Lesson 4: Signed Integers

In Lesson 3, we explored the use of unsigned binary integers to encode nonnegative decimal integers in the computer. We ended Lesson 3 by exploring MATLAB's four data classes for unsigned integers: `uint8`, `uint16`, `uint32`, and `uint64`. In Lesson 4, we extend our study of MATLAB by investigating four new data classes used to store signed integers. As we will see, MATLAB's data encoding scheme for signed integers requires additional creative thought beyond a simple binary representation.

Addition of Unsigned Integers

Before we dive into a study of how to encode signed integers in MATLAB, let's take a detour. Specifically, let's explore the operation of addition on a pair of unsigned integers. To begin our discussion of addition, we consider a simplified problem set. There are a total of four possible outcomes when we add two separate 1−bit unsigned integers. All four problems are shown below.

$$\begin{array}{c} \text{binary} \\ \begin{array}{rr} & 0 \\ + & 0 \\ \hline & 0 \end{array} \end{array} \quad = \quad \begin{array}{c} \text{decimal} \\ \begin{array}{rr} & 0 \\ + & 0 \\ \hline & 0 \end{array} \end{array}$$

$$\begin{array}{rr} & 1 \\ + & 0 \\ \hline & 1 \end{array} \quad = \quad \begin{array}{rr} & 1 \\ + & 0 \\ \hline & 1 \end{array}$$

$$\begin{array}{rr} & 0 \\ + & 1 \\ \hline & 1 \end{array} \quad = \quad \begin{array}{rr} & 0 \\ + & 1 \\ \hline & 1 \end{array}$$

$$\begin{array}{rrr} & & 1 \\ + & & 1 \\ \hline & 1 & 0 \end{array} \quad = \quad \begin{array}{rr} & 1 \\ + & 1 \\ \hline & 2 \end{array}$$

In each case, the addition proceeds very similar to decimal addition. However, out of the four of these possible outcomes, the last one is the most interesting. In particular, in binary arithmetic we have $1 + 1 = 10$. Moreover, we see that when adding two separate 1−bit unsigned binary integers, we can produce a 2−bit sum. We can use this observation to define a *carry* for addition of n−bit unsigned binary integers. Let's explore some examples of how addition between unsigned numbers generalizes when adding two integers encoded using MATLAB's `uint8` data class.

EXAMPLE 4.1

In this example, let's consider the sum $139 + 91$ carried out as a sum between two variables stored using the `uint8` data class. Below is the desired sum in both binary and decimal arithmetic.

$$\begin{array}{ccccccccc} & \multicolumn{8}{c}{\text{binary}} \\ & 1 & 0 & \overset{1}{0} & \overset{1}{0} & 1 & \overset{1}{0} & \overset{1}{1} & 1 \\ + & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \quad = \quad \begin{array}{cccc} & \multicolumn{3}{c}{\text{decimal}} \\ & \overset{1}{1} & \overset{1}{3} & 9 \\ + & & 9 & 1 \\ \hline & 2 & 3 & 0 \end{array}$$

Notice that we are able to perform this sum on our unsigned binary integers using essentially the same method we use to add our two decimal integers. In both cases we track the carry digit and use this information to calculate the proper value of the next most significant digit.

Let's take a look at another example of addition of unsigned binary integers. However, in this case, we will consider a more interesting phenomenon.

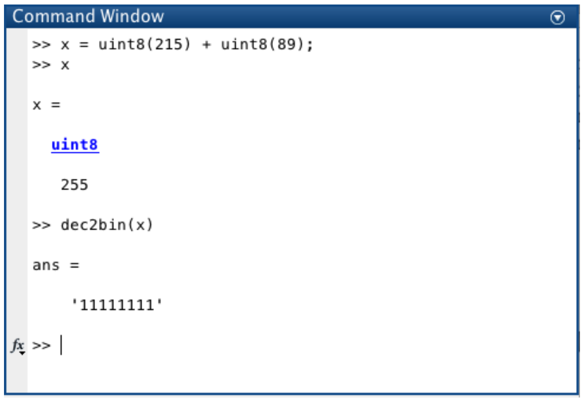
EXAMPLE 4.2

In this example, let's consider the sum $215 + 88$ carried out as a sum between two variables stored using the `uint8` data class. Below is the exact value of the sum in both binary and decimal arithmetic.

$$\begin{array}{cccccccccc} & & \multicolumn{8}{c}{\text{binary}} \\ & & \overset{1}{1} & 1 & \overset{1}{0} & \overset{1}{1} & \overset{1}{0} & \overset{1}{1} & \overset{1}{1} & 1 \\ + & & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline & \boxed{1} & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \quad = \quad \begin{array}{cccc} & \multicolumn{3}{c}{\text{decimal}} \\ & \overset{1}{2} & \overset{1}{1} & 5 \\ + & & 8 & 9 \\ \hline & 3 & 0 & 4 \end{array}$$

Here we have something very interesting. The carry over bit (highlighted in gray) on the most significant digit is nonzero and thus the exact binary result requires 9−bits to encode. However, the `uint8` data class only provides 8−bits of data to store our unsigned binary digits.

