FPLibrary v0.94
User documentation

Jérémie Detrey Florent de Dinechin

LIP – ÉNS Lyon
46, allée d'Italie
69364 Lyon cedex 07
France
{Jeremie.Detrey,Florent.de.Dinechin}@ens-lyon.fr

Contents

Chapter 1

Introduction

1.1 Description

FPLibrary is a library of parameterizable arithmetic operators for "real" numbers, such as floating-point numbers. It supports both floating-point and logarithmic number systems, and provides classical arithmetic operators ($+/-$, $\times$, $\div$ and $\sqrt{\ }$) along with some conversion operators for a number system to an other.

The whole library is written in portable VHDL code, mainly targeted for FPGAs (Field-Programable Gate Arrays). All operators are parameterizable in terms of precision and range for their operands and result, and are available in both combinatorial or pipelined flavours.

1.2 Installation

The latest version of FPLibrary can be freely downloaded from `http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/` as a tar-gzipped archive of the VHDL source tree.

To extract this archive:

```
$ tar xzvf FPLibrary-0.94.tgz
```

This will create the following directories:

`FPLibrary-0.94/doc/`	contains the documentation (*i.e.* this file),
`FPLibrary-0.94/vhdl/`	contains the source code of the library,
`FPLibrary-0.94/misc/`	contains some files for integrating FPLibrary to a VHDL project.

1.3 Usage

1.3.1 Synthesizing the library

In this section you will find how to synthesise the VHDL source of the library in order to import the operators it provides in your own designs. However, this procedure strongly depends on the VHDL environment you are using. We can only give the detailed procedure for Xilinx ISE and XST, but we hope the general guidelines will be precise enough for other environments.

Remark: if you feel like contributing to this section, you can send us the procedures for the missing VHDL environments, we will be glad to integrate them here.

General guidelines

First, you should integrate all the VHDL source files into your project as a library called `fplib`. This is very important, because the default working library for a VHDL project is `work`. You perhaps need to create the library `fplib` beforehand and then add the whole source tree (including all sub-directories) to this library.

The proper order of compilation from the file hierarchy, in case the synthesizer cannot figure it out by itself.

Finally, you will perhaps need to explicitly synthesize the library, but most synthesizers will probably automatically do so when synthesizing a design using FPLibrary operators.

Xilinx ISE

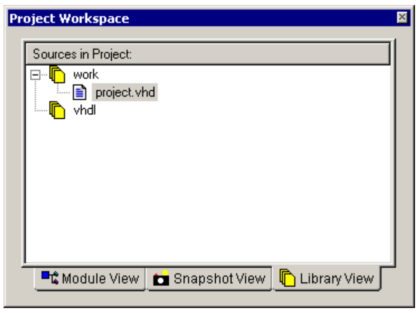
This section guides you through the different steps required to integrate FPLibrary to your project using Xilinx ISE tools and their graphical frontend (Project Navigator).

Before starting, you should have opened your project in the Xilinx Project Navigator. You should focus on the Sources window (you can show/hide it from the View menu), and more precisely on the Library View tab (Figure 1.1(a)).

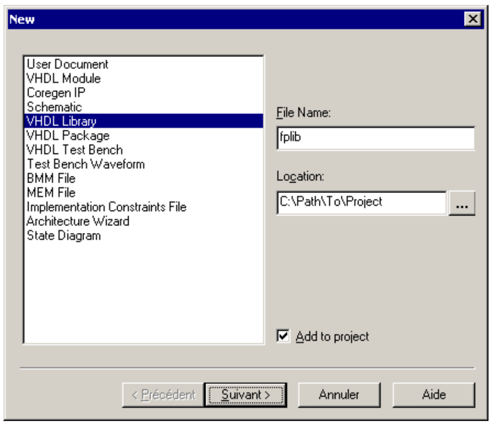
Then, right-click in this tab, and select New Source.... In the dialog box that opens, select VHDL Library for the type of source, and type `fplib` for the file name (Figure 1.1(b)). Make sure that the Add to project box is ticked, and click on Next and then Finish. Now the `fplib` library is created.

