Data sheet

Quixilica® Floating Point FPGA Cores

Floating Point Adder
- 169 MFLOPS on VirtexE-8*
Floating Point Multiplier
- 152 MFLOPS on VirtexE-8*
Floating Point Divider
- 189 MFLOPS on VirtexE-8*
Floating Point Square Root
- 180 MFLOPS on VirtexE-8*

IEEE-754 - compliant variable wordlength floating point arithmetic cores for Xilinx Virtex, Virtex E and Virtex-II FPGA families.

Features	Benefits
• Variable wordlength floating point (includes IEEE-754 single and double precision wordlengths)	• Optimise wordlength for application
	• Reduced FPGA area for each core
	• Increased computation available per FPGA
	• Reduced power consumption per core
	• Standard IEEE wordlengths available for integration with DSP / CPU
• Fully pipelined architecture	• Maximum throughput – new computation starts every clock cycle
• Optimised placement – as relationally placed macros (RPMs)	• High-density layout
	• Efficient use of FPGA resources
	• High-speed routing
	• High clock speeds achievable (189 MHz for VirtexE-8)
• Supports Virtex, Virtex-E and Virtex-II FPGA families	• Easy migration path for designs
	• Increased density and reduced power consumption on Virtex-II by using on-chip multipliers
• Bit-true models for simulation	• Rapid assessment of word-length requirements
	• Allows comprehensive bit-true testing of system design in software domain rather than in VHDL domain
• MATLAB MEX-function interfaces for algorithm modelling	• Direct modelling of bit-true design from algorithm development environment
• Compliance with IEEE-754 floating point standard	• Compatibility with microprocessor arithmetic, for heterogeneous systems
	• Rapid porting of designs from software to FPGA

* IEEE-754 single precision format.

1 IEEE Floating Point Representation

1.1 Floating Point Format

All of the floating-point cores are IEEE-754 compliant. The format of the binary floating point word is a total of *mw* + *ew* bits, where *mw* and *ew* are the generics defining the mantissa and exponent word lengths respectively. The floating-point number consists of three fields as shown in figure 1 and table 1.

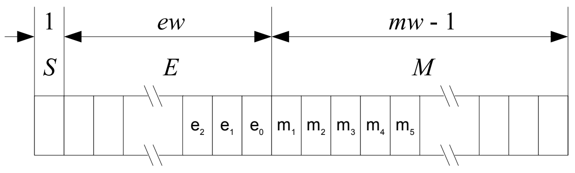

Figure 1– Floating point format

Field	Size / bits	Meaning
E	*ew*	exponent: unsigned biased representation where the binary value is the sum of the original signed exponent value and the bias number. Zero is represented by a binary value of 0.
S	1	sign: 0 indicates a positive number, 1 indicates a negative number.
M	*mw* – 1	mantissa: unsigned fixed point fraction, 1.0 <= mantissa < 2.0. The MSB is always 1, and is omitted, so only *mw*-1 bits are stored.

Table 1 - Floating point definitions

1.2 Floating Point Range

The floating point cores support both the IEEE-754 single and double format as shown in table 2, but also cover a much wider range of mantissa and exponent word length combinations.

1.3 Floating Point Conversion

For a normalised number the IEEE-754 floating point value, *f,* is constructed from the fields as follows:

$$f = (-1)^s 1.M \times 2^{E-E_{bias}}$$

where *M* is the fractional mantissa number constructed from the binary value as follows:

$$1.M = 1 + \sum_{i=1}^{mw-1} m_i 2^{-i}$$

(m_i is the mantissa bit relative to the msb.)

and where *E* is the exponent value constructed from the binary value as follows:

$$E = \sum_{i=0}^{ew-1} e_i 2^i$$

(e_i is the exponent bit relative to the lsb.)

and *S* is the value of the sign bit.

Note : These floating point cores will also support denormalised numbers in a future release.