The example above provides our first insight into a more general condition that occurs on finite precision machines. In particular, the result of the addition problem above is a number whose binary representation is larger than can be stored in a word size of 8−bits. This phenomenon is called *overflow*. Let's take a look at what happens when we attempt to execute this addition using MATLAB:



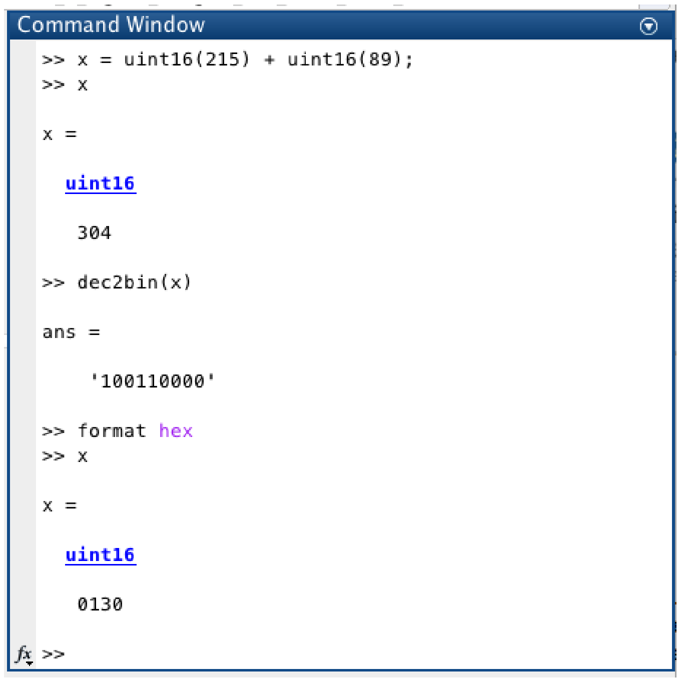
```
>> x = uint8(215) + uint8(89);
>> x

x =

  uint8

   255

>> dec2bin(x)

ans =

    '11111111'

>> |
```

Lesson 4, Figure 1: Overflow that occurs when adding two `uint8` integers

This result provides mathematical nonsense since MATLAB is claiming that $215 + 89 = 255$. However, the key understanding behind this addition is to recognize that overflow has occurred. As programmers, we need to be able to identify and address these type of issues as they arise in our work.

Range extension for unsigned integers

Example 4.2 above demonstrates that we should be careful to encode our data using a data class that is appropriate for our needs. Let's redo example 4.2 using MATLAB's `uint16` data class instead and see what happens.


```
>> x = uint16(215) + uint16(89);
>> x

x =

  uint16

   304

>> dec2bin(x)

ans =

    '100110000'

>> format hex
>> x

x =

  uint16

   0130

>>
```

Lesson 4, Figure 2: Fix overflow by encoding data using more bits via `uint16`

In the work above, we demonstrate a process of storing an 8−bit unsigned binary integer as a 16−bit unsigned binary integer. Let's take a look at the impact of this expansion on each of our addends from the overflow problem above:

decimal				binary																
2	1	5	=									1	1	0	1	0	1	1	1	(`uint8` encoding: 8-bits)
2	1	5	=	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	1	(`uint16` encoding: 16-bits)
	8	9	=									0	1	0	1	1	0	0	1	(`uint8` encoding: 8-bits)
	8	9	=	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	(`uint16` encoding: 16-bits)

We call the process of strategically expanding bit length *range expansion*, a term that represents the idea that by using more bits to encode our numbers, we can expand the range of unsigned integers that can be expressed and operated on correctly. Notice that by expanding our range, we can easily execute our addition problem because the carry continues into the 9−th most significant digit.

Signed Integer Representations

All of the work we've done so far has been focused on storing and manipulating nonnegative integers. Now, let's figure out how we might store and manipulate signed integers, which may take either positive or negative values. When working with computer arithmetic, there are several alternative conventions that we might use to store signed integers. All of these conventions depend on treating the most significant (leftmost) bit in our word as *sign bit.* If the sign bit is 0, then the number is positive. On the other hand, if the sign bit is 1, then the number is negative.

Simplest representation for signed integer (NOT used in MATLAB)

The simplest method we can use to represent signed integers called the *sign-magnitude representation.* In this encoding scheme, signed integers are represented as $n-$bit words where the rightmost $(n-1)$ bits store the magnitude of the integer and the most significant (leftmost) bit stores the sign. Let's take a look at how to encode positive and negative integers using $8-$bits via this convention:

$$\begin{array}{cccccccccccc} \multicolumn{8}{c}{\text{binary}} & & \multicolumn{3}{c}{\text{decimal}} \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & = & + & 9 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & = & - & 9 & 1 \end{array}$$

We can write this representation mathematically using sigma notation. Recall that for an $(m+1)-$bit sequence of binary digits in the form $y = b_m b_{m-1}...b_2 b_1 b_0$ where each digit b_i is either 0 or 1, if we interpreted y as an unsigned binary integer, then we could write the following sum

$$y = b_m \cdot 2^m + b_{m-1} \cdot 2^{m-1} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0 = \sum_{i=0}^{m} b_i \cdot 2^i$$

