Fixed-Point Designer™

Getting Started Guide

R2014**b**

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Fixed-Point Designer™ Getting Started Guide

Trademarks

Patents

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (Release 2013b)
March 2014	Online only	Revised for Version 4.2 (Release 2014a)
October 2014	Online only	Revised for Version 4.3 (Release 2014b)

Contents

1 Getting Started with Fixed Point

2 About Fixed-Point

Getting Started with Fixed Point

Create Fixed-Point Data in MATLAB

The following examples show how to create fixed-point data using the Fixed-Point Designer™ `fi` object.

Create a fixed-point number with default properties

Calling `fi` on a number produces a fixed-point number with default signedness and default word and fraction lengths.

```
 fi(pi)
```

```
ans =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Create a fixed-point number with specified signedness, word length, and fraction length

You can specify the signedness (1 for signed, 0 for unsigned) and the word and fraction lengths.

```
fi(pi,1,15,12)
```

```
ans =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 15
        FractionLength: 12
```

The `fi` and `numerictype` Objects

You can use the `fi` constructor to assign a fixed-point data type to a number or variable. Within the `fi` constructor, you can specify `numerictype` and `fimath` properties. There are two ways to create a `fi` object:

- At the MATLAB® command line using `fi`.
- Using a user interface. For more information on this second approach, see “Building fi Object Constructors in a GUI” in “Types of fi Constructors”.

Each `fi` object has an associated `numerictype` object. The `numerictype` object stores information about the `fi` object including word length, fraction length, and signedness. `numerictype` properties can be specified in the `fi` constructor, or assigned to a `fi` object later.

The `numerictype` object in MATLAB is equivalent to the`fixdt` object in Simulink®.

For more information on the properties of `numerictype` objects see “numerictype Object Properties”.

Create fixed-point integer values

To create fixed-point integer values, you specify a fraction length of 0.

```
fi(1:25,0,8,0)

ans =

  Columns 1 through 13
     1   2   3   4   5   6   7   8   9  10  11  12  13
  Columns 14 through 25
    14  15  16  17  18  19  20  21  22  23  24  25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

Create an array of random fixed-point values

```
fi(rand(4),0,12,8)

ans =

    0.1484    0.8125    0.1953    0.3516
    0.2578    0.2422    0.2500    0.8320
    0.8398    0.9297    0.6172    0.5859
    0.2539    0.3516    0.4727    0.5508
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 12
        FractionLength: 8
```

Create an array of zeros

When writing code, you sometimes want to test different data types for your variables. Separating the data types of your variables from your algorithm makes testing much simpler. By creating a table of data type definitions, you can programmatically toggle your function between floating point and fixed point data types. The following example shows how to use this technique and create an array of zeros.

```
 T.z = fi([],1,16,0);

z = zeros(2,3,'like',T.z)

z =

     0     0     0
     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

Note: For a full example showing this technique's implementation, see "Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types using cast and zeros".

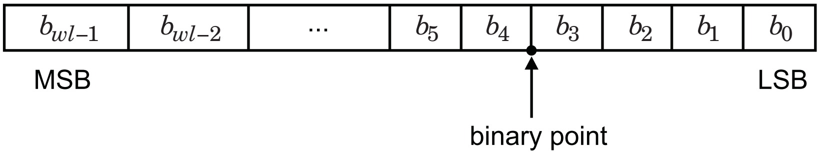
Fixed-Point Data Types

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. This chapter discusses many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the i^{th} binary digit.
- wl is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways:

- “Sign/magnitude”
- “One's complement”
- “Two's complement”

Two's complement is the most common representation of signed fixed-point numbers and is the only representation used by Fixed-Point Designer documentation.

Perform Fixed-Point Arithmetic

In this section...

Fixed-Point Arithmetic

Addition and subtraction

Whenever you add two fixed-point numbers, you may need a carry bit to correctly represent the result. For this reason, when adding two B-bit numbers (with the same scaling), the resulting value has an extra bit compared to the two operands used.

```
a = fi(0.234375,0,4,6);
c = a+a

c =

    0.4688

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 5
        FractionLength: 6

a.bin

ans =

1111

c.bin

ans =
11110
```

If you add or subtract two numbers with different precision, the radix point first needs to be aligned to perform the operation. The result is that there is a difference of more than one bit between the result of the operation and the operands.

```
a = fi(pi,1,16,13);
b = fi(0.1,1,12,14);
c = a + b

c =

    3.2416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 14
```

Multiplication

In general, a full precision product requires a word length equal to the sum of the word length of the operands. In the following example, note that the word length of the product `c` is equal to the word length of `a` plus the word length of `b`. The fraction length of `c` is also equal to the fraction length of `a` plus the fraction length of `b`.

```
a = fi(pi,1,20), b = fi(exp(1),1,16)

a =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 20
        FractionLength: 17

b =

    2.7183

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
    FractionLength: 13

c = a*b

c =

  8.5397

     DataTypeMode: Fixed-point: binary point scaling
       Signedness: Signed
       WordLength: 36
    FractionLength: 30
```

Math with other built in data types

Note that in C, the result of an operation between an integer data type and a double data type promotes to a double. However, in MATLAB, the result of an operation between a built-in integer data type and a double data type is an integer. In this respect, the `fi` object behaves like the built-in integer data types in MATLAB.

When doing arithmetic between `fi` and `double`, the double is cast to a `fi` with the same word length and signedness of the `fi`, and best precision fraction length. The result of the operation is a `fi`.

```
a = fi(pi);

a =

  3.1416

     DataTypeMode: Fixed-point: binary point scaling
       Signedness: Signed
       WordLength: 16
    FractionLength: 13

b = 0.5 * a

b =

  1.5708

     DataTypeMode: Fixed-point: binary point scaling
       Signedness: Signed
       WordLength: 32
```

```
        FractionLength: 28
```

When doing arithmetic between a `fi` and one of the built-in integer data types, `[u]int[8, 16, 32]`, the word length and signedness of the integer are preserved. The result of the operation is a `fi`.

```
a = fi(pi);
b = int8(2) * a
```

```
b =

    6.2832

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 13
```

The fimath Object

`fimath` properties define the rules for performing arithmetic operations on `fi` objects, including math, rounding, and overflow properties. A `fi` object can have a local `fimath` object, or it can use the default `fimath` properties. You can attach a `fimath` object to a `fi` object by using `setfimath`. Alternatively, you can specify `fimath` properties in the `fi` constructor at creation. When a `fi` object has a local `fimath` , rather than using the default properties, the display of the `fi` object shows the `fimath` properties. In this example, `a` has the `ProductMode` property specified in the constructor.

```
 a = fi(5,1,16,4,'ProductMode','KeepMSB')
```

```
a =

     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 4

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: KeepMSB
     ProductWordLength: 32
               SumMode: FullPrecision
```

The ProductMode property of a is set to KeepMSB while the remaining fimath properties use the default values.

Note: For more information on the fimath object, its properties, and their default values, see “fimath Object Properties”.

Bit Growth

The following table shows the bit growth of fi objects, A and B, when their SumMode and ProductMode properties use the default fimath value, FullPrecision.

	A	B	Sum = A+B	Prod = A*B
Format	fi(v_A,s_1,w_1,f	fi(v_B,s_2,w_2,f_2	—	—
Sign	s_1	s_2	$S_{sum} = (s_1 \vert\vert s_2)$	$S_{product} = (s_1 \vert\vert s_2)$
Integer bits	$I_1 = w_1 - f_1 - s_1$	$I_2 = w_2 - f_2 - s_2$	$I_{sum} = max(w_1 - f_1, w_2 - f_2) + 1 - S_{sum}$	$I_{product} = (w_1 + w_2) - (f_1 + f_2)$
Fraction bits	f_1	f_2	$F_{sum} = max(f_1, f_2)$	$F_{product} = f_1 + f_2$
Total bits	w_1	w_2	$S_{sum} + I_{sum} + F_{sum}$	$w_1 + w_2$

This example shows how bit growth can occur in a for-loop.

```
T.acc = fi([],1,32,0);
T.x = fi([],1,16,0);

x = cast(1:3,'like',T.x);
acc = zeros(1,1,'like',T.acc);

for n = 1:length(x)
    acc = acc + x(n)
end

acc =

     1
      s33,0
```

```
acc =

     3
      s34,0

acc =

     6
      s35,0
```

The word length of `acc` increases with each iteration of the loop. This increase causes two problems: One is that code generation does not allow changing data types in a loop. The other is that, if the loop is long enough, you run out of memory in MATLAB. See “Controlling Bit Growth” on page 1-12 for some strategies to avoid this problem.

Controlling Bit Growth

Using fimath

By specifying the `fimath` properties of a `fi` object, you can control the bit growth as operations are performed on the object.

```
F = fimath('SumMode', 'SpecifyPrecision', 'SumWordLength', 8,...
 'SumFractionLength', 0);
a = fi(8,1,8,0, F);
b = fi(3, 1, 8, 0);
c = a+b

c =

    11

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: SpecifyPrecision
         SumWordLength: 8
     SumFractionLength: 0
         CastBeforeSum: true
```

The fi object a has a local fimath object F. F specifies the word length and fraction length of the sum. Under the default fimath settings, the output, c, normally has word length 9, and fraction length 0. However because a had a local fimath object, the resulting fi object has word length 8 and fraction length 0.

You can also use fimath properties to control bit growth in a for-loop.

```
F = fimath('SumMode', 'SpecifyPrecision','SumWordLength',32,...
'SumFractionLength',0);
T.acc = fi([],1,32,0,F);
T.x = fi([],1,16,0);

x = cast(1:3,'like',T.x);
acc = zeros(1,1,'like',T.acc);

for n = 1:length(x)
    acc = acc + x(n)
end
```

```
acc =

     1
      s32,0

acc =

     3
      s32,0

acc =

     6
      s32,0
```

Unlike when T.acc was using the default fimath properties, the bit growth of acc is now restricted. Thus, the word length of acc stays at 32.

Subscripted Assignment

Another way to control bit growth is by using subscripted assignment. a(I) = b assigns the values of b into the elements of a specified by the subscript vector, I, while retaining the numerictype of a.

