Dillon Engineering, Inc.

3925 West 50th Street, Suite 202, Edina, MN 55424
Tel.: 952-836-2413 Fax: 952-927-6514
www.dilloneng.com

Floating Point Library IP Core

1 General Description

- ParaCore Architect™ IP Core
- Designed for IEEE 754 single and double precision math
- Designed for custom precision, any exponent and mantissa width
- Pipeline stages configurable via ParaCore parameters
- Full IEEE 754 Special Case tracking
- Parametric IP core for maximum flexibility
- Available in generic HDL or targeted EDIF formats
- Full test bench supplied
- Greater than 200MHz operation in Xilinx Virtex II Pro
- Accepts input data each clock cycle (pipelined version)

The Dillon Engineering Floating Point Library IP Core (FPLIC) is a set of modules designed to perform all floating point math functions. The HDL representation of the FPLIC functions is generated with ParaCore Architect so it can be targeted to any device.

The floating point representation in the DE Floating point unit follows the IEEE 754 standards for both single and double precision. The floating point modules can be pipelined (depth customized) or can be left non-pipelined depending on the requirement.

FPLIC is designed to address the design challenges of digital signal processing in FPGA's today. It is the only floating point IP library that can be tailored made to meet exact needs of your application.

By optimizing the mantissa and exponents lengths for the target technology and application, tremendous gains in dynamic range and precision can be attained versus fixed point results with a minimal impact on device size and cost.

2 IEEE 754 Description

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

2.1 Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit), double (64-bit) precision floating-point, and a custom precision example. The number of bits for each field are shown (bit ranges are in square brackets).

The custom size is shown as an example, any custom size is possible.

Floating Point Number Layout				
	Sign	***Exponent***	***Mantissa***	***Bias***
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023
Custom(e_width=6,m_width=16)	1 [22]	6 [21-16]	16 [15-0]	31

2.1.1 The Sign Bit

Zero is a positive number; one is a negative number. Flipping the value of this bit flips the sign of the number.

2.1.2 The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE 754 single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all zeros) and +128 (all ones) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

For custom precision, the bias is (2**e_width)/2 – 1.

2.1.3 The Mantissa

The mantissa represents the precision bits of the number.

Floating-point numbers are stored in normalized form. This puts the radix point after the first non-zero digit.

An optimization utilized by IEEE 754 is to discard the leading 1 and assume that it exists, saving 1 bit of storage. Thus, the mantissa has effectively 24 bits of resolution in single precision while only 23 bits are stored.

To summarize:

- The sign bit is 0 for positive, 1 for negative.
- The exponent's base is two.
- The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
- The first bit of the mantissa is assumed to be 1, and is not stored explicitly.

2.2 Ranges of Floating-Point Numbers

The range of positive numbers is defined by the following table:

	Range	***Decimal***
Single Precision	2^{-126} to $(2-2^{-23})x2^{127}$	$\sim10^{-37.93}$ to $\sim10^{38.53}$
Double Precision	2^{-1022} to $(2-2^{-52})x2^{1023}$	$\sim10^{-307.65}$ to $\sim10^{308.25}$
Custom (e_width=6, m_width=16)	2^{-30} to $(2-2^{-16})x2^{31}$	$\sim10^{-9.03}$ to $\sim10^{9.63}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that floating-point numbers are **not** able to represent (cases given relate to single precision):

1. Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (negative overflow)
2. Negative numbers greater than -2^{-126} (negative underflow)
3. Zero
4. Positive numbers less than 2^{-126} (positive underflow)
5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (positive overflow)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely ap-

proximated by zero.

Here's a table of the effective range (excluding infinite values) of IEEE floating-point numbers:

	Binary	***Decimal***
Single	$\pm (2\text{-}2^{-23})\text{x}2^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2\text{-}2^{-52})\text{x}2^{1023}$	$\sim \pm 10^{308.25}$
Custom (e_width=6, m_width=16)	$\pm (2\text{-}2^{-16})\text{x}2^{31}$	$\sim \pm 10^{9.63}$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double), and the mantissa is filled with ones (including the normalizing 1 bit).

2.3 Special Cases

IEEE 754 reserves exponent field values of all zeros and all ones to denote special cases in the floating-point scheme.