If, on the other hand, we interpret the sequence of $(m+1)-$bits given by $b_m b_{m-1}...b_2 b_1 b_0$ as a signed integer encoded using the signed-magnitude representation, then we have a different realization given by

$$y = (-1)^{b_m} \cdot \left(b_{m-1} \cdot 2^{m-1} + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0\right)$$

$$= \underbrace{(-1)^{b_m}}_{\text{sign}} \cdot \underbrace{\left(\sum_{i=0}^{m-1} b_i \cdot 2^i\right)}_{\text{magnitude}}$$

$$= \begin{cases} +\sum\limits_{i=0}^{m-1} b_i \cdot 2^i & \text{if } b_m = 0 \\ \\ -\sum\limits_{i=0}^{m-1} b_i \cdot 2^i & \text{if } b_m = 1 \end{cases}$$

There are two major drawbacks of the signed-magnitude representation. These include each of the following:

1. When adding signed integers that are encoded using the signed-magnitude representation, we must consider the signs and the magnitudes of both numbers being added together to correctly implement our addition.

2. There are two representations of the number zero in the signed-magnitude representation, given below. This ambiguity about the number 0 is inconvenient because it makes testing a value for zero more difficult. As we will see, testing a value for zero is a common operation that is used frequently in computation. Thus, we want this test to be as simple as possible.

binary		decimal
0 0 0 0 0 0 0 0	=	+ 0
1 0 0 0 0 0 0 0	=	− 0

Because of these drawbacks, the signed integers classes in MATLAB do not store signed integers in signed-magnitude representation. Instead, MATLAB uses the very popular scheme to store signed integers known as twos complement [1]

Twos complement representation for signed integer (used in MATLAB)

The twos complement encoding scheme for signed integers uses the most significant (leftmost) bit as a sign bit, where 0 encodes positive numbers and 1 encodes negative numbers. This feature makes it very easy to test whether a signed integer is positive or negative. However, the twos complement representation of signed integers differs from the signed-magnitude representation in a significant way. In particular, in the twos complement representation of signed integers, we weight the most significant bit using a very special scalar that provides desirable arithmetic properties and avoids the drawbacks of the signed-magnitude representation.

To begin our development of this encoding scheme, let's focus on the easiest case relevant to MATLAB data classes. Suppose that we want to encode a signed integer (either positive or negative) $y \in \mathbb{Z}$ using an $8-$bit encoding scheme. Then, the general binary representation for our integer will be as follows

$$y = b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

where $b_i \in \{0, 1\}$ for all $i = 0, 1, ..., 7$. If $y \geq 0$, we know that we want $b_7 = 0$. In this case, we the remaining $7-$bits will represent the magnitude of our integer $y \in \mathbb{Z}$ using our standard binary encoding (the same one we used to store unsigned integers and in our signed-magnitude representation). In other words, for $y \geq 0$, we have $b_7 = 0$ and

$$y = b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

$$= \sum_{i=0}^{6} b_i \cdot 2^i$$

This yields a range of nonnegative integer values $0 \leq y \leq 2^7 - 1$.

On the other hand, if we have a negative integer $y < 0$, the sign bit b_7 is set to one and we are free to define any encoding scheme we'd like for the remaining 7 bits. Because each bit has two possible values, we have a total of 2^7 different values that we can use encode negative integers using these 7 bits. Thus, we expect that with $8-$bits of information, we can assign negative values between $-1 \leq y \leq -2^7$.

[1] In computer science literature, the terms *two's complement* or *2's complement* often appear. However, the Institute of Electrical and Electronics Engineers (IEEE) publishes a number of documents to define standards and terms. In the *IEEE Std 100-1992, The New IEEE Standard Dictionary of Electrical and Electronics Terms*, the IEEE society suggests that we omit the apostrophe. We will adopt this convention here.

When designing our encoding scheme, we want to assign bit values to negative integers using an encoding scheme that makes arithmetic with negative numbers as simple as possible. Remember that in the unsigned integer representation `uint8`, we weight the most significant bit b_7 with the scalar 2^7. For our twos complement representation, it just so happens that if the weight on the most significant bit is -2^7, then our binary signed integer representation will exhibit a number of desirable arithmetic properties, as we will see in this lesson. In other words, the twos complement representation of signed integers is based on a weighted sum of bits, where we weight the most significant bit with a negative power of two. For our negative integer $y < 0$, we have

$$y = -b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

$$= -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i$$

Notice, that this weighted sum interpretation is compatible with the previous work we did. If $b_7 = 0$, then the first term $-b_7 \cdot 2^7 = 0$ and we get a nonnegative integer in the form mentioned above. If $b_7 = 1$, then we subtract 2^7 from the value of the summation and produce a negative number. Let's take a look at how to convert a few selected values of negative integers using this twos complement representation

