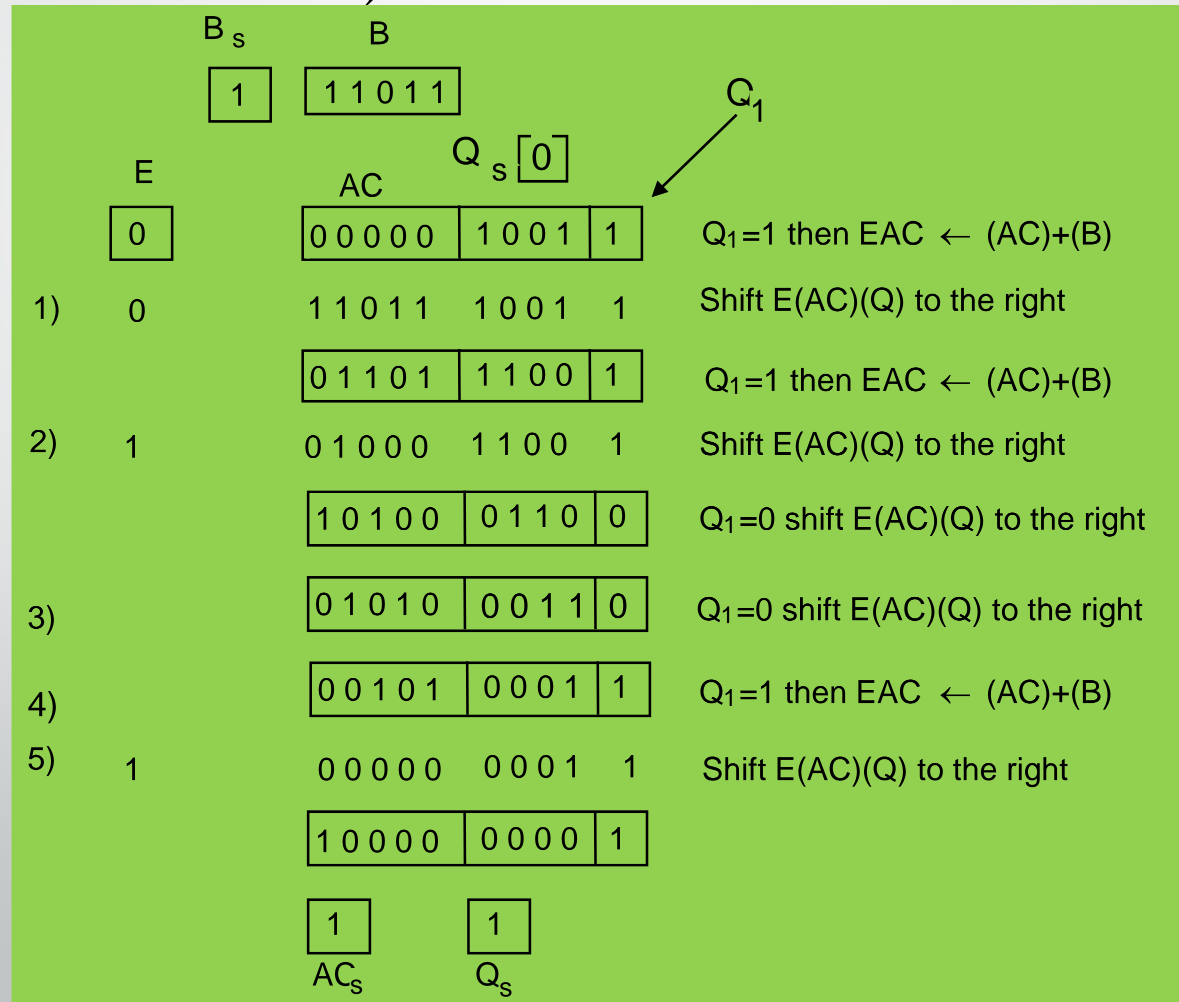
Background Module3

- **Arithmetic Logic Unit —** Multiplication Algorithm (Signed magnitude numbers)
  - Hardware Configuration
    - A double register (AC and Q) holds the final result.
    - Sign of the result is the EX-OR of the signs of multiplier and multiplicand.



