LogiCORE IP Floating-Point Operator Bit Accurate C Model

User Guide

UG812 (v1.0) June 22, 2011

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/22/11	1.0	Initial Xilinx release.

Table of Contents

XILINX®

Chapter 1

Introduction

The Xilinx® LogiCORE™ IP Floating-Point Operator v6.0 core bit accurate C model is a self-contained, linkable, shared library that models the functionality of this core with finite precision arithmetic. This model provides a bit accurate representation of the various modes of the Floating-Point Operator v6.0 core, and it is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

Features

- Bit accurate with Floating-Point Operator v6.0 core
- Available for 32-bit and 64-bit Linux platforms
- Available for 32-bit and 64-bit Windows platforms
- Supports all features of the Floating-Point Operator v6.0 core
- Designed for integration into a larger system model
- Example C code showing how to use the C model functions

Overview

This user guide provides information about the Xilinx LogiCORE IP Floating-Point Operator v6.0 bit accurate C model for 32-bit and 64-bit Linux, and 32-bit and 64-bit Windows platforms.

The model consists of a set of C functions that reside in a shared library. Example C code is provided to demonstrate how these functions form the interface to the C model. Full details of this interface are given in Chapter 3 of this document.

The model is bit accurate but not cycle-accurate; it performs exactly the same operations as the core. However, it does not model the core's latency or its interface signals.

Additional Core Resources

For detailed information and updates about the Floating-Point Operator v6.0 core, see the following documents:

- Floating-Point Operator v6.0 Product Specification (DS816)
- Floating-Point Operator v6.0 Release Notes

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the Floating-Point Operator v6.0 core.

Xilinx provides technical support for use of this product as described in *Floating-Point Operator Bit Accurate C Model User Guide (UG812)* and the *LogiCORE IP Floating-Point Operator v6.0 Product Specification (DS816)*. Xilinx cannot guarantee functionality or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the Floating-Point Operator v6.0 core and the accompanying documentation.

Floating-Point Operator v6.0 Bit Accurate C Model and IP Core

For comments or suggestions about the Floating-Point Operator v6.0 core and bit accurate C model, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Documentation

For comments or suggestions about the Floating-Point Operator v6.0 core documentation, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

XILINX®

Chapter 2

User Instructions

Unpacking and Model Contents

There are separate ZIP files containing all the files necessary for use with a specific computing platform. Each ZIP file contains:

- The C model shared library
- Multiple Precision Integers and Rationals (MPIR) [Ref 1] and Multiple Precision Floating-point Reliable (MPFR) [Ref 2] shared libraries, header files and source code
- The C model header file
- The example code showing customers how to call the C model
- Documentation

Note: The C model uses MPIR and MPFR libraries, which are provided in the ZIP files. MPIR is an interface-compatible version of the GNU Multiple Precision (GMP) [Ref 3] library, with greater support for Windows platforms. MPIR has been compiled using its GMP compatibility option, so the MPIR library and header file use GMP file names. MPFR uses GMP, but here has been configured to use MPIR instead.

Table 2-1: **Example C Model ZIP File Contents - Linux**

File	Description
floating_point_v6_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v6_0_bitacc_cmodel.so	Model shared object library
libgmp.so.7	MPIR library, used by the C model
libmpfr.so.4	MPFR library, used by the C model
gmp.h	MPIR header file, used by the C model
mpfr.h	MPFR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
allfns.c	Detailed example C code showing how to call every C model function
README.txt	Release notes
ug812_floating_point_cmodel.pdf	This user guide
mpir-2.2.1.tar.bz2	MPIR source code
mpfr-3.0.1.tar.bz2	MPFR source code

Table 2-2: **Example C Model ZIP File Contents - Windows**

File	Description
floating_point_v6_0_bitacc_cmodel.h	Header file which defines the C model API
libIp_floating_point_v6_0_bitacc_cmodel.dll	Model dynamically linked library
libIp_floating_point_v6_0_bitacc_cmodel.lib	Model .lib file for compiling
libgmp.dll	MPIR library, used by the C model
libgmp.lib	MPIR .lib file for compiling
libmpfr.dll	MPFR library, used by the C model
libmpfr.lib	MPFR .lib file for compiling
gmp.h	MPIR header file, used by the C model
mpfr.h	MPFR header file, used by the C model
run_bitacc_cmodel.c	Example program for calling the C model
allfns.c	Detailed example C code showing how to call every C model function
README.txt	Release notes
ug812_floating_point_cmodel.pdf	This user guide
mpir-2.2.1.tar.bz2	MPIR source code
mpfr-3.0.1.tar.bz2	MPFR source code
mpfr.build.vc9.zip	Microsoft Visual Studio 2008 project files for compiling MPFR on Windows
mpfr.build.vc10.zip	Microsoft Visual Studio 2010 project files for compiling MPFR on Windows
mpfr_nt_stdint.h	Header file to enable some MPFR functions when compiling MPFR on Windows

Installation

Linux

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_floating_point_v6_0_bitacc_cmodel.so`, `libgmp.so.7` and `libmpfr.so.4` files reside is included in the path of the environment variable LD_LIBRARY_PATH.

Windows

- Unpack the contents of the ZIP file.
- Ensure that the directory where the `libIp_floating_point_v6_0_bitacc_cmodel.dll`, `libgmp.dll` and `libmpfr.dll` files reside is
  a. included in the path of the environment variable PATH or
  b. the directory in which the executable that calls the C model is run.

XILINX®

Chapter 3

Floating-Point Operator v6.0 Bit Accurate C Model

C Model Interface

The Floating-Point Operator C model has a C function based Application Programming Interface (API), which is very similar to the APIs of other floating-point arithmetic libraries MPIR (Multiple Precision Integers and Rationals) and MPFR (GNU Multiple Precision Floating-point Reliable library). The C model uses these libraries internally and provides functions to convert between their data types.