Parameter	Symbol	Quixilica® range	Value for IEEE-754 single precision	Value for IEEE-754 double precision
Mantissa width	*mw*	$5 \le mw \le 53$	24	53
Exponent width	*ew*	$4 \le ew \le 15$	8	11
Wordlength		*mw* + *ew*	32	64
Exponent bias	E_{bias}	$2^{ew-1} - 1$	+127	+1023

Table 2 – Parameters for Quixilica® floating point cores and standard IEEE-754 precisions

1.4 Rounding

The floating point cores use the IEEE-754 round to even functionality where the result is rounded to the nearest representable value. If two values are equally near then the one with its lsb 0 is chosen giving a round to even result.

1.5 Underflow

An underflow exception is signalled when tininess has been detected. Tininess is the creation of a non-zero result between $\pm 2^{2-2^{ew-1}}$. This may cause overflow at some later point.

1.6 Overflow

When the largest finite number is exceeded in magnitude by what would be the rounded floating point value, overflow occurs. Overflow is only detected after rounding has been performed. When overflow occurs the overflow flag is signalled and the resultant output becomes infinity with the sign corresponding to the precise result.

1.7 Invalid Operation

An invalid operation is raised when the combination of zero and infinity input values cause an invalid result. The final output is a quiet NaN.

1.8 Not a number (NaN)

NaN is an IEEE-754 floating point representation for the result of an operation which cannot return a valid value. NaNs are detected at the inputs to the cores resulting in a quiet NaN at the output of the floating point operation. The binary representation of a NaN at the resultant output is defined as follows:

S	e	m
x	1111……….1111	1111……………..1111

Figure 2 - NaN representation

1.9 Division By Zero

Division by zero is only applicable to the floating point divider core and is signalled when the numerator is a finite non-zero number and the denominator is zero.

2 fpAdd: Floating Point Adder

2.1 fpAdd Component Description

The component symbol for fpAdd is shown in Figure 3 and the VHDL component declaration is shown in Figure 5.

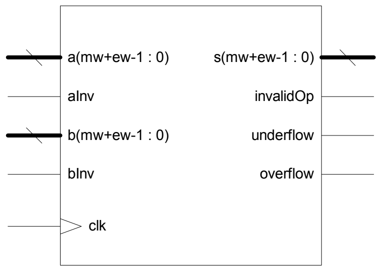

Figure 3 – fpAdd component symbol

2.2 fpAdd Functional Description

A functional representation of the operation of fpAdd is shown in Figure 4. Note that this does not necessarily represent the architectural description.

Input normalisation. The exponents of the two numbers are compared and the mantissa of the smaller number is shifted to align the two mantissae.

Sign calculation. The sign of the two input numbers, along with the aInv and bInv inputs are combined together to determine whether to add or subtract the two unsigned mantissae. The sign of this operation is then used to calculate the final sign.

Mantissa addition. The two unsigned mantissae are added or subtracted according to the result of the sign comparison. The result is a signed integer

Normalisation shift. The mantissa is normalised by shifting so that its MSB is '1'. For each shift, the exponent is adjusted accordingly.

Mantissa round. The normalised mantissa is rounded. This may cause an overflow, in which case the exponent is adjusted and the mantissa is normalised. Rounding is performed in accordance with the IEEE-754 "round to even" mode.

Output pack. The mantissa, exponent and sign are packed together into the output S. The flags are set according to exceptions that occur in the arithmetic.