2.3.1 Zero

Zero is a special value denoted with an exponent field of 0 and a mantissa of 0. Note that -0 and +0 are distinct values, though they both compare as equal.

2.3.2 Infinity

The values +infinity and -infinity are denoted with an exponent of all ones and a mantissa of all zeros. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations.

2.3.3 Indeterminate

The value indeterminate is represented by an exponent of all ones, a mantissa with a leading one followed by all zeros, and a sign bit of one. This value is used to represent results that are indeterminate, such as (infinity - infinity), or (0 x infinity).

2.3.4 Not A Number

Finally, the value NaN (*Not a Number*) is used to represent a value that is an error of some form. This is is represented with an exponent field of all ones, a zero sign

bit, and a non zero mantissa. This is a special value that might be used to denote a variable that doesn't yet hold a value.

2.4 Special Operations

See the individual Fp module section for the special case logic for each of them.

3 Module Descriptions

3.1 Common Information

This section describes features that are common to all FPLIC modules.

3.1.1 Number Format

The basic number format for all FPLIC numbers is <sign><exponent><mantissa>. The sign bit is always the MSB, followed by the exponent, and the the mantissa. Width of exponent and mantissa are customized when module is generated.

3.1.2 Special Case Status

Standard special case tracking, with special cases encoded into all numbers as per IEEE 754, is supported by all modules that require the information.

All modules are built with special case status output ports (port_sp_out). This status port is defined by the following table:

SP[3:0] Bits	***Function***	***Description***
'0001'	Zero	Number is zero. Sign bit in number also applies.
'0010'	INF	Infinity. Sign bit in number also applies.
'0100'	NAN	Not a number.
'1000'	IND	Indeterminate.

If more than a single bit is set, then the number is set to indeterminate.

IEEE 754 encodes this status information into the numbers as described in Section 2. This is the default encoding for FPLIC numbers.

3.1.3 Performance

The number of pipeline stages is customized when the module is generated. There is a trade off between pipeline stages, clock frequency and logic usage. Increasing the pipeline stages will increase the clock frequency and logic usage. Decreasing the pipeline stages decreases the clock frequency and logic usage.

The first pipeline stage is always applied to the output, the second to the input, and the remainder are internal pipelines.

3.1.4 SYNC_IN/SYNC_OUT

Each module has an I/O pair that can be used to track the data through the pipeline stages. SYNC_OUT tracks SYNC_IN delayed by the number of pipeline stages the module in configured to have. If this feature is not used, then tie the SYNC_IN to 0.

3.1.5 I/O Timing

In pipelined versions, all I/O is synchronous with port_clk with a input set on every block and an output on every clock following the pipeline delay.

Versions of each module are available that don't operate on continuous data, thus allowing logic reduction through logic sharing.

3.2 FpAdd/FpSub

In the FpAdd (or FpSub) operation the exponents of the two numbers are compared with each other. The mantissa of the lesser of the exponents is shifted by the difference of the exponents. Then the two mantissa's are added and the resultant exponent is the greater of the two exponents. If a carry is produced, the exponent of the result is shifted to account for the carry which is produced. The sign of the resultant number is the sign of the greater number.

3.2.1 FpAdd Ports

FpAdd/FpSub I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A. Operation is A+B (FpAdd) and A-B (FpSub)
port_b	In	Operand B.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Result.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_sp_out	Out	Special case status for the result.

3.2.2 Special Case Logic

Special case logic for the FpAdd/FpSub modules:

FpAdd/FpSub Special Case Logic				
Precedence	**a**	**b**	**x**	**Notes**
1	NAN	any	NAN	“any” is any Fp number
1	any	NAN	NAN	
2	IND	any	IND	
2	any	IND	IND	
3	±INF	any	INF	x.sign = a.sign
4	any	±INF	INF	x.sign = !b.sign
5	any	any	ZERO	Underflow, sign is 0.
5	any	any	INF	Overflow, sign is 1 for negative overflow, 0 for positive.
5	any	any	a±b	Valid result

3.3 FpMult

In the FpMult operation, the exponents of the two numbers are added and the result of this becomes the resultant exponent. The mantissa's are multiplied together to produce the resultant mantissa. The exponent is adjusted appropriately after the mantissa is normalized.