Note: MPIR [Ref 1] and MPFR [Ref 2] are free, open source software libraries, distributed under the GNU Lesser General Public License. The source code and a compiled version of each library is provided with the C model. MPIR is a compatible alternative to GMP (GNU Multiple Precision Arithmetic) [Ref 3] that provides greater support for Windows platforms. MPIR and GMP can be used interchangeably.

Two example C files, `run_bitacc_cmodel.c` and `allfns.c`, are included, that demonstrate how to call the C model. See these files for examples of using the interface described in the following sections.

The Application Programming Interface (API) of the C model is defined in the header file `floating_point_v6_0_bitacc_cmodel.h`. The interface consists of data structures and functions as described in the following sections.

Data Types

The C types defined for the Floating-Point Operator C model are shown in Table 3-1.

Table 3-1: **Floating-Point Operator C Model Data Types**

Name	Type	Description
xip_fpo_prec_t	long	Precision of mantissa or exponent (bits)
xip_fpo_sign_t	int	Sign bit of a floating-point number
xip_fpo_exp_t	long	Exponent of a floating-point number
xip_fpo_t	struct[1]	Custom precision floating-point number (internally defined as a one-element array of a structure)
xip_fpo_fix_t	struct[1]	Custom precision fixed-point number (internally defined as a one-element array of a structure)
xip_fpo_ptr	struct *	Pointer to underlying custom precision floating-point struct. Equivalent to `xip_fpo_t` but easier to use in certain situations (for example, terminator in `xip_fpo_inits2` function).

Table 3-1: **Floating-Point Operator C Model Data Types *(Cont'd)***

Name	Type	Description
xip_fpo_fix_ptr	struct *	Pointer to underlying custom precision fixed-point struct. Equivalent to `xip_fpo_fix_t` but easier to use in certain situations (for example, terminator in `xip_fpo_fix_inits2` function).
xip_fpo_exc_t	int	Bitwise flags which when set indicate exceptions that occurred during an operation: bit 0: underflow bit 1: overflow bit 2: invalid operation bit 3: divide by zero bit 4: operation not supported by Floating-Point Operator v6.0 core (for example, add with different precision operands)

`xip_fpo_prec_t` is used for initializing variables of type `xip_fpo_t` and `xip_fpo_fix_t`.

`xip_fpo_prec_t` and `xip_fpo_exp_t` are of type `long` for compatibility with MPFR, not because they need a greater numerical range than provided by `int`.

The Floating-Point Operator C model functions use `xip_fpo_t` and `xip_fpo_fix_t` for input and output variables. Users should use these types for all custom precision floating-point and fixed-point variables. Defining this type as a one-element array of the underlying struct means that when a user declares a variable of this type, the memory for the struct members is automatically allocated, and the user can pass the variable as-is to functions with no need to add a * to pass a pointer, and it is automatically passed by reference. This is the same method as used by MPIR [Ref 1] and MPFR [Ref 2].

Note that `xip_fpo_t` is an IEEE-754 compatible floating-point type, except that signalling NaNs and denormalized numbers are not supported. If a signalling NaN is stored in an `xip_fpo_t` variable, the value becomes a quiet NaN. Similarly, denormalized numbers are converted to zero (with an underflow exception, if appropriate). See the Floating-Point Operator v6.0 Product Specification (DS816) for more details.

`xip_fpo_exc_t` is the return value type of most functions.

The C model API also provides versions of its operation functions for single and double precision, using standard C data types `float` and `double` respectively. This provides an easy use model for applications that do not require custom precision.

Functions

There are a number of C model functions accessible to the user.

Information Functions

The Floating-Point Operator C model information functions are shown in Table 3-2.

Table 3-2: **Floating-Point Operator C Model Information Functions**

Name	Return	Arguments	Description
xip_fpo_get_version	const char *	void	Return the Floating-Point Operator C model version, as a null-terminated string. For v6.0 this is "6.0".

Initialization Functions

The Floating-Point Operator C model initialization functions are shown in Table 3-3. Most functions have variants to handle floating-point and fixed-point variables.