Background Module3

- **Arithmetic Logic Unit —** Multiplication Algorithm (Signed magnitude numbers)
  - Example



Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm (2^s Complement numbers)
  - Pre-Post Complementation
    - Convert negative numbers to positive numbers.
    - Apply "add and shift" algorithm.
    - Convert the result - if necessary.

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm (2^s Complement numbers)
  - Extension of "add and shift"
    - Convert the multiplier to a positive number - i.e., if multiplier is negative then multiply the multiplier and the multiplicand by -1.
    - Apply a modified "add and shift" algorithm in which addition and shift operations are done for 2^s complement numbers.

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm (2s Complement numbers)
  - Booth's Algorithm
    - Instead of checking one bit of the multiplier at a time, check two bits at a time and take proper actions according to the following table:

00	no action, shift right.
01	add multiplicand, shift right.
10	subtract multiplicand, shift right.
11	no action, shift right.

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm (2^s Complement numbers)
  - Booth's Algorithm
    - It is suitable for 2^s complement numbers.
    - On average, Booth's Algorithm is faster than "add and shift".
    - Always, extend the multiplier by a zero on the right.

Background Module3

- **Arithmetic Logic Unit –** Multiplication Algorithm (2s Complement numbers)
  - Booth's Algorithm (Example)

Extension

1101101

0000000 | 0000111 | 0

0010011 — 1st 2 bits are 10 ⇒ subtract (add 2s complement)

0010011 0000111 0 — Shift right
0001001 1 000011 1 — 1st 2 bits are 11 ⇒ shift right
0000100 11 00001 1 — 1st 2 bits are 11 ⇒ shift right
0000010 011 0000 1 — 1st 2 bits are 01 ⇒ add
1101101

1101111 011 0000 1 — Shift right
1110111 1011 000 0 — 1st 2 bits are 00 ⇒ shift right
1111011 11011 00 0 — 1st 2 bits are 00 ⇒ shift right
1111101 111011 0 0 — 1st 2 bits are 00 ⇒ shift right

A 1111110 | 1111011 | 0

B * Q

Background Module3

- **Questions**
  - Why is Booth's algorithm faster than "add and shift" algorithm (in general)?
  - Why does Booth's Algorithm work?

Background Module3

- **Arithmetic Logic Unit —** Division Algorithm
  - Similar to multiplication, one can develop a software routine which performs division as a sequence of subtractions. Naturally, such an algorithm is very inefficient and slow.

Background Module3

- **Arithmetic Logic Unit –** Division Algorithm
  - Similar to the pencil and paper routine, one can develop an algorithm which performs division as a sequence of Compare, Shift, and Subtract operations.
  - One should note that division in a binary system is much simpler than the division in decimal system, since the quotient digits are either 0 or 1.

Background Module3

- **Arithmetic Logic Unit —** Division Algorithm
  - To minimize the hardware requirements for division, we should remember that:
    - Comparison can be performed via arithmetic operation(s).
    - Subtraction can be performed via complement-addition.
  - In other words; division requires almost the same hardware modules as multiplication does.

Background Module3

- **Arithmetic Logic Unit –** Division Algorithm (Signed magnitude numbers)
  - Division can be carried out as a sequence of n (n is the length of divisor) iterations.
  - Dividend is a double register.
  - One bit of the quotient is generated in each iteration.
  - At the end of the operation, the quotient is in the 1st half part of the double register (low-order part), and remainder is in the 2nd half part.
  - Sign of the quotient is the XOR of the signs of dividend and divisor.
  - Sign of the remainder is the same as the sign of the dividend.

Background Module3

- **Arithmetic Logic Unit** – Division Algorithm (Divide Overflow)
  - In the division algorithm for signed magnitude numbers, if the 1st half part of dividend is larger than or equal to the divisor, then the quotient overflows to the remainder part. This phenomenon is called divide overflow. As a result, a wrong answer will be generated.
  - To avoid divide overflow and also due to the fact that division is a time consuming operation, the divide overflow condition is checked at the beginning of the operation. In case of divide overflow, proper actions will be taken (in the form of say an interrupt).

Background Module3

- **Arithmetic Logic Unit –** Methods of Division
  - There are several different algorithms for division:
    - Restoring Method
    - Non-Restoring Method
    - Direct Comparison

Background Module3

- **Arithmetic Logic Unit —** Methods of Division
  - Restoring Method: In the restoring method, the contents of the partial remainder is restored whenever it is detected that the divisor is larger than the partial remainder.