```
T.acc = fi([],1,32,0);
```

```
T.x = fi([],1,16,0);

x = cast(1:3,'like',T.x);
acc = zeros(1,1,'like',T.acc);

% Assign in to acc without changing its type
for n = 1:length(x)
    acc(:) = acc + x(n)
end
```

acc (:) = acc + x(n) dictates that the values at subscript vector, `(:)`, change. However, the `numerictype` of output `acc` remains the same. Because `acc` is a scalar, you also receive the same output if you use `(1)` as the subscript vector.

```
  for n = 1:numel(x)
    acc(1) = acc + x(n);
  end
```

```
acc =

     1
      s32,0

acc =

     3
      s32,0

acc =

     6
      s32,0
```

The `numerictype` of `acc` remains the same at each iteration of the `for`-loop.

Subscripted assignment can also help you control bit growth in a function. In the function, `cumulative_sum`, the `numerictype` of `y` does not change, but the values in the elements specified by *n* do.

```
function y = cumulative_sum(x)
% CUMULATIVE_SUM Cumulative sum of elements of a vector.
%
%   For vectors, Y = cumulative_sum(X) is a vector containing the
%   cumulative sum of the elements of X.  The type of Y is the type of X.
    y = zeros(size(x),'like',x);
    y(1) = x(1);
    for n = 2:length(x)
        y(n) = y(n-1) + x(n);
    end
end

y = cumulative_sum(fi([1:10],1,8,0))

y =

     1     3     6    10    15    21    28    36    45    55

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

Note: For more information on subscripted assignment, see the `subsasgn` function.

accumpos and accumneg

Another way you can control bit growth is by using the `accumpos` and `accumneg` functions to perform addition and subtraction operations. Similar to using subscripted assignment, `accumpos` and `accumneg` preserve the data type of one of its input `fi` objects while allowing you to specify a rounding method, and overflow action in the input values.

For more information on how to implement `accumpos` and `accumneg`, see "Avoid Multiword Operations in Generated Code"

Overflows and Rounding

When performing fixed-point arithmetic, consider the possibility and consequences of overflow. The `fimath` object specifies the overflow and rounding modes used when performing arithmetic operations.

Overflows

Overflows can occur when the result of an operation exceeds the maximum or minimum representable value. The `fimath` object has an `OverflowAction` property which offers two ways of dealing with overflows: saturation and wrap. If you set `OverflowAction` to saturate, overflows are saturated to the maximum or minimum value in the range. If you set OverflowAction to wrap, any overflows wrap using modulo arithmetic, if unsigned, or two's complement wrap, if signed.

For more information on how to detect overflow see "Underflow and Overflow Logging Using fipref".

Rounding

There are several factors to consider when choosing a rounding method, including cost, bias, and whether or not there is a possibility of overflow. Fixed-Point Designer software offers several different rounding functions to meet the requirements of your design.

Rounding Method	Description	Cost	Bias	Possibility of Overflow
`ceil`	Rounds to the closest representable number in the direction of positive infinity.	Low	Large positive	Yes
`convergent`	Rounds to the closest representable number. In the case of a tie, `convergent` rounds to the nearest even number. This approach is the least-biased rounding method provided by the toolbox.	High	Unbiased	Yes
`floor`	Rounds to the closest representable number in the direction of negative infinity, equivalent to two's complement truncation.	Low	Large negative	No
`nearest`	Rounds to the closest representable number. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity. This rounding method is the default for `fi` object creation and `fi` arithmetic.	Moderate	Small positive	Yes
`round`	Rounds to the closest representable number. In the case of a tie, the `round` method rounds:	High	• Small negative for negative samples	Yes

Rounding Method	Description	Cost	Bias	Possibility of Overflow
	• Positive numbers to the closest representable number in the direction of positive infinity. • Negative numbers to the closest representable number in the direction of negative infinity.		• Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples	
`fix`	Rounds to the closest representable number in the direction of zero.	Low	• Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples	No

Perform Fixed-Point Arithmetic

This example shows how to perform basic fixed-point arithmetic operations.

Addition and Subtraction

Whenever you add two unsigned fixed-point numbers, you may need a carry bit to correctly represent the result. For this reason, when adding two B-bit numbers (with the same scaling), the resulting value has an extra bit compared to the two operands used.

```
a = ufi(0.234375,4,6);
c = a + a
```

```
c = 

    0.4688

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 5
        FractionLength: 6
```

```
a.bin
```

```
ans =

1111
```

```
c.bin
```

```
ans =

11110
```

With signed, two's-complement numbers, a similar scenario occurs because of the sign extension required to correctly represent the result.

```
a = sfi(0.078125,4,6);
b = sfi(-0.125,4,6);
```

```
c = a + b


c =

   -0.0469

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 5
        FractionLength: 6

a.bin


ans =

0101


b.bin


ans =

1000


c.bin


ans =

11101
```

If you add or subtract two numbers with different precision, the radix point first needs to be aligned to perform the operation. The result is that there is a difference of more than one bit between the result of the operation and the operands (depending on how far apart the radix points are).

```
a = sfi(pi,16,13);
b = sfi(0.1,12,14);
c = a + b
```

```
c =

    3.2416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 14
```

Further Considerations for Addition and Subtraction

Note that the following pattern is **not** recommended. Since scalar additions are performed at each iteration in the for-loop, a bit is added to temp during each iteration. As a result, instead of a ceil(log2(Nadds)) bit-growth, the bit-growth is equal to Nadds.

```
s = rng; rng('default');
b = sfi(4*rand(16,1)-2,32,30);
rng(s); % restore RNG state
Nadds = length(b) - 1;
temp  = b(1);
for n = 1:Nadds
    temp = temp + b(n+1); % temp has 15 more bits than b
end
```

If the `sum` command is used instead, the bit-growth is curbed as expected.

```
c = sum(b) % c has 4 more bits than b
```

```
c =

    7.0059

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 36
        FractionLength: 30
```

Multiplication

In general, a full precision product requires a word length equal to the sum of the word lengths of the operands. In the following example, note that the word length of the product `c` is equal to the word length of `a` plus the word length of `b`. The fraction length of `c` is also equal to the fraction length of `a` plus the fraction length of `b`.

```
a = sfi(pi,20);
b = sfi(exp(1),16);
c = a * b


c =

    8.5397

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 36
        FractionLength: 30
```

Assignment

When you assign a fixed-point value into a pre-defined variable, quantization might be involved. In such cases, the right-hand-side of the expression is quantized by rounding to nearest and then saturating, if necessary, before assigning to the left-hand-side.

```
N = 10;
a = sfi(2*rand(N,1)-1,16,15);
b = sfi(2*rand(N,1)-1,16,15);
c = sfi(zeros(N,1),16,14);
for n = 1:N
    c(n) = a(n).*b(n);
end
```

Note that when the product `a(n).*b(n)` is computed with full precision, an intermediate result with wordlength 32 and fraction length 30 is generated. That result is then quantized to a wordlength of 16 and a fraction length of 14, as explained above. The quantized value is then assigned to the element `c(n)`.

Quantizing Results Explicitly

Often, it is not desirable to round to nearest or to saturate when quantizing a result because of the extra logic/computation required. It also may be undesirable to have to assign to a left-hand-side value to perform the quantization. You can use `QUANTIZE` for such purposes. A common case is a feedback-loop. If no quantization is introduced, unbounded bit-growth will occur as more input data is provided.

```
a = sfi(0.1,16,18);
x = sfi(2*rand(128,1)-1,16,15);
y = sfi(zeros(size(x)),16,14);
```

```
for n = 1:length(x)
    z    = y(n);
    y(n) = x(n) - quantize(a.*z, true, 16, 14, 'Floor', 'Wrap');
end
```

In this example, the product `a.*z` is computed with full precision and is subsequently quantized to a wordlength of 16 bits and a fraction length of 14. The quantization is done by rounding to floor (truncation) and allowing for wrapping if overflow occurs. Quantization still occurs at assignment, because the expression `x(n) - quantize(a.*z, ...)` produces an intermediate result of 18 bits and y is defined to have 16 bits. To eliminate the quantization at assignment, you can introduce an additional explicit quantization as shown below. The advantage of doing this is that no round-to-nearest/saturation logic is used. The left-hand-side result has the same 16-bit wordlength and fraction length of 14 as `y(n)`, so no quantization is necessary.

```
a = sfi(0.1,16,18);
x = sfi(2*rand(128,1)-1,16,15);
y = sfi(zeros(size(x)),16,14);
T = numerictype(true, 16, 14);
for n = 1:length(x)
    z    = y(n);
    y(n) = quantize(x(n), T, 'Floor', 'Wrap') - ...
           quantize(a.*z, T, 'Floor', 'Wrap');
end
```

Non-Full-Precision Sums

Full-precision sums are not always desirable. For example, the 18-bit wordlength corresponding to the intermediate result `x(n) - quantize(...)` above may result in complicated and inefficient code, if C code is generated. Instead, it may be desirable to keep all results of addition/subtraction to 16 bits. You can use the `accumpos` and `accumneg` functions for this purpose.

```
a = sfi(0.1,16,18);
x = sfi(2*rand(128,1)-1,16,15);
y = sfi(zeros(size(x)),16,14);
T = numerictype(true, 16, 14);
for n = 1:length(x)
    z    = y(n);
    y(n) = quantize(x(n), T);                 % defaults: 'Floor','Wrap'
    y(n) = accumneg(y(n), quantize(a.*z, T)); % defaults: 'Floor','Wrap'
end
```