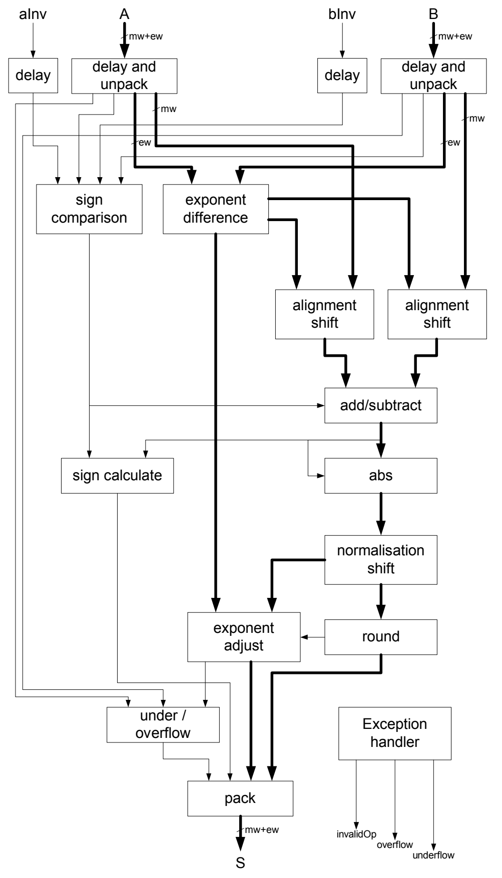

Figure 4 – fpAdd functional block diagram

```
component fpAdd
  generic (
    ew        : integer;      -- bit width of the exponent, see table 7 for possible values
    mw        : integer);     -- bit width of the mantissa, see table 7 for possible values
  port (
    clk       : in  std_logic;                         -- synchronous clock
    a         : in  std_logic_vector(mw+ew-1 downto 0);  -- operand input A
    b         : in  std_logic_vector(mw+ew-1 downto 0);  -- operand input B
    aInv      : in  std_logic := '0';                  -- invert flag A (inverts sign of A input)
    bInv      : in  std_logic := '0';                  -- invert flag B (inverts sign of B input)
    s         : out std_logic_vector(mw+ew-1 downto 0);  -- sum output
    invalidOp : out std_logic;                         -- invalid operation exception flag
    overflow  : out std_logic;                         -- overflow exception flag
    underflow : out std_logic);                        -- underflow exception flag
end component fpAdd;
```

Figure 5 – fpAdd component declaration

3 fpMul: Floating Point Multiplier

3.1 fpMul Component Description

The component symbol for fpMul is shown in Figure 6 and the VHDL component declaration is shown in Figure 8.

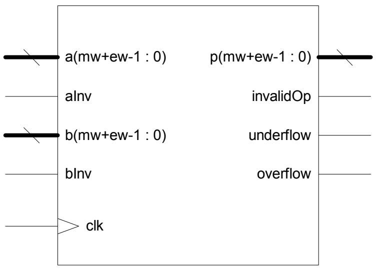

Figure 6 - fpMul component symbol

3.2 fpMul Functional Description

A functional representation of the operation of fpMul is shown in Figure 7. Note that this does not necessarily represent the architectural description.

Exponent addition. The exponents of the two floating point numbers are added together.
Mantissa multiplication. The mantissae of the two floating point numbers are multiplied together within a fixed-point multiplier core.
Normalisation and rounding. The result from the mantissa multiplication is used to adjust the exponent if necessary, and the result is rounded in accordance with the IEEE-754 “Round to even” mode.
Output pack. The sign, mantissa and exponent are packed into the output word. Exceptions that occur within the arithmetic are indicated using the three exception flag outputs.