Table 3-3: **Floating-Point Operator C Model Initialization Functions**

Name	Return	Arguments	Description
xip_fpo_init2	void	xip_fpo_t x, xip_fpo_prec_t exp, xip_fpo_prec_t mant	Initialize floating-point variable *x*, set its exponent precision to *exp*, its mantissa precision to *mant*, and its value to NaN.
xip_fpo_fix_init2	void	xip_fpo_fix_t x, xip_fpo_prec_t i, xip_fpo_prec_t frac	Initialize fixed-point variable *x*, set its integer precision to *i*, its fraction precision to *frac*, and its value to zero.
xip_fpo_inits2	void	xip_fpo_prec_t exp, xip_fpo_prec_t mant, xip_fpo_t x, ...	Initialize all `xip_fpo_t` variables pointed to by the argument list, set their exponent precision to *exp*, their mantissa precision to *mant*, and their value to NaN. The last item in the list must be a null pointer of type `xip_fpo_t` (or equivalently `xip_fpo_ptr`).
xip_fpo_fix_inits2	void	xip_fpo_prec_t i, xip_fpo_prec_t frac, xip_fpo_fix_t x, ...	Initialize all `xip_fpo_fix_t` variables pointed to by the argument list, set their integer precision to *i*, their fraction precision to *frac*, and their value to zero. The last item in the list must be a null pointer of type `xip_fpo_fix_t` (or equivalently `xip_fpo_fix_ptr`).
xip_fpo_clear	void	xip_fpo_t x	Free the memory used by *x*.
xip_fpo_fix_clear	void	xip_fpo_fix_t x	Free the memory used by *x*.
xip_fpo_clears	void	xip_fpo_t x, ...	Free the memory used by all `xip_fpo_t` variables pointed to by the argument list. The last item in the list must be a null pointer of type `xip_fpo_t` (or equivalently `xip_fpo_ptr`).
xip_fpo_fix_clears	void	xip_fpo_fix_t x, ...	Free the memory used by all `xip_fpo_fix_t` variables pointed to by the argument list. The last item in the list must be a null pointer of type `xip_fpo_fix_t` (or equivalently `xip_fpo_fix_ptr`).
xip_fpo_set_prec	void	xip_fpo_t x, xip_fpo_prec_t exp, xip_fpo_prec_t mant	Reset *x* to an exponent precision of *exp*, a mantissa precision of *mant*, and set its value to NaN. The previous value of *x* is lost.
xip_fpo_fix_set_prec	void	xip_fpo_fix_t x, xip_fpo_prec_t i, xip_fpo_prec_t frac	Reset *x* to an integer precision of *i*, a fraction precision of *frac*, and set its value to zero. The previous value of *x* is lost.
xip_fpo_get_prec_mant	xip_fpo_prec_t	xip_fpo_t x	Return the mantissa precision (in bits) of *x*.
xip_fpo_get_prec_exp	xip_fpo_prec_t	xip_fpo_t x	Return the exponent precision (in bits) of *x*.
xip_fpo_fix_get_prec_frac	xip_fpo_prec_t	xip_fpo_fix_t x	Return the fraction precision (in bits) of *x*.
xip_fpo_fix_get_prec_int	xip_fpo_prec_t	xip_fpo_fix_t x	Return the integer precision (in bits) of *x*.

A floating-point number has a minimum exponent required to support normalization:

$$\text{minimum exponent width} = ceil(\log_2(\text{fraction width} + 3)) + 1$$

If the exponent width specified for `xip_fpo_init2` or `xip_fpo_set_prec` for initializing or resetting a floating-point variable is too small, it is internally increased to the minimum permitted width.

A variable should be initialized only once, or be cleared using `xip_fpo_clear` between initializations. To change the precision of a variable that has already been initialized, use `xip_fpo_set_prec`.

An example of initializing and clearing floating-point variables is shown:

```
xip_fpo_t x, y, z;
xip_fpo_init2 (x, 11, 53);     // double precision
xip_fpo_inits2 (7, 17, y, z, (xip_fpo_ptr) 0);  // custom precision
// perform operations
xip_fpo_set_prec (8, 24, y);   // change to single precision
// more operations
xip_fpo_clears (x, y, z, (xip_fpo_ptr) 0);
```

Assignment Functions

The Floating-Point Operator C model assignment functions are shown in Table 3-4. Most functions have variants to handle both floating-point and fixed-point variables. Functions are provided for assigning Floating-Point Operator C model variables from MPIR and MPFR variables for ease of use alongside these existing libraries.

Table 3-4: **Floating-Point Operator C Model Assignment Functions**

Name	Return	Arguments	Description
xip_fpo_set	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set the value of *rop* to *op*.(1)
xip_fpo_fix_set	xip_fpo_exc_t	xip_fpo_fix_t rop, xip_fpo_fix_t op	
xip_fpo_set_ui	xip_fpo_exc_t	xip_fpo_t rop, unsigned long op	
xip_fpo_fix_set_ui	xip_fpo_exc_t	xip_fpo_fix_t rop, unsigned long op	
xip_fpo_set_si	xip_fpo_exc_t	xip_fpo_t rop, signed long op	
xip_fpo_fix_set_si	xip_fpo_exc_t	xip_fpo_fix_t rop, signed long op	
xip_fpo_set_uj	xip_fpo_exc_t	xip_fpo_t rop, uintmax_t op	
xip_fpo_fix_set_uj	xip_fpo_exc_t	xip_fpo_fix_t rop, uintmax_t op	
xip_fpo_set_sj	xip_fpo_exc_t	xip_fpo_t rop, intmax_t op	
xip_fpo_fix_set_sj	xip_fpo_exc_t	xip_fpo_fix_t rop, intmax_t op	
xip_fpo_set_flt	xip_fpo_exc_t	xip_fpo_t rop, float op	
xip_fpo_fix_set_flt	xip_fpo_exc_t	xip_fpo_fix_t rop, float op	
xip_fpo_set_d	xip_fpo_exc_t	xip_fpo_t rop, double op	
xip_fpo_fix_set_d	xip_fpo_exc_t	xip_fpo_fix_t rop, double op	
xip_fpo_set_z	xip_fpo_exc_t	xip_fpo_t rop, mpz_t op	Set the value of *rop* to the value of GMP/MPIR integer *op*.(1)
xip_fpo_fix_set_z	xip_fpo_exc_t	xip_fpo_fix_t rop, mpz_t op	
xip_fpo_set_q	xip_fpo_exc_t	xip_fpo_t rop, mpq_t op	Set the value of *rop* to the value of GMP/MPIR rational number *op*.(1)
xip_fpo_fix_set_q	xip_fpo_exc_t	xip_fpo_fix_t rop, mpq_t op	
xip_fpo_set_f	xip_fpo_exc_t	xip_fpo_t rop, mpf_t op	Set the value of *rop* to the value of GMP/MPIR floating-point number *op*.(1)
xip_fpo_fix_set_f	xip_fpo_exc_t	xip_fpo_fix_t rop, mpf_t op	

