

# **MaxCompiler**

## **Kernel Numerics Tutorial**

Version 2014.2



# Contents

<b>Contents</b>	<b>1</b>
<b>Preface</b>	<b>2</b>
<b>1 Floating-Point Data Types</b>	<b>4</b>
1.1 Floating Point Overview	4
1.1.1 Range of a Floating-Point Number	4
1.1.2 Special Values	5
1.1.3 Exponent Bias	5
1.2 MaxCompiler IEEE Floating-Point Compliance	5
1.3 Allowable Mantissa and Exponent Sizes	6
1.4 Floating-Point Utilities	6
1.5 Casting Floating-Point Streams	6
<b>2 Fixed-Point Data Types</b>	<b>8</b>
2.1 Fixed Point Overview	8
2.2 Offset Fixed Point	8
2.3 Inferring Offsets via Maximum Values	10
2.4 Rounding Modes	10
<b>3 Numeric Exceptions</b>	<b>11</b>
3.1 Floating-Point Exceptions	11
3.2 Fixed-Point Exceptions	12
3.3 Numeric Exception Behavior	14
3.4 Enabling Numeric Exceptions	14
3.5 Reading Numeric Exception Data	16
3.6 Mapping Numeric Exceptions to the Kernel Source	18
3.7 Conditionally Enabling Numeric Exceptions	22

---

## Preface

### Purpose of this document

This document covers features of MaxCompiler that allow you to optimize numerics within a Kernel, including **numeric exceptions** (such as overflow). Numeric exceptions allow you to see *which numeric exceptions* occurred for *which operations*.

The first two sections offer an overview of **floating-point** and **fixed-point** data types in MaxCompiler.

Each section introduces a new set of features and goes through examples showing their use, where appropriate.

### Document Conventions

When important concepts are introduced for the first time, they appear in **bold**. *Italics* are used for emphasis. Directories and commands are displayed in typewriter font. Variable and function names are also displayed in typewriter font.

Java methods and classes are shown using the following format:

```
void debug.pushEnableNumericExceptions(boolean enable);
```

C function prototypes are similar:

```
int max_numeric_ex_enabled(max_maxfile_t *maxfile, const char *design_name)
```

Actual Java usage is shown without italics:

```
debug.simPrintf(result.hasAnyDoubt(), "cycle %d, result = %d?\n", cnt, result);
```

C usage is similarly without italics:

```
max.show_numeric_ex_masked(device, "MovingAverageExceptionsKernel", 0, stdout, 1);
```

Sections of code taken from the source of the examples appear with a border and line numbers:

```
24
25     // Control
26     DFEVar cnt = control.count.simpleCounter(32, N);
27
28     DFEVar sel_n1 = cnt > 0;
29     DFEVar sel_nu = cnt < N-1;
30
31     DFEVar sel_m = sel_n1 & sel_nu;
32
33     DFEVar prev = sel_n1 ? x_prev : 0;
34     DFEVar next = sel_nu ? x_next : 0;
35
36     DFEVar divisor = sel_m ? constant.var(dfeInt(8), 3) : 2;
37
38     // Calculation
39     debug.pushEnableNumericExceptions(true);
40
41     DFEVar sum = prev+x+next;
42     DFEVar result = sum/divisor;
43
44     debug.popEnableNumericExceptions();
45
46     // Output
47     io.output("y", result, result.getType());
48 }
49 }
```

## 1 Floating-Point Data Types

MaxCompiler supports floating-point data streams both in IEEE 754 standard formats (half-, single- and double-precision) and with user-specified sizes of mantissa and exponent.

A floating-point type is parameterized with mantissa and exponent bit-widths in MaxCompiler using `dfeFloat`:

```
DFEFloat dfeFloat(int exponent_bits, int mantissa_bits)
```

### 1.1 Floating Point Overview

Floating-point representation allows a wider range of numbers to be represented than integer or fixed-point representation. Floating point uses a fixed number of bits to represent an approximation of the number (the mantissa) which is scaled by an exponent.

Floating-point numbers are normalized such that the most significant bit of the mantissa is 1 (i.e.  $1 \leq \text{mantissa} < 2$ ). This allows an optimization where the most significant bit of the mantissa is not stored: it is implicit. This spare bit is then used to represent the sign of the number.

Consider a 32-bit floating-point number with a 24-bit mantissa ( $m$ ) and an 8-bit exponent ( $e$ ). This would be declared as `dfeFloat(8, 24)` in MaxCompiler. The storage required for the mantissa is actually 23 bits, leaving one bit for the sign ( $s$ ), as shown in [Figure 1](#).

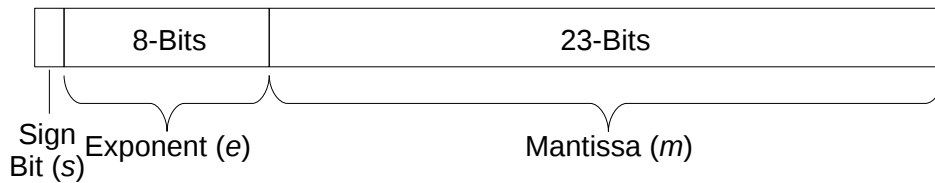


Figure 1: 32-bit Floating-point number.