Modeling Accumulators

`accumpos` and `accumneg` are well-suited to model accumulators. The behavior corresponds to the += and -= operators in C. A common example is an FIR filter in which the coefficients and input data are represented with 16 bits. The multiplication is performed in full-precision, yielding 32 bits, and an accumulator with 8 guard-bits, i.e. 40-bits total is used to enable up to 256 accumulations without the possibility of overflow.

```
b = sfi(1/256*[1:128,128:-1:1],16); % Filter coefficients
x = sfi(2*rand(300,1)-1,16,15);     % Input data
z = sfi(zeros(256,1),16,15);        % Used to store the states
y = sfi(zeros(size(x)),40,31);      % Initialize Output data
for n = 1:length(x)
    acc = sfi(0,40,31); % Reset accumulator
    z(1) = x(n);        % Load input sample
    for k = 1:length(b)
        acc = accumpos(acc,b(k).*z(k)); % Multiply and accumulate
    end
    z(2:end) = z(1:end-1); % Update states
    y(n) = acc;            % Assign output
end
```

Matrix Arithmetic

To simplify syntax and shorten simulation time, you can use matrix arithmetic. For the FIR filter example, you can replace the inner loop with an inner product.

```
z = sfi(zeros(256,1),16,15); % Used to store the states
y = sfi(zeros(size(x)),40,31);
for n = 1:length(x)
    z(1) = x(n);
    y(n) = b*z;
    z(2:end) = z(1:end-1);
end
```

The inner product `b*z` is performed with full precision. Because this is a matrix operation, the bit growth is due to both the multiplication involved and the addition of the resulting products. Therefore, the bit growth depends on the length of the operands. Since `b` and `z` have length 256, that accounts for an 8-bit growth due to the additions. This is why the inner product results in 32 + 8 = 40 bits (with fraction length 31). Since this is the format `y` is initialized to, no quantization occurs in the assignment `y(n) = b*z`.

If you had to perform an inner product for more than 256 coefficients, the bit growth would be more than 8 bits beyond the 32 needed for the product. If you only had a 40-bit accumulator, you could model the behavior by either introducing a quantizer, as in `y(n) = quantize(Q,b*z)`, or you could use the `accumpos` function as has been shown.

Modeling a Counter

`accumpos` can be used to model a simple counter which naturally wraps after reaching its maximum value. For example, you can model a 3-bit counter as follows.

```
c = ufi(0,3,0);
Ncounts = 20; % Number of times to count
for n = 1:Ncounts
    c = accumpos(c,1);
end
```

Since the 3-bit counter naturally wraps back to 0 after reaching 7, the final value of the counter is mod(20,8) = 4.

Math With Other Built-In Data Types

FI * DOUBLE

When doing arithmetic between `fi` and `double`, the double is cast to a `fi` with the same word length and signedness of the `fi`, and best-precision fraction length. The result of the operation is a `fi`.

```
a = fi(pi);
b = 0.5 * a
```

```
b =

    1.5708

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28
```

Some Differences Between MATLAB® and C

Note that in C, the result of an operation between an integer data type and a double data type promotes to a double.

However, in MATLAB, the result of an operation between a built-in integer data type and a double data type is an integer. In this respect, the `fi` object behaves like the built-in integer data types in MATLAB. The result of an operation between a `fi` and a double is a `fi`.

FI * INT8

When doing arithmetic between fi and one of the built-in integer data types [u]int[8,16,32], the word length and signedness of the integer are preserved. The result of the operation is a fi.

```
a = fi(pi);
b = int8(2) * a
```

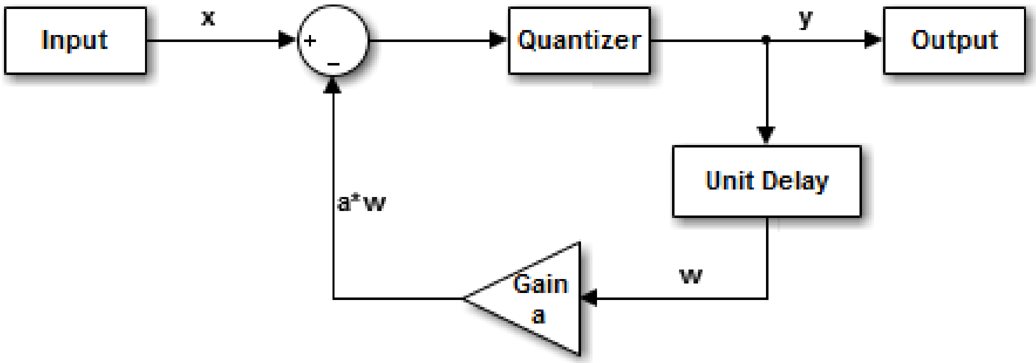
```
b =

    6.2832

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 13
```

Accelerate Fixed-Point Simulation

This example shows how to accelerate fixed-point algorithms using `fiaccel` function. You generate a MEX function from MATLAB® code, run the generated MEX function, and compare the execution speed with MATLAB code simulation.

Description of the Example

This example uses a first-order feedback loop. It also uses a quantizer to avoid infinite bit growth. The output signal is delayed by one sample and fed back to dampen the input signal.



Copy Required File

You need this MATLAB-file to run this example. Copy it to a temporary directory. This step requires write-permission to the system's temporary directory.

```
tempdirObj = fidemo.fiTempdir('fiaccelbasicsdemo');
fiacceldir = tempdirObj.tempDir;
fiaccelsrc = ...
    fullfile(matlabroot,'toolbox','fixedpoint','fidemos','+fidemo','fiaccelFeedback.m'
copyfile(fiaccelsrc,fiacceldir,'f');
```

Inspect the MATLAB Feedback Function Code

The MATLAB function that performs the feedback loop is in the file `fiaccelFeedback.m`. This code quantizes the input, and performs the feedback loop action :

```
type(fullfile(fiacceldir,'fiaccelFeedback.m'))
```

```
function [y,w] = fiaccelFeedback(x,a,y,w)
%FIACCELFEEDBACK Quantizer and feedback loop used in FIACCELBASICSDEMO.

% Copyright 1984-2013 The MathWorks, Inc.
%#codegen

for n = 1:length(x)
    y(n) =  quantize(x(n) - a*w, true, 16, 12, 'floor', 'wrap');
    w    = y(n);
end
```

The following variables are used in this function:

- `x` is the input signal vector.
- `y` is the output signal vector.
- `a` is the feedback gain.
- `w` is the unit-delayed output signal.

Create the Input Signal and Initialize Variables

```
rng('default');                     % Random number generator
x = fi(2*rand(1000,1)-1,true,16,15); % Input signal
a = fi(.9,true,16,15);              % Feedback gain
y = fi(zeros(size(x)),true,16,12);  % Initialize output. Fraction length
                                    % is chosen to prevent overflow
w = fi(0,true,16,12);               % Initialize delayed output
A = coder.Constant(a);              % Declare "a" constant for code
                                    % generation
```

Run Normal Mode

```
tic,
y = fiaccelFeedback(x,a,y,w);
t1 = toc;
```

Build the MEX Version of the Feedback Code

```
fiaccel fiaccelFeedback -args {x,A,y,w} -o fiaccelFeedback_mex
```

Run the MEX Version

```
tic
y2 = fiaccelFeedback_mex(x,y,w);
t2 = toc;
```

Acceleration Ratio

Code acceleration provides optimizations for accelerating fixed-point algorithms through MEX file generation. Fixed-Point Designer™ provides a convenience function `fiaccel` to convert your MATLAB code to a MEX function, which can greatly accelerate the execution speed of your fixed-point algorithms.

```
r = t1/t2
```

```
r =

  119.0292
```

Clean up Temporary Files

```
clear fiaccelFeedback_mex;
tempdirObj.cleanUp;
```

Generate Fixed-Point C Code

Note: To generate fixed-point code from MATLAB you must have both the Fixed-Point Designer product and the MATLAB Coder™ product. You also must have a C compiler.

This example shows how to generate code for a simple function that multiples and accumulates two input values. This is the type of code that you could embed in external hardware. The function is

```
function acc = mult_acc(x,a,acc)
acc = accumpos(acc,x*a);
```

This code defines the test bench inputs, sets up the required code generation properties, and generates the code. The test bench inputs are specified as fixed-point numbers. The `x` input is a random number, `a` is 0.9, and the accumulator, `acc`, is initialized to 0. The `coder.HardwareImplementation` object specifies properties of the external hardware that impact the generated code. The examples specifies a 40-bit accumulator. The `coder.CodeConfig` object has properties that directly affect code generation. The `codegen` command takes the function, the configuration object as the input arguments and generates embeddable C code.

```
x = fi(rand,true,16,15);
a = fi(0.9,true,16,15);
acc = fi(0,true,40,30);


%%
hi = coder.HardwareImplementation;
hi. ProdHWDeviceType = 'Generic->Custom'
hi. TargetHWDeviceType = 'Generic->Custom'
hi.TargetBitPerLong = 40;
hi.ProdBitPerLong   = 40;

hc = coder.config('lib');
hc.HardwareImplementation = hi;
hc.GenerateReport         = true;

codegen mult_acc -config hc -args {x,a,acc}
```

The generated C code is

```
/*
 * mult_acc.c
 *
 * Code generation for function 'mult_acc'
*/

/* Include files */
#include "rt_nonfinite.h"
#include "mult_acc.h"

/* Function Definitions */
void mult_acc(short x, short a, long *acc)
{
  *acc += x * a;
}

/* End of code generation (mult_acc.c) */
```

Note: For a list of functions supported for code generation, see “Functions and Objects Supported for C and C++ Code Generation — Alphabetical List”.