Table 3-4: **Floating-Point Operator C Model Assignment Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_set_fr	xip_fpo_exc_t	xip_fpo_t rop, mpfr_t op	Set the value of *rop* to the value of MPFR floating-point number *op*.(1)
xip_fpo_fix_set_fr	xip_fpo_exc_t	xip_fpo_fix_t rop, mpfr_t op	
xip_fpo_set_ui_2exp	xip_fpo_exc_t	xip_fpo_t rop, unsigned long op, xip_fpo_exp_t e	Set the value of *rop* to *op* multiplied by two to the power of *e*.(1)
xip_fpo_set_si_2exp	xip_fpo_exc_t	xip_fpo_t rop, signed long op, xip_fpo_exp_t e	
xip_fpo_set_uj_2exp	xip_fpo_exc_t	xip_fpo_t rop, uintmax_t op, intmax_t e	
xip_fpo_set_sj_2exp	xip_fpo_exc_t	xip_fpo_t rop, intmax_t op, intmax_te	
xip_fpo_set_str	xip_fpo_exc_t	xip_fpo_t rop, const char *s, int base	Set the value of *rop* to the string in *s* which is in the base *base*. See xip_fpo_set_str and xip_fpo_fix_set_str for details.(1)
xip_fpo_fix_set_str	xip_fpo_exc_t	xip_fpo_fix_t rop, const char *s, int base	
xip_fpo_set_nan	void	xip_fpo_t x	Set the value of *x* to NaN.
xip_fpo_set_inf	void	xip_fpo_t x, int sign	Set the value of *x* to plus infinity if *sign* is non-negative, minus infinity otherwise.
xip_fpo_set_zero	void	xip_fpo_t x, int sign	Set the value of *x* to plus zero if *sign* is non-negative, minus zero otherwise.

Notes:

1. Any exceptions that occur are signalled in the return value.
When assigning to a fixed-point variable, if overflow occurs, the result is saturated and the return value is the largest representable fixed-point number of the correct sign. Converting a NaN returns the most negative representable fixed-point number and the invalid operation exception is signalled in the return value.

xip_fpo_set_str and xip_fpo_fix_set_str

The functions `xip_fpo_set_str` and `xip_fpo_fix_set_str` take a string argument (actually const char *) and an integer base. They have the same usage as the MPFR function `mpfr_set_str`.

The base is a value between 2 and 62 or zero. The string is a representation of numeric data to be read and stored in the floating-point variable. The whole string must represent a valid floating-point number.

The form of numeric data is a non-empty sequence of significand digits with an optional decimal point, and an optional exponent consisting of an exponent prefix followed by an optional sign and a non-empty sequence of decimal digits. A significand digit is either a decimal digit or a Latin letter (62 possible characters), with A = 10, B = 11, ..., Z = 35; case is

ignored in bases less or equal to 36, in bases larger than 36, a = 36, b = 37, ..., z = 61. The value of a significand digit must be strictly less than the base. The decimal point can be either the one defined by the current locale or the period (the first one is accepted for consistency with the C standard and the practice, the second one is accepted to allow the programmer to provide numbers from strings in a way that does not depend on the current locale). The exponent prefix can be e or E for bases up to 10, or @ in any base; it indicates a multiplication by a power of the base. In bases 2 and 16, the exponent prefix can also be p or P, in which case the exponent, called binary exponent, indicates a multiplication by a power of 2 instead of the base (there is a difference only for base 16); in base 16 for example 1p2 represents 4 whereas `1@2` represents 256.

If the argument *base* is 0, then the base is automatically detected as follows. If the significand starts with 0b or 0B, base 2 is assumed. If the significand starts with 0x or 0X, base 16 is assumed. Otherwise base 10 is assumed.

Note: The exponent (if present) must contain at least a digit. Otherwise, the possible exponent prefix and sign are not part of the number (which ends with the significand). Similarly, if 0b, 0B, 0x or 0X is not followed by a binary/hexadecimal digit, then the subject sequence stops at the character 0, thus 0 is read.

Special data (for infinities and NaN) can be `@inf@` or `@nan@(n-char-sequence-opt)`, and if base <= 16, it can also be `infinity`, `inf`, `nan` or `nan(n-char-sequence-opt)`, all case insensitive. A `n-char-sequence-opt` is a possibly empty string containing only digits, Latin letters and the underscore (0, 1, 2, ..., 9, a, b, ..., z, A, B, ..., Z, _).

Note: There is an optional sign for all data, even NaN. For example, `-@nAn@(This_Is_Not_17)` is a valid representation for NaN in base 17.

If the whole string cannot be parsed into a floating-point or fixed-point number, then an invalid operation exception is signalled. In this case, *rop* might have changed. Overflow or underflow can occur if the string is parsed to a floating-point or fixed-point number that is too large or too small to represent in the floating-point or fixed-point variable's precision.

Conversion functions

The Floating-Point Operator C model conversion functions are shown in Table 3-5. Most functions have variants to handle both floating-point and fixed-point variables.

Functions that convert to a standard C data type return the converted result as that data type. Any exceptions that occur are ignored. Functions that convert to GMP or MPFR data types place the result in the first argument and return exception flags, as with most Floating-Point Operator C model functions.