The equation for working out the real number that this represents is:

$$x = (-1)^s \times m \times 2^e$$

#### 1.1.1 Range of a Floating-Point Number

The range of a floating point number in MaxCompiler is between  $\pm 1 \times 2^{-\frac{2^N}{2}-2}$  and approximately  $\pm 2 \times 2^{\frac{2^N}{2}-1}$ , where  $N$  is the number of bits in the exponent. For single precision (`dfeFloat(8, 24)`), this is between  $\pm 1 \times 2^{-126}$  and approximately  $\pm 2 \times 2^{127}$ .



Note that the maximum exponent is +127 and the minimum -126 as two values are reserved for special values.

### 1.1.2 Special Values

Certain values of the exponent and mantissa represent special values (shown in [Table 1](#)). These values are the same for all bit-widths of exponent and mantissa.

Value	Exponent	Mantissa
$\pm 0$	All zeroes	0
NaN (Not a Number)	All ones	Non-zero
$\pm\text{infinity}$	All ones	0

Table 1: Special Values for floating-point numbers.

### 1.1.3 Exponent Bias

The exponent is *biased*, which means it is offset from its actual value when stored. For single precision (`dfloat(8,24)`) the bias is +127. This needs to be subtracted from the stored value when working out the number represented in floating point. So, for a single-precision number with an exponent of -100, this is stored as 27. For double precision (`dfloat(11,53)`), the bias is +1023.

The general equation for calculating the bias is:

$$2^{N-1} - 1$$

Where N is the number of bits in the exponent.

## 1.2 MaxCompiler IEEE Floating-Point Compliance

When there are 8 exponent bits and 24 mantissa bits in a floating-point type in MaxCompiler, the floating-point format is equivalent to IEEE 754 single-precision. Similarly, for an exponent of 11 bits and mantissa 53 bits the type is in IEEE 754 double-precision format.

MaxCompiler floating-point numbers vary from the IEEE standard in the following ways:

- No support for denormalized numbers. Results that would have been denormalized are set to an appropriately signed zero.
  - To increase the dynamic range in MaxCompiler, you can increase the width of the exponent.
  - If slight variations between results from a CPU implementation and a MaxCompiler implementation are to be avoided, we recommend that the software model either disables denormalized numbers in the CPU (if possible) or checks for denormalized numbers and explicitly rounds to zero.
- Only the default rounding mode, round to nearest (`RoundingMode.TONEAREVEN` in MaxCompiler), is supported.
- All NaNs are treated as Quiet-NaN: no exceptions are raised. The only exception to this rule is as an input to a floating-point to fixed-point conversion, when an exception is raised (see [section 3](#)).

### 1.3 Allowable Mantissa and Exponent Sizes

The valid range for the number of bits of the exponent (`exponent_bits`) is between 4 and 16 inclusive.

The valid range for the number of bits of the mantissa (`mantissa_bits`) depends on the exponent width. The minimum mantissa size is always 4, including the implicit bit due to normalization, and the maximum is determined by [Table 2](#).

Exponent Width	Max. Mantissa Size
4	5
5	13
6	29
7	61
8 - 16	64

Table 2: Allowable Exponent and Mantissa size combinations.



All mantissa widths in [Table 2](#) include the implicit bit due to normalization.

### 1.4 Floating-Point Utilities

MaxCompiler provides a number of utility methods for querying and manipulating floating-point types.

The `getMantissaBits` method returns the number of bits for the mantissa in a floating-point stream type:

```
int DFEFloat.getMantissaBits()
```



`getMantissaBits()` returns the width of the mantissa as declared, *including the implicit bit* e.g. 24 for a floating point number declared as `dfeFloat(8,24)`.

`getExponentBits` returns the number of bits for the exponent:

```
int DFEFloat.getExponentBits()
```

The method `getExponentBias()` is available to get the bias of the exponent (see [subsubsection 1.1.3](#)):

```
int DFEFloat.getExponentBias()
```

### 1.5 Casting Floating-Point Streams

Casting from `DFERawBits` to and from floating point is free. However, casting to a different Kernel type, for example between `DFEFloat` and `DFEFloat`, can be expensive in terms of resource usage, so should be used judiciously.

## 1 Floating-Point Data Types

---

A numeric exception is raised if a cast from a floating-point stream causes overflow or underflow.



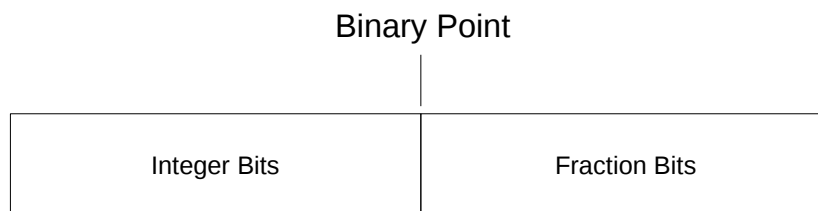
## 2 Fixed-Point Data Types

An important task in the optimization of many MaxCompiler designs is conversion from using floating-point data to using fixed-point data. Many applications do not require either the full dynamic range or the precision of a floating-point representation. This is especially true for applications where the inputs are of limited precision in the first place.

MaxCompiler has built-in fixed-point types with various modes of rounding and scaling.

### 2.1 Fixed Point Overview

In an integer, the binary point is implicitly to the right of the least significant bit of the binary word. In fixed-point representation, the position of the binary point is moved to leftwards, so some bits are used to represent fractional values, as shown in [Figure 2](#).