3.3.1 FpMult Ports

FpMult I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A.
port_b	In	Operand B.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.

FpMult I/O Ports		
Port	**Dir**	**Function**
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Result.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_sp_out	Out	Special case status for the result.

3.3.2 Special Case Logic

Special case logic for the FpMult modules:

FpMult Special Case Logic				
Precedence	**a**	**b**	**x**	**Notes**
1	IND	any	IND	“any” is any Fp number
1	any	IND	IND	
2	NAN	any	NAN	
2	any	NAN	NAN	
3	±INF	any	INF	x.sign = a.sign ^ b.sign
4	any	±INF	INF	x.sign = a.sign ^ b.sign
5	any	any	ZERO	Underflow, sign is 0.
5	any	any	INF	Overflow, x.sign = a.sign ^ b.sign
5	any	any	a*b	Valid result

3.4 FpDiv

The FpDiv in FPLIC is done using the Goldschmidt's Algorithm. In this algorithm the result of the division is calculated by multiplying the numerator and denominator with a series of approximations for the reciprocal of the denominator. The mantissa of the numerator is divided by the mantissa of the denominator in this fashion. The resultant exponent is the difference between the exponents of the numerator and the denominator.

3.4.1 FpDiv Ports

FpDiv I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A. Operation is A/B.
port_b	In	Operand B.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Result.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_sp_out	Out	Special case status for the result.

3.4.2 Special Case Logic

Special case logic for the FpDiv modules:

FpDiv Special Case Logic				
Precedence	**a**	**b**	**x**	**Notes**
1	NAN	any	NAN	“any” is any Fp number
1	any	NAN	NAN	
2	IND	any	IND	
2	any	IND	IND	
3	any	±ZERO	INF	x.sign = a.sign ^ b.sign
4	±ZERO	any	ZERO	x.sign = a.sign ^ b.sign
5	any	±INF	ZERO	x.sign = a.sign ^ b.sign
5	±INF	any	INF	x.sign = a.sign ^ b.sign
6	any	any	ZERO	Underflow, sign is 0.
6	any	any	INF	Overflow, x.sign = a.sign ^ b.sign

FpDiv Special Case Logic				
6	any	any	a*b	Valid result

3.5 Int2Fp

Int2Fp takes a integer and converts it to a floating point number. The width of the integer and size of floating point number are customized when the module is generated.

The preprocessing takes the two's complement of a negative number and keeps track of the sign bit. The positive integer is normalized so that the most significant 1 becomes the hidden bit and the exponent is set to reflect the number of shifts.

3.5.1 Int2Fp Ports

In2Fp I/O Ports		
Port	**Dir**	**Function**
port_a	In	Integer input.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Floating point output.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.

3.5.2 Special Case Logic

No special cases for this module.

3.6 Fp2Int

Fp2Int takes as input a floating point number and convert it to an integer.

The conversion barrel shifts the mantissa using the exponent and integer width to set the shift amount, then post processes the shifted result to encode the special cases.

3.6.1 Fp2Int Ports

Fp2Int I/O Ports		
Port	**Dir**	**Function**
port_a	In	Floating point input.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_sp_a	In	Special case status for operand A.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Integer output.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_oflow	Out	Indicates Fp number was too big for integer.

3.6.2 Special Case Logic

No special cases for this module.

3.7 FpSqrt

The FpSqrt (square root) in FPLIC is done using the restoring shift/subtract algorithm. In this algorithm the result of the square root operation is calculated by multiplying the series of approximations for the root of reciprocal of the number.

3.7.1 FpSqrt Ports

FpSqrt I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A, radicand.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Result square root.

FpSqrt I/O Ports		
Port	**Dir**	**Function**
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_sp_out	Out	Special case status for the result.

3.7.2 Special Case Logic

Special case logic for the FpSqrt modules:

FpSqrt Special Case Logic			
Precedence	**a**	**x**	**Notes**
1	NAN	NAN	
2	IND	IND	
3	-any	IND	Negative input produces IND
4	INF	IND	
5	any	$\sqrt{a}$	Valid result

3.8 FpRcp

The FpRcp in FPLIC is done using the Goldschmidt's Algorithm. In this algorithm the result of the division is calculated by multiplying the numerator and denominator with a series of approximations for the reciprocal of the denominator. The mantissa of the numerator is divided by the mantissa of the denominator in this fashion. The resultant exponent is the difference between the exponents of the numerator and the denominator.