Table 3-5: **Floating-Point Operator C Model Conversion Functions**

Name	Return	Arguments	Description
xip_fpo_get_ui	unsigned long	xip_fpo_t op	Convert *op* to an unsigned long int after rounding.
xip_fpo_fix_get_ui	unsigned long	xip_fpo_fix_t op	
xip_fpo_get_si	signed long	xip_fpo_t op	Convert *op* to a signed long int after rounding.
xip_fpo_fix_get_si	signed long	xip_fpo_fix_t op	
xip_fpo_get_uj	uintmax_t	xip_fpo_t op	Convert *op* to an unsigned maximum size integer after rounding.
xip_fpo_fix_get_uj	uintmax_t	xip_fpo_fix_t op	
xip_fpo_get_sj	intmax_t	xip_fpo_t op	Convert *op* to a signed maximum size integer after rounding.
xip_fpo_fix_get_sj	intmax_t	xip_fpo_fix_t op	

Table 3-5: **Floating-Point Operator C Model Conversion Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_get_flt	float	xip_fpo_t op	Convert *op* to a float.
xip_fpo_fix_get_flt	float	xip_fpo_fix_t op	
xip_fpo_get_d	double	xip_fpo_t op	Convert *op* to a double.
xip_fpo_fix_get_d	double	xip_fpo_fix_t op	
xip_fpo_get_d_2exp	double	long *exp, xip_fpo_t op	Convert the mantissa of *op* to a double such that 0.5<=abs(mantissa)<1, and set the value pointed to by *exp* to the exponent of *op*. If *op* is zero, zero is returned and *exp* is zero. If *op* is NaN or infinity, NaN or infinity respectively is returned and *exp* is undefined.
xip_fpo_get_z	xip_fpo_exc_t	mpz_t rop, xip_fpo_t op	Convert *op* to a GMP/MPIR integer after rounding and store in *rop*. If *op* is NaN or infinity, *rop* is set to 0 and an invalid operation exception is returned.
xip_fpo_fix_get_z	xip_fpo_exc_t	mpz_t rop, xip_fpo_fix_t op	
xip_fpo_get_f	xip_fpo_exc_t	mpf_t rop, xip_fpo_t op	Convert *op* to a GMP/MPIR floating-point number and store it in *rop*. If *op* is NaN or infinity, *rop* is set to 0 and an invalid operation exception is returned.
xip_fpo_fix_get_f	xip_fpo_exc_t	mpf_t rop, xip_fpo_fix_t op	
xip_fpo_get_fr	xip_fpo_exc_t	mpfr_t rop, xip_fpo_t op	Convert *op* to an MPFR floating-point number and store it in *rop*.
xip_fpo_fix_get_fr	xip_fpo_exc_t	mpfr_t rop, xip_fpo_fix_t op	
xip_fpo_get_str	char *	char * str, xip_fpo_exp_t * exp, int base, int n_digits, xip_fpo_t op	Convert *op* to a string of digits in base *base*, returning the exponent separately in the variable pointed to by *exp*. See xip_fpo_get_str for details.
xip_fpo_fix_get_str	char *	char * str, int base, xip_fpo_fix_t op	Convert *op* to a string of digits in base *base*. See xip_fpo_fix_get_str for details.
xip_fpo_free_str	void	char * str	Free a string allocated by `xip_fpo_get_str` or `xip_fpo_fix_get_str`.
xip_fpo_fix_free_str	void	char * str	A synonym for `xip_fpo_free_str`.
xip_fpo_sizeinbase	int	xip_fpo_t op, int base	Return the size of *op* measured in number of digits in the given *base*. *base* can vary from 2 to 62. The sign of *op* is ignored. Returns -1 if an error occurs. Use to determine the space required when converting *op* to a string using `xip_fpo_get_str` or `xip_fpo_fix_get_str`.
xip_fpo_fix_sizeinbase	int	xip_fpo_fix_t op, int base	

xip_fpo_get_str

The function `xip_fpo_get_str` has the same usage as the MPFR function `mpfr_get_str`. *n_digits* is either zero or the number of significant digits output in the string; in the latter case, *n_digits* must be greater or equal to 2. The base can vary from 2 to 62. If the input number is an ordinary number, the exponent is written through the pointer *exp* (for input 0, the exponent is set to 0).

The generated string is in the base specified by *base*. Each string character is either a decimal digit or a Latin letter (62 possible characters). For *base* in the range 2 to 36, decimal digits and lower case letters are used, with a = 10, b = 11, … z = 35. For *base* in the range 37 to 62, digits, upper case, and lower case letters are used, with A = 10, B = 11, ..., Z = 35, a = 36, b = 37, ..., z = 61.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. For example, the number -3.1416 would be returned as "-31416" in the string and 1 written at *exp*. The value is rounded to provide *n_digits* of output, using round to nearest even: if *op* is exactly in the middle of two consecutive possible outputs, the one with an even significand is chosen, where both significands are considered with the exponent of *op*. Note that for an odd base, this might not correspond to an even last digit: for example with 2 digits in base 7, (14) and a half is rounded to (15) which is 12 in decimal, (16) and a half is rounded to (20) which is 14 in decimal, and (26) and a half is rounded to (26) which is 20 in decimal.

If *n_digits* is zero, the number of digits of the significand is chosen large enough so that re-reading the printed value with the same precision recovers the original value of *op*. More precisely, in most cases, the chosen precision of *str* is the minimal precision m depending only on p = PREC(*op*) and b that satisfies the above property, that is,
m = 1 + ceil(p*log(2)/log(b)), with p replaced by p-1 if b is a power of 2.

If *str* is a null pointer, space for the significand is allocated using the GMP/MPIR current allocation function which is `malloc()` by default, and a pointer to the string is returned. To free the memory used by the returned string, you must use `xip_fpo_free_str`.

If *str* is not a null pointer, it should point to a block of storage large enough for the significand, that is, at least max(*n_digits* + 2, 7) if *n_digits* > 0, or `xip_fpo_sizeinbase`(*op*, *base*) + 2 otherwise. The extra two bytes are for a possible minus sign, and for the terminating null character, and the value 7 accounts for `-@Inf@` plus the terminating null character.

A pointer to the string is returned, unless there is an error, in which case a null pointer is returned.

xip_fpo_fix_get_str

The function `xip_fpo_fix_get_str` has the same usage as the GMP/MPIR function `mpz_get_str`. The base can vary from 2 to 62.