EXAMPLE 4.3

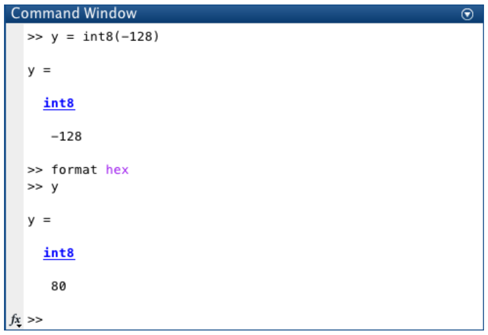
Let's take a look at how to encode the negative number $y = -128$ via the twos complement representation. We claim, without formal argumentation, that the proper encoding scheme for this number should be a binary number given by

$$10000000 = 1000\ \ 0000 = (80)_{16}$$

Let's confirm this claim by checking the math:

Weight (power)	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
Weight (decimal)	-128	64	32	16	8	4	2	1		
	1	0	0	0	0	0	0	0	=	-128
	twos compliment representation									decimal

We are now in a position to investigate MATLAB's use of twos complement to store 8−bit signed integers. The `int8` data class is used precisely for this purpose. Let's take a look at storing the number $y = -128$ using this data class:



```
Command Window
>> y = int8(-128)

y =

  int8

   -128

>> format hex
>> y

y =

  int8

   80

fx >>
```

Lesson 4, Figure 3: Using `int8` to store −128 in memory

EXAMPLE 4.4

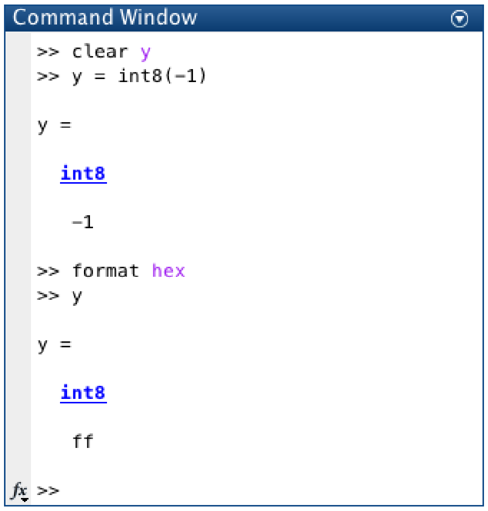
Let's use an 8−bit twos complement representation to encode the negative decimal integer $y = -1$. The proper encoding for -1 in this representation is

$$11111111 = 1111\ \ 1111 = (\text{ ff })_{16}$$

Let's confirm this claim by checking the math:

Weight (power)	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
Weight (decimal)	-128	64	32	16	8	4	2	1		
	1	1	1	1	1	1	1	1	=	-1
	twos compliment representation									decimal

Again, let's use the `int8` data class to store the number $y = -1$:



```
>> clear y
>> y = int8(-1)

y =

  int8

   -1

>> format hex
>> y

y =

  int8

   ff

>>
```

Lesson 4, Figure 4: Using `int8` to store -1 in memory

EXAMPLE 4.5

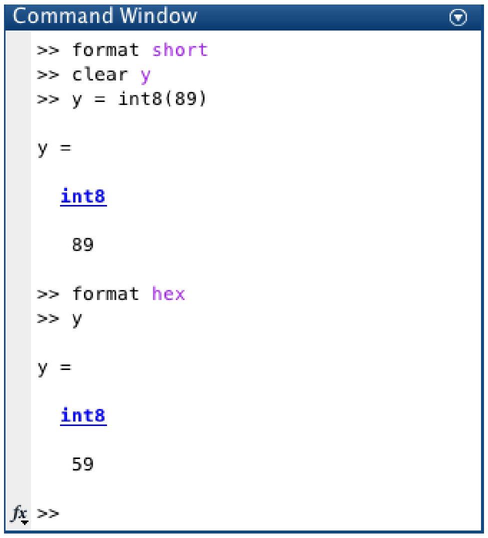
In our final example, let's confirm that the twos compliment representation used for the `int8` data class does indeed use our standard binary representation for nonnegative integers. We expect that the number $y = 89$ has the following 8−bit binary representation

$$0\,1\,0\,1\,1\,0\,0\,1 = 0\,1\,0\,1 \;\; 1\,0\,0\,1 = (\,5\,9\,)_{16}$$

Let's confirm this claim by checking the math:

Weight (power)	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
Weight (decimal)	-128	64	32	16	8	4	2	1		
	0	1	0	1	1	0	0	1	=	89
	twos compliment representation									decimal

Again, let's use the `int8` data class to store the number $y = 89$:



```
Command Window
>> format short
>> clear y
>> y = int8(89)

y =

  int8

   89

>> format hex
>> y

y =

  int8

   59

fx >>
```

Lesson 4, Figure 5: Using `int8` to store 89 in memory

The examples above demonstrate a useful mechanism we can use to convert twos complement representations back into decimal form. These tables are based on a more general illustration known as a *value box* in which the binary digit on the far right of the box is weighted by $2^0 = 1$ and the next most significant binary digit (to the left) has double the weight until the leftmost position is reached which has weight -2^7. The value box is a table without labels, as seen below as seen below:

-128	64	32	16	8	4	2	1
0	1	0	1	1	0	0	1

Convert from decimal to twos complement

The next natural question we might ask ourselves is how to convert a signed integer in decimal representation into the corresponding twos compliment representation. To do so, we will use a three-step process known as the *taking the twos complement of an integer*. Suppose we start with a nonnegative integer $0 \leq y \leq 2^7 - 1$. To find the value $x = -y$ and store this negative number via the twos complement representation, we use the following three steps:

0. Find the binary representation of $y = |x|$ as an 8−bit integer:
$$y = b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0 \qquad\qquad \text{where } b_7 = 0.$$
1. Take the *bitwise (Boolean) complement* of each bit of the binary string (including the sign bit b_7) by changing each 1 to 0 and each 0 to 1.
2. Treat the result of step 2 as an unsigned binary integer and add 1

Let's take a look at an example of how to accomplish this conversion.

