UNIT 3 – ARITHMETIC AND LOGIC UNIT

Introduction

- ALU is that part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out.
- An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

ALU Inputs and Outputs

- Figure 3.1 indicates how the ALU is interconnected with the rest of the processor
- Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored back in registers
- These registers are temporary storage locations within the processor that are connected by signal paths to the ALU
- The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

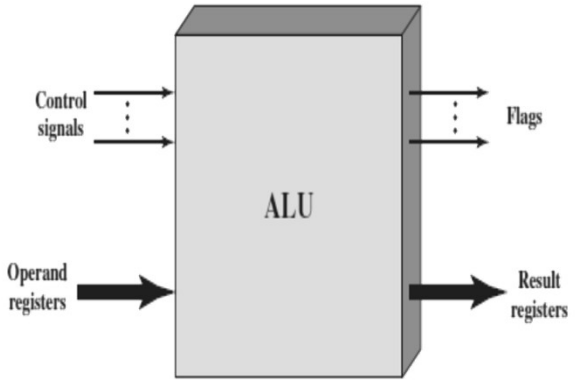


Fig 3.1 ALU Inputs and Outputs

Hardware for Addition and Subtraction

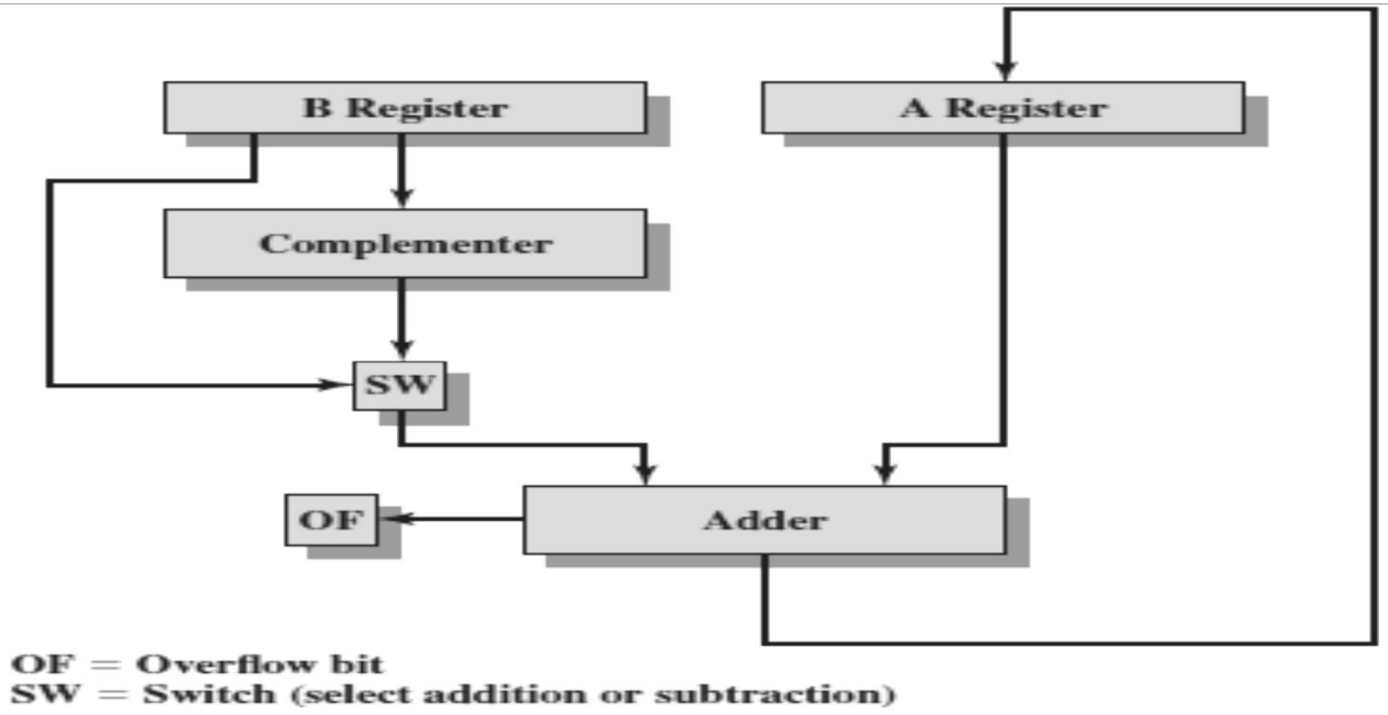


Fig 3.2 Block Diagram of Hardware for Addition and Subtraction

Hardware for Addition and Subtraction

- The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication
- The binary adder treats the two numbers as unsigned integers
- For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers
