Celoxica

Platform Developer's Kit

Pipelined Floating-point Library Manual

Authors: RG

Document number: 1

Customer Support at http://www.celoxica.com/support/

Celoxica in Europe	Celoxica in Japan	Celoxica in the Americas
T: +44 (0) 1235 863 656	T: +81 (0) 45 331 0218	T: +1 800 570 7004
E: sales.emea@celoxica.com	E: sales.japan@celoxica.com	E: sales.america@celoxica.com

Contents

Assumptions & Omissions

This manual assumes that you:

- have used Handel-C or have the Handel-C Language Reference Manual
- are familiar with common programming terms (e.g. functions)
- are familiar with your operating system (Linux or MS Windows)

This manual does not include:

- instruction in VHDL or Verilog
- instruction in the use of place and route tools
- tutorial example programs. These are provided in the Handel-C User Manual

Conventions

The following conventions are used in this document.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

copy ***SourceFileName DestinationFileName***

Optional elements are enclosed in square brackets like this:

```
struct [type_Name]
```

Curly brackets around an element show that it is optional but it may be repeated any number of times.

string ::= "{***character***}"

1 Pipelined floating-point library

The pipelined floating-point library is a platform-independent core. It allows you to perform floating-point operations in a pipelined manner on single-precision floating-point numbers.

The library contains:

- The `Float` type: type for a single-precision floating-point number.
- Macro procedures to perform arithmetic operations on floating-point numbers. These are useful if you want to use the operations within pipelined code.
- Macros to create "cores" which perform arithmetic operations on floating-point numbers. These are useful if you want to use operations within sequential code.
- Macros for casting and converting between integers and floating-point numbers.
- `FloatPipeChipType`***`X`*** macros: optimize code by allowing embedded cores to be used for floating-point operations

To use the library, you need to include `float_pipe.hch` in your code and link to `float_pipe.hcl` and `stdlib.hcl`. The libraries are in ***`InstallDir`***`/PDK/Hardware/Lib`.

1.1 Float type

```
Float;
```

Description

The `Float` type is the single-precision floating-point format handled by the Pipelined floating-point library. You use it in a similar way to a float type in software. Accessing of members is done by treating the `Float` type as a `struct`.

Example

```
Float A, B, C[64];
...
if (A.Sign == Negative)
    {
        par
       {
           A.Exponent = B.Exponent;
           A.Mantissa = B.Mantissa;
       }
        ...
    }
```

1.2 Floating-Point Introduction

Floating-point numbers

This is a single precision pipelined floating-point library. Each number consists of a one bit sign, an eight bit exponent and a 23 bit fraction.

The sign bit :	0 denotes a positive number and 1 denotes a negative number.
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 single-precision floats as used in this library this value is 127. For example a stored exponent value of 150 is an exponent of 200-150= 50.
The mantissa:	The mantissa or significand is the precision bit of the number is usually stored in normalized form. This means there is always one digit in front of the binary point. As this is always there we can save space by assuming its presence and not representing it explicitly, as a result the mantissa has effectively 24 bits of resolution.

Special reserved values

Some exponent values are reserved to denote special values in floating point. These can all be represented using the floating-point library.

See FloatNegX and FloatPosX common floating-point values and FloatIsSNAN() and FloatISQNAN()

Zero:	Zero is denoted by an exponent field of zero and a fraction field of zero. The sign bit denotes -0 and +0, these are equivalent.
Infinity:	An exponent of all 1's and a fraction of all 0's. The sign bit denotes -infinity and +infinity.
Not a Number (NAN):	represented as an exponent of all ones and a non-zero fraction. There are two types of NaN: QNaN (Quiet NaN) and SNaN (Signalling NaN). QNaN's denote indeterminate operations, while SNaN's denote invalid operations. A QNaN is a NaN with the most significant fractional bit set. An SNaN is a NaN with the most significant fractional bit clear.

1.3 FloatNegX and FloatPosX: common floating-point values

```
extern macro expr FloatNegInf();
extern macro expr FloatNegOne();
extern macro expr FloatNegOneHalf();
extern macro expr FloatNegZero();
extern macro expr FloatPosZero();
extern macro expr FloatPosOneHalf();
extern macro expr FloatPosOne();
extern macro expr FloatPosInf();
```

Parameters:	None.
Timing:	Compile-time.
Description:	`Float` values representing the associated numbers. The "Neg" or "Pos" indicates the sign bit value.

