WB_FPU Floating-point Unit

Summary

This document provides detailed reference information with respect to the WB_FPU peripheral device.

Core Reference
CR0171 (v1.1) September 20, 2007

Altium Designer's floating-point unit – WB_FPU – facilitates the conversion of 32-bit integer values into single precision floating-point numbers, and vice-versa. This hardware peripheral performs these conversions and additional, standard mathematical calculations, with greater speed when compared with previous, software-based floating-point support.

Incorporating the standard Wishbone interface, the WB_FPU can be used with any of the 32-bit processors available in Altium Designer.

Features

- Converts 32-bit integer values to single precision floating-point numbers, in accordance with the IEEE-754 standard
- Converts single precision floating-point numbers to 32-bit integer values
- Perform addition of two single precision floating-point numbers
- Perform subtraction of two single precision floating-point numbers
- Perform multiplication of two single precision floating-point numbers
- Perform division of two single precision floating-point numbers
- Wishbone-compliant.

Available Devices

The WB_FPU device can be found in the FPGA Peripherals integrated library (`\Program Files\Altium Designer 6\Library\Fpga\FPGA Peripherals.IntLib`).

IEEE 754 Standard

Before discussing the actual WB_FPU peripheral in detail – including its functional and hardware descriptions – it is worth spending some time to look at the standard to which the floating-point numbers adhere, the IEEE Standard for Binary Floating-point Arithmetic (IEEE 754). This standard not only specifies how floating-point numbers are to be represented, but also how arithmetic calculations on these numbers should be performed.

Single precision format

The WB_FPU supports single precision (32-bit) binary floating-point numbers, formatted in accordance with the IEEE 754 Standard. Figure 1 shows the composition of a binary floating-point number under this standard.

MSB LSB

31	30 23	22 0
Sign	Exponent	Significand

Figure 1. IEEE 754 floating-point number format.

Sign

The Sign consists of a single bit. If this bit is '1', then the number is negative. If this bit is '0', then the number is positive.

Exponent

The Exponent consists of 8 bits. The base system used is binary (base 2). This base is implicit and is therefore not stored as part of the 32-bit format.

To facilitate both positive and negative exponent values, a bias value is added to the actual exponent to arrive at the 8-bit value that gets stored. For IEEE 754 single precision floats, the bias value used is 127.

Significand

The significand represents the precision bits of the number. The WB_FPU supports normalized floating-point numbers (see Normalized numbers). Normalization leads to two important points:

- The radix point is always placed after the first non-zero digit (e.g. `1101` becomes 1.101×2^{3})
- The leading bit – to the left of the radix point – is required to be non-zero. As we are using the binary system, this bit can only be `1`. As such, the leading bit is implicit and not stored as part of the significand.

  The significand therefore effectively has 24-bit resolution, but only 23 bits are used to represent the fractional part of the number only – the digits to the right of the radix point. If the fractional part of the number (written in binary notation) is less than 23 bits, the remaining right-hand bits are padded with zeroes. For example 1.101×2^{3} would yield a significand of `10100000000000000000000`.

Floating-point encoding example

Understanding more clearly how an integer is encoded into this floating-point format can be demonstrated using a simple example. Consider the integer value `276`. We need to obtain the Sign, Exponent and Significand in order to fully represent this integer in binary floating-point format.

- This is a positive integer, so the sign bit of the binary floating-point representation will be `0`.
- Convert the integer into binary notation – giving us `100010100`.
- If we normalize this value, moving the radix point to the right of the leading 1, our binary value `100010100` becomes 1.00010100×2^{8}.
- The actual exponent for this value is `8`. The exponent that gets stored is `8 + 127` (the bias value). Thus, `135` will be stored as the 8-bit exponent value `10000111`.
- The entry for the significand is the fractional part of the number (highlighted in gray), padded with zeroes to obtain the required 23 bits. In this example, the significand is `00010100000000000000000`.