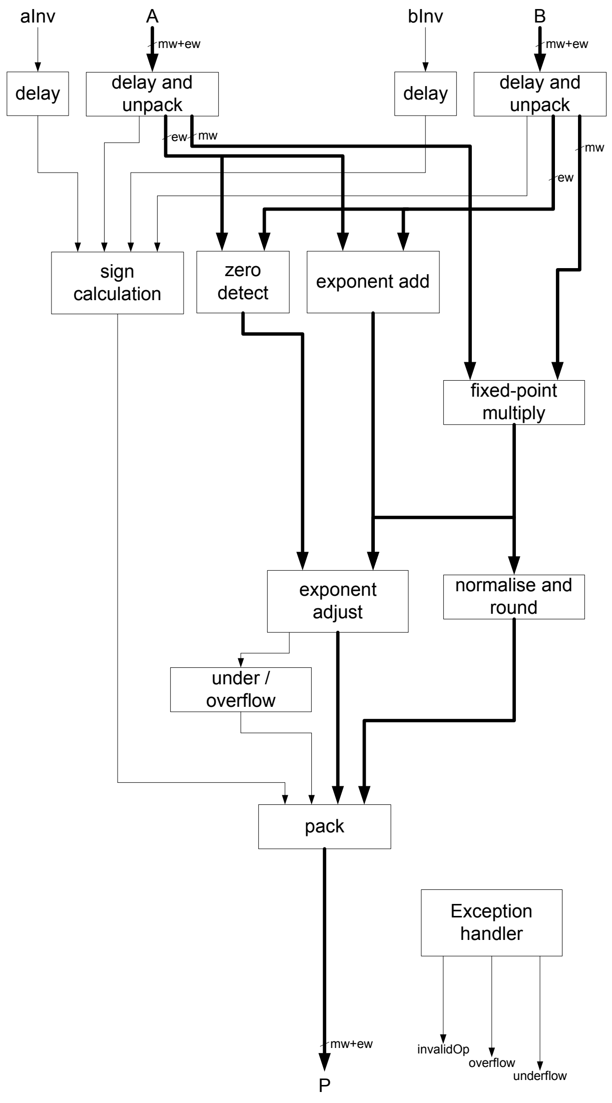

Figure 7 – fpMul functional block diagram

```
component fpMul
  generic (
    ew        : integer;      -- bit width of the exponent, see table 7 for possible values
    mw        : integer);     -- bit width of the mantissa, see table 7 for possible values
  port (
    clk       : in  std_logic;                          -- synchronous clock
    a         : in  std_logic_vector(mw+ew-1 downto 0);  -- operand input A
    b         : in  std_logic_vector(mw+ew-1 downto 0);  -- operand input B
    aInv      : in  std_logic := '0';                   -- invert flag A (inverts sign of A input)
    bInv      : in  std_logic := '0';                   -- invert flag B (inverts sign of B input)
    p         : out std_logic_vector(mw+ew-1 downto 0);  -- product output
    invalidOp : out std_logic;                          -- invalid operation exception flag
    overflow  : out std_logic;                          -- overflow exception flag
    underflow : out std_logic);                         -- underflow exception flag
end component fpMul;
```

Figure 8 – fpMul component declaration

4 fpDiv: Floating Point Divider

4.1 fpDiv Component Description

The component symbol for fpDiv is shown in Figure 9 and the VHDL component declaration is shown in Figure 11.

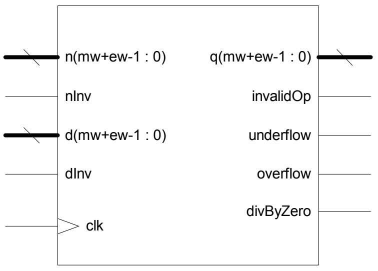

Figure 9 – fpDiv component symbol

4.2 fpDiv Functional Description

A functional representation of the operation of fpDiv is shown in Figure 10. Note that this does not necessarily represent the architectural description.
Exponent subtraction. The exponent of the denominator is subtracted from the exponent of the numerator.
Mantissa division. The mantissa of the numerator is divided by the mantissa of the denominator, within a fixed-point divider core.
Normalisation and rounding. The result from the mantissa division is used to adjust the exponent if necessary, and the result is rounded in accordance with the IEEE-754 "Round to even" mode.
Output pack. The sign, mantissa and exponent are packed into the output word. Exceptions that occur within the arithmetic are indicated using the four exception flag outputs.