3.8.1 FpRcp Ports

FpRcp I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A.
port_sync_in	In	Input used to track the pipeline delay, does not effect math operation only used to create port_sync_out.
port_clk	In	Clock input, only used on pipelined versions. All I/O are synchronous to the clock.
port_x	Out	Result.
port_sync_out	Out	Delayed version of port_sync_in, delayed by pipeline depth.
port_sp_out	Out	Special case status for the result.

3.8.2 Special Case Logic

Special case logic for the FpRcp modules:

FpRcp Special Case Logic			
Precedence	**a**	**x**	**Notes**
1	NAN	NAN	“any” is any Fp number
2	IND	IND	
3	±INF	ZERO	x.sign = a.sign ^ b.sign
4	±ZERO	INF	x.sign = a.sign ^ b.sign
5	any	$1/a$	Valud result

3.9 FpSplit

FpSplit is used to split a Fp number into the sign, exponent and mantissa.

3.9.1 FpSplit Ports

FpRcp I/O Ports		
Port	**Dir**	**Function**
port_fp	In	Fp number to be split.
port_sign	Out	Sign of Fp number.
port_exp	Out	Exponent of Fp number.
port_man	Out	Mantissa of Fp number.

3.9.2 Special Case Logic

No special cases for this module.

3.10 FpCat

The FpCat module is used to construct a Fp vector from the sign, exponent and mantissa.

3.10.1 FpCat Ports

FpCat I/O Ports		
Port	**Dir**	**Function**
port_sign	In	Sign input
port_exp	In	Expoent
port_man	In	mantissa
port_fp	Out	Floating point number concatinated from the sign, exp, and man.

3.10.2 Special Case Logic

No special cases for this module.

3.11 FpCmp

The FpCmp is used to compare two Fp numbers, eq (equal), gt (greater than), lt (less than) and er (error) are produced.

3.11.1 FpCmp Ports

FpCmp I/O Ports		
Port	**Dir**	**Function**
port_a	In	Operand A.
port_b	In	Operand B.
port_clk	In	clock
port_er	Out	Error, a or b IND or NAN
port_eq	Out	A == B is True
port_gt	Out	A > B is True
port_lt	Out	A < B is True

3.11.2 Special Case Logic

No special cases for this module.

4 ParaCore Options

The DE FPLIC is very flexible since it was designed using ParaCore Architect. All configuration parameters are used to customize the module when it is generated.

The parameters for all Math operations are similar and defined by the following table.

Math Operation Parameters		
Parameter	***Type***	***Description***
exp_width	integer	Sets the exponent width for a floating point I/O to the module.
man_width	integer	Sets the mantissa width for a floating point I/O to the module.
width	integer	Set the integer width for any integer I/O of the module.
stages	integer	Defines the number of pipeline stages in the module. Stage == 0 is valid and results in a completely combinatorial module (clk is connected but not used). Stages > 15 is valid but will not increase the clock frequency of the module.

All modules are built by and delivered to the clients specifications.

Bit accurate C models are available for all modules.

5 FPGA Area Usage

The following section give a few samples of area usage in a couple of FPGA technologies.

5.1 Virtex II area Usage

The following is the area used by the FPLIC in a Virtex II FPGA. Units are in slices with mult18x18s in parenthesis if used.

Virtex II Area Usage (Slice)			
Function	***Single Precision***	***Double Precision***	***e_width=6,m_width=16***
FpAdd/FpSub	525	3100	503
FpMult	141 (4)	458 (9)	79 (1)
FpDiv	1200 (28)	4600 (112)	465

5.2 Altera Area Usage

The following is the area used by the FPLIC in Altera Apex FPGAs.

Apex Usage (Slice)			
Function	***Single Precision***	***Double Precision***	***e_width=6,m_width=16***
FpAdd/FpSub	1673	6742	1022
FpMult	1360	6009	737
FpDiv	12397	55178	7410