Manually Convert a Floating-Point MATLAB Algorithm to Fixed Point

This example shows how to convert a floating-point algorithm to fixed point and then generate C code for the algorithm. The example uses the following best practices:

- Separate your algorithm from the test file.
- Prepare your algorithm for instrumentation and code generation.
- Manage data types and control bit growth.
- Separate data type definitions from algorithmic code by creating a table of data definitions.

For a complete list of best practices, see "Manual Fixed-Point Conversion Best Practices".

Separate Your Algorithm From the Test File

Write a MATLAB function, `mysum`, that sums the elements of a vector.

```
function y = mysum(x)
  y = 0;
  for n = 1:length(x)
    y = y + x(n);
  end
end
```

Since you only need to convert the algorithmic portion to fixed-point, it is more efficient to structure your code so that the algorithm, in which you do the core processing, is separate from the test file.

Write a Test Script

In the test file, create your inputs, call the algorithm, and plot the results.

1 Write a MATLAB script, `mysum_test`, that verifies the behavior of your algorithm using double data types.

```
n = 10;
rng default
```

```
x = 2*rand(n,1)-1;

% Algorithm
y = mysum(x);

% Verify results
y_expected = sum(double(x));

err = double(y) - y_expected
```

`rng default` puts the settings of the random number generator used by the rand function to its default value so that it produces the same random numbers as if you restarted MATLAB.

2 Run the test script.

```
mysum_test
```

```
err =

     0
```

The results obtained using `mysum` match those obtained using the MATLAB `sum` function.

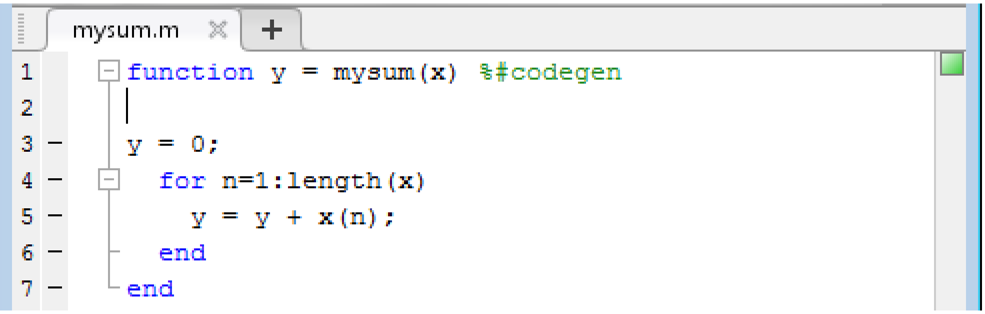
For more information, see "Create a Test File".

Prepare Algorithm for Instrumentation and Code Generation

In your algorithm, after the function signature, add the `%#codegen` compilation directive to indicate that you intend to instrument the algorithm and generate C code for it. Adding this directive instructs the MATLAB code analyzer to help you diagnose and fix violations that would result in errors during instrumentation and code generation.

```
function y = mysum(x) %#codegen
  y = 0;
  for n = 1:length(x)
    y = y + x(n);
  end
end
```

For this algorithm, the code analyzer indicator in the top right corner of the editor window remains green telling you that it has not detected any issues.



```
function y = mysum(x) %#codegen

y = 0;
  for n=1:length(x)
    y = y + x(n);
  end
end
```

For more information, see “Prepare Your Algorithm for Code Acceleration or Code Generation”.

Generate C Code for Your Original Algorithm

Generate C code for the original algorithm to verify that the algorithm is suitable for code generation and to see the floating-point C code. Use the `codegen` function (requires MATLAB Coder) to generate a C library.

1. Add the following line to the end of your test script to generate C code for `mysum`.

   ```
   codegen mysum -args {x} -config:lib -report
   ```
2. Run the test script again.

   MATLAB Coder generates C code for `mysum` function and provides a link to the code generation report.
3. Click the link to open the code generation report and view the generated C code for `mysum`.

   ```
   /* Function Definitions */
   double mysum(const double x[10])
   {
     double y;
     int n;
     y = 0.0;
     for (n = 0; n < 10; n++) {
       y += x[n];
    }

    return y;
   ```

```
 }
```

Because C does not allow floating-point indices, the loop counter, `n`, is automatically declared as an integer type. You do not need to convert `n` to fixed point.

Input `x` and output `y` are declared as double.

Manage Data Types and Control Bit Growth

Test Your Algorithm With Singles to Check for Type Mismatches

1. Modify your test file so that the data type of `x` is single.

```
n = 10;
rng default
x = single(2*rand(n,1)-1);

% Algorithm
y = mysum(x);

% Verify results
y_expected = sum(double(x));

err = double(y) - y_expected
codegen mysum -args {x} -config:lib -report
```

2. Run the test script again.

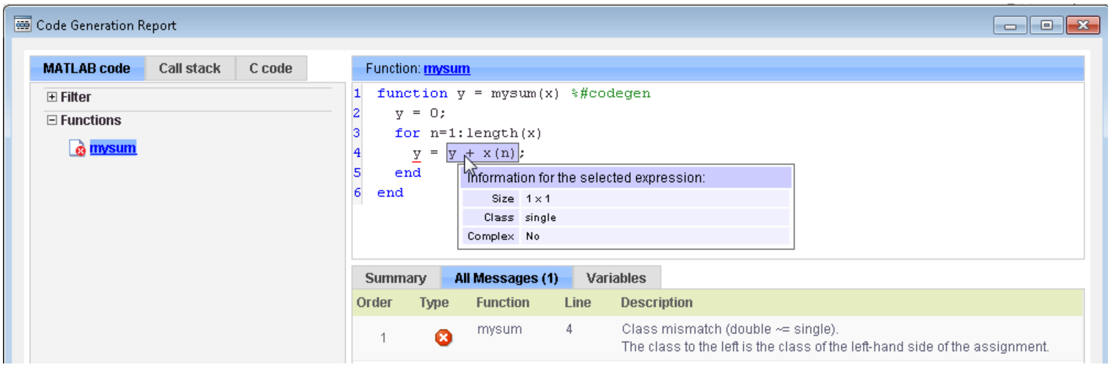
```
mysum_test
```

```
err =

     8.940696716308594e-08

??? Class mismatch (double ~= single).
The class to the left is the class of the left-hand side of the assignment.
```

Code generation fails, reporting a data type mismatch on line `y = y + x(n);`.

3. Click the link to the error report to open the report.

In the report, on the line `y = y + x(n)`, report underlines the `y` on the left side of the assignment in red to indicate that there is an error. The issue is that `y` is declared as a double but is being assigned to a single.`y + x(n)` is the sum of a double and a single which is a single. If you place your cursor over variables and expressions in the report, you can see information about their types. Here, you can see that the expression, `y + x(n)` is a single.



```
function y = mysum(x) %#codegen
  y = 0;
  for n=1:length(x)
    y = y + x(n);
  end
end
```

Information for the selected expression:

Size	1 x 1
Class	single
Complex	No

Order	Type	Function	Line	Description
1	✖	mysum	4	Class mismatch (double ~= single). The class to the left is the class of the left-hand side of the assignment.

4 To fix the type mismatch, update your algorithm to use subscripted assignment for the sum of elements. Change `y = y + x(n)` to `y(:) = y + x(n)`.

```
function y = mysum(x) %#codegen
  y = 0;
  for n = 1:length(x)
    y(:) = y + x(n);
  end
end
```

Using subscripted assignment, you also prevent the bit growth which is the default behavior when you add fixed-point numbers. For more information, see “Bit Growth”. Preventing bit growth is important because you want to maintain your fixed-point types throughout your code. For more information, see “Controlling Bit Growth”.

5 Regenerate C code and open the code generation report. In the C code, the result is now cast to double to resolve the type mismatch.

Build Instrumented Mex

Use the `buildInstrumentedMex` function to instrument your algorithm for logging minimum and maximum values of all named and intermediate variables. Use the `showInstrumentationResults` function to propose fixed-point data types based on these logged values. Later, you use these proposed fixed-point types to test your algorithm.

1 Update the test script:

a After you declare `n`, add `buildInstrumentedMex mySum —args {zeros(n,1)} -histogram`.

b Change `x` back to double. Replace `x = single(2*rand(n,1)-1);` with `x = 2*rand(n,1)-1;`

c Instead of calling the original algorithm, call the generated MEX function. Change `y = mysum(x)` to `y=mysum_mex(x)`.

d After calling the MEX function, add `showInstrumentationResults mysum_mex -defaultDT numerictype(1,16) -proposeFL`. The `-defaultDT numerictype(1,16) -proposeFL` flags indicate that you want to propose fraction lengths for a 16-bit word length.

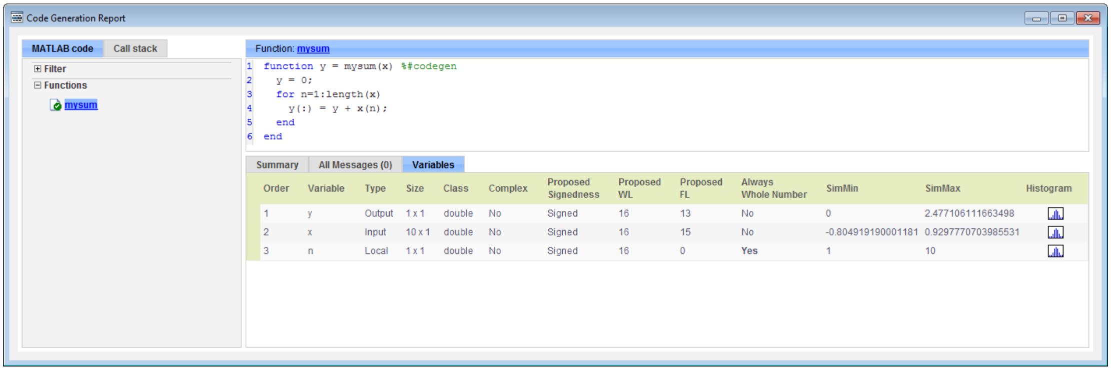
  Here is an updated test script.

```
%% Build instrumented mex
n = 10;

buildInstrumentedMex mysum -args {zeros(n,1)} -histogram

%% Test inputs
rng default
x = 2*rand(n,1)-1;

% Algorithm
y = mysum_mex(x);

% Verify results

showInstrumentationResults mysum_mex ...
  -defaultDT numerictype(1,16) -proposeFL
y_expected = sum(double(x));

err = double(y) - y_expected

%% Generate C code

codegen mysum -args {x} -config:lib -report
```

2 Run the test script again.

The `showInstrumentationResults` function proposes data types and opens the code generation report to display the results.