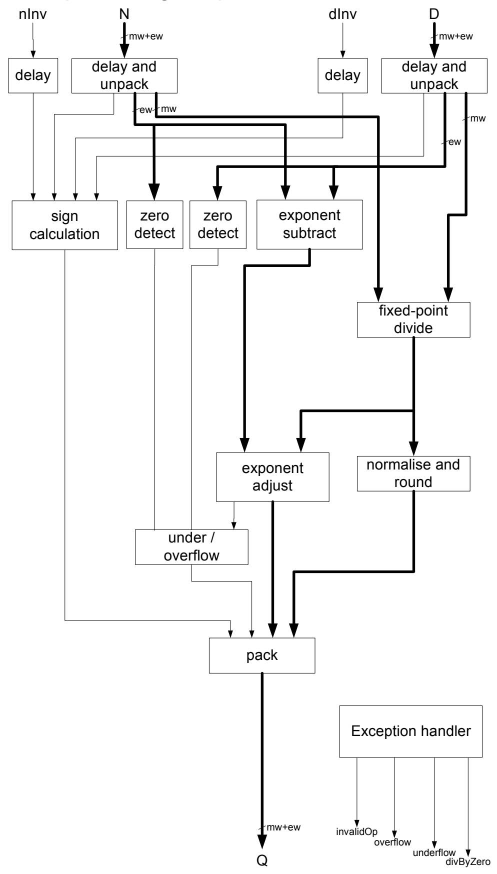

Figure 10 – fpDiv functional block diagram

```
component fpDiv
  generic (
    ew        : integer;      -- bit width of the exponent, see table 7 for possible values
    mw        : integer);     -- bit width of the mantissa, see table 7 for possible values
  port (
    clk       : in  std_logic;                           -- synchronous clock
    n         : in  std_logic_vector(mw+ew-1 downto 0);  -- numerator input
    d         : in  std_logic_vector(mw+ew-1 downto 0);  -- denominator input
    nInv      : in  std_logic := '0';                    -- invert flag - inverts sign of numerator
    dInv      : in  std_logic := '0';                    -- invert flag - inverts sign of denominator
    q         : out std_logic_vector(mw+ew-1 downto 0);  -- quotient output
    invalidOp : out std_logic;                           -- invalid operation exception flag
    divByZero : out std_logic;                           -- division by zero exception flag
    overflow  : out std_logic;                           -- overflow exception flag
    underflow : out std_logic);                          -- underflow exception flag
end component fpDiv;
```

Figure 11 – fpDiv component declaration

5 fpSqrt: Floating Point Square Root

5.1 fpSqrt Component Description

The component symbol for fpSqrt is shown in Figure 12 and the VHDL component declaration is shown in Figure 14.

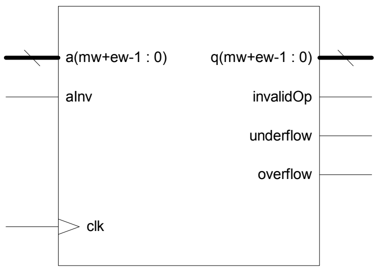

Figure 12 – fpSqrt component symbol

5.2 fpSqrt Functional Description

A functional representation of the operation of fpMul is shown in Figure 13. Note that this does not necessarily represent the architectural description.

Exponent division. The exponent of the input is divided by two.
Mantissa denormalise. If the exponent was odd, the mantissa is divided by 2 to give an even exponent.
Square root. The square root of the mantissa is found using a fixed-point square root core.
Normalisation and rounding. The result from the mantissa square root operation is used to adjust the exponent if necessary, and the result is rounded in accordance with the IEEE-754 “Round to even” mode.

Output pack. The sign, mantissa and exponent are packed into the output word. Exceptions that occur within the arithmetic are indicated using the three exception flag outputs.

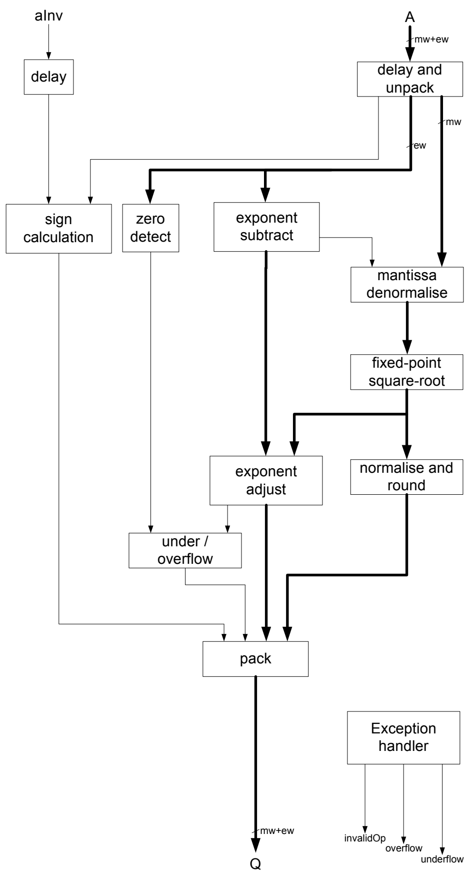

Figure 13 - fpSqrt Functional Block Diagram

```
component fpSqrt
  generic (
    ew        : integer;      -- bit width of the exponent, see table 7 for possible values
    mw        : integer);     -- bit width of the mantissa, see table 7 for possible values
  port (
    clk       : in  std_logic;                          -- synchronous clock
    a         : in  std_logic_vector(mw+ew-1 downto 0); -- operand input
    aInv      : in  std_logic := '0';                   -- invert flag - inverts sign of input operand
    q         : out std_logic_vector(mw+ew-1 downto 0); -- quotient output
    invalidOp : out std_logic;                          -- invalid operation exception flag
    overflow  : out std_logic;                          -- overflow exception flag
    underflow : out std_logic);                         -- underflow exception flag
end component fpSqrt;
```

