

WEEK- 7

2.3 Signed Integer Representations.

- The conversions we have so far presented have involved only positive numbers.
- To represent negative values, computer systems allocate the high-order bit to indicate the sign of a value.
 - The high-order bit is the leftmost bit in a byte. It is also called the most significant bit.
- The remaining bits contain the value of the number.
- There are three ways in which signed binary numbers may be expressed:
 - ***Signed magnitude,***
 - ***One's complement***
 - ***Two's complement.***
- In an 8-bit word, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit.
- For example, in 8-bit signed magnitude, positive 3 is: 00000011
- Negative 3 is: 10000011
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
 - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.
- Binary addition is as easy as it gets. You need to know only four rules:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

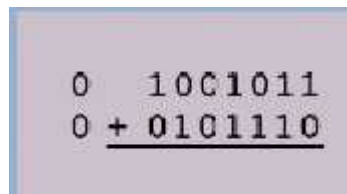
- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.

Let's see how the addition rules work with signed magnitude numbers.

- Example:

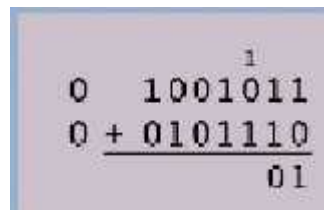
Using signed magnitude binary arithmetic, find the sum of 75 and 46.

- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.



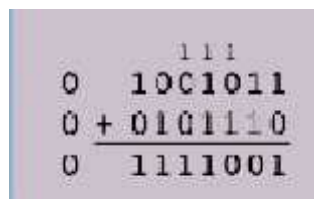
A screenshot of a digital logic simulator showing the addition of two signed magnitude numbers. The first number is 0 1001011 (representing +75) and the second number is 0 + 0101110 (representing +46). The numbers are aligned for addition, with a horizontal line under the second number.

- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.



A screenshot of the same digital logic simulator showing the first step of the addition. A carry of 1 is shown above the third bit position. The result of the first bit addition (0 + 0) is 01, shown below the horizontal line.

- In the second bit, we have a carry, so we note it above the third bit.
- The third and fourth bits also give us carries.



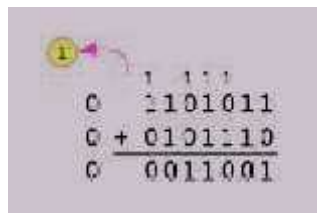
A screenshot of the digital logic simulator showing the third step of the addition. Carries of 111 are shown above the fifth bit position. The result of the third bit addition (0 + 0 + carry 1) is 111, shown below the horizontal line.

- Once we have worked our way through all eight bits, we are done.

In this example, we were careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

- Example:

- Using signed magnitude binary arithmetic, find the sum of 107 and 46.



Handwritten binary addition showing a carry overflow from the seventh bit:

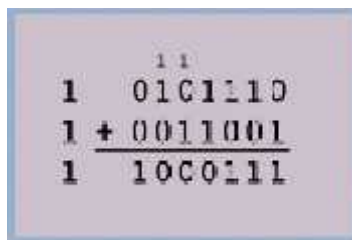
$$\begin{array}{r}
 0 \quad 1101011 \\
 0 + 0101110 \\
 \hline
 0 \quad 0011001
 \end{array}$$

A yellow circle with the number '1' and a pink arrow points to the carry bit '1' above the seventh column.

- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: $107 + 46 = 25$.

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.

- Example: Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.



Handwritten binary addition for negative numbers in signed magnitude representation:

$$\begin{array}{r}
 1 \quad 0101110 \\
 1 + 0011001 \\
 \hline
 1 \quad 1000111
 \end{array}$$

- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.
 - Mixed sign addition (or subtraction) is done the same way.
- Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$\begin{array}{r}
 \begin{array}{c} 02 \quad 02 \\ 0 \quad 0101110 \end{array} \\
 1 + 0011001 \\
 \hline
 0 \quad 0010101
 \end{array}$$

- The sign of the result gets the sign of the number that is larger.
 - Note the “borrows” from the second and sixth bits.
- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.
- In complement systems, negative values are represented by some difference between a number and its base.
- In *diminished radix complement* systems, a negative value is given by the difference between the absolute value of a number and one less than its base.

- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.
- For example:

In 8-bit one's complement positive 3 is: 00000011

Negative 3 is: 11111100

- In one's complement, as with signed magnitude, a 1 in the high order bit indicates negative values.

- Complement systems are useful because they eliminate the need for special circuitry for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.²

- With one's complement addition, the carry bit is “carried around” and added to the sum.

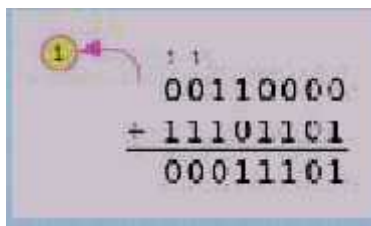
- Example: Using one's complement binary arithmetic; find the sum of 48 and – 19.

$$\begin{array}{r}
 \text{1} \quad 00110000 \\
 11101100 \\
 \hline
 00011100 \\
 + 1 \\
 \hline
 00011101
 \end{array}$$

We note that 19 in one's complement is 00010011, so -19 in one's complement is: 11101100.

- Although the “end carry around” adds some complexity, one's complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.