3 In the code generation report, click the **Variables** tab. `showInstrumentationResults` proposed a fraction length of for `y` and `x`. Here, it proposes a fraction length of 13 for `y` and 15 for `x`.



Function: mysum

```
function y = mysum(x) %#codegen
  y = 0;
  for n=1:length(x)
    y(:) = y + x(n);
  end
end
```

Order	Variable	Type	Size	Class	Complex	Proposed Signedness	Proposed WL	Proposed FL	Always Whole Number	SimMin	SimMax	Histogram
1	y	Output	1 x 1	double	No	Signed	16	13	No	0	2.477106111663498	📊
2	x	Input	10 x 1	double	No	Signed	16	15	No	-0.804919190001181	0.9297770703985531	📊
3	n	Local	1 x 1	double	No	Signed	16	0	**Yes**	1	10	📊

In the report, you can:

- View the simulation minimum and maximum values for the input `x` and output `y`.
- View the proposed data types for `x` and `y`.
- View information for all variables, intermediate results, and expressions in your code.

  To view this information, place your cursor over the variable or expression in the report.
- View the histogram data for `x` and `y` to help you identify any values that are outside range or below precision based on the current data type.

  To view the histogram for a particular variable, click its histogram icon, 📊.

Separate Data Type Definitions From Algorithmic Code

Rather than manually modifying the algorithm to examine the behavior for each data type, separate the data type definitions from the algorithm.

Modify `mysum` so that it takes an input parameter, `T`, which is a structure that defines the data types of the input and output data. When `y` is first defined, use the `cast` function like syntax — `cast(x,'like',y)` — to cast `x` to the desired data type.

```
function y = mysum(x,T) %#codegen
```

```
  y = cast(0,'like',T.y);
  for n = 1:length(x)
    y(:) = y + x(n);
  end
end
```

Create a Table of Data Type Definitions

Write a function, mytypes, that defines the different data types that you want to use to test your algorithm. In your data types table, include double, single, and scaled double data types as well as the fixed-point data types proposed earlier. Before converting your algorithm to fixed point, it is good practice to:

- Test the connection between the data type definition table and your algorithm using doubles.
- Test the algorithm with singles to find data type mismatches and other problems.
- Run the algorithm using scaled doubles to check for overflows.

```
function T = mytypes(dt)
  switch dt
    case 'double'
      T.x = double([]);
      T.y = double([]);
    case 'single'
      T.x = single([]);
      T.y = single([]);
    case 'fixed'
      T.x = fi([],true,16,15);
      T.y = fi([],true,16,13);
    case 'scaled'
      T.x = fi([],true,16,15,...
           'DataType','ScaledDouble');
      T.y = fi([],true,16,13,...
           'DataType','ScaledDouble');
  end
end
```

For more information, see “Separate Data Type Definitions from Algorithm”.

Update Test Script to Use Types Table

Update the test script, mysum_test, to use the types table.

1. For the first run, check the connection between table and algorithm using doubles. Before you declare `n`, add `T = mytypes('double');`
2. Update the call to `buildInstrumentedMex` to use the type of `T.x` specified in the data types table: `buildInstrumentedMex mysum -args {zeros(n,1,'like',T.x),T} -histogram`
3. Cast `x` to use the type of `T.x` specified in the table: `x = cast(2*rand(n,1)-1,'like',T.x);`
4. Call the MEX function passing in `T`: `y = mysum_mex(x,T);`
5. Call `codegen` passing in `T`: `codegen mysum -args {x,T} -config:lib -report`

   Here is the updated test script.

   ```
   %% Build instrumented mex
   T = mytypes('double');

   n = 10;

   buildInstrumentedMex mysum ...
       -args {zeros(n,1,'like',T.x),T} -histogram

   %% Test inputs
   rng default
   x = cast(2*rand(n,1)-1,'like',T.x);

   % Algorithm
   y = mysum_mex(x,T);

   % Verify results

   showInstrumentationResults mysum_mex ...
       -defaultDT numerictype(1,16) -proposeFL

   y_expected = sum(double(x));

   err = double(y) - y_expected

   %% Generate C code

   codegen mysum -args {x,T} -config:lib -report
   ```
6. Run the test script and click the link to open the code generation report.

The generated C code is the same as the code generated for the original algorithm. Because the variable `T` is used to specify the types and these types are constant at code generation time; `T` is not used at run time and does not appear in the generated code.

Generate Fixed-Point Code

Update the test script to use the fixed-point types proposed earlier and view the generated C code.

1 Update the test script to use fixed-point types. Replace `T = mytypes('double');` with `T = mytypes('fixed');` and then save the script.

2 Run the test script and view the generated C code.

   This version of C code is not very efficient; it contains a lot of overflow handling. The next step is to optimize the data types to avoid overflows.

Optimize Data Types

Use Scaled Doubles to Detect Overflow

Scaled doubles are a hybrid between floating-point and fixed-point numbers. Fixed-Point Designer stores them as doubles with the scaling, sign, and word length information retained. Because all the arithmetic is performed in double-precision, you can see any overflows that occur.

1 Update the test script to use scaled doubles. Replace `T = mytypes('double');` with `T = mytypes('scaled');`

2 Run the test script again.

   The test runs using scaled doubles and displays the report. No overflows are detected.

   So far, you've run the test script using random inputs which means that it is unlikely that the test has exercised the full operating range of the algorithm.

3 Find the full range of the input.

```
range(T.x)
```

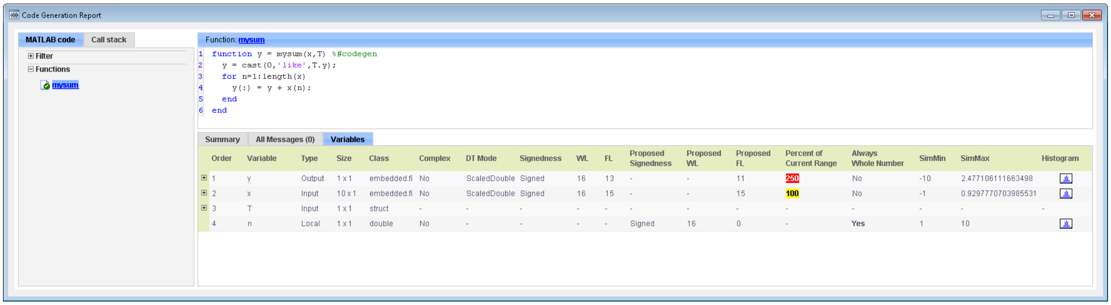
```
-1.000000000000000   0.999969482421875

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

4 Update the script to test the negative edge case. Run `mysum_mex` with the original random input and with an input that tests the full range and aggregate the results.

```
%% Build instrumented mex
T = mytypes('scaled');
n = 10;

buildInstrumentedMex mysum ...
    -args {zeros(n,1,'like',T.x),T} -histogram

%% Test inputs
rng default
x = cast(2*rand(n,1)-1,'like',T.x);
y = mysum_mex(x,T);
 % Run once with this set of inputs
y_expected = sum(double(x));
err = double(y) - y_expected

% Run again with this set of inputs. The logs will aggregate.
x = -ones(n,1,'like',T.x);
y = mysum_mex(x,T);
y_expected = sum(double(x));
err = double(y) - y_expected

% Verify results

showInstrumentationResults mysum_mex ...
    -defaultDT numerictype(1,16) -proposeFL

y_expected = sum(double(x));

err = double(y) - y_expected

%% Generate C code

codegen mysum -args {x,T} -config:lib -report
```

5 Run the test script again.

The test runs and `y` overflows the range of the fixed-point data type. `showInstrumentationResults` proposes a new fraction length of 11 for `y`.



```
function y = mysum(x,T) %#codegen
  y = cast(0,'like',T.y);
  for n=1:length(x)
    y(:) = y + x(n);
  end
end
```

Order	Variable	Type	Size	Class	Complex	DT Mode	Signedness	WL	FL	Proposed Signedness	Proposed WL	Proposed FL	Percent of Current Range	Always Whole Number	SimMin	SimMax	Histogram
⊞ 1	y	Output	1 x 1	embedded.fi	No	ScaledDouble	Signed	16	13	-	-	11	250	No	-10	2.477106111663498	📊
⊞ 2	x	Input	10 x 1	embedded.fi	No	ScaledDouble	Signed	16	15	-	-	15	100	No	-1	0.9297770703985531	📊
⊞ 3	T	Input	1 x 1	struct	-	-	-	-	-	-	-	-	-	-	-	-	-
4	n	Local	1 x 1	double	No	-	-	-	-	Signed	16	0	-	Yes	1	10	📊

6. Update the test script to use scaled doubles with the new proposed type for `y`. In `myTypes.m`, for the `'scaled'` case, `T.y = fi([],true,16,11,'DataType','ScaledDouble')`

7. Rerun the test script.

   There are now no overflows.

Generate Code for the Proposed Fixed-Point Type

Update the data types table to use the proposed fixed-point type and generate code.

1. In `myTypes.m`, for the `'fixed'` case, `T.y = fi([],true,16,11)`
2. Update the test script, `mysum_test`, to use `T = mytypes('fixed');`
3. Run the test script and then click the View Report link to view the generated C code.