Third, you need to add all the VHDL sources of FPLibrary to `fplib`. Right-click on the fplib library, and select Add Source... (Figure 1.1(c)), and select all the FPLibrary source files. Note that this operation does not recursively add all sub-folders, so you will have to do this manually[1].

Eventually, when all the source files are added, FPLibrary will be ready to be used (Figure 1.1(d)). ISE will automatically synthesize the library when sythesizing your project.

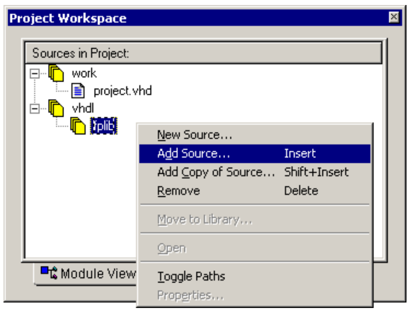


(a) Library View tab.

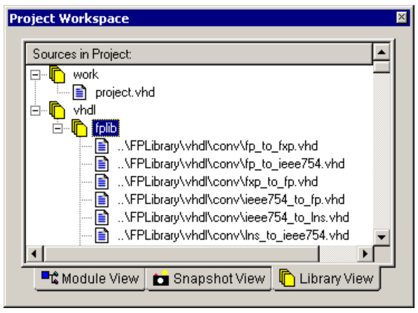


File Name: fplib
Location: C:\Path\To\Project
☑ Add to project

(b) Creating the library.



(c) Adding source files.



```
..\FPLibrary\vhdl\conv\fp_to_fxp.vhd
..\FPLibrary\vhdl\conv\fp_to_ieee754.vhd
..\FPLibrary\vhdl\conv\fxp_to_fp.vhd
..\FPLibrary\vhdl\conv\ieee754_to_fp.vhd
..\FPLibrary\vhdl\conv\ieee754_to_lns.vhd
..\FPLibrary\vhdl\conv\lns_to_ieee754.vhd
```

(d) FPLibrary is loaded.

Figure 1.1: Xilinx Project Navigator screenshots.

[1]This task is quite painful, but an ugly patch of the `.npl` project file can do the trick:
Edit the project file and look for a line that should be something like: `SUBLIB fplib VhdlLibrary vhdl`
After this line, insert the contents of file `FPLibrary-0.94/misc/ise/npl_patch` and replace the dummy path `PATH:\` by the actual path to the FPLibrary source code.
Then, you just have to reload your project to take these modifications into account.

XST (Xilinx Synthesis Technology)

This section describes the FPLibrary installation steps for those who use Xilinx tools in a command-line fashion, and therefore do not want to create a project using ISE.

To use the library, you just have to use the file `FPLibrary-0.94/misc/xst/fplib.prj` and merge it into your own `.prj` file.

You can also compile each VHDL source file separatly using a script, as the order of compilation is given by `FPLibrary-0.94/misc/xst/fplib.prj`. Be sure to compile the FPLibrary files into the `fplib` library, using the command-line option `-work_lib fplib`.

1.3.2 Using FPLibrary operators

To use FPLibrary operators in your own circuits, just add the following lines to the source code of the concerned components:

```
library fplib;
use fplib.pkg_fplib.all;
```

You then just have to instantiate the operators in the usual way. If FPLibrary was correctly integrated into your project as the `fplib` library (as described in Section 1.3.1), the synthesizer will automatically import them.

See Sections 3 and 4 for a complete description of the operators and their interfaces.

1.4 Examples

A few simple examples can be found in the directory`FPLibrary-0.94/vhdl/test/examples`.

Chapter 2

Number representation formats

2.1 Floating-point

The floating-point (FP) number format used in FPLibrary is mainly inspired from the IEEE-754 standard [2]. Its main idea is to represent numbers with a fixed-point normalized mantissa multiplied by an order of magnitude (an integer power of 2). For example: 1.25×2^{21}, -1.75×2^{18}, 1.00×2^{-5}, ...