This is all the information we need to fully represent the integer in floating-point form:

276 = 0 10000111 00010100000000000000000

Valid range for floating-point numbers

The valid range for normalized (non-zero) numbers in single precision floating-point format can be defined as:

$$2^{minexp} \leq \text{FloatValue} \leq (2 - 2^{1-precisionbits}) \times 2^{maxexp}$$ ...Positive values

$$-2^{minexp} \geq \text{FloatValue} \geq -(2 - 2^{1-precisionbits}) \times 2^{maxexp}$$ ...Negative values

where,

`minexp` is the minimum legal exponent value for normalized numbers being encoded. For single precision floating-point format this value is `-126` which, when the bias value (`127`) is added, will give a biased exponent of `1` (`00000001b`).

`maxexp` is the maximum legal exponent value for normalized numbers being encoded. For single precision floating-point format this value is `127` which, when the bias value (`127`) is added, will give a biased exponent of `254` (`11111110b`).

`precisionbits` is the number of bits of significand precision for the floating-point format you are using. For single precision, this value is `24` – remembering that the MSB is always `1` and so is hidden (or implicit).

Feeding the above values back into the expressions gives the following ranges:

$$2^{-126} \leq \text{Value} \leq (2 - 2^{-23}) \times 2^{127}$$ ...Positive values

$$-2^{-126} \geq \text{Value} \geq -(2 - 2^{-23}) \times 2^{127}$$ ...Negative values

These ranges can be reduced to a single range governing all values, positive and negative:

$$+/-(2 - 2^{-23}) \times 2^{127}$$

The value 0 (or more specifically +0 and -0) is treated as a special case and is discussed in the next section.

Special values

The IEEE 754 standard reserves effective exponent values of 0 and 255, which are used in conjunction with particular significand values to denote special binary floating-point values. Table 1 lists the special values supported by the WB_FPU.

Table 1. Special values.

Sign	Exponent	Significand	Describes
0	00h	000000h	Positive zero
1	00h	000000h	Negative zero
0	FFh	000000h	Positive infinity
1	FFh	000000h	Negative infinity
0	FFh	> 000000h	NaN – Not a Number

Note: -0 and +0 are distinct values, but are equal for comparison purposes.

A word about NaNs

The value NaN (Not a Number) is used to represent a value that does not represent a real number. As illustrated in Table 1, NaNs are represented by a bit pattern with an exponent of all ones and a non-zero significand. The sign bit can be 0 or 1 – it has no bearing.

There are two categories of NaNs:

- **QNaN (Quiet NaN)** – arising when the result of an arithmetic operation is mathematically undefined. The MSB of the significand is '1' for this type of NaN.
- **SNaN (Signaling NaN)** – used to signal an exception when an invalid operation is performed. The MSB of the significand is '0' for this type of NaN.

The difference between QNaN and SNaN is not implemented in the WB_FPU.

Normalized numbers

The majority of numbers in the IEEE 754 floating-point format are normalized. Such a value has an assumed '1' for the hidden 24th bit of the significand, after which directly follows the radix point. A normalized floating-point value can be summarized using the following expression:

$$\text{NormalizedFloatValue} = s \times 2^{e-b} \times 1.f$$

where,

`s` is defined by the sign bit and is +1 for sign=0 and -1 for sign=1.

`e` is the biased exponent

`b` is the bias value of 127.

`f` is the 23-bit 'fractional value' of the significand, to the right of the radix point.

The 24-bit significand (1.f) of a normalized binary floating-point number will therefore always be in the range:

$$1 \leq \text{significand} < 2$$

Denormalized numbers

When a calculation involving two floating-point values results in an exponent that is too small to be properly represented – at values less than $1.f \times 2^{-127}$ – an underflow event occurs. The IEEE 754 standard includes the provision for 'gradual underflow', through the use of denormalized (or subnormal) numbers.