The generated string is in the base specified by *base*. Each string character is either a decimal digit or a Latin letter (62 possible characters). For base in the range 2 to 36, decimal digits and lower case letters are used, with a = 10, b = 11, … z = 35. For base in the range 37 to 62, digits, upper case, and lower case letters are used, with A = 10, B = 11, ..., Z = 35, a = 36, b = 37, ..., z = 61.

The generated string is either an integer value with no radix point, or a fraction with an explicit radix point. All significant digits are returned, but no leading or trailing zeros are returned. No rounding is carried out.

If *str* is a null pointer, space for the significand is allocated using the current allocation function, and a pointer to the string is returned. To free the memory used by the returned string, you must use `xip_fpo_fix_free_str`.

If *str* is not a null pointer, it should point to a block of storage large enough for the result, that being `xip_fpo_fix_sizeinbase`(*op*, *base*) + 2. The extra two bytes are for a possible minus sign, and the terminating null character.

Operation Functions

The Floating-Point Operator C model functions that model operations of the core are shown in Table 3-6. In addition to functions using `xip_fpo_t` and `xip_fpo_fix_t` type arguments to provide custom precision, alternative versions of functions using standard C data types `float` and `double` are also provided, to make it easy for customers who do not need custom precision. For fixed to float and float to fixed functions, `float` and `double` to and from `int` are provided. For float to float functions, all combinations of `float` and `double` are provided: where these data types are the same, the function provides a means to condition numbers (convert signalling NaNs to quiet NaNs, convert denormalized numbers to zero).

Table 3-6: **Floating-Point Operator C Model Operation Functions**

Name	Return	Arguments	Description
xip_fpo_add	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* + *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_add_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* + *op2*. Single precision version.
xip_fpo_add_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* + *op2*. Double precision version.
xip_fpo_sub	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* - *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_sub_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* - *op2*. Single precision version.
xip_fpo_sub_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* - *op2*. Double precision version.
xip_fpo_mul	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1* × *op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_mul_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* × *op2*. Single precision version.
xip_fpo_mul_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1* × *op2*. Double precision version.

Table 3-6: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_div	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op1, xip_fpo_t op2	Set *rop* = *op1*/*op2*. *rop*, *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_div_flt	xip_fpo_exc_t	float * rop, float op1, float op2	Set *rop* = *op1* / *op2*. Single precision version.
xip_fpo_div_d	xip_fpo_exc_t	double * rop, double op1, double op2	Set *rop* = *op1*/*op2*. Double precision version.
xip_fpo_rec[(1)]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = 1/*op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_rec_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = 1/*op*. Single precision version.
xip_fpo_rec_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = 1/*op*. Double precision version.
xip_fpo_sqrt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = square root of *op*. *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_sqrt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = square root of *op*. Single precision version.
xip_fpo_sqrt_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = square root of *op*. Double precision version.
xip_fpo_recsqrt[(1)]	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = 1/(square root of *op*). *rop* and *op* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_recsqrt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = 1/(square root of *op*). Single precision version.
xip_fpo_recsqrt_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = 1/(square root of *op*). Double precision version.
xip_fpo_unordered	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. op1 and op2 must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_unordered_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. Single precision version.
xip_fpo_unordered_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* or *op2* is a NaN, 0 otherwise. Double precision version.
xip_fpo_equal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. op1 and op2 must have identical precisions, otherwise an operation not supported exception is returned.

Table 3-6: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_equal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. Single precision version.
xip_fpo_equal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* = *op2*, 0 otherwise. Double precision version.
xip_fpo_less	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_less_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. Single precision version.
xip_fpo_less_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* < *op2*, 0 otherwise. Double precision version.
xip_fpo_lessequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_lessequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. Single precision version.
xip_fpo_lessequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* <= *op2*, 0 otherwise. Double precision version.
xip_fpo_greater	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_greater_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. Single precision version.
xip_fpo_greater_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* > *op2*, 0 otherwise. Double precision version.
xip_fpo_greaterequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_greaterequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. Single precision version.
xip_fpo_greaterequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* >= *op2*, 0 otherwise. Double precision version.
xip_fpo_notequal	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.