```
short mysum(const short x[10])
{
  short y;
  int n;
  int i0;
  int i1;
  int i2;
  int i3;
  y = 0;
  for (n = 0; n < 10; n++) {
    i0 = y << 4;
    i1 = x[n];
    if ((i0 & 1048576) != 0) {
      i2 = i0 | -1048576;
    } else {
      i2 = i0 & 1048575;
   }

    if ((i1 & 1048576) != 0) {
     i3 = i1 | -1048576;
    } else {
      i3 = i1 & 1048575;
    }

  i0 = i2 + i3;
  if ((i0 & 1048576) != 0) {
    i0 |= -1048576;
  } else {
    i0 &= 1048575;
  }

  i0 = (i0 + 8) >> 4;
  if (i0 > 32767) {
    i0 = 32767;
  } else {
      if (i0 < -32768) {
        i0 = -32768;
      }
    }

   y = (short)i0;
  }
  return y;
}
```

By default, `fi` arithmetic uses saturation on overflow and nearest rounding which results in inefficient code.

Modify fimath Settings

To make the generated code more efficient, use fixed-point math (`fimath`) settings that are more appropriate for C code generation: wrap on overflow and floor rounding.

1 In `myTypes.m`, add a `'fixed2'` case:

```
case 'fixed2'
     F = fimath('RoundingMethod', 'Floor', ...
          'OverflowAction', 'Wrap', ...
          'ProductMode', 'FullPrecision', ...
          'SumMode', 'KeepLSB', ...
          'SumWordLength', 32, ...
          'CastBeforeSum', true);
     T.x = fi([],true,16,15,F);
     T.y = fi([],true,16,11,F);
```

Tip Instead of manually entering `fimath` properties, you can use the MATLAB Editor **Insert fimath** option. For more information, see “Building fimath Object Constructors in a GUI”.

2 Update the test script to use `'fixed2'`, run the script, and then view the generated C code.

```
short mysum(const short x[10])
{
 short y;
 int n;
 y = 0;
 for (n = 0; n < 10; n++) {
   y = (short)(((y << 4) + x[n]) >> 4);
 }

  return y;
}
```

The generated code is more efficient, but `y` is shifted to align with `x` and loses 4 bits of precision.

3 To fix this precision loss, update the word length of `y` to 32 bits and keep 15 bits of precision to align with `x`.

In `myTypes.m`, add a `'fixed32'` case:

```
 case 'fixed32'
      F = fimath('RoundingMethod', 'Floor', ...
           'OverflowAction', 'Wrap', ...
           'ProductMode', 'FullPrecision', ...
           'SumMode', 'KeepLSB', ...
           'SumWordLength', 32, ...
           'CastBeforeSum', true);
      T.x = fi([],true,16,15,F);
      T.y = fi([],true,32,15,F);
```

4 Update the test script to use `'fixed32'` and run the script to generate code again.

Now, the generated code is very efficient.

```
int mysum(const short x[10])
{
  int y;
  int n;
  y = 0;
  for (n = 0; n < 10; n++) {
    y += x[n];
  }

  return y;
}
```

For more information, see “Optimize Your Algorithm”.

About Fixed-Point

Fixed-Point Designer Product Description

Design and simulate fixed-point systems

Fixed-Point Designer provides data types and tools for developing fixed-point algorithms in MATLAB code, Simulink models, and Stateflow® charts. It automatically proposes fixed-point data types and attributes such as word length and rounding mode. You can also specify these attributes manually. You can perform accelerated bit-true simulations to observe the impact of limited range and precision.

Fixed-Point Designer lets you convert floating-point algorithms to fixed point. You can create and optimize fixed-point data types that meet your numerical accuracy requirements and target-hardware constraints. You can simulate your design and analyze the results to obtain range information. Fixed-Point Designer uses this information to recommend word lengths and scalings for your fixed-point algorithm and enable you to compare fixed-point results with floating-point baselines.

Fixed-Point Designer supports C, HDL, and PLC code generation.

Key Features

- Fixed-point data types in MATLAB, Simulink, and Stateflow
- Bit-true, accelerated simulation of fixed-point algorithms
- Recommendations for word length and scaling data type
- Floating- and fixed-point results comparison
- Fixed-point data type override to double or single precision
- Support for C, HDL, and PLC code generation

Benefits of Using Fixed-Point Hardware

Digital hardware is becoming the primary means by which control systems and signal processing filters are implemented. Digital hardware can be classified as either off-the-shelf hardware (for example, microcontrollers, microprocessors, general-purpose processors, and digital signal processors) or custom hardware. Within these two types of hardware, there are many architecture designs. These designs range from systems with a single instruction, single data stream processing unit to systems with multiple instruction, multiple data stream processing units.

Within digital hardware, numbers are represented as either fixed-point or floating-point data types. For both these data types, word sizes are fixed at a set number of bits. However, the dynamic range of fixed-point values is much less than floating-point values with equivalent word sizes. Therefore, in order to avoid overflow or unreasonable quantization errors, fixed-point values must be scaled. Since floating-point processors can greatly simplify the real-time implementation of a control law or digital filter, and floating-point numbers can effectively approximate real-world numbers, then why use a microcontroller or processor with fixed-point hardware support?

- Size and Power Consumption — The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. For example, consider a portable telephone where one of the product design goals is to make it as portable (small and light) as possible. If one of today's high-end floating-point, general-purpose processors is used, a large heat sink and battery would also be needed, resulting in a costly, large, and heavy portable phone.
- Memory Usage and Speed — In general fixed-point calculations require less memory and less processor time to perform.
- Cost — Fixed-point hardware is more cost effective where price/cost is an important consideration. When digital hardware is used in a product, especially mass-produced products, fixed-point hardware costs much less than floating-point hardware and can result in significant savings.

After making the decision to use fixed-point hardware, the next step is to choose a method for implementing the dynamic system (for example, control system or digital filter). Floating-point software emulation libraries are generally ruled out because of timing or memory size constraints. Therefore, you are left with fixed-point math where binary integer values are scaled.

View Fixed-Point Data

In Fixed-Point Designer software, the `fipref` object determines the display properties of `fi` objects. Code examples generally show `fi` objects as they appear with the following `fipref` object properties:

- `NumberDisplay` — `'RealWorldValue'`
- `NumericTypeDisplay` — `'full'`
- `FimathDisplay` — `'full'`

Setting `'FimathDisplay'` to `'full'` provides a quick and easy way to differentiate between `fi` objects with a local fimath and those that are associated with the default fimath. When `'FimathDisplay'` is set to `'full'`, MATLAB displays `fimath` object properties for `fi` objects with a local fimath. MATLAB never displays `fimath` object properties for `fi` objects that are associated with the default fimath. Because of this display difference, you can tell when a `fi` object is associated with the default fimath just by looking at the output.

Additionally, unless otherwise specified, examples throughout the Fixed-Point Designer documentation use the following default configuration of fimath:

```
        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

For more information on display settings, refer to “fi Object Display Preferences Using fipref”.

Displaying the fimath Properties of fi Objects

To see the output as it appears in most Fixed-Point Designer code examples, set your `fipref` properties as follows and create two `fi` objects:

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'full', 'FimathDisplay', 'full');
a = fi(pi,'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap')
b = fi(pi)
```

MATLAB returns the following:

```
a =
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision

b =
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

MATLAB displays `fimath` object properties in the output of `fi` object `a` because `a` has a local fimath.

MATLAB does not display any `fimath` object properties in the output of `fi` object `b` because `b` associates itself with the default fimath.

Hiding the fimath Properties of fi Objects

If you are working with multiple `fi` objects that have local fimaths, you may want to turn off the `fimath` object display:

- `NumberDisplay` — `'RealWorldValue'`
- `NumericTypeDisplay` — `'full'`
- `FimathDisplay` — `'none'`

For example,

```
p = fipref('NumberDisplay','RealWorldValue',...
'NumericTypeDisplay','full','FimathDisplay','none')

p =
         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'none'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'

F = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
a = fi(pi, F)

a =
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Although this setting helps decrease the amount of output produced, it also makes it impossible to tell from the output whether a `fi` object uses the default fimath. To do so, you can use the `isfimathlocal` function. For example,

```
isfimathlocal(a)

ans =
     1
```

When the `isfimathlocal` function returns `1`, the `fi` object has a local fimath. If the function returns `0`, the `fi` object uses the default fimath.

Shortening the numerictype Display of fi Objects

To reduce the amount of output even further, you can set the `NumericTypeDisplay` to `'short'`. For example,