A denormalized number has a biased exponent of zero. When a number becomes denormalized, the significand is shifted by one bit to the right, in order to include the hidden 24th bit. This bit is set to zero. To compensate, the un-biased exponent is incremented by 1.

A denormalized floating-point value can be summarized using the following expression:

$$\text{DenormalizedFloatValue} = s \times 2^{-126} \times 0.f$$

The actual term for the un-biased exponent is 2^{e-b+1}, but as the biased exponent (e) is 0 for denormalized numbers and the bias (b) is 127, this reduces to 2^{-126}.

Denormalized numbers are not implemented in the WB_FPU. Instead, numbers with an exponent overflow will be rounded to infinity and numbers with an exponent underflow will be rounded to zero.

Floating-point algorithms

The following sections look at algorithms defined by the IEEE 754 standard for conversion and base arithmetic calculations, and which are implemented in the WB_FPU.

Conversions

The WB_FPU provides conversion from integer to the IEEE 754 single precision floating-point format and vice-versa.

Calculations

The WB_FPU supports the following calculations involving two floating-point values – Addition, Subtraction, Multiplication and Division.

Addition

Addition of two floating-point values requires that the exponents of both operands be first made equal. This involves shifting the significand of the operand with the smallest exponent, to the right, such that both exponents become equal. The significands are then added.

When adding significands, an overflow may occur. This overflow can be corrected by shifting the resulting significand one bit position to the right and adjusting the exponent accordingly.

The addition process takes a single clock cycle.

Table 2 summarizes the possible resulting values based on additions involving the various combinations of floating-point operands, including the special values listed in Table 1 (refer back to the section *Special values*).

Table 2. Addition of operands.

OP1 / OP2	**NaN**	**+Infinity**	**-Infinity**	**+0**	**-0**	**+Num**	**-Num**
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Infinity	NaN	+Infinity	NaN	+Infinity	+Infinity	+Infinity	+Infinity
-Infinity	NaN	NaN	-Infinity	-Infinity	-Infinity	-Infinity	-Infinity
+0	NaN	+Infinity	-Infinity	+0	+0	+Num	-Num
-0	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
+Num	NaN	+Infinity	-Infinity	+Num	+Num	+Num +Infinity	±Num +0
-Num	NaN	+Infinity	-Infinity	-Num	-Num	±Num +0	-Num -Infinity

Subtraction

Subtraction is similar to addition. It involves making the exponents of both operands equal and then subtracting the resulting significands. In order to make things less complicated, care is taken to ensure that the smaller operand is always subtracted from the larger one.

In order to keep the WB_FPU synthesizable for reasonable clock frequencies (e.g. 50MHz on a Spartan Virtex-II device) the subtraction process takes two clock cycles instead of one.

Table 3 summarizes the possible resulting values based on subtractions involving the various combinations of floating-point operands, including the special values listed in Table 1(refer back to the section *Special values*).

Table 3. Subtraction of operands.

OP1 / OP2	**NaN**	**+Infinity**	**-Infinity**	**+0**	**-0**	**+Num**	**-Num**
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Infinity	NaN	NaN	-Infinity	-Infinity	-Infinity	-Infinity	-Infinity
-Infinity	NaN	+Infinity	NaN	+Infinity	+Infinity	+Infinity	+Infinity
+0	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
-0	NaN	+Infinity	-Infinity	+0	+0	+Num	-Num

OP1 / OP2	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
+Num	NaN	+Infinity	-Infinity	-Num	-Num	±Num +0	-Num -Infinity
-Num	NaN	+Infinity	-Infinity	+Num	+Num	+Num +Infinity	±Num +0

Multiplication

Multiplication of two floating-point values involves addition of their exponents and multiplication of their significands. The resulting sign is an XOR function of the signs of both operands.

Table 4 summarizes the possible resulting values based on multiplications involving the various combinations of floating-point operands, including the special values listed in Table 1 (refer back to the section *Special values*).

Table 4. Multiplication of operands.