- The result may be stored in one of these registers or in a third
- The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow)
- For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder.

Multiplication

- Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software
- A wide variety of algorithms have been used in various computers
- We begin with simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation

Multiplication of Unsigned Integers

- Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
- The multiplication of two *n*-bit binary integers results in a product of up to 2*n* bits in length (e.g., 11 * 11 = 1001).

Multiplication of Unsigned Integers

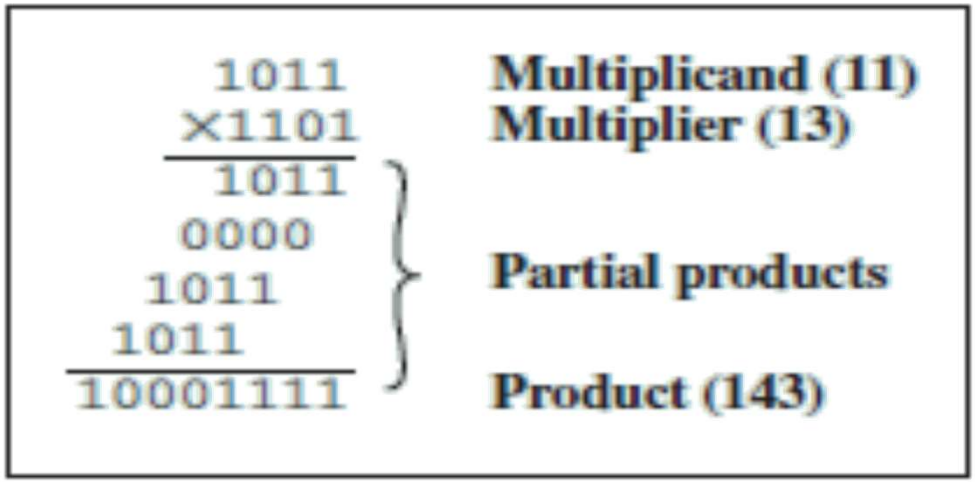


Figure 3.3 Multiplication of Unsigned Integers

Example 2

Multiply 15 and 6

Hardware Implementation of Unsigned Binary Integer

- Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient
- First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed.
- Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required