EXAMPLE 4.6

Suppose we want to find an 8−bit twos complement representation of the number $x = -53$. To do so, we execute all three steps in our process above. We begin by finding the binary representation of $y = +53$

$$53 = 0\,0\,1\,1\,0\,1\,0\,1 = 0\,0\,1\,1\;\;0\,1\,0\,1 = (\,3\,5\,)_{16}$$

Now we find the bitwise complement of our binary number

binary representation of +53:	0	0	1	1	0	1	0	1
bitwise complement:	1	1	0	0	1	0	1	0

Now, to complete step 3, we add one to our result to find

$$\begin{array}{ccccccccc c}
 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
+ & & & & & & & & 1 \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1
\end{array}$$

With this, we see that the twos complement representation of $x = -53$ is given by $1\,1\,0\,0\,1\,0\,1\,1 = 1\,1\,0\,0\;\;1\,0\,1\,1 = (\,c\,b\,)_{16}$.

EXAMPLE 4.7

Suppose we want to find an 8−bit twos complement representation of the number $x = -120$. Once again, we run through our three-step algorithm. We begin by finding the binary representation of $y = +120 = |x|$, given by

$$120 = 0\,1\,1\,1\,1\,0\,0\,0 = 0\,1\,1\,1\;\;1\,0\,0\,0 = (\,7\,8\,)_{16}$$

Now we find the bitwise complement of our binary number

binary representation of +120:	0	1	1	1	1	0	0	0
bitwise complement:	1	0	0	0	0	1	1	1

Now, to complete step 3, we add one to our result to find

$$\begin{array}{ccccccccc}
 & 1 & 0 & 0 & 0 & \overset{1}{0} & \overset{1}{1} & \overset{1}{1} & 1 \\
+ & & & & & & & & 1 \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}$$

With this, we see that the twos complement representation of $x = -53$ is given by $1\,0\,0\,0\,1\,0\,0\,0 = 1\,0\,0\,0\;\;1\,0\,0\,0 = (\,8\,8\,)_{16}$.

Decimal Representation	8−bit Twos Complement Representation	Hexadecimal Representation
128	None	None
127	0 1 1 1 1 1 1 1	7 f
126	0 1 1 1 1 1 1 0	7 e
125	0 1 1 1 1 1 0 1	7 d
124	0 1 1 1 1 1 0 0	7 c
123	0 1 1 1 1 0 1 1	7 b
122	0 1 1 1 1 0 1 0	7 a
121	0 1 1 1 1 0 0 1	7 9
120	0 1 1 1 1 0 0 0	7 8
119	0 1 1 1 0 1 1 1	7 7
118	0 1 1 1 0 1 1 0	7 6
117	0 1 1 1 0 1 0 1	7 5
116	0 1 1 1 0 1 0 0	7 4
115	0 1 1 1 0 0 1 1	7 3
114	0 1 1 1 0 0 1 0	7 2
113	0 1 1 1 0 0 0 1	7 1
112	0 1 1 1 0 0 0 0	7 0
⋮	⋮	⋮
8	0 0 0 0 1 0 0 0	0 8
7	0 0 0 0 0 1 1 1	0 7
6	0 0 0 0 0 1 1 0	0 6
5	0 0 0 0 0 1 0 1	0 5
4	0 0 0 0 0 1 0 0	0 4
3	0 0 0 0 0 0 1 1	0 3
2	0 0 0 0 0 0 1 0	0 2
1	0 0 0 0 0 0 0 1	0 1
0	0 0 0 0 0 0 0 0	0 0
-1	1 1 1 1 1 1 1 1	f f
-2	1 1 1 1 1 1 1 0	f e
-3	1 1 1 1 1 1 0 1	f d
-4	1 1 1 1 1 1 0 0	f c
-5	1 1 1 1 1 0 1 1	f b
-6	1 1 1 1 1 0 1 0	f a
-7	1 1 1 1 1 0 0 1	f 9
-8	1 1 1 1 1 0 0 0	f 8
⋮	⋮	⋮
-120	1 0 0 0 1 0 0 0	8 8
-121	1 0 0 0 0 1 1 1	8 7
-122	1 0 0 0 0 1 1 0	8 6
-123	1 0 0 0 0 1 0 1	8 5
-124	1 0 0 0 0 1 0 0	8 4
-125	1 0 0 0 0 0 1 1	8 3
-126	1 0 0 0 0 0 1 0	8 2
-127	1 0 0 0 0 0 0 1	8 1
-128	1 0 0 0 0 0 0 0	8 0

The table of values seen above presents the twos complement representation of a selection of signed integers that can be stored via the `int8` data class.