*Figure 2: Fixed point number*

In this example of 8 integer bits and 8 fractional bits, in two's-complement mode, the largest possible value is  $127 + 255/256 = 127.99609375$ , the smallest possible value is  $-128$  and each fractional bit represents a  $1/256 = 0.00390625$ .

### 2.2 Offset Fixed Point

In MaxCompiler, fixed-point types are described using an offset of the binary point. A negative offset moves the binary point to the left through the binary word from immediately to the right of the least significant bit. A positive offset moves the binary point to the right.

A negative offset greater than the width of the word leads to a smaller range of fractions represented with greater precision. A positive offset allows us to represent a larger range of integer values losing the precision of the lower bits, so we can only represent integers divisible by  $2^{\text{offset}}$ .

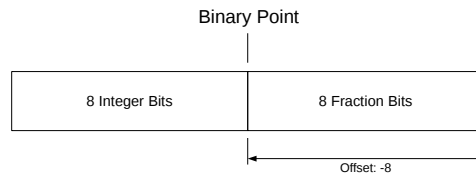
A number of examples of offsets on a 16-bit binary word are shown in [Figure 3](#).

An offset fixed-point type is declared using `dfeFixOffset`:

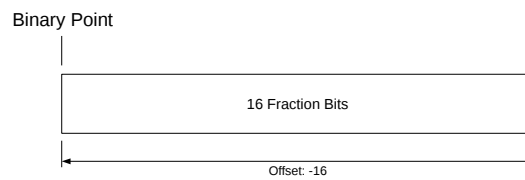
```
DfeFix dfeFixOffset(int num_bits, int offset, SignMode sign_mode)
```

where `num_bits` is the total number of bits in the word, and `offset` is the position of the offset relative to the least significant bit in the binary word (negative to the left, positive to the right). `sign_mode` can be one of `SignMode.TWOSCOMPLEMENT` or `SignMode.UNSIGNED`.

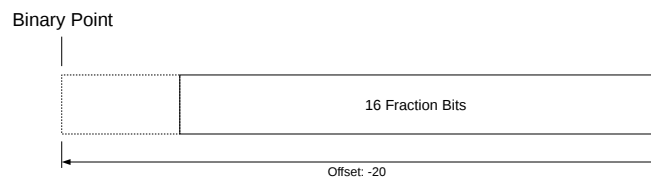
## 2 Fixed-Point Data Types



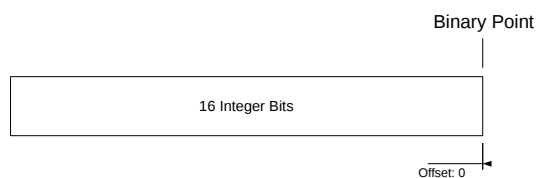
(a) -8: 8 integer bits and 8 fractional bits representing  $(128 > x \geq -128)$



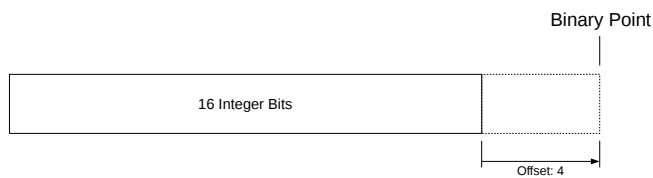
(b) -16: All fraction bits ( $0.5 > x \geq -0.5$ )



(c) -20: All fractions bits ( $0.03125 > x \geq -0.03125$ )



(d) 0: Integer Equivalent ( $128 > x \geq -128$ )



(e) 4: Integers that are multiples of 16

Figure 3: Different offsets on a 16-bit word in two's-complement mode

### 2.3 Inferring Offsets via Maximum Values

As well as allowing the fixed-point offset to be expressed explicitly, MaxCompiler can infer the offset from a maximum value. This is the recommended method for creating fixed-point types.

The maximum value can be passed in as either a `long` or `double` value:

```
DFEFix dfeFixMax(int num_bits, long max, SignMode sign_mode)
DFEFix dfeFixMax(int num_bits, double max, SignMode sign_mode)
```

For example, passing in a maximum of 127.0 in two's-complement mode returns a fixed-point type with an offset of -8:

```
DFEType fixedType = dfeFixMax(16, 127.0, SignMode.TWOSCOMPLEMENT);
```

### 2.4 Rounding Modes

There are three rounding modes available for fixed-point numbers in MaxCompiler: truncation, round-to-nearest and round-to-nearest-even.

These modes can be pushed and popped with in a Kernel to apply to the operations by a call to `push` and `pop`:

```
void pushRoundingMode(RoundingMode mode)
void popRoundingMode()
```

The rounding modes are defined in the `RoundingMode` enum:

```
public enum RoundingMode {
    TRUNCATE,
    TONEAR,
    TONEAREVEN;
}
```

The default rounding mode for fixed-point, and therefore integer, arithmetic in MaxCompiler is Round-to-Nearest.



This can result in different answers to a CPU, where results are truncated for integer arithmetic.

### 3 Numeric Exceptions

Arithmetic operations in Kernel designs have the capability of raising numeric exceptions. Enabling numeric exceptions is helpful during the design process to debug any numerical issues and can be used in a production application to detect run-time data errors. Numeric exceptions require extra logic on the Data Flow Engine, so are disabled by default.

Numeric exceptions are raised in similar circumstances to a CPU, but the Kernel always continues processing, raising a flag to indicate that a numeric exception has occurred. For floating-point numbers, the type of numeric exceptions that can be raised closely follow the IEEE 754 standard. For fixed-point numbers, overflow and divide-by-zero exceptions can be raised.