```
p = fipref('NumberDisplay','RealWorldValue',...
'NumericTypeDisplay','short','FimathDisplay','full');

a = fi(pi)

a =
    3.1416
      s16,13
```

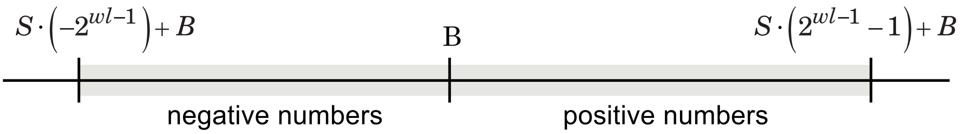
Precision and Range

In this section...
"Range" on page 2-7
"Precision" on page 2-8

Note: You must pay attention to the precision and range of the fixed-point data types and scalings you choose in order to know whether rounding methods will be invoked or if overflows or underflows will occur.

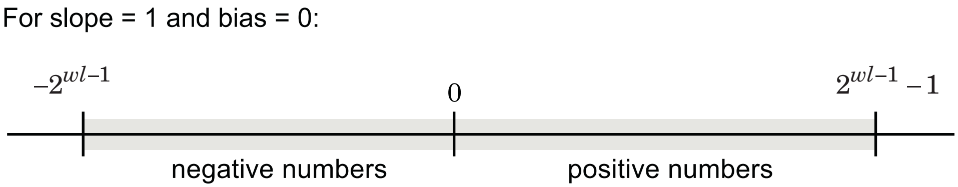
Range

The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl, scaling S and bias B is illustrated below:


$S \cdot \left(-2^{wl-1}\right) + B$
B
$S \cdot \left(2^{wl-1} - 1\right) + B$
negative numbers
positive numbers

For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is $2^{wl-1} - 1$. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1}:



Overflow Handling

Because a fixed-point data type represents numbers within a finite range, overflows and underflows can occur if the result of an operation is larger or smaller than the numbers in that range.

Fixed-Point Designer software allows you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type.

When you create a `fi` object, any overflows are saturated. The `OverflowAction` property of the default fimath is `saturate`. You can log overflows and underflows by setting the `LoggingMode` property of the `fipref` object to `on`. Refer to “LoggingMode” for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Methods

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, a rounding method is used to cast the value to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding method itself. To provide you with greater flexibility in the trade-off between cost and bias, Fixed-Point Designer software currently supports the following rounding methods:

- `Ceiling` rounds to the closest representable number in the direction of positive infinity.
- `Convergent` rounds to the closest representable number. In the case of a tie, `convergent` rounds to the nearest even number. This is the least biased rounding method provided by the toolbox.
- `fix` rounds to the closest representable number in the direction of zero.
- `Floor`, which is equivalent to two's complement truncation, rounds to the closest representable number in the direction of negative infinity.
- `Nearest` rounds to the closest representable number. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity. This rounding method is the default for `fi` object creation and `fi` arithmetic.
- `Round` rounds to the closest representable number. In the case of a tie, the `round` method rounds:
  - Positive numbers to the closest representable number in the direction of positive infinity.
  - Negative numbers to the closest representable number in the direction of negative infinity.

Choosing a Rounding Method

Each rounding method has a set of inherent properties. Depending on the requirements of your design, these properties could make the rounding method more or less desirable to you. By knowing the requirements of your design and understanding the properties of each rounding method, you can determine which is the best fit for your needs. The most important properties to consider are:

- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?
  - Low — The method requires few processing cycles.
  - Moderate — The method requires a moderate number of processing cycles.
  - High — The method requires more processing cycles.

Note: The cost estimates provided here are hardware independent. Some processors have rounding modes built-in, so consider carefully the hardware you are using before calculating the true cost of each rounding mode.

- Bias — What is the expected value of the rounded values minus the original values: $\mathrm{E}\left(\hat{\theta}-\theta\right)$?
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)<0$ — The rounding method introduces a negative bias.
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)=0$ — The rounding method is unbiased.
  - $\mathrm{E}\left(\hat{\theta}-\theta\right)>0$ — The rounding method introduces a positive bias.
- Possibility of Overflow — Does the rounding method introduce the possibility of overflow?
  - Yes — The rounded values may exceed the minimum or maximum representable value.
  - No — The rounded values will never exceed the minimum or maximum representable value.

The following table shows a comparison of the different rounding methods available in the Fixed-Point Designer product.

Fixed-Point Designer Rounding Mode	Cost	Bias	Possibility of Overflow
`Ceiling`	Low	Large positive	Yes
`Convergent`	High	Unbiased	Yes
`Zero`	Low	• Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples	No
`Floor`	Low	Large negative	No
`Nearest`	Moderate	Small positive	Yes
`Round`	High	• Small negative for negative samples	Yes

Fixed-Point Designer Rounding Mode	Cost	Bias	Possibility of Overflow
		• Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples	
`Simplest` (Simulink only)	Low	Depends on the operation	No

Scaling

Fixed-point numbers can be encoded according to the scheme

$$\text{real-world value} = (\text{slope} \times \text{integer}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In Fixed-Point Designer documentation, the negative of the fixed exponent is often referred to as the *fraction length*.

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in [Slope Bias] representation that has a bias equal to zero and a slope adjustment factor equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

$$\text{real-world value} = 2^{\text{fixed exponent}} \times \text{integer}$$

or

$$\text{real-world value} = 2^{\text{-fixed exponent}} \times \text{integer}$$

Fixed-Point Designer software supports both binary point-only scaling and [Slope Bias] scaling.

Fixed-Point Arithmetic

In this section...

Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

$$\begin{array}{r l} 010010.1 & (18.5) \\ +0110.110 & (6.75) \\ \hline 011001.010 & (25.25) \end{array}$$

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign-extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

$$\begin{array}{r l} 010010.100 & (18.5) \\ -0110.110 & (6.75) \\ \hline \end{array}$$

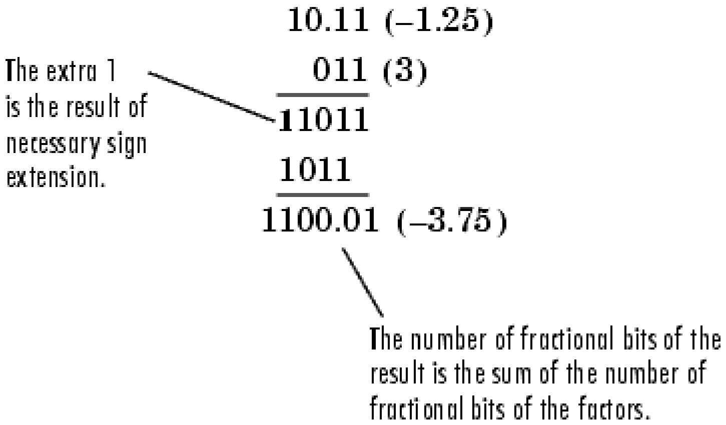
The default global fimath has a value of `1` (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

If `CastBeforeSum` has a value of `0` (false), the addends are added with full precision maintained. After the addition the sum is then quantized.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign-extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):



The extra 1 is the result of necessary sign extension.

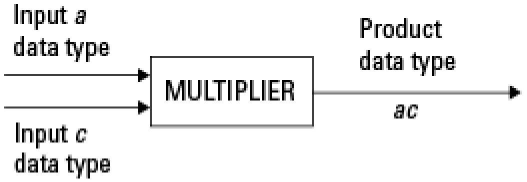
The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication using Fixed-Point Designer software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication.

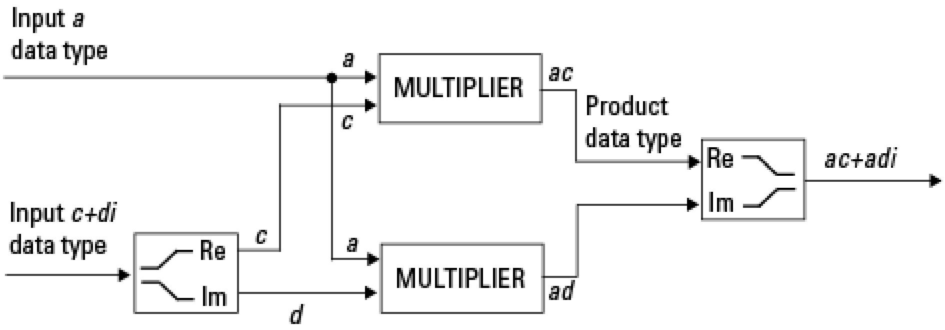
Real-Real Multiplication

The following diagram shows the data types used by the toolbox in the multiplication of two real numbers. The software returns the output of this operation in the product data type, which is governed by the `fimath` object `ProductMode` property.



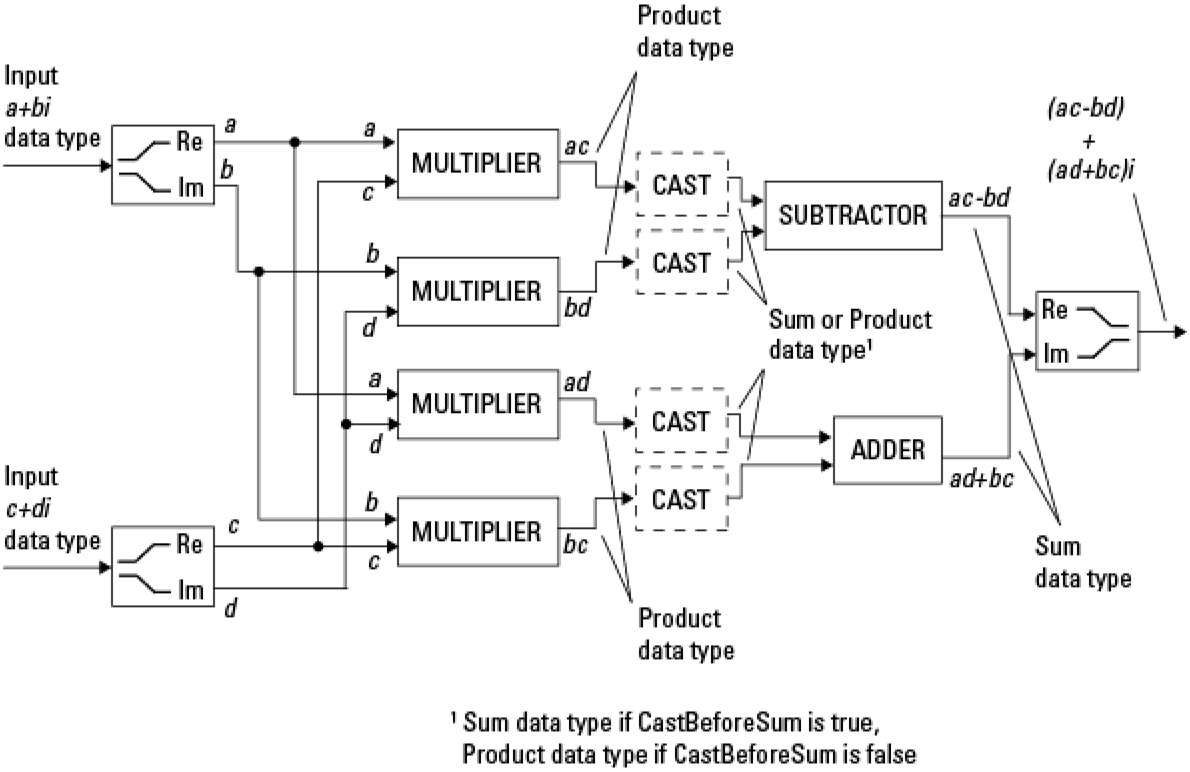
Real-Complex Multiplication

The following diagram shows the data types used by the toolbox in the multiplication of a real and a complex fixed-point number. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product data type, which is governed by the `fimath` object `ProductMode` property:



Complex-Complex Multiplication

The following diagram shows the multiplication of two complex fixed-point numbers. Note that the software returns the output of this operation in the sum data type, which is governed by the `fimath` object `SumMode` property. The intermediate product data type is determined by the `fimath` object `ProductMode` property.



When the `fimath` object `CastBeforeSum` property is `true`, the casts to the sum data type are present after the multipliers in the preceding diagram. In C code, this is equivalent to

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator. When the `CastBeforeSum` property is `false`, the casts are not present, and the data remains in the product data type before the subtraction and addition operations.

Multiplication with fimath

In the following examples, let