Background Module3

- **Arithmetic Logic Unit –** Methods of Division
  - Non-Restoring Method: In non-restoring method, if it is detected that the divisor is larger than the partial remainder, its contents are not restored.
  - However, in the next iteration, instead of subtracting the divisor from the partial remainder, it will be added to the partial remainder.

Background Module3

- **Arithmetic Logic Unit –** Methods of Division
  - Direct Comparison: divisor is compared against partial remainder. If it is smaller than or equal to partial remainder, it will be subtracted and quotient digit is set to 1. Otherwise, just the quotient digit is set to zero.

Background Module3

- **Arithmetic Logic Unit** – Methods of Division
  - Naturally, restoring and non-restoring techniques are equivalent.
  - Consider the consecutive sequence of iterations for both restoring and non-restoring techniques (next slide).
  - As can be seen, both schemes generate the same value as the partial remainder. However, the non-restoring technique should be faster than the restoring method since it requires fewer number of operations.

Background Module3

- **Arithmetic Logic Unit —** Methods of Division

Restoring

Subtract in the second Iteration

(2 ((A-B)+B) -B) =2A-B

Subtract, Restore, and Shift in the first iteration

Non-Restoring

Addition in the second Iteration

2 (A-B) +B =2A-B

Subtraction and shift in the first iteration

Background Module3

- **Arithmetic Logic Unit –** Division Algorithm (Example)
  - Restoring Method

Extended Sign Bit → E

E	A	Q (Q_n)	B / Operation
0	0000	1011	0011
0	0000	1011	Shift left EAQ.
0	0001	011 □	Subtract B.
1	1101		
1	1110	011 □	E=1 $\Rightarrow Q_1 \leftarrow 0$.
0	0011		Restore A.
0	0001	0110	Shift left EAQ.
0	0010	110 □	Subtract B.
1	1101		
1	1111	110 □	E=1 $\Rightarrow Q_1 \leftarrow 0$.
0	0011		Restore A.
0	0010	1100	Shift left EAQ.
0	0101	100 □	Subtract B.
1	1101		
0	0010	100 □	E=0 $\Rightarrow Q_1 \leftarrow 1$.
0	0010	1001	Shift left EAQ.
0	0101	001 □	Subtract B.
1	1101		
0	0010	001 □	E=0 $\Rightarrow Q_1 \leftarrow 1$.
0	0010	0011	
	R	Q	

Background Module3

- **Arithmetic Logic Unit — ** Division Algorithm (Example)
  - Non-restoring Method

E	A	Q	B
0	0000	1011	0011
0	0000	1011	Shift left EAQ.
0	0001	011☐	Subtract B.
1	1101		
1	1110	011☐	E=1 ⇒ Q_1 ← 0.
1	1110	0110	Shift left EAQ.
1	1100	110☐	Add B.
0	0011		
1	1111	110☐	E=1 ⇒ Q_1 ← 0.
1	1111	1100	Shift left EAQ.
1	1111	100☐	Add B.
0	0011		
0	0010	100☐	E=0 ⇒ Q_1 ← 1.
0	0010	1001	Shift left EAQ.
0	0101	001☐	Subtract B.
1	1101		
0	0010	0011	E=0 ⇒ Q_1 ← 1.
0	0010	0011	
	R	Q	

Background Module3

- **Arithmetic Logic Unit —** Floating Point Numbers
  - A number *N* in base *r* can be represented in many ways. For example, decimal number 2.945 can be represented as:

$.2945 * 10^{+1}$

$29.45 * 10^{-1}$

•

•

•

  - A scientific or floating point format of a number is a unique representation in which a number is represented as the multiple of 2 parts:

a power of the base (called exponent)* a fraction (called mantissa)

  - For example, $+16.23 = +.1623 * 10^{+2}$

Background Module3

- **Arithmetic Logic Unit** – Floating Point Numbers
  - Such a representation in the computer requires two registers, one to represent the exponent and one to represent the mantissa.
  - Note that each part is a signed number which could be represented in signed magnitude or signed complement format.
  - A floating point number is normalized if the most significant digit of the mantissa is non-zero.
  - In most systems the exponent is represented in excess base. This means positive exponent. Therefore, a fixed value (e.g., biased value) will be added to the exponent. For example, IBM uses excess 64 for the power, and DEC10 uses excess 128 to represent the power.