Negating an integer in twos complement

One of the really interesting features of the three-step conversion process outlined above is that steps 1 and 2 are the exact process we need to negate an integer.

EXAMPLE 4.8

Suppose, for example, we want to find the twos complement representation of $-(-53)$. Then, all we need to do is to negate

binary representation of -53:	1	1	0	0	1	0	1	1
bitwise complement:	0	0	1	1	0	1	0	0
	+							1
binary representation of +53:	0	0	1	1	0	1	0	1

EXAMPLE 4.9

Suppose we know that the 8−bit twos complement representation of the number $x = -18$ is given as

$$-18 = 11101110 = 1110\ \ 1110 = (\,e\,e\,)_{16}$$

To find the value of $y = -x = 18$, we can complete steps 2 and 3 from our process outlined above. Let's begin with step 2:

binary representation of -18:	1	1	1	0	1	1	1	0
bitwise complement:	0	0	0	1	0	0	0	1

Now, to complete step 3, we add one to our result to find

$$\begin{array}{ccccccccc} & 0 & 0 & 0 & 1 & 0 & 0 & \overset{1}{0} & 1 \\ + & & & & & & & & 1 \\ \hline & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}$$

By negating our twos complement representation of $x = -18$ we see that $y = +18 = -x$ has a twos complement representation given by

$$00010011 = 0001\ \ 0010 = (\,1\,2\,)_{16}.$$

Mathematically, we can show this negation operation works by using summation notation. Let's suppose that our $8-$bit sequence of binary digits

$$y = b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

encodes a signed integer in twos complement representation with $-127 \leq y \leq 127$. Thus, it's decimal representation is given as

$$y = -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i$$

Suppose also that the bitwise complement of binary digit b_i is denoted as $\overline{b_i}$. In other words, we say

$$\overline{b_i} = \begin{cases} 1 & \text{if } b_i = 0, \\ 0 & \text{if } b_i = 1. \end{cases}$$

With this definition in mind, we note that $b_i + \overline{b_i} = 1$ for all values of i. To form the bitwise complement of our number y, we consider the sequence of bits

$$\overline{b_7}\, \overline{b_6}\, \overline{b_5}\, \overline{b_4}\, \overline{b_3}\, \overline{b_2}\, \overline{b_1}\, \overline{b_0}.$$

As outlined in our two-step process, we treat this string of binary digits as an unsigned integer and add 1 to the value and interpret the sum as a twos complement representation of a signed integer, with

$$\begin{aligned} x &= \overline{b_7}\, \overline{b_6}\, \overline{b_5}\, \overline{b_4}\, \overline{b_3}\, \overline{b_2}\, \overline{b_1}\, \overline{b_0} + 1 \\ &= -\overline{b_7} \cdot 2^7 + \sum_{i=0}^{6} \overline{b_i} \cdot 2^i + 1 \end{aligned}$$

Now, we want to check that $x = -y$. To do so, we can equivalently confirm that $y + x = 0$. Consider

$$\begin{aligned} y + x &= \left(-b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i \right) + \left(-\overline{b_7} \cdot 2^7 + \sum_{i=0}^{6} \overline{b_i} \cdot 2^i + 1 \right) \\ &= -\left(b_7 + \overline{b_7} \right) \cdot 2^7 + 1 + \sum_{i=0}^{6} \left(b_i + \overline{b_i} \right) \cdot 2^i \\ &= -2^7 + 1 + \sum_{i=0}^{6} 2^i \\ &= -2^7 + 1 + 2^7 - 1 \\ &= 0. \end{aligned}$$

This is exactly what was to be shown.

The derivation on the previous page depends on the key assumption that we can do treat the bitwise complement of y as an unsigned integer, then add 1 to this value, and finally treat the result as an 8−bit signed integer in twos complement representation. However, this assumption breaks down in two special cases. First, let's consider what happens when $y = 0$. When we take the bitwise complement, we find the following:

binary representation of 0:	0	0	0	0	0	0	0	0
bitwise complement:	1	1	1	1	1	1	1	1

Then, when we add 1 to the result, assuming the form of an unsigned integer addition, we get a carry out of the most significant bit position. This yields a 9−bit output, as seen below:

$$\begin{array}{ccccccccccc}
 & & & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} \\
+ & & & & & & & & & & 1 \\
\hline
 & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Since our output is stored in 8−bit format, we ignore the carry over bit (highlighted in gray) and see that the negation of zero produces 0, as we expect from our intuition in mathematics.

The other special case is more interesting. Suppose we want to negate $y = -128$. Let's do that below:

binary representation of -128:	1	0	0	0	0	0	0	0
bitwise complement:	0	1	1	1	1	1	1	1
	+							1
result :	1	0	0	0	0	0	0	0

Thus, our algorithm indicates that when we negate the bit pattern for $y = -128$, we get the same bit pattern back again. This is clearly not correct mathematically. However, this freak occurrence is unavoidable. Specifically, in 8−bits of information, we have $2^8 = 256$ possible bit patterns, which is an even number. When encoding signed integers, we want to represent positive and negative integers as well as the number 0. If we try to guarantee the same number of positive and negative representations, then there must be two representations for the number zero (as in the signed magnitude representation). On the other hand, if we want to guarantee there is a unique representation of the number 0 (as in twos complement), then we must have an unequal number of representations for positive and negative integers. For an 8−bit twos complement representation, we are able to represent numbers greater than or equal to -2^7 and less than or equal to $+2^7 - 1$. As we will see, these type of special considerations are important when storing numbers in a digital computer.