Hardware Implementation of Unsigned Binary Integer

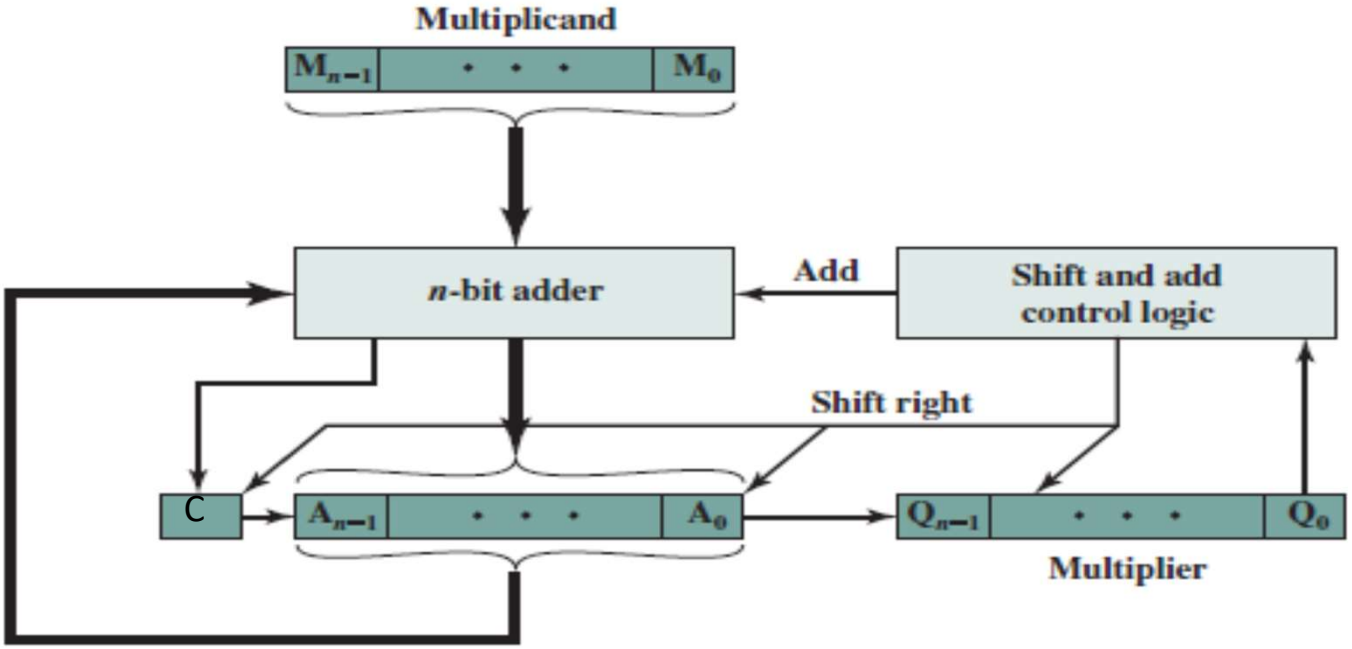


Figure 3.4 Block Diagram

Hardware Implementation of Unsigned Binary Integer

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

Figure 3.5 Example

Hardware Implementation of Unsigned Binary Integer

- The multiplier and multiplicand are loaded into two registers (Q and M)
- A third register, the A register, is also needed and is initially set to 0
- There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition

Operation of the Multiplier

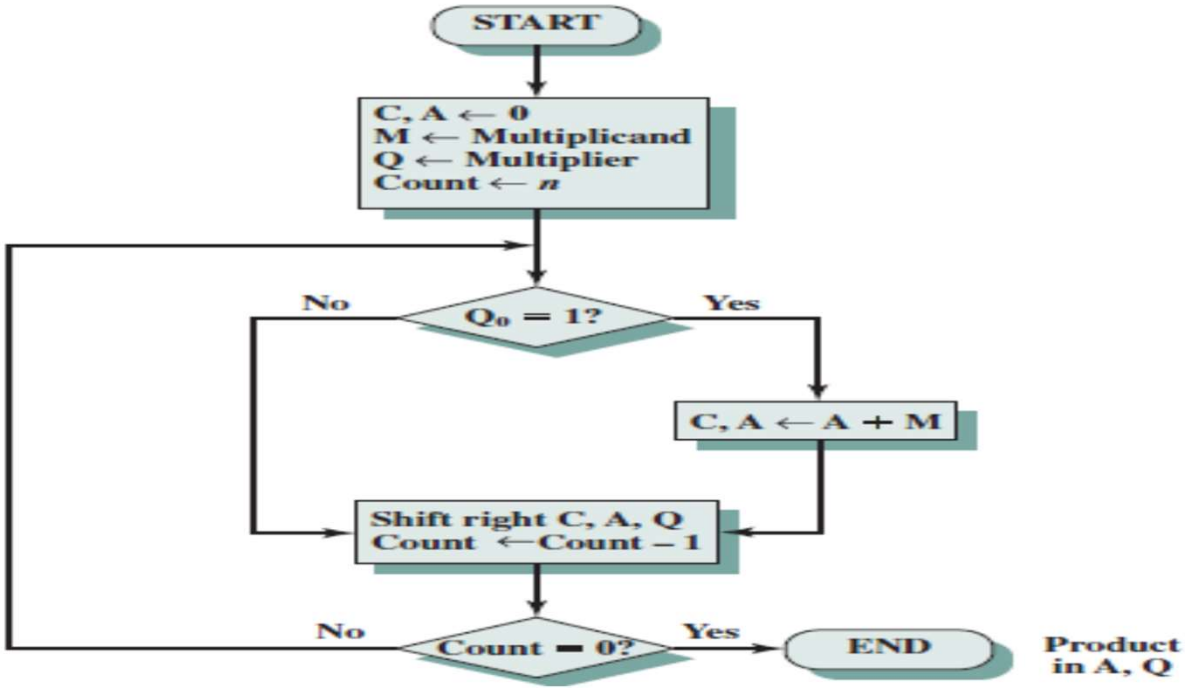
- Control logic reads the bits of the multiplier one at a time
- If Q0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow
- Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into An-1, A0 goes into Qn-1, and Q0 is lost
- If Q0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier
- The resulting 2n-bit product is contained in the A and Q registers
- Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