This representation is parameterized by two bitwidths w_E and w_F. An FP number X is then represented as a vector of $w_E + w_F + 3$ bits, and can be partitioned in 4 fields as shown Figure 2.1:

- exn (2 bits): the exception tag;
- S_X (1 bit): the sign bit;
- E_X (w_E bits): the exponent (biased by $E_0 = 2^{w_E-1} - 1$);
- F_X (w_F bits): the fraction (mantissa).

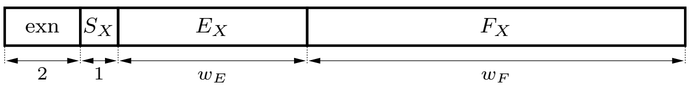


Figure 2.1: Floating-point number format.

The value of X is given according to the exception tag:

- zero ($\mathrm{exn} = 00$): $X = (-1)^{S_X} \times 0$
  note that 0 is signed, as stated in the IEEE-754 standard;
- normalized number ($\mathrm{exn} = 01$): $X = (-1)^{S_X} \times 1.F_X \times 2^{E_X - E_0}$;
- infinity ($\mathrm{exn} = 10$): $X = (-1)^{S_X} \times \infty$;
- not-a-number ($\mathrm{exn} = 11$): $X = \mathrm{NaN}$
  NaN is an undefined value, such as $0/0$ or $\infty - \infty$.

Remark: FPLibrary does not handle the denormalized numbers described by the IEEE-754 standard.

2.2 Logarithmic number system

The logarithmic number system (LNS) was first introduced by Schwartzlander in [3]. The idea here is to use a fixed-point exponent instead of a mantissa. For example: $2^{42.25}$, $2^{-12.50}$, $-2^{23.75}$, ...

In FPLibrary, this representation is also parameterized by w_E and w_F, and an LNS number X is represented as a vector of $w_E + w_F + 3$ bits, and can be partitioned in 4 fields as shown Figure 2.2:

- exn (2 bits): the exception tag;
- S_X (1 bit): the sign bit;
- E_{L_X} (w_E bits): the integer part of the logarithm L_X;
- F_{L_X} (w_F bits): the fractional part of the logarithm.

Remark: the logarithm $L_X = E_{L_X}.F_{L_X}$ is represented in 2's complement.

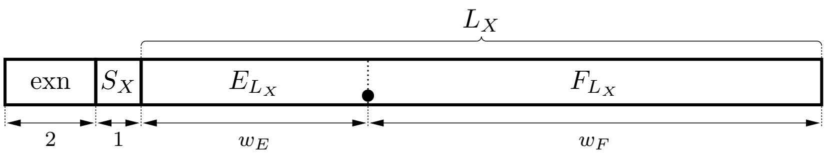


Figure 2.2: Logarithmic number format.

As for FP numbers, the exception tag dictates the value of X:

- zero ($\mathrm{exn} = 00$): $X = (-1)^{S_X} \times 0$;
- general case ($\mathrm{exn} = 01$): $X = (-1)^{S_X} \times 2^{L_X}$ where $L_X = E_{L_X}.F_{L_X}$;
- infinity ($\mathrm{exn} = 10$): $X = (-1)^{S_X} \times \infty$;
- not-a-number ($\mathrm{exn} = 11$): $X = \mathrm{NaN}$.

Chapter 3

Arithmetic operators

3.1 Addition / subtraction

3.1.1 Synopsis

```
component Add is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13 );
  port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
         nB : in  std_logic_vector(wE+wF+2 downto 0);
         nR : out std_logic_vector(wE+wF+2 downto 0) );
end component;

component Add_Clk is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13;
            reg : boolean  := true );
  port ( nA  : in  std_logic_vector(wE+wF+2 downto 0);
         nB  : in  std_logic_vector(wE+wF+2 downto 0);
         nR  : out std_logic_vector(wE+wF+2 downto 0);
         clk : in  std_logic );
end component;

function addLatency( fmt    : format;
                     wE, wF : positive ) return natural;
```

3.1.2 Parameter mapping

Generic parameters

fmt The number system for the operands and result. Should be set to either `FP` or `LNS`.

wE The value of w_E used in the representation of the operands and result (default: `6`).

wF The value of w_F used in the representation of the operands and result (default: `13`).

reg Indicates if the design should be pipelined or not (default: `yes`).