Table 3-6: **Floating-Point Operator C Model Operation Functions *(Cont'd)***

Name	Return	Arguments	Description
xip_fpo_notequal_flt	xip_fpo_exc_t	int * res, float op1, float op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. Single precision version.
xip_fpo_notequal_d	xip_fpo_exc_t	int * res, double op1, double op2	Set *res* = 1 if *op1* <> *op2* or either *op1* or *op2* are NaN, 0 otherwise. Double precision version.
xip_fpo_condcode	xip_fpo_exc_t	int * res, xip_fpo_t op1, xip_fpo_t op2	Compare *op1* and *op2*, and set the least significant 4 bits of *res* to the resulting condition code. See Table 3-7 for the condition code encoding. *op1* and *op2* must have identical precisions, otherwise an operation not supported exception is returned.
xip_fpo_condcode_flt	xip_fpo_exc_t	int * res, float op1, float op2	Compare *op1* and *op2*, and set the least significant 4 bits of res to the resulting condition code. See Table 3-7 for the condition code encoding. Single precision version.
xip_fpo_condcode_d	xip_fpo_exc_t	int * res, double op1, double op2	Compare *op1* and *op2*, and set the least significant 4 bits of res to the resulting condition code. See Table 3-7 for the condition code encoding. Double precision version.
xip_fpo_flttofix	xip_fpo_exc_t	xip_fpo_fix_t rop, xip_fpo_t op	Set *rop* = *op*, rounding as required. *rop* and *op* must have compatible precisions (see xip_fpo_flttofix and xip_fpo_fixtoflt), otherwise an operation not supported exception is returned.
xip_fpo_flttofix_int_flt	xip_fpo_exc_t	int * rop, float op	Set *rop* = *op*, rounding as required. Single precision to integer version.
xip_fpo_flttofix_int_d	xip_fpo_exc_t	int * rop, double op	Set *rop* = *op*, rounding as required. Double precision to integer version.
xip_fpo_fixtoflt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_fix_t op	Set *rop* = *op*, rounding as required. *rop* and *op* must have compatible precisions (see xip_fpo_flttofix and xip_fpo_fixtoflt), otherwise an operation not supported exception is returned.
xip_fpo_fixtoflt_flt_int	xip_fpo_exc_t	float * rop, int op	Set *rop* = *op*, rounding as required. Integer to single precision version.
xip_fpo_fixtoflt_d_int	xip_fpo_exc_t	double * rop, int op	Set *rop* = *op*, rounding as required. Integer to double precision version.
xip_fpo_flttoflt	xip_fpo_exc_t	xip_fpo_t rop, xip_fpo_t op	Set *rop* = *op*, rounding as required. *rop* and *op* can have different precisions.
xip_fpo_flttoflt_flt_flt	xip_fpo_exc_t	float * rop, float op	Set *rop* = *op*, rounding as required. Single to single precision version (for conditioning numbers).
xip_fpo_flttoflt_flt_d	xip_fpo_exc_t	float * rop, double op	Set *rop* = *op*, rounding as required. Double to single precision version.
xip_fpo_flttoflt_d_flt	xip_fpo_exc_t	double * rop, float op	Set *rop* = *op*, rounding as required. Single to double precision version.
xip_fpo_flttoflt_d_d	xip_fpo_exc_t	double * rop, double op	Set *rop* = *op*, rounding as required. Double to double precision version (for conditioning numbers).

1. Only supported for `xip_fpo_t` operands with IEEE-754 single precision (exponent=8, mantissa=24) or double precision (exponent=11, mantissa=53).

For all functions, the result is guaranteed to match exactly the numerical output of the Floating-Point Operator v6.0 core, and the returned exceptions are guaranteed to match exactly the signalled exceptions of the Floating-Point Operator v6.0 core, for identical inputs.

When the operand and result variables do not meet constraints of the Floating-Point Operator v6.0 core, an operation not supported exception is returned. In this case, no other exception bits are set in the return value, and the result variable is not modified.

`xip_fpo_condcode` functions set the 4 least significant bits of their integer result to a condition code, which has the encoding shown in Table 3-7. Encodings not shown are reserved and are not returned by the functions.

Table 3-7: **Condition Code Encoding**

Integer result	Condition code bit				Meaning
	3	2	1	0	
	Unordered	Greater than	Less than	Equal	
1	0	0	0	1	*op1* = *op2*
2	0	0	1	0	*op1* < *op2*
4	0	1	0	0	*op1* > *op2*
8	1	0	0	0	*op1*, *op2* or both are NaN

For all comparison functions, the sign of zero is ignored, such that -0 = +0.

xip_fpo_flttofix and xip_fpo_fixtoflt

`xip_fpo_flttofix` and `xip_fpo_fixtoflt` functions have restrictions on the precisions of the fixed-point and floating-point operand and result. The exponent width of the floating-point variable must be at least:

$$\text{minimum floating-point exponent width} = \text{ceil}(\log_2(\text{fixed-point total width} + 3)) + 1$$

If the operand and result variable do not meet this condition, an operation not supported exception is returned and the result variable is not modified.

Compiling

Compilation of user code requires access to the `floating_point_v6_0_bitacc_cmodel.h` header file and the header files of the MPIR [Ref 1] and MPFR [Ref 2] dependent libraries, `gmp.h` and `mpfr.h`. The header files should be copied to a location where they are available to the compiler. Depending on the location chosen, the include search path of the compiler might need to be modified.

The `floating_point_v6_0_bitacc_cmodel.h` header file must be included first, because it defines some symbols that are used in the MPIR and MPFR header files. The `floating_point_v6_0_bitacc_cmodel.h` header file includes the MPIR and MPFR header files, so these do not need to be explicitly included in source code that uses the C model. When compiling on Windows, the symbol `NT` must be defined, either by a compiler option, or in user source code before the `floating_point_v6_0_bitacc_cmodel.h` header file is included.

Linking

To use the C model the user executable must be linked against the correct libraries for the target platform.

Note: The C model uses MPIR and MPFR libraries. Pre-compiled MPIR and MPFR libraries are provided with the C model. It is also possible to use GMP or MPIR, and MPFR libraries from other sources, for example, compiled from source code. For details, see Dependent Libraries.

Linux

The executable must be linked against the following shared object libraries:

- `libgmp.so.7`
- `libmpfr.so.4`
- `libIp_floating_point_v6_0_bitacc_cmodel.so`

Using GCC, linking is typically achieved by adding the following command line options:

```
-L. -lgmp -lmpfr -lIp_floating_point_v6_0_bitacc_cmodel
```

This assumes the shared object libraries are in the current directory. If this is not the case, the `-L.` option should be changed to specify the library search path to use.

Using GCC, the provided example program `run_bitacc_cmodel.c` can be compiled and linked using the following command:

```
gcc run_bitacc_cmodel.c -o run_bitacc_cmodel -I. -L. -lgmp -lmpfr
-lIp_floating_point_v6_0_bitacc_cmodel
```

Windows

The executable must be linked against the following dynamic link libraries:

- `libgmp.dll`
- `libmpfr.dll`
- `libIp_floating_point_v6_0_bitacc_cmodel.dll`

Depending on the compiler, the import libraries might also be required:

- `libgmp.lib`
- `libmpfr.lib`
- `libIp_floating_point_v6_0_bitacc_cmodel.lib`

Using Microsoft Visual Studio, linking is typically achieved by adding the import libraries to the Additional Dependencies entry under the Linker section of Project Properties.