Example

```
Float A;
A = FloatPosOne ();
```

Macro:	**Description:**
FloatNegInf()	Floating-point minus infinity.
FloatNegOne()	Floating-point minus one (-1).
FloatNegOneHalf()	Floating-point minus a half (-.5).
FloatNegZero()	Floating-point minus one (-1).
FloatPosZero()	Floating-point zero (0).
FloatPosOneHalf()	Floating-point a half (.5).
FloatPosOne()	Floating-point one (1).
FloatPosInf()	Floating-point infinity.

1.4 FloatIsSNaN() and FloatIsQNaN()

```
extern macro expr FloatIsSNaN (f); /* Invalid */
extern macro expr FloatIsQNaN (f); /* Indeterminate */
```

Parameters:	***f***: Floating-point value of type `Float`.
Timing:	0 clock cycles or more.
Description:	Returns true for the check specified by the expression name.

Example

```
Float A;
if (FloatIsSNaN (A))
{
    ...
}
```

1.5 Converting between Integer and floating-point numbers

The following macros can be used to convert between integers and floating-point numbers when you are using the Pipelined floating-point library:

Converting recalculates the value. These operations take a number of clock cycles to produce a result and are pipelined.

Macros are also provided in the Pipelined floating-point library to 'pack' a floating-point number into an int container and unpack an int to a floating-point number. These macros are useful for passing values between Handel-C and C/C++ for simulation.

1.5.1 Packing and unpacking floating-point to an integer container

The following macros can be used to pack a floating point number into an integer and unpacking an integer representation to a floating-point type when you are using the Pipelined floating-point library.

These macros are useful in simulation when passing values from Handel-C to C or C++.

The order of the "int 32" representation is:

Bits:	**Description:**
[31]	Sign
[30:23]	Exponent
[22:0]	Mantissa

- FloatPackInInt32(): Creates a 32-bit int representation of the floating-point value.
- FloatUnpackFromInt32(): Creates a float from a 32-bit representation of a floating-point number.

Macros are also provided in the pipelined floating-point library for converting between integer and floating-point.

FloatUnpackFromInt32() macro

```
extern macro expr FloatUnpackFromInt32 (B);
```

Parameters:	***B***: Value of type `int 32`.
Timing:	0 clock cycles or more.
Description:	Returns a floating point value (Float) from an "int 32" value representation.The floating-point value is unpacked from an "int 32" type. The order of the int 32 representation is (from MSB): bit [31] Sign bits [30:23] Exponent bits [22:0] Mantissa

Example

```
while (1)
    {
        par
       {
            Float  A;
            int 32 B;
            A = FloatUnpackFromInt32 (B);
            ...
        }
    }
```

Macros are also provided in the pipelined floating-point library for converting between integer and floating-point.

FloatPackInInt32() macro

```
extern macro expr FloatPackInInt32 (A);
```

Parameters:	***A***: Value of type `Float`.
Timing:	0 clock cycles or more.
Description:	Creates a 32-bit int representation of the floating-point value. The floating-point value is packed into a "int 32" type. The order of the int representation is (from MSB): bit [31] Sign bits [30:23] Exponent bits [22:0] Mantissa

Example

```
Float  A;
int 32 B;
B = FloatPackInInt32 (A);
```

Macros are also provided in the pipelined floating-point library for converting between integer and floating-point.

1.5.2 FloatPipeToInt() and FloatPipeToUInt()

```
extern macro proc FloatPipeToInt (A, QPtr, IsInt64);
extern macro proc FloatPipeToUInt (A, QPtr, IsUInt64);
```

Parameters:	***A***: Floating-point value of type `Float`.
	QPtr: Integer of type `int 32` or `int 64` for `FloatPipeToInt()`, and `unsigned 32` or `unsigned 64` for `FloatPipeToUInt()`.
	IsInt64: Compile-time constant. If ***IsInt64*** is 1 this indicates that parameter ***QPtr*** is of type `int 64` or `unsigned 64`.
Timing:	10 clock cycles or more for `FloatPipeToInt()`.
	9 clock cycles or more for `FloatPipeToUInt()`.
Description:	`FloatPipeToInt()` converts a floating-point value of type Float to `int 32` or `int 64`.
	`FloatPipeToUInt()` converts a floating-point value of type `Float` to `unsigned 32` or `unsigned 64` storage.
	IsInt64 determines whether the storage created for the int or unsigned number is 32 or 64-bit.