Successful Integer addition via twos complement

One of the appealing features of twos complement is that addition (and subtraction) of a pair of signed integers proceed in the same manner as if the numbers were unsigned integers. Let's take a look at some successful examples of how addition of signed integers represented in twos complement plays out.

EXAMPLE 4.10

Suppose we want to find the sum $-45 + 22$ using an $8-$bit, twos complement representation of these signed integers. The first step is to translate each number into twos complement representation using our algorithm mentioned above. After doing so, we see that

$$\begin{aligned} -45 &= 1\,1\,0\,1 \quad 0\,0\,1\,1 = (\ d\,3\ )_{16}, \\ +22 &= 0\,0\,0\,1 \quad 0\,1\,1\,0 = (\ 1\,6\ )_{16}. \end{aligned}$$

Then, we find the sum of the two numbers in the same exact way we did for signed integers

$$\begin{array}{r|cccccccccc} \text{binary representation of -45:} & & 1 & 1 & \overset{1}{0} & 1 & \overset{1}{0} & \overset{1}{0} & 1 & 1 & = & -45 \\ \text{binary representation of +22:} & + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & = & +22 \\ \hline \text{sum of -23:} & & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & = & -23 \end{array}$$

Let's confirm that our result is indeed what we need. Using our twos complement representation we see:

$$\begin{aligned} 11101001 &= -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i \\ &= -b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ &= -1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= -2^7 + 2^6 + 2^5 + 2^3 + 2^0 \\ &= -128 + 64 + 32 + 8 + 1 \\ &= -128 + 105 = -23 = (\ \mathrm{e}\,9\ )_{16}. \end{aligned}$$

This is exactly what we expect.

EXAMPLE 4.11

For this next successful example, let's consider an addition problem of two positive numbers. In particular, suppose we want to find the sum $75 + 113$ using an $8-$bit, twos complement representation of these signed integers. Once again, we begin by transforming each number into twos complement representation to see that

$$
\begin{aligned}
75 &= 0100 \;\; 1011 = (\,4\,\mathrm{b}\,)_{16},\\
+13 &= 0000 \;\; 1101 = (\,0\,\mathrm{d}\,)_{16}.
\end{aligned}
$$

Then, we find the sum of the two numbers in the same exact way we did for signed integers

					1	1	1	1			
binary representation of +75:		0	1	0	0	1	0	1	1	=	+75
binary representation of +13:	+	0	0	0	0	1	1	0	1	=	+13
sum of + 88:		0	1	0	1	1	0	0	0	=	+88

Let's confirm that our result is indeed what we need. Using our twos complement representation we see:

$$
\begin{aligned}
01011000 &= -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i\\
&= -b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0\\
&= -0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0\\
&= 2^6 + 2^4 + 2^3\\
&= 64 + 16 + 8\\
&= 88 = (\,5\,8\,)_{16}
\end{aligned}
$$

This is exactly what we expect.

EXAMPLE 4.12

Let's consider adding two numbers in twos complement to produce a sum of 0. For instance, lets find $57 + -57$ using an $8-$bit twos complement representation of these signed integers. We begin by noting that

$$
\begin{aligned}
+57 &= 0011 \;\; 1001 = (\,3\,9\,)_{16},\\
-57 &= 1100 \;\; 0111 = (\,\mathrm{c}\,7\,)_{16}.
\end{aligned}
$$

We then sum these two numbers as if they were unsigned integers:

			1	1	1	1	1	1	1			
binary representation of +57			0	0	1	1	1	0	0	1	=	+57
binary representation of -57:	+		1	1	0	0	0	1	1	1	=	−57
sum of 0:		1	0	0	0	0	0	0	0	0	=	0

In this case, there is a carry over bit beyond the end of the $8-$bit word used to store the output of this sum. This carry over bit (highlighted in gray above) is ignored when adding signed integers stored in twos complement. Our result is indeed correct since $00000000 = 0$

EXAMPLE 4.13

For this final successful example of twos complement addition, let's consider the addition of two negative numbers. Specifically, let's find the sum $-31 + -81$ using an $8-$bit, twos complement representation of these signed integers. Just as before, we start our work by writing each number using a twos complement representation with

$$\begin{aligned} -31 &= 1110 \;\; 0001 = (\, e1 \,)_{16}, \\ -81 &= 1010 \;\; 1111 = (\, af \,)_{16}. \end{aligned}$$

Then, we find the sum of the two numbers in the same exact way we did for unsigned integers

			1	1		1	1	1	1			
binary representation of -31			1	1	1	0	0	0	0	1	=	-31
binary representation of -81:	+		1	0	1	0	1	1	1	1	=	-81
sum of -112:		1	1	0	0	1	0	0	0	0	=	-112