Example 2

Multiply 15 and 6

Operation of the Multiplier

Figure 3.6 Flowchart



2's Complement Multiplication

- We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers
- If these numbers are considered to be unsigned integers, then we are adding

  9 (1001) + 3 (0011) = 12 (1100)

- As twos complement integers, we are adding

  - 7(1001) + 3 (0011) = - 4(1100)

2's Complement Multiplication

- Unfortunately, this simple scheme will not work for multiplication
- We multiplied 11 (1011) X 13 (1101) = 143(10001111)
- If we interpret these as twos complement numbers, we have

-5(1011) X - 3 (1101) = - 113 (10001111)

This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative

2's Complement Multiplication

- Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

  $1101 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 2^3 + 2^2 + 2^0$

- Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits.
- This technique is used to make the generation of partial products by multiplication explicit
- The only difference is that it recognizes that the partial products should be viewed as 2n-bit numbers generated from the n-bit multiplicand

2's Complement Multiplication

- Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011
- Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

```
      1011
    × 1101
  --------
  00001011   1011 × 1 × 2^0
  00000000   1011 × 0 × 2^1
  00101100   1011 × 1 × 2^2
  01011000   1011 × 1 × 2^3
  --------
  10001111
```

Figure 3.7 Multiplication of Two Unsigned 4-bit Integers yielding an 8-bit result

Example 2

Multiply 15 and 6

2's Complement Multiplication

- Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative
- The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a 2n-bit field; the sign bits of the partial products must line up
- This is demonstrated in Figure 3.8, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of 9 * 3 = 27 proceeds simply
- However, if 1001 is interpreted as the twos complement value - 7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in Figure 3.8b
- Note that this is accomplished by padding out each partial product to the left with binary 1s.

2's Complement Multiplication

1001 (9) x 0011 (3) 00001001 1001 x 2^0 00010010 1001 x 2^1 00011011 (27)	1001 (-7) x 0011 (3) 11111001 (-7) x 2^0 = (-7) 11110010 (-7) x 2^1 = (-14) 11101011 (-21)

Figure 3.8 Comparison of Multiplication of Unsigned and Twos Complement Integers

2's Complement Multiplication

- If the multiplier is negative, straightforward multiplication also will not work
- The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place.