```
F = fimath('ProductMode','FullPrecision',...
'SumMode','FullPrecision');
T1 = numerictype('WordLength',24,'FractionLength',20);
T2 = numerictype('WordLength',16,'FractionLength',10);
```

Real*Real

Notice that the word length and fraction length of the result `z` are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the `fimath` `SumMode` and `ProductMode` properties are set to `FullPrecision`:

```
P = fipref;
P.FimathDisplay = 'none';
x = fi(5, T1, F)

x =

     5


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 20

y = fi(10, T2, F)

y =

    10


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

z = x*y

z =
    50


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 30
```

Real*Complex

Notice that the word length and fraction length of the result `z` are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the `fimath` `SumMode` and `ProductMode` properties are set to `FullPrecision`:

```
x = fi(5,T1,F)

x =

     5


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 20

y = fi(10+2i,T2,F)

y =

  10.0000 + 2.0000i


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

z = x*y

z =

  50.0000 +10.0000i
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 30
```

Complex*Complex

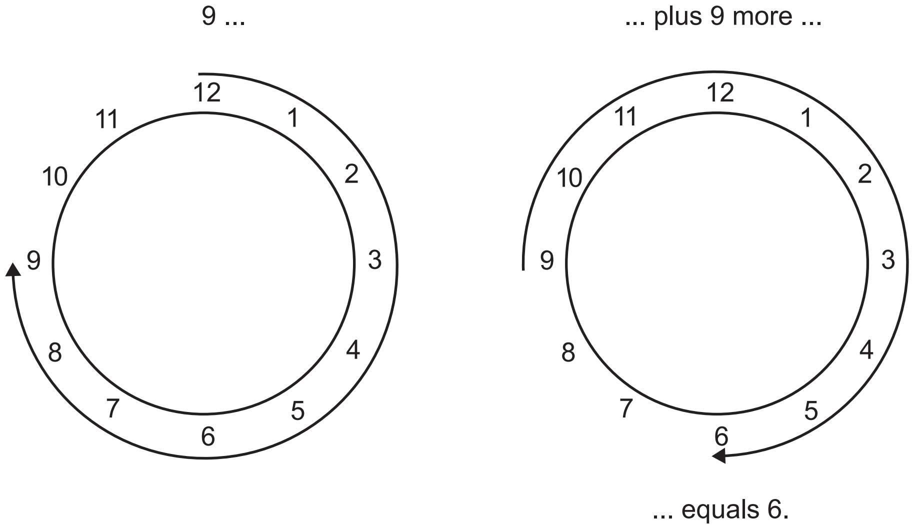
Complex-complex multiplication involves an addition as well as multiplication, so the word length of the full-precision result has one more bit than the sum of the word lengths of the multiplicands:

```
x = fi(5+6i,T1,F)

x =

   5.0000 + 6.0000i


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 20

y = fi(10+2i,T2,F)

y =

  10.0000 + 2.0000i


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

z = x*y

z =

  38.0000 +70.0000i


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 41
        FractionLength: 30
```

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the “clock” system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:



Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped “around the circle” to either 0 or 1.

Two's Complement

Two's complement is a way to interpret a binary number. In two's complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit of a two's complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two's complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 = (0 + 2^0) = 1$$
$$11 = \left(\left(-2^1\right) + \left(2^0\right)\right) = (-2 + 1) = -1$$

To compute the negative of a binary number using two's complement,

1 Take the one's complement, or “flip the bits.”

2 Add a 2^(-*FL*) using binary math, where *FL* is the fraction length.

3 Discard any bits carried beyond the original word length.

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

$$11010 \rightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

$$\begin{array}{r} 00101 \\ +1 \\ \hline 00110 \end{array} \quad (6)$$

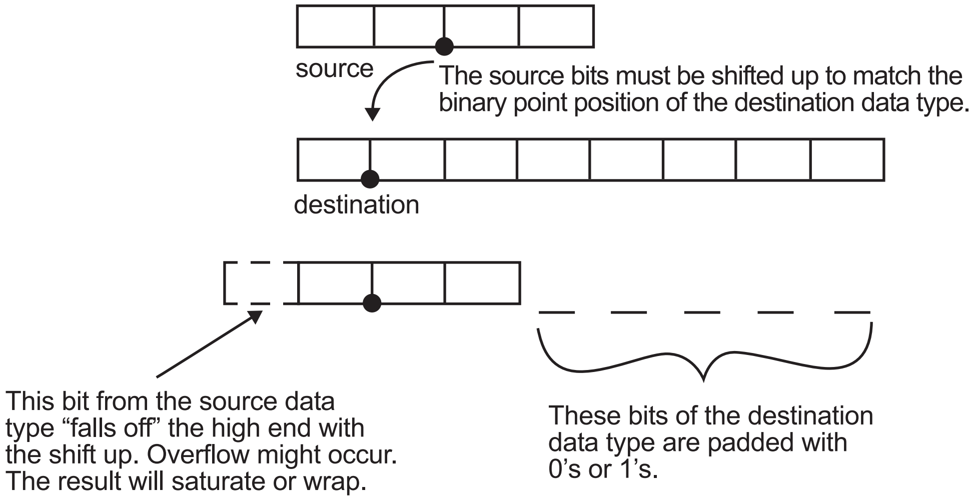
Casts

The `fimath` object allows you to specify the data type and scaling of intermediate sums and products with the `SumMode` and `ProductMode` properties. It is important to keep in mind the ramifications of each cast when you set the `SumMode` and `ProductMode` properties. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples show cases where overflow and rounding can occur.

Note For more examples of casting, see “Cast fi Objects”.

Casting from a Shorter Data Type to a Longer Data Type

Consider the cast of a nonzero number, represented by a 4-bit data type with two fractional bits, to an 8-bit data type with seven fractional bits:


source
The source bits must be shifted up to match the binary point position of the destination data type.
destination
This bit from the source data type “falls off” the high end with the shift up. Overflow might occur. The result will saturate or wrap.
These bits of the destination data type are padded with 0’s or 1’s.

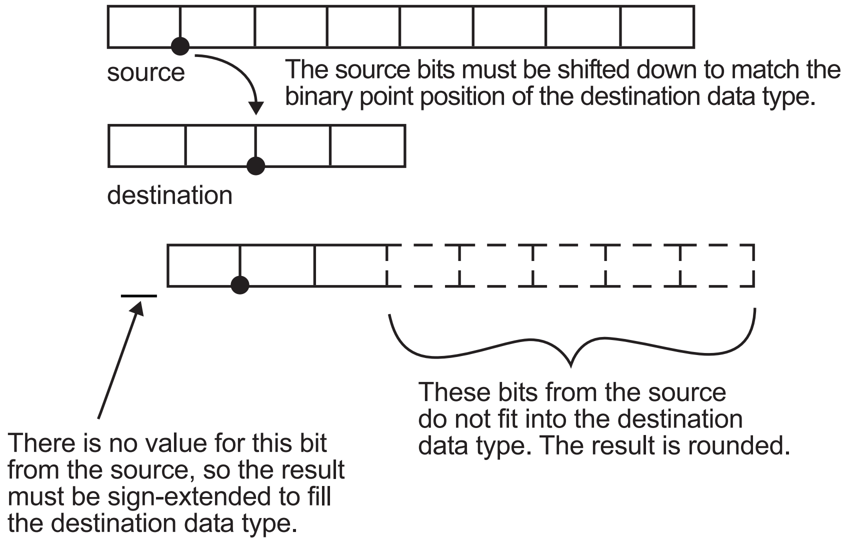
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.
- If wrapping occurs, the empty bits are padded with 0's.
- If saturation occurs,
  - The empty bits of a positive number are padded with 1's.
  - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow can still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

Casting from a Longer Data Type to a Shorter Data Type

Consider the cast of a nonzero number, represented by an 8-bit data type with seven fractional bits, to a 4-bit data type with two fractional bits:


source
The source bits must be shifted down to match the binary point position of the destination data type.
destination
There is no value for this bit from the source, so the result must be sign-extended to fill the destination data type.
These bits from the source do not fit into the destination data type. The result is rounded.

As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so “sign extension” is used to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision can be lost as the result is rounded.

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow can occur.

The worst case occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.