#### 3.1 Floating-Point Exceptions

MaxCompiler supports four of the five floating-point exceptions specified in the IEEE 754 standard. A brief explanation of each numeric exception type follows:

**Overflow** The result is larger than the maximum value that the chosen representation can contain.

**Underflow** The result is smaller than the minimum *normalized* value that the chosen representation can contain *but greater than zero*.



Because denormalized numbers are not supported in MaxCompiler, any result that would result in a denormalized number is set to an appropriately signed zero and an underflow exception is signaled.

**Divide-by-0** Division operation with a divisor of zero and a finite non-zero dividend.

**Invalid Operation** A real value cannot be returned (e.g.  $\sqrt{-1}$ ).



Because MaxCompiler treats all NaNs as quiet NaNs, no Invalid Operation exceptions are raised on any operation on a signaling NaN.

**Inexact** *Not supported.* This exception is raised if the result of an operation cannot be exactly represented, i.e. the result is *rounded*. Because most floating-point operations produce rounded results most of the time, the inexact exception is not usually considered to be an error.

[Table 3](#) shows the floating-point operations and the types of numeric exception that they can raise. [subsection 3.4](#) shows how to enable these numeric exceptions for each operation.

Operation	Operator	OFlow	UFlow	Div0	InvOp	Notes
ADD	+	✓			✓	
SUB	-	✓			✓	
MUL	*	✓	✓		✓	
DIV	/	✓	✓	✓	✓	1
ACCUMULATOR	makeAccumulator	✓			✓	
CAST_FROM_FIX						2
CAST_FROM_FLOAT		✓			✓	3
SQRT	sqrt		✓		✓	

<sup>1</sup> 0/0 raises an Invalid Operation exception instead of a Divide-by-Zero exception.

<sup>2</sup> Fixed-to-float casts cannot overflow or underflow.

<sup>3</sup> Applies to both float-to-float and float-to-fixed.

Table 3: Floating-point operations and the exceptions they can raise.

## 3.2 Fixed-Point Exceptions

MaxCompiler supports two exceptions for fixed-point operations:

**Overflow** The result is larger than the maximum value that the chosen representation can contain.

**Divide-by-Zero** Division operation with a divisor of zero.

Table 4 shows the fixed-point operations and the types of numeric exception that they can raise. subsection 3.4 shows how to enable these numeric exceptions for each operation.

Operation	Operator	OFlow	Div0	Notes
ADD	+	✓		
SUB	-	✓		
MUL	*	✓		
DIV	/	✓	✓	<sup>1</sup>
NEG	-	✓		<sup>2</sup>
ACCUMULATOR	makeAccumulator	✓		
SHIFT_LEFT	<<	✓		
CAST_FROM_FIX		✓		<sup>3</sup>

<sup>1</sup> Division of a non-zero dividend by a zero divider raises a Divide-by-Zero exception. This is different behavior from floating point, where a Invalid Operation exception may be raised (see [Table 3](#)).

<sup>2</sup> Only in the specific circumstance of negating the maximum negative value (e.g. -128 in a signed 8-bit integer).

<sup>3</sup> Applies to fixed-to-fixed only.

*Table 4:* Fixed-point operations and the exceptions they can raise.

### 3 Numeric Exceptions

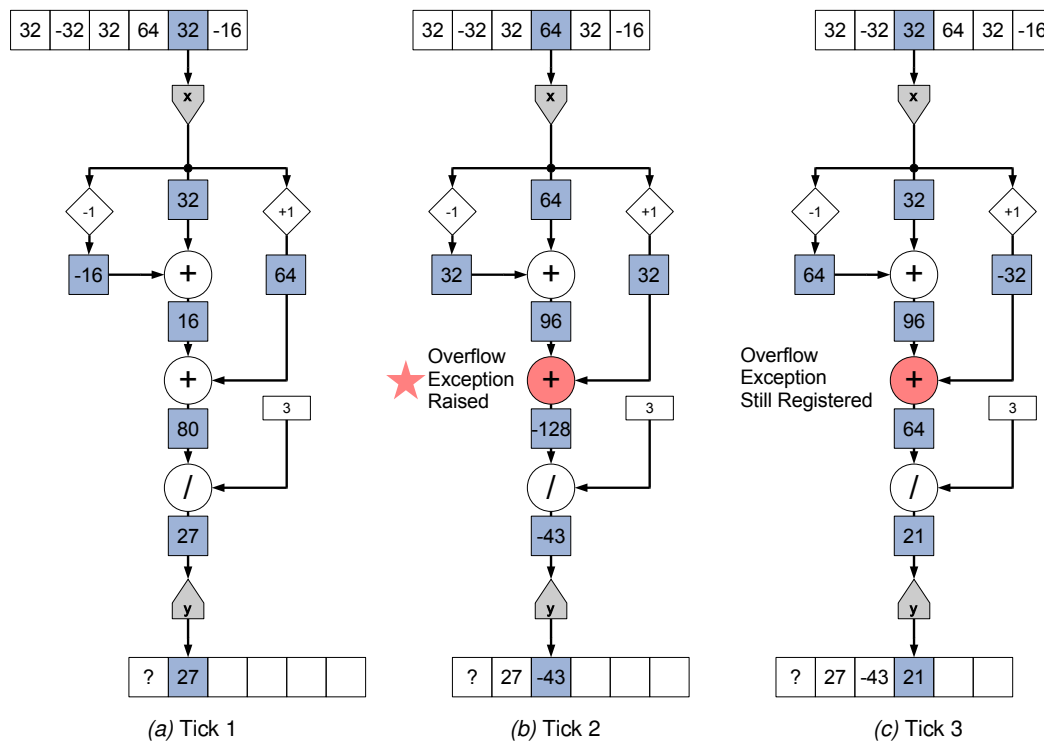


Figure 4: 3 Kernel ticks of the moving average filter showing an overflow exception being raised for 8-bit signed integer data.