Dependent Libraries

The C model uses MPIR and MPFR libraries. Pre-compiled MPIR and MPFR libraries are provided with the C model, using the following versions of the libraries:

- MPIR 2.2.1
- MPFR 3.0.1

As MPIR is a compatible alternative to GMP, the GMP library may be used in place of MPIR. It is possible to use GMP or MPIR and MPFR libraries from other sources, for example, compiled from source code.

GMP and MPIR in particular, and MPFR to a lesser extent, contain many low level optimizations for specific processors. The libraries provided are compiled for a generic processor on each platform, using no optimized processor-specific code. These libraries will work on any processor, but will run more slowly than libraries compiled to use optimized processor-specific code. For the fastest performance, compile libraries from source on the machine on which you run the executables.

Source code and compilation scripts are provided for the versions of MPIR and MPFR that were used to compile the provided libraries. Source code and compilation scripts for any version of the libraries can be obtained from the GMP [Ref 3], MPIR [Ref 1] and MPFR [Ref 2] web sites. Microsoft Visual Studio project files for compiling MPFR on Windows can be obtained from Brian Gladman's web site [Ref 4].

Note: If compiling MPIR using its `configure` script (for example, on Linux platforms), use the `--enable-gmpcompat` option when running the `configure` script. This generates a `libgmp.so` library and a `gmp.h` header file that provide full compatibility with the GMP library. This compatibility is required by the MPFR compilation scripts.

Note: Some Windows compilers, for example Microsoft Visual Studio versions prior to 2010, do not have full support for the C99 standard of the C programming language. The MPFR library contains functions that use the C99 types `intmax_t` and `uintmax_t` (for example, functions with `_sj` and `_uj` suffixes). When MPFR is compiled, it checks if these types are present, and excludes these functions if not. The C model requires these functions in MPFR. Therefore, when compiling MPFR using a Windows compiler without C99 support, include the provided `mpfr_nt_stdint.h` header file, which defines the types `intmax_t` and `uintmax_t`. Using Microsoft Visual Studio, this header file can be included without modifying source code by adding it to the Force Includes entry under the Advanced sub-section of the C/C++ section of Project Properties.

Example

The `run_bitacc_cmodel.c` file contains example code to show basic operation of the C model. Part of this example code is shown here. The comments assist in understanding the code.

This code calculates *e*, the base of natural logarithms, in the given precision. The Taylor Series expansion for the exponential function e^x is:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

To calculate *e*, set $x = 1$:

$$e^x = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!} + \ldots$$

This code calculates terms iteratively until the accuracy of *e* no longer improves.

```
#include <stdio.h>
#include "floating_point_v6_0_bitacc_cmodel.h"
int main()
{
xip_fpo_exp_t exp_prec, mant_prec;
  // The algorithm will work for any legal combination
  // of values for exp_prec and mant_prec
  exp_prec  = 16;
  mant_prec = 64;
  printf("Using Taylor Series expansion to calculate e, the base of
natural logarithms, in %d-bit mantissa precision\n", mant_prec);
```

```
int i, done;
  xip_fpo_t n, fact, one, term, e, e_old;
  xip_fpo_exc_t ex;
  xip_fpo_exp_t exp;
  char * result = 0;
  double e_d;

xip_fpo_inits2 (exp_prec, mant_prec, n, fact, one, term, e,
                  e_old, (xip_fpo_ptr) 0);
  xip_fpo_set_ui (one, 1);

// 0th term
  i = 0;
  xip_fpo_set_ui (fact, 1);
  xip_fpo_set_ui (e, 1);

// Main iteration loop
  do {

// Set up this iteration
    i++;
    xip_fpo_set_ui (n, i);
    xip_fpo_set (e_old, e);

// Calculate the next term: 1/n!
    ex  = xip_fpo_mul (fact, fact, n);  // n!
    ex |= xip_fpo_div (term, one, fact);  // 1/n!
    // Note: an alternative to the preceding line is:
    //   ex |= xip_fpo_rec (term, fact);
    // but this is only possible if using single or double
    // (exp_prec, mant_prec = 8, 24 or 11, 53 respectively)
    // because xip_fpo_rec only supports single and double

// Calculate the estimate of e
    ex |= xip_fpo_add (e, e, term);

// Are we done?
    ex |= xip_fpo_equal (&done, e, e_old);

// Check for exceptions (none should occur)
    if (ex) {
      printf ("Iteration %d: exception occurred: %d\n", i, ex);
      return 1;
    }

// Print result so far
    result = xip_fpo_get_str (result, &exp, 10, 0, e);
    printf ("After %2d iteration(s), e is 0.%s * 10 ^ %d\n",
            i, result, exp);

} while (!done);

// Convert result to C's double precision type
  e_d = xip_fpo_get_d (e);
  printf ("As a C double, e is %.20f\n", e_d);

// Free up memory
  xip_fpo_clears (n, fact, one, term, e, e_old, xip_fpo_ptr) 0);
  xip_fpo_free_str (result);
return 0;
}
```

XILINX®

Appendix A

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx Support website at:

http://www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

http://www.xilinx.com/support/documentation/sw_manuals/glossary.pdf.

References

1. The Multiple Precision Integers and Rationals (MPIR) Library: http://www.mpir.org/
2. The GNU Multiple Precision Floating-Point Reliable (MPFR) Library: http://www.mpfr.org/
3. The GNU Multiple Precision Arithmetic (GMP) Library: http://gmplib.org/
4. Multiple Precision Arithmetic on Windows, Brian Gladman: http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php