Figure 14 – fpSqrt component declaration

6 Performance

All of the floating point cores are fully pipelined so a new floating-point calculation starts on each clock cycle. Latency varies depending upon the chosen mantissa and exponent wordlengths. Preliminary performance characteristics for each floating point core for a given mantissa and exponent wordlength on Virtex-E devices are given in Tables 3, 4, 5 and 6.

The maximum clock speeds that can be achieved on different speed grades of Virtex-E FPGA were obtained using Xilinx Foundation tools version 3.3, service pack 7.

7 Layout

The layout of the floating point cores are designed for efficiency over a range of sizes. The layout may be optimised for any particular size of mantissa and exponent. The cores are synthesised assuming that their top left corner is aligned with a CLB. This may result in wasting a column of slices or a row of LUTs if the adjacent cores do not finish at a CLB boundary. The layout of the cores may be adjusted so that their top left corner aligns with either slice in a CLB and with either LUT in a slice. Contact your distributor for further details.

The Xilinx Foundation tools support placement of components through the use of the RLOC attribute in VHDL.

Floating Point Adder (fpAdd)							
Parameters		Frequency (VIRTEXE)			Size		Latency of output
mw	ew	-6 (MHz)	-7 (MHz)	-8 (MHz)	Slices	RPM Bounds as Cols x Rows	Clock cycles
8	4						
10	4						
12	4						
8	6	153		199	121	28 × 12 (LUTS)	10
10	6						
12	6	143		193	158	30 × 16 (LUTS)	10
14	6						
16	6	153		188	208	28 × 20 (LUTS)	11
18	6						
20	6	147		182	247	27 × 24 (LUTS)	11
22	6						
24	6						
14	8	137		169	306	27 × 28 (LUTS)	11
16	8						
18	8						
20	8						
22	8						
24	8						
16	10						
18	10						
20	10						
22	10						
24	10						

Table 3 – Performance characteristics for the fpAdd core (other values TBC)

Floating Point Divider (fpDiv)							
Parameters		Frequency (VIRTEXE)			Size		Latency of output
mw	ew	-6 (MHz)	-7 (MHz)	-8 (MHz)	Slices	RPM Bounds as Cols x Rows	Clock cycles
8	4						
10	4						
12	4						
8	6	202		259	124	27 × 10 (LUTS)	11
10	6						
12	6	182		238	220	37 × 14 (LUTS)	15
14	6						
16	6	166		216	348	41 × 18 (LUTS)	19
18	6						
20	6	154		203	512	50 × 22 (LUTS)	23
22	6						
24	6						
14	8						
16	8						
18	8						
20	8						
22	8						
24	8	142		189	711	58 × 26 (LUTS)	27
16	10						
18	10						
20	10						
22	10						
24	10						

Table 4 – Performance characteristics for the fpDiv core (other values TBC)

Floating Point Multiplier (fpMul)							
Parameters		Frequency (VIRTEXE)			Size		Latency of output
mw	ew	-6 (MHz)	-7 (MHz)	-8 (MHz)	Slices	RPM Bounds as Cols x Rows^	Clock cycles
8	4						
10	4						
12	4						
8	6	130		164	67	10 x 20 (LUTS)	5
10	6						
12	6	129		165	119	14 x 20 (LUTS)	6
14	6						
16	6	125		165	171	16 x 23 (LUTS)	6
18	6						
20	6	122		157	229	18 x 31 (LUTS)	6
22	6						
24	6						
14	8						
16	8						
18	8						
20	8						
22	8						
24	8	122		152	326	22 x 37 (LUTS)	6
16	10						
18	10						
20	10						
22	10						
24	10						