Notice that there is a carry over bit beyond the end of the $8-$bit word used to store the output of this sum. This carry over bit (highlighted in gray above) is ignored when adding signed integers stored in twos complement. Our result is indeed correct since:

$$\begin{aligned} 10010000 &= -b_7 \cdot 2^7 + \sum_{i=0}^{6} b_i \cdot 2^i \\ &= -b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \\ &= -1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\ &= -2^7 + 2^4 \\ &= -128 + 16 \\ &= -112 = (\, 90 \,)_{16} \end{aligned}$$

In all of the four successful examples of twos complement addition given above, we verified that twos complement addition may proceed as if the two numbers were unsigned integers. In particular, if the result of our addition operation is positive (as in examples 4.11), we produce the proper positive sum in twos complement form. On the other hand, if the result of our addition operation is negative (as in examples 4.10 and 4.13), we get the proper negative sum in twos complement form. In two instances (examples 4.12 and 4.13), the twos complement addition produced a carry over bit (highlighted in gray). In general, we ignore the carry over bit that results from twos complement addition.

Overflow rule for twos complement addition

Not all addition in twos complement will produce a successful sum. In fact, some addition problems may result in a sum that is larger than what we can store in the word size being used to store the output. Let's take a look at how this plays out in practice by considering two examples.

EXAMPLE 4.14

For our first example of when twos complement addition is unsuccessful due to overflow, let's consider the addition of two positive numbers. Specifically, let's find the sum $113 + 91$ using an 8−bit, twos complement representation of these signed integers. Just as before, we start our work by writing each number using a twos complement representation with

$$\begin{aligned} 113 &= 0111\ \ 0001 = (\,71\,)_{16}, \\ 91 &= 0101\ \ 1011 = (\,5b\,)_{16}. \end{aligned}$$

Then, we find the sum of the two numbers using unsigned addition:

$$\begin{array}{r|cccccccccccc} \text{binary representation of } +113 & & \overset{1}{0} & \overset{1}{1} & \overset{1}{1} & 1 & 0 & \overset{1}{0} & \overset{1}{0} & 1 & = & 113 \\ \text{binary representation of } +91\text{:} & + & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & = & +91 \\ \hline \text{sum of } +204\text{:} & & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & = & \text{Overflow} \end{array}$$

Even though both summands are positive, the twos complement addition produces a result with opposite sign. This is a key feature of detecting *overflow* in two complement addition.

EXAMPLE 4.15

For our next example illustrating when twos complement addition is unsuccessful due to overflow, let's addition of two negative numbers. Specifically, let's find the sum $-113 + -91$ using an 8−bit, twos complement representation of these signed integers. We commence by finding the twos complement representation of each summand:

$$\begin{aligned} -91 &= 1010\ \ 0101 = (\,a5\,)_{16}, \\ -113 &= 1000\ \ 1111 = (\,8f\,)_{16}. \end{aligned}$$

We find the sum of the two numbers via unsigned addition:

$$\begin{array}{r|cccccccccccc} \text{binary representation of -91} & & & 1 & 0 & 1 & \overset{1}{0} & \overset{1}{0} & \overset{1}{1} & \overset{1}{0} & 1 & = & -91 \\ \text{binary representation of -113:} & + & & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & = & -113 \\ \hline \text{sum of -204:} & & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & = & \text{Overflow} \end{array}$$

Both summands are negative and yet twos complement addition produces a positive result, which is nonsense.

Examples 4.14 and 4.15 above lead to a general rule for detecting overflow in twos complement addition.

OVERFLOW RULE FOR TWOS COMPLEMENT ADDITION

If two numbers with the same sign are added in twos complement, then overflow occurs if and only if the resulting sum has the opposite sign of our two summands.

Subtraction rule for twos complement addition

SUBTRACTION RULE FOR TWOS COMPLEMENT

To subtract one number B from another number A, negate B (find the twos complement of B) and then add this to the number A.

Range extension for twos complement representation of signed integers

RANGE EXPANSION RULE FOR TWOS COMPLEMENT

To take an $8-$bit signed integer stored using the `int8` data class and store it as an $m-$bit signed integer (where $m = 16, 32$, or 64 corresponding to the `int16`, `int32`, and `int64` data classes, respectively), we move the sign bit to the new leftmost significant position and fill in all the rest of the bits with copies of the sign bit. For positive numbers, we fill in all zeros. For negative numbers, we fill in all ones. This is called *sign extension.*

Let's recapitulate our major finding for the `int8` signed integers class below.

Table 4.1: Characteristics of 8−Bit twos complement representation and arithmetic as related to MATLAB's `int8` data class

PROPERTY	DESCRIPTION
Range	$-2^7 \leq y \leq 2^7 - 1$
Treatment of zero	Provides a unique representation of zero.
Negation	Take the bitwise (Boolean) complement of each bit of the corresponding positive number. Treat the resulting bit pattern as an unsigned integer and add 1 to this result.
Overflow rule	If two numbers with identical sign are added in twos complement, then overflow occurs if and only if the result has the opposite sign.
Subtraction rule	To subtract B from A, take the twos complement of B and add it to A.
Range Expansion	To expand an `int8` signed integer to a sign integer data class with larger range, move the sign bit to the new leftmost position and fill in all extra bits with copies of the sign bit.