### 3.3 Numeric Exception Behavior

The Kernel records which numeric exceptions have been raised for each operation during the Kernel execution using a separate flag for each numeric exception type. Once a numeric exception of a particular type has been raised for an operation, that flag can only be cleared when the device is reset and the Kernel restarted.

Example 1 performs a 3-tap moving average of an input stream using 8-bit signed integer data. The Kernel source code is shown in [Listing 1](#).

Figure 5 shows the Kernel graph for Example 1 over multiple ticks (with the boundary conditions removed for clarity). Once an overflow exception has been raised in tick 2, the overflow exception flag stays set until the Kernel is reset.

### 3.4 Enabling Numeric Exceptions

Numeric exceptions can be enabled for all or just part of a Kernel design. Two methods on the debug property of the Kernel class are available to enable and disable all numeric exceptions:

```
void debug.pushEnableNumericExceptions(boolean enable);
void debug.popEnableNumericExceptions();
```

Numeric exceptions are enabled for parts of the Kernel between calls to `pushEnabledExceptions(true)` and `popEnabledExceptions()`. The calls to push and pop the enabling of numeric exceptions behave as a stack during the compilation of the Kernel, which enables fine-grained control of the individual operations to have numeric exceptions enabled.

Exceptions for stream type	Notes
<code>EnableExceptionTypes.DFE_FIX_DFE_FLOAT</code>	Both fixed point and floating point
<code>EnableExceptionTypes.DFE_FIX</code>	Fixed point only
<code>EnableExceptionTypes.DFE_FLOAT</code>	Floating point only

Table 5: Exceptions for fixed-point and floating-point types.

Operation	Notes
<code>EnableExceptionOps.ADD</code>	
<code>EnableExceptionOps.SUB</code>	
<code>EnableExceptionOps.MUL</code>	
<code>EnableExceptionOps.DIV</code>	
<code>EnableExceptionOps.NEG</code>	(Arithmetic negation) Fixed point only
<code>EnableExceptionOps.ACCUMULATOR</code>	
<code>EnableExceptionOps.CAST_FROM_FIX</code>	Produces fixed-point exceptions for fix-to-fix casts and cannot produce exceptions for fix-to-float operations
<code>EnableExceptionOps.CAST_FROM_FLOAT</code>	Only produces floating-point exceptions
<code>EnableExceptionOps.SHIFT_LEFT</code>	Fixed point only
<code>EnableExceptionOps.SQRT</code>	Floating point only
<code>EnableExceptionOps.ALL</code>	All the above operations

Table 6: Operations that can raise exceptions.

In Example 1, numeric exceptions are enabled for the section of code that does the calculation:

```

39     debug.pushEnableNumericExceptions(true);
40
41     DFEVar sum = prev+x+next;
42     DFEVar result = sum/divisor;
43
44     debug.popEnableNumericExceptions();


```

You can also enable and disable exceptions for specific data types and operations:

```
void debug.pushEnableNumericExceptions(boolean enable, EnableExceptionTypes type, EnableExceptionOps... ops)
```

This version of the method takes an `EnableExceptionTypes` enumeration instance to specify for which stream data type to disable the exceptions: floating-point, fixed-point or both (see [Table 5](#)). The varargs argument takes a list of one or more `EnableExceptionOps` enumeration instances, the possible values for which are shown in [Table 6](#)

These calls to push and pop the enabling of exceptions for certain data types and operations behave as a stack during compilation, with each push overriding the setting for all of the data types and operations that it specifies. The exceptions that these operations can raise are shown in [Table 3](#) and [Table 4](#).

 Enabling numeric exceptions for a section of code is a compile-time option that determines whether the logic for numeric exceptions is built into the final design.



### 3 Numeric Exceptions

---

For example, the following code enables all numeric exceptions in line 4, then disables exceptions for the cast from floating point in line 9:

```

1  DFEVar x = io.input("x", dfeFloat(8,24));
2
3  // Enable all numeric exceptions
4  debug.pushEnableNumericExceptions(true);
5
6  DFEVar y = x*x;
7
8  // Disable numeric exceptions for float-to-fix casts
9  debug.pushEnableNumericExceptions(false, EnableExceptionTypes.DFE_FIX_DFE_FLOAT, EnableExceptionOps.CAST_FROM_FLOAT);
10
11 y = y.cast(dfeOffsetFix(24,8, SignMode.TWOSCOMPLEMENT));
12
13 debug.popEnableNumericExceptions();
14 debug.popEnableNumericExceptions();

```

Any overflows from the cast on line 11 from floating point to fixed point are therefore ignored.

#### 3.5 Reading Numeric Exception Data

Each operation that has exceptions enabled, keeps a record of which exceptions were raised during the execution of the Kernel. The Kernel can be instructed, via a run-time configuration setting, to save this information into a file for analysis. For the rest of this document, we refer to this file as the "binary cache" file.