Table 5 – Performance characteristics for the fpMul core (other values TBC)

Floating Point Square-Root (fpSqrt)							
Parameters		Frequency (VIRTEXE)			Size		Latency of output
mw	ew	-6 (MHz)	-7 (MHz)	-8 (MHz)	Slices	RPM Bounds as Cols x Rows^	Clock cycles
8	4						
10	4						
12	4						
8	6						
10	6						
12	6						
14	6						
16	6	173		191	620	38 x 20 (LUTS)	19
18	6						
20	6						
22	6						
24	6						
14	8						
16	8						
18	8						
20	8						
22	8						
24	8						
16	10						
18	10						
20	10						
22	10						
24	10						

Table 6 – Performance Characteristics of the fpSqrt core (other values TBC)

8 Bit-true model

A bit-true model of each floating point core is available as a dynamic link library for Microsoft Windows NT 4.0. This may be called from the user's own programs (e.g. written in C/C++), or may be used with an accompanying MATLAB MEX function dynamic link library to enable the floating point core function to be used within MATLAB.

The bit-true model allows evaluation of the floating point core, and also allows numerical simulation to be carried out in the user's algorithm development environment, rather than in a VHDL simulation. The bit-true model executes much more rapidly than a corresponding VHDL simulation, allowing more comprehensive modelling.
The Matlab function interfaces are shown in Figure 15, 16, 17 and 18.

```
[sum, flags] = fpAdd (a, b, mw, ew, rounding, bypassModel, verbose)

 * where flags, rounding, bypassModel and verbose are optional
 * sum                   is the sum output
 * flags                 is an integer representing flag status
 * a                     is one input operand
 * b                     is the other input operand
 * mw                    is the mantissa width used to represent each operand
 * ew                    is the exponent width used to represent each operand
 * rounding              is 1 to use standard IEEE-754 rounding, 0 to use the basic rounding in the
 *                       current VHDL models (defaults to 0)
 * bypassModel           is 0 to use the VHDL model, 1 to bypass this and use double precision floating
 *                       point arithmetic
 * verbose               is 0 to provide minimal warning and error messages, 1 to provide full warning
 *                       and error messages (use 1 for debugging algorithms, then switch to 0 to avoid
 *                       underflow and overflow warning messages once the algorithm is debugged).
 *                       Defaults to 1 (full warnings)
```

Figure 15 - fpAdd function prototype for the bit-true model

```
[prod, flags] = fpMul (a, b, mw, ew, rounding, bypassModel, verbose)

 * where flags, rounding, bypassModel and verbose are optional
 * prod                  is the product output
 * flags                 is an integer representing flag status
 * a                     is one input operand
 * b                     is the other input operand
 * mw                    is the mantissa width used to represent each operand
 * ew                    is the exponent width used to represent each operand
 * rounding              is 1 to use standard IEEE-754 rounding, 0 to use the basic rounding in the
 *                       current VHDL models (defaults to 0)
 * bypassModel           is 0 to use the VHDL model, 1 to bypass this and use double precision floating
 *                       point arithmetic
 * verbose               is 0 to provide minimal warning and error messages, 1 to provide full warning
 *                       and error messages (use 1 for debugging algorithms, then switch to 0 to avoid
 *                       underflow and overflow warning messages once the algorithm is debugged).
 *                       Defaults to 1 (full warnings)
```

Figure 16 - fpMul function prototype for the bit-true model

```
[quotient, flags] = fpDiv (numerator, denominator, mw, ew, rounding, bypassModel)

 * where flags, rounding and bypassModel are optional
 * quotient              is the quotient output
 * flags                 is an integer representing flag status
 * numerator             is the numerator input operand
 * denominator           is the denominator input operand
 * mw                    is the mantissa width used to represent each operand
 * ew                    is the exponent width used to represent each operand
 * rounding              is 1 to use standard IEEE-754 rounding, 0 to use the basic rounding in the
 *                       current VHDL models (defaults to 0)
 * bypassModel           is 0 to use the VHDL model, 1 to bypass this and use double precision floating
 *                       point arithmetic
```