- Two's complement solves this problem.
- Two's complement is the *radix complement* of the binary numbering system.
- To express a value in two's complemented:
 - If the number is positive, just convert it to binary and you're done.
 - If the number is negative, find the one's complement of the number and then add 1.
- Example:
 - In 8-bit one's complement, positive 3 is: 00000011
 - Negative 3 in one's complement is: 11111100
 - Adding 1 gives us -3 in two's complement form: 11111101.
- With two's complement arithmetic, all we do is adding our two binary numbers. Just discard any carries emitting from the high order bit.



$$\begin{array}{r}
 00110000 \\
 + 11101101 \\
 \hline
 00011101
 \end{array}$$

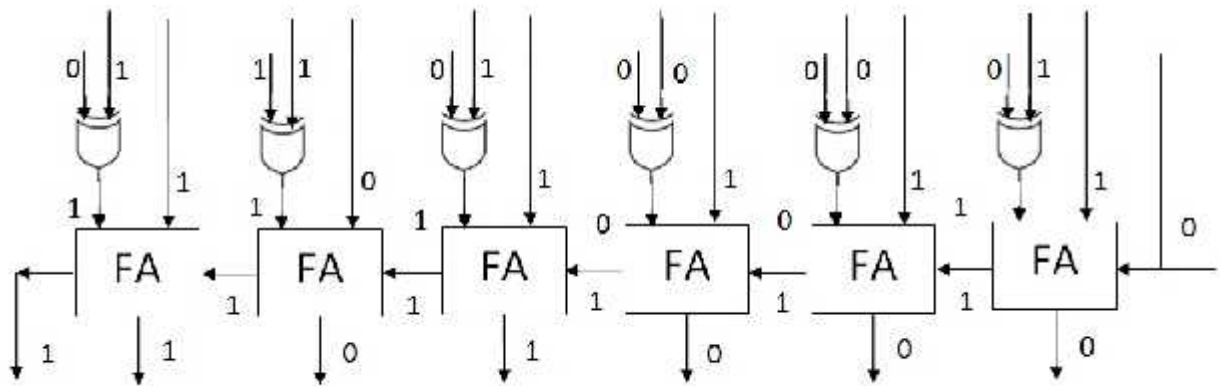
We note that 19 in one's complement is: 00010011.

So -19 in one's complement is: 11101100.

And -19 in two's complement is: 11101101.

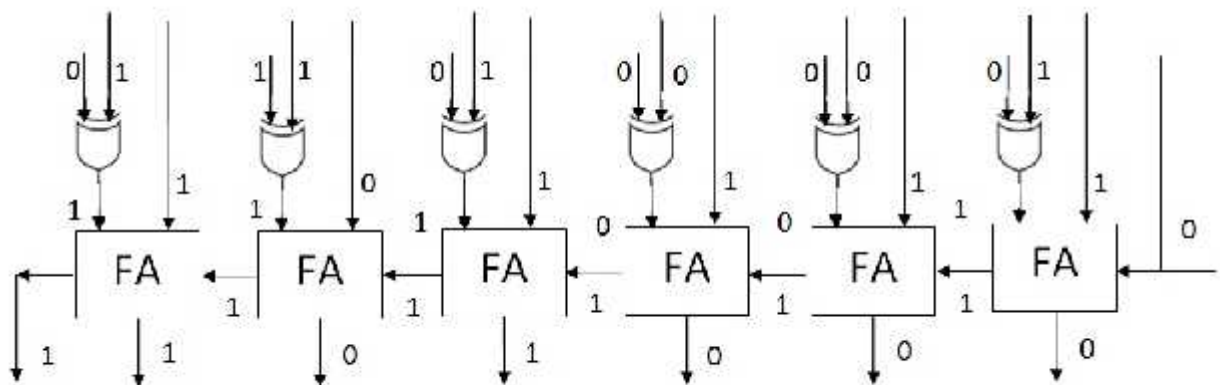
- Example-1:

Using two's complement binary arithmetic, find the sum of 57 and 47.



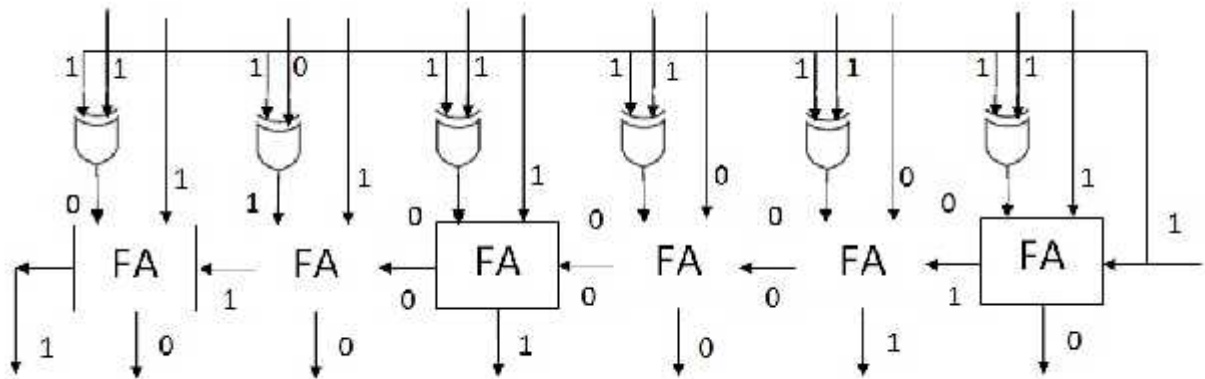
Example 2:

Using two's complement binary arithmetic, find the sum of - 57 and - 47.



Example3:

Using two's complement binary arithmetic, find the sum of 57 and
- 47.



Example-4:

Using two's complement binary arithmetic, find the sum of - 57
and 47.