Signal ports

nA, nB The two operands A and B.

nR The result $R = A + B$.

`clk` The clock signal.

3.1.3 Description

`Add` The pure combinatorial version of the addition/subtraction operator.

`Add_Clk` The combinatorial version if `reg` is set to `no` or the pipelined version if `reg` is set to `yes`.

`addLatency` The number of stages of the pipelined version of the operator.

Remark: the result is rounded to nearest for FP operators, but this rounding cannot be achieved in LNS. See [1] for more details.

3.2 Multiplication

3.2.1 Synopsis

```
component Mul is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13 );
  port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
         nB : in  std_logic_vector(wE+wF+2 downto 0);
         nR : out std_logic_vector(wE+wF+2 downto 0) );
end component;

component Mul_Clk is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13;
            reg : boolean  := true );
  port ( nA  : in  std_logic_vector(wE+wF+2 downto 0);
         nB  : in  std_logic_vector(wE+wF+2 downto 0);
         nR  : out std_logic_vector(wE+wF+2 downto 0);
         clk : in  std_logic );
end component;

function mulLatency( fmt    : format;
                     wE, wF : positive ) return natural;
```

3.2.2 Parameter mapping

Generic parameters

fmt The number system for the operands and result. Should be set to either `FP` or `LNS`.

wE The value of w_E used in the representation of the operands and result (default: `6`).

wF The value of w_F used in the representation of the operands and result (default: `13`).

reg Indicates if the design should be pipelined or not (default: `yes`).

Signal ports

nA, nB The two operands A and B.

nR The result $R = A \times B$.

clk The clock signal.

3.2.3 Description

Mul The pure combinatorial version of the multiplication operator.

Mul_Clk The combinatorial version if `reg` is set to `no` or the pipelined version if `reg` is set to `yes`.

mulLatency The number of stages of the pipelined version of the operator.

Remark: the result is rounded to nearest for both FP and LNS operators.

3.3 Division

3.3.1 Synopsis

```
component Div is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13 );
  port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
         nB : in  std_logic_vector(wE+wF+2 downto 0);
         nR : out std_logic_vector(wE+wF+2 downto 0) );
end component;

component Div_Clk is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13;
            reg : boolean  := true );
  port ( nA  : in  std_logic_vector(wE+wF+2 downto 0);
         nB  : in  std_logic_vector(wE+wF+2 downto 0);
         nR  : out std_logic_vector(wE+wF+2 downto 0);
         clk : in  std_logic );
end component;

function divLatency( fmt    : format;
                     wE, wF : positive ) return natural;
```

3.3.2 Parameter mapping

Generic parameters

`fmt` The number system for the operands and result. Should be set to either `FP` or `LNS`.

`wE` The value of w_E used in the representation of the operands and result (default: `6`).

`wF` The value of w_F used in the representation of the operands and result (default: `13`).

`reg` Indicates if the design should be pipelined or not (default: `yes`).

Signal ports

`nA` The dividend A.

`nB` The divisor B.

`nR` The result $R = A/B$.

`clk` The clock signal.

3.3.3 Description

`Div` The pure combinatorial version of the division operator.

`Div_Clk` The combinatorial version if `reg` is set to `no` or the pipelined version if `reg` is set to `yes`.

`divLatency` The number of stages of the pipelined version of the operator.

Remark: the result is rounded to nearest for both FP and LNS operators.

3.4 Square root

3.4.1 Synopsis

```
component Sqrt is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13 );
  port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
         nR : out std_logic_vector(wE+wF+2 downto 0) );
end component;

component Sqrt_Clk is
  generic ( fmt : format;
            wE  : positive := 6;
            wF  : positive := 13;
            reg : boolean  := true );
  port ( nA  : in  std_logic_vector(wE+wF+2 downto 0);
         nR  : out std_logic_vector(wE+wF+2 downto 0);
         clk : in  std_logic );
end component;

function sqrtLatency( fmt    : format;
                      wE, wF : positive ) return natural;
```

3.4.2 Parameter mapping

Generic parameters

fmt The number system for the operand and result. Should be set to either `FP` or `LNS`.

wE The value of w_E used in the representation of the operand and result (default: `6`).

wF The value of w_F used in the representation of the operand and result (default: `13`).

reg Indicates if the design should be pipelined or not (default: `yes`).