For example, the 4-bit decimal number - 3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$1101 = (1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0) = -(2^3 + 2^2 + 2^0)$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

2’s Complement Multiplication

- Solution:
- First solution is both multiplier and multiplicand can be converted to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed
- Implementers have preferred to use techniques that do not require this final transformation step
- Second solution is to use Booth’s algorithm
- This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach

Booth's Algorithm

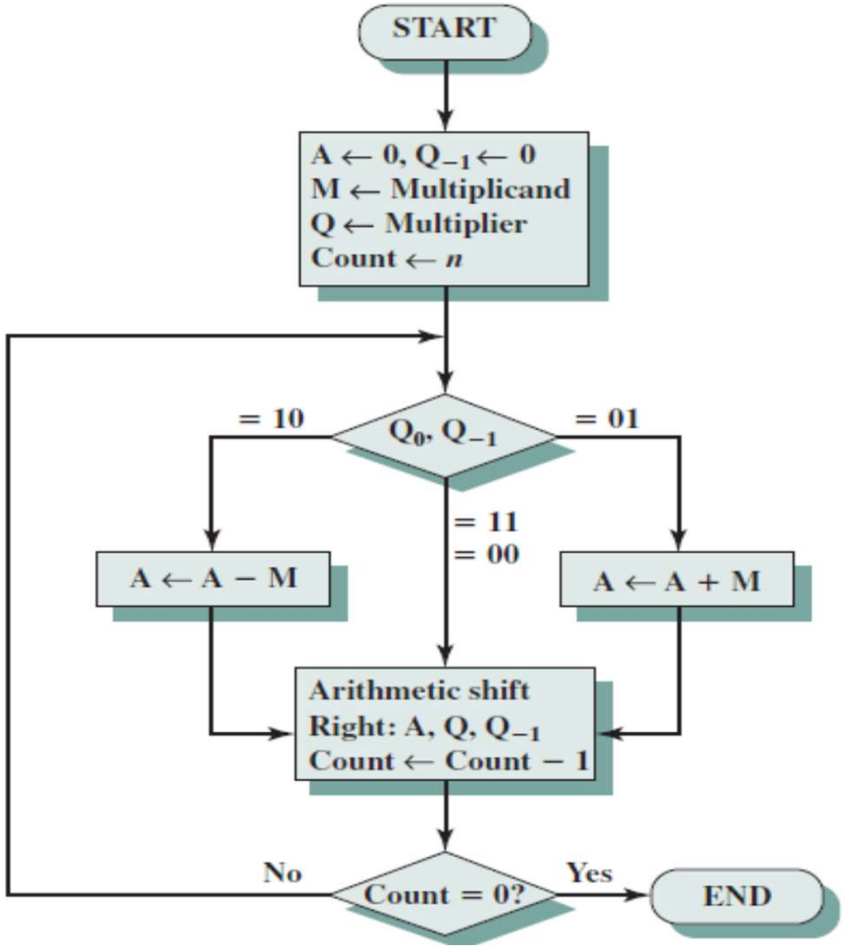


Figure 3.9 Booth's Algorithm for 2's Complement Multiplication

Booth's Algorithm

1) The multiplier and multiplicand are placed in the Q and M registers, respectively
2) There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}
3) The results of the multiplication will appear in the A and Q registers
4) Initially ,A and Q_{-1} are initialized to 0
5) The control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined
6) If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q-1 registers are shifted to the right 1 bit.

Booth's Algorithm

7) If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0
8) In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}
9) This is required to preserve the sign of the number in A and Q. It is known as an arithmetic shift, because it preserves the sign bit

Booth's Algorithm

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**

Figure 3.10 Example of Booth's Algorithm(7*3)

Booth's Algorithm

For (-7)x(3)

Booth's Algorithm

For 7x(-3)

Booth's Algorithm

For (-7)x(-3)

Booth's Algorithm

For (5)x(-4)

Booth's Algorithm

For (-5)x(-4)

Why does Booth's algorithm works?

- Consider a positive multiplier consisting of one block of 1s surrounded by 0s (e.g., 00011110)
- As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

- The number of such operations can be reduced to two if we observe that

  $2^n + 2^{n-1} + \ldots + 2^{n-K} = 2^{n+1} - 2^{n-K}$

Why does Booth's algorithm works?

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

- So the product can be generated by one addition and one subtraction of the multiplicand
- This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Why does Booth's algorithm works?

- Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1)
- The same scheme works for negative multiplier also