To capture exceptions raised, additional settings must be included in the \$SLIC\_CONF environment setting. If you are a bash user, these are (*without line breaks*):

```
[user@machine]$ export SLIC_CONF=$SLIC_CONF;default_maxdebug_mode=2;
                        default_maxdebug_name=MyAction
```

For a csh user, these are:

```
[user@machine]$ setenv SLIC_CONF $SLIC_CONF;default_maxdebug_mode=2;
                        default_maxdebug_name=MyAction
```

The default\_maxdebug\_mode setting enables the collection of the debug data, including exceptions data; and the default\_maxdebug\_name setting defines part of the generated name of the binary cache file that is created. The complete name of the binary cache file is formed by the concatenation of "maxdebug-", the default\_maxdebug\_name environment setting, a time-stamp, and a device id, for example:

```
maxdebug_MyAction.2012-10-15_15-04-46.dev0.1831731826237
```

This long name is necessary to ensure that data from multiple runs on multiple DFEs is not overwritten. For simplicity in the examples used hereafter in this document, we assume that we have renamed this binary cache file to a shorter name:

```
[user@machine]$ mv
    maxdebug_MyAction.2012-10-15_15-04-46.dev0.1831731826237
    maxdebug_MyAction.out
```

Incidentally, the binary cache file also contains debug information, and can be queried using MaxDebug with its "-f" option.

The MaxQueryExceptions utility is used to read the binary cache file and report on exceptions.

```
[user@machine]$ maxQueryExceptions -h
Usage:      maxQueryExceptions [options]
```

where:

- b, --binfile <maxdebugfile> \*\*\* MANDATORY ARGUMENT \*\*\*  
specifies the binary file which is written when maxdebug is enabled in the SLiC actions; this contains the raw values of any numerical exceptions.
- m, --maxfile <maxfile> \*\*\* MANDATORY ARGUMENT \*\*\*  
specifies the maxfile containing the design.

Input options:

- k, --kernel <kernelname>[,<kernelname2>[,<kernelname3 >...]]  
name of kernel, or list of kernels, to be queried.  
If not specified, all kernels in the design are queried.
- x, --excludemask <dbgfile>[,<dbgfile2>[,<dbgfile3 >...]]  
set one or more binary maxdebug files to use for masking the exceptions register. These binary files must have all been generated from the same maxfile, so that their exceptions registers have the same size, address, and node ID associations as the target binary file specified with the '-b' argument.  
If more than one exclude file is supplied, the effect is cumulative, and the bit-masks for each kernel are OR-ed together. Exceptions corresponding to set bits in the exclude-mask will not be printed.

Output options:

- i, --id <nodeid>[,<nodeid2>[,<nodeid3 >...]]  
show node details for list of node IDs.
- d, --dump  
output in "dump" format; the numerical exceptions are printed out as EXCEPTION\_VALUE macros.
- s, --show  
output in "show" format; the numerical exceptions are printed out in tabular form.
- n, --nodeinfo  
show node details where exception occurred.
- c, --count  
get count of the number of set bits in the exceptions register.  
Any exclude-masks supplied are applied before the bits are counted.
- g, --getmask  
get contents of numerical exceptions as a bit-mask string;  
Any exclude-masks supplied are applied beforehand.

Three arguments are mandatory:

- the maxfile used to build the design;
- the binary cache file created as described above; and
- one of the output options, to select how any exceptions are presented. More than one output option can be used: the different format outputs are listed one after another.

After running the example, we run the `maxQueryExceptions` utility from a command prompt, using the binary cache file created, and the location of the `.max` file:

```
[user@machine]$ maxQueryExceptions -s -b maxdebug_MyAction.out
-m RunRules/Simulation/maxfiles/MovingAverageExceptionsHostSim.max
```

The "`maxQueryExceptions -s`" output option produces a table of results, such as:

#	node ID	exceptions	kernel
#	-----	-----	-----
	13	DIV	Kernel1
	55	OVR	Kernel1
	13	DIV	Kernel2
	55	OVR	Kernel2

Example 1 streams data through the moving-average Kernel, and then queries the binary cache file for any exceptions.

The output for this example is shown in [Listing 2](#), and the corresponding `maxQueryExceptions` output is shown in [Listing 3](#). On line 4 of [Listing 2](#), we can see that the third output value is `-43`, when the correct answer is `43` (using the default round-to-nearest rounding mode). Line 3 of [Listing 3](#) shows that an overflow (OVR) has occurred at node ID 36 (node IDs are covered in [subsection 3.6](#)).

The exceptions in a binary cache file can be used to mask the exceptions in a binary cache file from another run using the same `.max` file, and this allows differences in exceptions from separate runs to be seen more easily.

```
[user@machine]$ maxQueryExceptions -s -b maxdebug_MyAction.out
-m RunRules/Simulation/maxfiles/MovingAverageExceptionsHostSim.max
-x maxdebug_MyAction_original.out
```

### 3.6 Mapping Numeric Exceptions to the Kernel Source

The output of `maxQueryExceptions` displays the exceptions by **node ID** (see line 1 of [Listing 3](#)), which is the ID of the operation in the Kernel graph. For anything but a small example, the node ID alone makes it difficult to work out which operation in the Kernel produced the numeric exception.

The output option "`-n`" of `maxQueryExceptions` lists the node IDs along with their stack trace from the Kernel compilation, which shows the line in the source code that created the node in the graph, as shown in [Listing 4](#).