Signal ports

nA The operand A.

nR The result $R = \sqrt{A}$.

clk The clock signal.

3.4.3 Description

Sqrt The pure combinatorial version of the division operator.

Sqrt_Clk The combinatorial version if `reg` is set to `no` or the pipelined version if `reg` is set to `yes`.

sqrtLatency The number of stages of the pipelined version of the operator.

Remark: the result is rounded to nearest for both FP and LNS operators.

Chapter 4

Conversion operators

4.1 Fixed-point/floating-point conversions

4.1.1 Synopsis

```
component FXP_To_FP is
  generic ( wE    : positive := 6;
            wF    : positive := 13;
            wFX_I : positive := 6;
            wFX_F : positive := 13 );
  port ( nA : in  std_logic_vector(wFX_I+wFX_F-1 downto 0);
         nR : out std_logic_vector(wE+wF+2 downto 0));
end component;

component FP_To_FXP is
  generic ( wE    : positive := 6;
            wF    : positive := 13;
            wFX_I : positive := 6;
            wFX_F : positive := 13 );
  port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
         nR : out std_logic_vector(wFX_I+wFX_F-1 downto 0));
end component;
```

4.2 Conversion between IEEE754 and FPLibrary floating-point formats

The IEEE754 format is the standard format used in most PCs. Single precision is (wE=8, wF=23), double-precision is (wE=11, wF=52). This format is more memory-efficient, since it encodes infinities and zeros respectively as the largest and smallest values of the exponent. Conversely, the internal FPLibrary format is more hardware-efficient. It codes these special values as special bits, and therefore doesn't need to decode and encode special exponent values into the corresponding bits at each operation.

Therefore, a pipeline of operators should use the FPLibrary format. Conversion from and to the IEEE754 format should be performed essentially at input and output.

Remark: If you store single precision data in internal memory blocks such as BlockRams or M9K, you will have 36 bits to store a single precision data, which means that you may keep the single precision FPLibrary format (8+23+3=34 bits).

4.2.1 Synopsis

```
component IEEE754_To_FP is
  generic ( wE : positive := 6;
            wF : positive := 13 );
```

```
    port ( nA : in  std_logic_vector(wE+wF downto 0);
           nR : out std_logic_vector(wE+wF+2 downto 0));
  end component;

  component FP_To_IEEE754 is
    generic ( wE : positive := 6;
              wF : positive := 13 );
    port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
           nR : out std_logic_vector(wE+wF downto 0));
  end component;
```

4.2.2 Description

IEEE754_To_FP Converts the IEE-754 format on wF+wE+1 bits into the internal FPLibrary format on wF+wE+3 bits.

FP_To_IEEE754 Converts the internal FPLibrary format on wF+wE+3 bits into the IEE-754 format on wF+wE+1 bits.

4.3 Conversion between IEEE754 and LNS formats

4.3.1 Synopsis

```
  component IEEE754_To_LNS is
    generic ( wE : positive := 6;
              wF : positive := 13 );
    port ( nA : in  std_logic_vector(wE+wF downto 0);
           nR : out std_logic_vector(wE+wF+2 downto 0));
  end component;

  component LNS_To_IEEE754 is
    generic ( wE : positive := 6;
              wF : positive := 13 );
    port ( nA : in  std_logic_vector(wE+wF+2 downto 0);
           nR : out std_logic_vector(wE+wF downto 0));
  end component;
end package;
```

Bibliography

[1] J. Detrey and F. de Dinechin. A VHDL library of LNS operators. In *37th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, USA, October 2003.

[2] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985, 1985.

[3] E. E. Swartzlander and G. Alexopoulos. The sign/logarithm number system. *IEEE Transactions on Computers*, 24(12):1238–1242, December 1975.