Background Module3

- **Arithmetic Logic Unit –** Floating Point Numbers
  - Convert $(-25.75)_{10}$ to a floating point number for IBM:

$$(-25.75)_{10} = -(11001.11)_2 = -(.1100111 * 2^5)_2 =$$

$$(.0001100111 * 2^8)_2 = - .000110011100 * 16^2$$

Background Module3

- **Arithmetic Logic Unit –** Floating Point Addition and Subtraction
  - Check for zeroes.
  - Align the mantissas (smaller exponent should be equated to the larger one).
  - Add or subtract the mantissas.
  - Normalize the result.
  - If $exp_1 \geq exp_2$ then

$$fl_1 \pm fl_2 = \left((mantissa_1) \pm (mantissa_2) * r_b^{-(exp_1 - exp_2)} \right) * r_b^{exp_1}$$

  - If $exp_1 < exp_2$ then

$$fl_1 \pm fl_2 = \left((mantissa_1) * r_b^{-(exp_2 - exp_1)} \pm (mantissa_2) \right) * r_b^{exp_2}$$

Background Module3

- **Arithmetic Logic Unit –** Floating Point Multiplication
  - Check for zeros.
  - Add the exponents.
  - Adjust the exponent of the result.
  - Multiply the mantissas.
  - Normalize the result.

$$fl_1 * fl_2 = (mantissa_1) * (mantissa)_2 * r_b^{(exp_1 + exp_2)}$$

Background Module3

- **Arithmetic Logic Unit –** Floating Point Division
  - Check for zeroes.
  - Align the dividend (for divide overflow condition).
  - Subtract the exponents.
  - Adjust the exponent of the result.
  - Divide the mantissas.

$$fl_1 / fl_2 = (mantissa_1) / (mantissa)_2 * r_b^{(exp_1 - exp_2)}$$

Background Module3

- **Arithmetic Logic Unit**
  - Singed magnitude number: A method to represent signed numbers.
  - 2s complement number: A method to represent signed numbers.
  - 1s complement number: A method to represent signed numbers.
  - Addition overflow: A case where the addition result is too big to fit in the destination register.

Background Module3

- **Arithmetic Logic Unit**
  - Divide overflow: A case when the quotient result is too big to fit in the quotient register.
  - Add-and-shift algorithm: A method to perform multiplication.
  - Booth's algorithm: A method to perform multiplication for 2^s complement numbers.
  - Restoring technique: A method to perform division.
  - Non-restoring technique: A faster method than restoring technique to perform division.

Background Module3

- **Arithmetic Logic Unit**
  - Biased value: A fixed and positive value added to the exponent of a floating point number. The goal is to deal with positive (unsigned) exponents.
  - Normalized number: A floating point number is normalized if the first digit in the mantissa is a non zero digit.
  - Aligning the mantissa: A process by which the mantissa is shifted to the right by the number of differences between two exponents.
  - Mantissa: The fraction portion of a floating point number.

Background Module3

- Questions:
  - Prove that two unsigned numbers can be compared by means of 1s complement addition.
  - How do we detect overflow for signed magnitude addition?
  - How do we detect overflow for 2s complement addition (note I am looking for another technique than the one discussed in this unit)?
  - Prove that the add-and-shift algorithm, as implemented in this unit, does not generate an overflow.
  - Justify the following statement. On average, Booth's algorithm is faster than add-and-shift algorithm.

Background Module3

- Questions:
  - On average, Booth's algorithm is faster than add-and-shift algorithm, this means that in some cases Booth's algorithm would be slower than add-and-shift algorithm. Can you give an example that supports this?
  - Add-and-shift checks one bit of multiplier during each iteration, Booth's algorithm checks two bits of multiplier during each iteration and as we discussed Booth's algorithm is faster that add-and-shift. Can you come up with an algorithm that multilies two numbers faster than Booth's algorithm?

Background Module3

- Questions:
  - In case of a floating point addition or subtraction operation, why should we always align the mantissa of the smaller number?
  - In the case of multiplication of two floating point numbers, after adding the exponents, why should we subtract one excess base from the result?
  - In the case of division of two floating point numbers, after subtracting the exponents, why should we add one excess base to the result?