Example

```
int 64 Q1;
unsigned 32 Q2;
Float A[2];

while (1)
{
    par
    {
        A[0] = FloatUnpackFromInt32 (ABus.in);
        A[1] = FloatUnpackFromInt32 (BBus.in);
        FloatPipeToInt  (A[0], &Q1, 1);
        FloatPipeToUInt (A[1], &Q2, 0);
        ...
    }
}
```

1.6 FloatPipeFromInt() macro

`extern macro proc FloatPipeFromInt (`***`A`***`, `***`QPtr`***`, `***`IsInt64`***`);`

Parameters:	***A***: Integer of type `int 32` or `int 64`.
	QPtr: Value of type `Float` to store the conversion.
	IsInt64: Compile-time constant. If ***IsInt64*** is 1 this indicates that parameter ***A*** is of type `int 64`.
Timing:	8 clock cycles or more.
Description:	Converts a signed integer of type `int 32` or `int 64` to a single precision floating-point value of type `Float`. ***IsInt64*** should be set to 1 for integers of type `int 64`.

Example

```
int 64 A;
Float  Q;
while (1)
    {
        par
        {
            A = ABus.in;
            FloatPipeFromInt (A, &Q, 1);
            ...
        }
    }
```

1.7 FloatPipeX: Floating-point arithmetic macros

```
extern macro proc FloatPipeAdd (A, B, QPtr, Carry, APtr, BPtr);
extern macro proc FloatPipeSub (A, B, QPtr, Carry, APtr, BPtr);
extern macro proc FloatPipeMult (A, B, QPtr, Carry, APtr, BPtr, Clock, ChipType);
extern macro proc FloatPipeDiv (A, B, QPtr, Carry, APtr, BPtr);
extern macro proc FloatPipeSqrt (A, QPtr, Carry, PPtr);
extern macro proc FloatPipeEq (A, B, QPtr);
extern macro proc FloatPipeGt (A, B, QPtr);
extern macro proc FloatPipeLs (A, B, QPtr);
```

Parameters:	***A***: Value of type `Float` as parameter 1.
	B: Value of type `Float` as parameter 2.
	QPtr: Pointer of type `Float` or `unsigned 1` for return value.
	Carry: Value indicating whether parameters need to be pipelined and returned with the result. Possible values:
	0 - no parameters
	1 - first parameter
	2 - second parameter
	3 - both parameters
	APtr: Pointer of type `Float` for return of pipelined parameter 1.
	BPtr: Pointer of type `Float` for return of pipelined parameter 2.
	Clock: Clock signal.
	ChipType: Value identifying the chip type. Set this using the `FloatPipeChipType`***`X`*** macros.
Timing:	5 clock cycles or more.
Description:	Instantiates an operator pipeline identified by the name of the procedure. Each operator has a different latency which can be queried using the `FloatPipe`***`X`***`Cycles` macros.
	The ***Carry*** parameter helps to pipeline the inputs into the operation so that they can be extracted with the result.
	If you want to perform arithmetic operations on floating-point numbers within sequential code, use the ***floating-point arithmetic cores*** (see page 16) instead.

Example

```
Float A, B, Q, Ar, Br;
while (1)
    {
        par
        {
             A = FloatUnpackFromInt32 (ParameterABus.in);
             B = FloatUnpackFromInt32 (ParameterBBus.in);
             FloatPipeSub (A, B, &Q, 3, &Ar, &Br);
         }
     }
```

Macro:	**Description:**
FloatPipeAdd()	Pipelined floating-point adder
FloatPipeSub()	Pipelined floating-point subtracter
FloatPipeMult()	Pipelined floating-point multiplier

FloatPipeDiv()	Pipelined floating-point divider
FloatPipeSqrt()	Pipelined floating-point square-rooter
FloatPipeEq()	Pipelined floating-point comparator, if A is equal to B result is 1.
FloatPipeGt()	Pipelined floating-point comparator, if A is greater than B result is 1.
FloatPipeLs()	Pipelined floating-point comparator, if A is less than B result is 1.