Line 10 shows the line from the Kernel source that created this operation. Line 6 shows the type of operation that this node performs, which is useful when there are multiple nodes instantiated from a single line of code. In this case, a `NodeCondTriAdd` operation means that MaxCompiler has created a three-input add block for the line:

```
41 DFEVar sum = prev+x+next;
```

### 3 Numeric Exceptions

---

From this information, we conclude that it is this add operation that has overflowed the 8-bit signed integer data storage.

★ When debugging numeric exceptions in a complex expression, breaking the expression into multiple expressions over many lines can simplify debugging because the stack trace for each node ID then refers to a different line in the source code.

★ There is not a one-to-one mapping between operations declared in the Java source code and nodes in the graph output, due to transformations and optimizations performed by MaxCompiler. For example, breaking  $a=x+y+z$  into multiple expressions may still result in a single tri-input adder.

### 3 Numeric Exceptions

*Listing 1: Moving average Kernel with numeric exceptions enabled (MovingAverageExceptionsKernel.maxj).*

```

1  /**
2   * Document: Kernel Numerics Tutorial (maxcompiler-kernel-numeric.pdf)>
3   * Chapter: 3      Example: 1      Name: Moving Average Exceptions
4   * MaxFile name: MovingAverageExceptions
5   * Summary:
6   *   A kernel that calculates a moving average and tracks numeric exceptions.
7   */
8   package movingaverageexceptions;
9
10  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
11  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
12  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
13
14  public class MovingAverageExceptionsKernel extends Kernel {
15      public MovingAverageExceptionsKernel(KernelParameters parameters, int N) {
16          super(parameters);
17
18          // Input
19          DFEVar x = io.input("x", dfelnt(8));
20
21          // Data
22          DFEVar x_prev = stream.offset(x, -1);
23          DFEVar x_next = stream.offset(x, 1);
24
25          // Control
26          DFEVar cnt = control.count.simpleCounter(32, N);
27
28          DFEVar sel_n1 = cnt > 0;
29          DFEVar sel_nu = cnt < N-1;
30
31          DFEVar sel_m = sel_n1 & sel_nu;
32
33          DFEVar prev = sel_n1 ? x_prev : 0;
34          DFEVar next = sel_nu ? x_next : 0;
35
36          DFEVar divisor = sel_m ? constant.var(dfelnt(8), 3) : 2;
37
38          // Calculation
39          debug.pushEnableNumericExceptions(true);
40
41          DFEVar sum = prev+x+next;
42          DFEVar result = sum/divisor;
43
44          debug.popEnableNumericExceptions();
45
46          // Output
47          io.output("y", result, result.getType());
48      }
49  }

```

*Listing 2: Standard output for Example 1.*

```

1 Running DFE
2   8
3  27
4 -43
5  21
6  11
7   5
8  11
9  -5
10  0
11  27
12 -43
13  21
14  11
15   5
16  11
17   0

```

*Listing 3: Numeric exception output for Example 1.*

1	# node ID	exceptions	kernel
2	# -----	-----	-----
3	44	OVR	MovingAverageExceptionsKernel

*Listing 4: maxQueryExceptions output for Example 1.*

```

1 [user@machine]$ maxQueryExceptions -n -b maxdebug_MyAction.out
2   -m RunRules/Simulation/maxfiles/MovingAverageExceptionsHostSim.max
3
4   node ID: 44
5   design: MovingAverageExceptionsKernel
6   type: NodeCondTriAdd
7   value: 3 +/-0
8 stack trace:
9   com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar.add(DFEVar.
10      java:775)
11 movingaverageexceptions.MovingAverageExceptionsKernel.<init>(  
    MovingAverageExceptionsKernel.maxj:41)  
movingaverageexceptions.MovingAverageExceptionsManager.main(  
    MovingAverageExceptionsManager.maxj:18)

```

### 3.7 Conditionally Enabling Numeric Exceptions

You can suppress numeric exceptions for particular operations in the Kernel based on a run-time condition. A Boolean stream is used to enable (1) or disable (0) numeric exceptions. This can help get meaningful exception data out of a design when various parts of the Kernel are active at different times.

The methods `pushNumericExceptionCondition` and `popNumericExceptionCondition` delimit the section of the Kernel where numeric exceptions are to be enabled when the Boolean stream argument `condition` to `pushNumericExceptionCondition` is true:

```
void debug.pushNumericExceptionCondition(DFEVar condition)
void debug.popNumericExceptionCondition()
```

These calls to push and pop the conditional enabling of numeric exceptions for sections of the Kernel behave as a stack during compilation, with each condition stream being *ANDed* with the previously pushed streams for the subsequent section of code, up to the corresponding pop.

Example 2 shows conditional numeric exceptions in use. The full Kernel source is shown in [Listing 5](#).

The Kernel takes two 8-bit signed integer streams as input either adds or subtracts them, according to the following pseudo-code:

```
if ((x >= 0 and y <= 0) or (x < 0 and y > 0))
    result = x + y
else
    result = x - y
```

In a streaming implementation, both the add and the subtract are performed every Kernel tick and the result selected from on or the other depending on the condition. The operator for which the result is discarded, however, can still produce a numeric exception, which we want to suppress as it has no bearing on the correctness of our output.