Figure 17 - fpDiv function prototype for the bit-true model

```
[quotient, flags] = fpSqrt (radicand, mw, ew, rounding, bypassModel)

 * where flags, rounding and bypassModel are optional
 * quotient           is the quotient output
 * flags              is an integer representing flag status
 * numerator          is the radicand input operand
 * mw                 is the mantissa width used to represent each operand
 * ew                 is the exponent width used to represent each operand
 * rounding           is 1 to use standard IEEE-754 rounding, 0 to use the basic rounding in the
 *                    current VHDL models (defaults to 0)
 * bypassModel        is 0 to use the VHDL model, 1 to bypass this and use double precision floating
 *                    point arithmetic
```

Figure 18 - fpSqrt function prototype for the bit-true model

9 Ordering information

9.1 Part numbers

To order the **EDIF** netlist for a floating point core specify the part number:

FP2-m-e

Where:
m **=** Mantissa wordlength
e **=** Exponent wordlength

Part No.	**Description**
FP2-8-4	Fileset for FP2 core, Mantissa size 8, Exponent size 4
FP2-8-6	Fileset for FP2 core, Mantissa size 8, Exponent size 6
FP2-10-4	Fileset for FP2 core, Mantissa size 10, Exponent size 4
FP2-10-6	Fileset for FP2 core, Mantissa size 10, Exponent size 6
FP2-12-4	Fileset for FP2 core, Mantissa size 12, Exponent size 4
FP2-12-6	Fileset for FP2 core, Mantissa size 12, Exponent size 6
FP2-14-6	Fileset for FP2 core, Mantissa size 14, Exponent size 6
FP2-14-8	Fileset for FP2 core, Mantissa size 14, Exponent size 8
FP2-16-6	Fileset for FP2 core, Mantissa size 16, Exponent size 6
FP2-16-8	Fileset for FP2 core, Mantissa size 16, Exponent size 8
FP2-16-10	Fileset for FP2 core, Mantissa size 16, Exponent size 10
FP2-18-6	Fileset for FP2 core, Mantissa size 18, Exponent size 6
FP2-18-8	Fileset for FP2 core, Mantissa size 18, Exponent size 8
FP2-18-10	Fileset for FP2 core, Mantissa size 18, Exponent size 10
FP2-20-6	Fileset for FP2 core, Mantissa size 20, Exponent size 6
FP2-20-8	Fileset for FP2 core, Mantissa size 20, Exponent size 8
FP2-20-10	Fileset for FP2 core, Mantissa size 20, Exponent size 10
FP2-22-6	Fileset for FP2 core, Mantissa size 22, Exponent size 6
FP2-22-8	Fileset for FP2 core, Mantissa size 22, Exponent size 8
FP2-22-10	Fileset for FP2 core, Mantissa size 22, Exponent size 10
FP2-24-6	Fileset for FP2 core, Mantissa size 24, Exponent size 6
FP2-24-8	Fileset for FP2 core, Mantissa size 24, Exponent size 8
FP2-24-10	Fileset for FP2 core, Mantissa size 24, Exponent size 10
FP2-ALL-ALL	Fileset containing all filesets identified in items above

To order the bit-true model, including Matlab™ and GEDAE™ interface, specify part number:

FP2-BTM

9.2 Availability

The following sizes of floating point cores are available as the precompiled EDIF netlists. Other sizes can be produced upon request.

mw	ew			
	4	6	8	10
8	✓	✓		
10	✓	✓		
12	✓	✓		
14		✓	✓	
16		✓	✓	✓
18		✓	✓	✓
20		✓	✓	✓
22		✓	✓	✓
24		✓	✓	✓

Table 7 – floating point core availability

For further information please contact:

Transtech DSP
20 Thornwood Drive
Ithaca, NY 14850-1263
USA
Tel: 607 257 8678
Fax: 607 257 8679

Manor Courtyard
Hughenden Avenue
High Wycombe
HP13 5RE, UK
Tel: +44(0)1494 464432
Fax: +44(0)1494 464472

11 Saharov Street NIA
Rishon LeZion 75707
Israel
Tel: 972-3-9518718
Fax: 972-3-9518719

email: sales@transtech-dsp.com
www.transtech-dsp.com