1.8 Floating-point constants

The following constants are provided in the pipelined floating-point library:

extern macro expr FloatPi();

extern macro expr FloatE();

Constant:	**Description:**
Pi	3.14159265
E	2.71828183

Parameters:	None.
Timing:	Compile-time.
Description:	`Float` values representing the associated numbers.

Example

```
Float A;
A = FloatPi();
```

1.9 FloatPipeXCycles: clock cycles for floating-point operations

```
extern macro expr FloatPipeFromIntCycles;
extern macro expr FloatPipeToIntCycles;
extern macro expr FloatPipeToUIntCycles;
extern macro expr FloatPipeAddCycles;
extern macro expr FloatPipeSubCycles;
extern macro expr FloatPipeDivCycles;
extern macro expr FloatPipeMultCycles;
extern macro expr FloatPipeSqrtCycles;
extern macro expr FloatPipeEqCycles;
extern macro expr FloatPipeGtCycles;
extern macro expr FloatPipeLsCycles;
```

Parameters:	None.
Timing:	Compile-time.
Description:	Constant. Value representing the number of clock cycles needed for the result to be registered for the operation described by the routine name.

FloatPipeXCycles macro	Corresponding floating-point operation
FloatPipeFromIntCycles	FloatPipeFromInt()
FloatPipeToIntCycles	FloatPipeToInt()
FloatPipeToUIntCycles	FloatPipeToUInt()
FloatPipeAddCycles	FloatPipeAdd()
FloatPipeSubCycles	FloatPipeSub()
FloatPipeDivCycles	FloatPipeDiv()
FloatPipeMultCycles	FloatPipeMult()
FloatPipeSqrtCycles	FloatPipeSqrt()
FloatPipeEqCycles	FloatPipeEq()
FloatPipeGtCycles	FloatPipeGt()
FloatPipeLsCycles	FloatPipeLs()

Example

```
int x;
x = 0;
Float A, B, C;
while (x != FloatPipeAddCycles)
{
    par
    {
        FloatPipeAdd (A, B, &C ... )
        x ++;
    }
}
```

1.10 Floating-point arithmetic cores

The Pipelined floating-point library contains macros which allow you set up "cores" to perform arithmetic operations on floating-point numbers. The cores are best used in sequential code. If you have pipelined code, use the ***floating-point arithmetic macros*** (see page 12).

- `FloatPipeCoreCreateHandle()`: creates a Handle for the core.
- `FloatPipeCoreSet()`: Sets the input parameters for the operation.
- `FloatPipeCore`***X*** macros: Set the arithmetic operation to be performed.
- `FloatPipeCoreResult()`: Determines if the operation is ready to return the result.
- `FloatPipeCoreGet()`: Captures a pipelined result.
- `FloatPipeCoreRun()`: Runs the arithmetic core.

1.10.1 FloatPipeCoreCreateHandle() macro

```
#define FloatPipeCoreCreateHandle(Name)
```

Parameters:	***Name***: Name of handle for operation you want to create.
Timing:	Handle created at compile time.
Description:	Creates a handle for a core operation. The name should be intuitive of the type of operation to be created. There must not be any space between the name and the bracket encapsulating the name.

Example

```
FloatPipeCoreCreateHandle(MyMUL);
FloatPipeCoreCreateHandle(MySUB);
```

1.10.2 FloatPipeCoreSet() macro

```
extern macro proc FloatPipeCoreSet (HandlePtr, A, B);
```

Parameters: ***HandlePtr***: Pointer to an operation handle created using `FloatPipeCoreCreateHandle()`.

A: Float value that is the first parameter of the operation.

B: Float value that is the second parameter of the operation.

You must include this parameter, even if the operation does not take a second parameter.

Timing: 0 or more clock cycles.

Description: Sets the input parameters for the operation to which ***HandlePtr*** is connected. ***HandlePtr*** is connected to an operation using `FloatPipeCoreRun()`. In addition to the inputs being set, this also triggers the execution of the operation.

You need to run this macro in parallel with `FloatPipeCoreRun()`.