We first enable numeric exceptions:

```
24    debug.pushEnableNumericExceptions(true);
```

We then calculate the condition for whether we want to perform an add or a subtract:

```
26    DFEVar doAdd = ((x >= 0) & (y <= 0)) | ((x < 0) & (y > 0));
```

Numeric exceptions are then conditionally disabled for the inactive operator using the condition stream:

```
28    debug.pushNumericExceptionCondition(doAdd);
29    DFEVar add=x+y;
30    debug.popNumericExceptionCondition();
31    debug.pushNumericExceptionCondition(!doAdd);
32    DFEVar sub=x-y;
33    debug.popNumericExceptionCondition();
```

Finally, the same condition is used to select the output of the add or the subtract:

```
35    DFEVar result = doAdd ? add : sub;
36
37    debug.popEnableNumericExceptions();
```

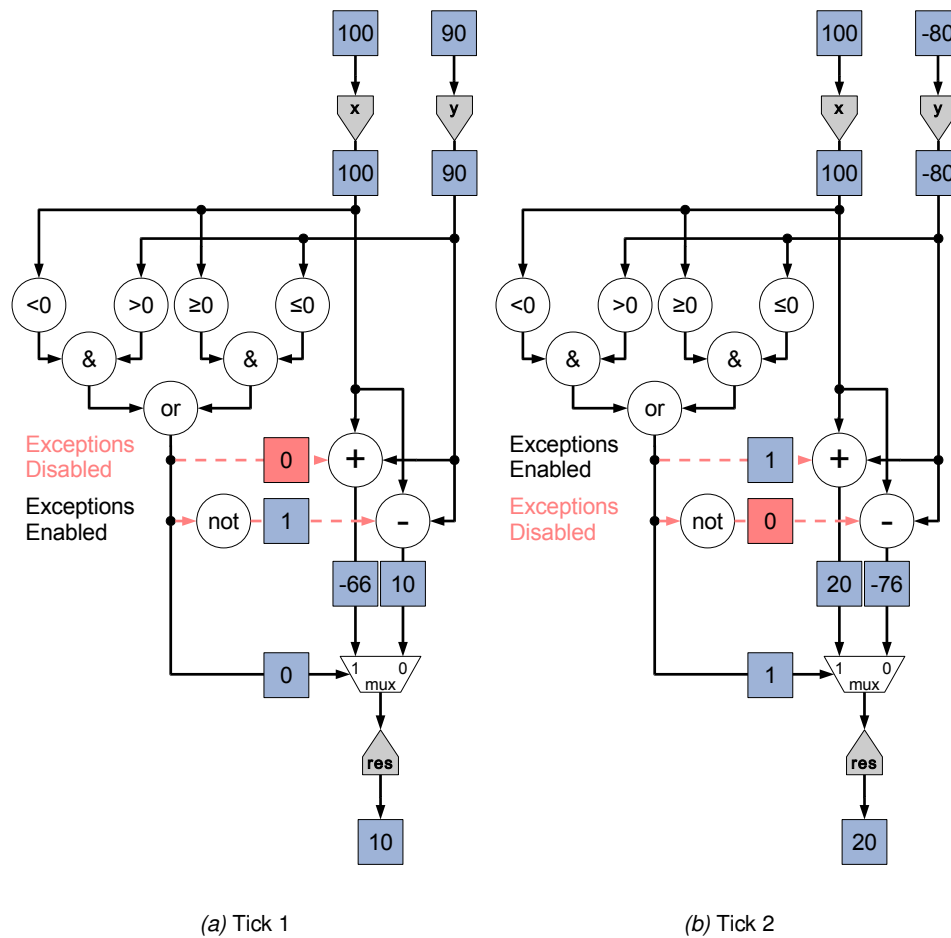


Figure 5: 2 Kernel ticks of Example 2 showing conditional numeric exceptions being used for 8-bit signed integer data.

★ Conditionally disabling numeric exceptions in the Kernel stops the exceptions from being recorded at certain times: this is different to simply masking the display of numeric exceptions in the CPU code using `max_show_numeric_ex_masked`.



*Listing 5: Moving average Kernel showing conditional numeric exceptions (ConditionalExceptionsKernel.maxj).*

```

1  /**
2   * Document: Kernel Numerics Tutorial (maxcompiler-kernel-numerics.pdf)>
3   * Chapter: 3      Example: 2      Name: Conditional Exceptions
4   * MaxFile name: ConditionalExceptions
5   * Summary:
6   *   A kernel that calculates the element-wise difference between the
7   *   modulus of the two inputs. Results can be both positive and negative.
8   */
9  package conditionalexceptions;
10
11  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
14
15  public class ConditionalExceptionsKernel extends Kernel {
16      public ConditionalExceptionsKernel(KernelParameters parameters, int N) {
17          super(parameters);
18
19          // Input
20          DFEVar x = io.input("x", dfelnt(8));
21          DFEVar y = io.input("y", dfelnt(8));
22
23          // Calculation
24          debug.pushEnableNumericExceptions(true);
25
26          DFEVar doAdd = ((x >= 0) & (y <= 0)) | ((x < 0) & (y > 0));
27
28          debug.pushNumericExceptionCondition(doAdd);
29          DFEVar add=x+y;
30          debug.popNumericExceptionCondition();
31          debug.pushNumericExceptionCondition(~doAdd);
32          DFEVar sub=x-y;
33          debug.popNumericExceptionCondition();
34
35          DFEVar result = doAdd ? add : sub;
36
37          debug.popEnableNumericExceptions();
38
39          // Output
40          io.output("result", result, result.getType());
41      }
42  }

```