Example

```
par
{
    FloatPipeCoreRun (&MyDivider,
                      FloatPipeCoreDiv,
                       __clock,
                      FloatPipeChipTypeGeneric);
    ...
    seq
    {
        ...
        FloatPipeCoreSet (&MySquareRoot, A, B);
    }
}
```

1.10.3 FloatPipeCoreX: Floating-point core arithmetic macros

```
extern macro expr FloatPipeCoreAdd;
extern macro expr FloatPipeCoreSub;
extern macro expr FloatPipeCoreMult;
extern macro expr FloatPipeCoreDiv;
extern macro expr FloatPipeCoreSqrt;
```

Parameters:	None.
Timing:	Compile-time.
Description:	Identifier for the type of operation selected as a core. These expressions are fed as the second parameter for `FloatPipeCoreRun()`.

Example

```
FloatPipeCoreRun (&MyMultiplier,
                  FloatPipeCoreMult,
                  __clock,
                  FloatPipeChipTypeGeneric);
```

1.10.4 FloatPipeCoreResult() macro

`extern macro expr FloatPipeCoreResult (`***`HandlePtr`***`);`

Parameters:	***`HandlePtr`***: Pointer to an operation handle created using `FloatPipeCoreCreateHandle()`.
Timing:	0 or more clock cycles.
Description:	An expression that returns true if the operation selected is ready to return the result in the current clock cycle.

Example

```
seq
{
    ...
    FloatPipeCoreSet (&MyDivider, A, B);
    while (FloatPipeCoreResult (&MyDivider) == 0)
        delay;
    ...
}
```

1.10.5 FloatPipeCoreGet() macro

`extern macro proc FloatPipeCoreGet (`***`HandlePtr`***`, `***`ResultPtr`***`);`

Parameters:	***HandlePtr***: Pointer to an operation handle created using `FloatPipeCoreCreateHandle()`.
	ResultPtr: Pointer of type `Float` for the core to latch the result.
Timing:	0 or more clock cycles.
Description:	A procedure that can be used to latch the result from the core result register. This procedure is used in conjunction with `FloatPipeCoreResult()` to capture the pipelined result.

Example

```
while (FloatPipeCoreResult (&MyDivider) == 0)
    delay;
    FloatPipeCoreGet (&MyDivider, &Result);
```

1.10.6 FloatPipeCoreRun() macro

extern macro proc FloatPipeCoreRun (***HandlePtr***, ***Operation***, ***Clock***, ***ChipType***);

Parameters:	***HandlePtr***: Pointer to an operation handle created using `FloatPipeCoreCreateHandle()`.
	Operation: Type of arithmetic operation the core will perform. Set this using the `FloatPipeCore`***X*** macros.
	Clock: Interface, expression or signal returning the clock pulse.
	ChipType: Identifier for the FPGA/PLD target. Enables the core to use embedded features (primitives) available on the FPGA. Set using the `FloatPipeChipType`***X*** macros. Use `FloatPipeChipTypeGeneric` for simulation.
Timing:	Does not terminate in normal use.
Description:	Runs the arithmetic operation selected, with the selected parameters and links the handle to the operation.
	There is no check done for matching the name of the handle to that of the operation.
	You should run this macro in parallel with the rest of your code.

Example

```
FloatPipeCoreCreateHandle(MySquareRoot);
...

par
{
    FloatPipeCoreRun (&MySquareRoot,
                     FloatPipeCoreSqrt,
                      __clock,
                     FloatPipeChipTypeGeneric);
    ...

    seq
    {
        Float In, Result;
        ...
        FloatPipeCoreSet (&MySquareRoot, In, FloatPosZero ());
        while (!FloatPipeCoreResult (&MySquareRoot))
            delay;
        FloatPipeCoreGet (&MySquareRoot, &Result);
    }
    ...
}
```

1.11 FloatPipeChipTypeX macros

```
extern macro expr FloatPipeChipTypeVirtex2;
extern macro expr FloatPipeChipTypeGeneric;
```

Parameters:	None.
Timing:	Compile-time.
Description:	Identifier for the type of chip chosen. Some chip types contain embedded hardware cores that can be used by floating-point operations. The identifiers available indicate which chips are currently supported. For simulation you must manually set the parameter to `FloatPipeChipTypeGeneric`. This expression is fed as the last parameter of `FloatPipeMult()` and `FloatPipeCoreRun()`.

Example

```
FloatPipeCoreRun (&MyMultiplier,
                  FloatPipeCoreMult,
                  __clock,
                  FloatPipeChipTypeVirtex2);
```

2 Index