OP1 / OP2	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Infinity	NaN	+Infinity	-Infinity	NaN	NaN	+Infinity	-Infinity
-Infinity	NaN	-Infinity	+Infinity	NaN	NaN	-Infinity	+Infinity
+0	NaN	NaN	NaN	+0	-0	+0	-0
-0	NaN	NaN	NaN	-0	+0	-0	+0
+Num	NaN	+Infinity	-Infinity	+0	-0	+Num +Infinity	-Num -Infinity
-Num	NaN	+Infinity	+Infinity	-0	+0	-Num -Infinity	+Num +Infinity

Division

Division of two floating-point numbers is similar to multiplication. It involves subtraction of their exponents and division of their significands. The resulting sign is an XOR function of the signs of both operands.

Table 5 summarizes the possible resulting values based on divisions involving the various combinations of floating-point operands, including the special values listed in Table 1 (refer back to the section *Special values*).

Table 5. Division of operands.

OP1 / OP2	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+Infinity	NaN	NaN	NaN	+0	-0	+0	-0
-Infinity	NaN	NaN	NaN	-0	+0	-0	+0
+0	NaN	+Infinity	-Infinity	NaN	NaN	+Infinity	-Infinity
-0	NaN	-Infinity	+Infinity	NaN	NaN	-Infinity	+Infinity
+Num	NaN	+Infinity	-Infinity	+0	-0	+Num	-Num
-Num	NaN	-Infinity	+Infinity	-0	+0	-Num	+Num

Rounding

The IEEE 754 standard describes the following four rounding methods:

- Truncate
- Round down (towards –infinity)
- Round up (towards +infinity)
- Round to nearest even.

The WB_FPU implements the last of these – round to nearest even – since this is the method used in standard C programming. This method implements rounding to the nearest possible value, except where the initial value is mid-way between the higher and lower values. In such cases, even values will be rounded down and odd values will be rounded up.

Rounding in the WB_FPU is performed as follows:

```
if (guard && (LSB or sticky)) then
   round_up();
else
   round_down();
end if;
```

LSB is the least significant bit of the significand resulting from a calculation. The Guard and Sticky bits are used as an aid in rounding and can be summarized as follows:

- **Guard bit** – this is the first bit that does not fit into a significand (i.e. the bit to the right of the significand's LSB)
- **Sticky bit** – this is an OR-reduced term for the complete set of bits that do not fit in the significand (i.e. bits to the right of the LSB and guard bits).

Functional Description

Symbol

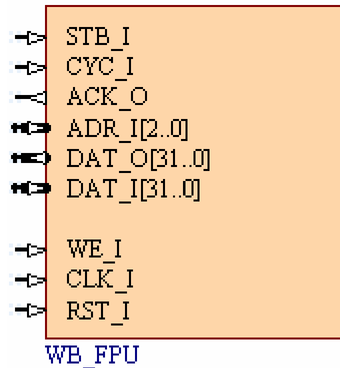

Figure 2. WB_FPU Symbol.

Pin description

Table 6. WB_FPU pin description

Name	Type	Polarity/ Bus size	Description
		Control Signals	
CLK_I	I	Rise	External (system) clock signal
RST_I	I	High	External (system) reset
		Host Processor Interface Signals	
STB_I	I	High	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle
CYC_I	I	High	Cycle signal. When asserted, indicates the start of a valid Wishbone cycle
ACK_O	O	High	Standard Wishbone device acknowledgement signal. When this signal goes high, the Floating-point Unit (Wishbone Slave) has finished execution of the requested action and the current bus cycle is terminated
ADR_I	I	3	Address bus, used to select an internal register of the device for writing to/reading from
DAT_O	O	32	Data to be sent to host processor

Name	Type	Polarity/ Bus size	Description
DAT_I	I	32	Data received from host processor
WE_I	I	Level	Write enable signal. Used to indicate whether the current local bus cycle is a Read or Write cycle: 0 = Read 1 = Write

Hardware Description

Block Diagram

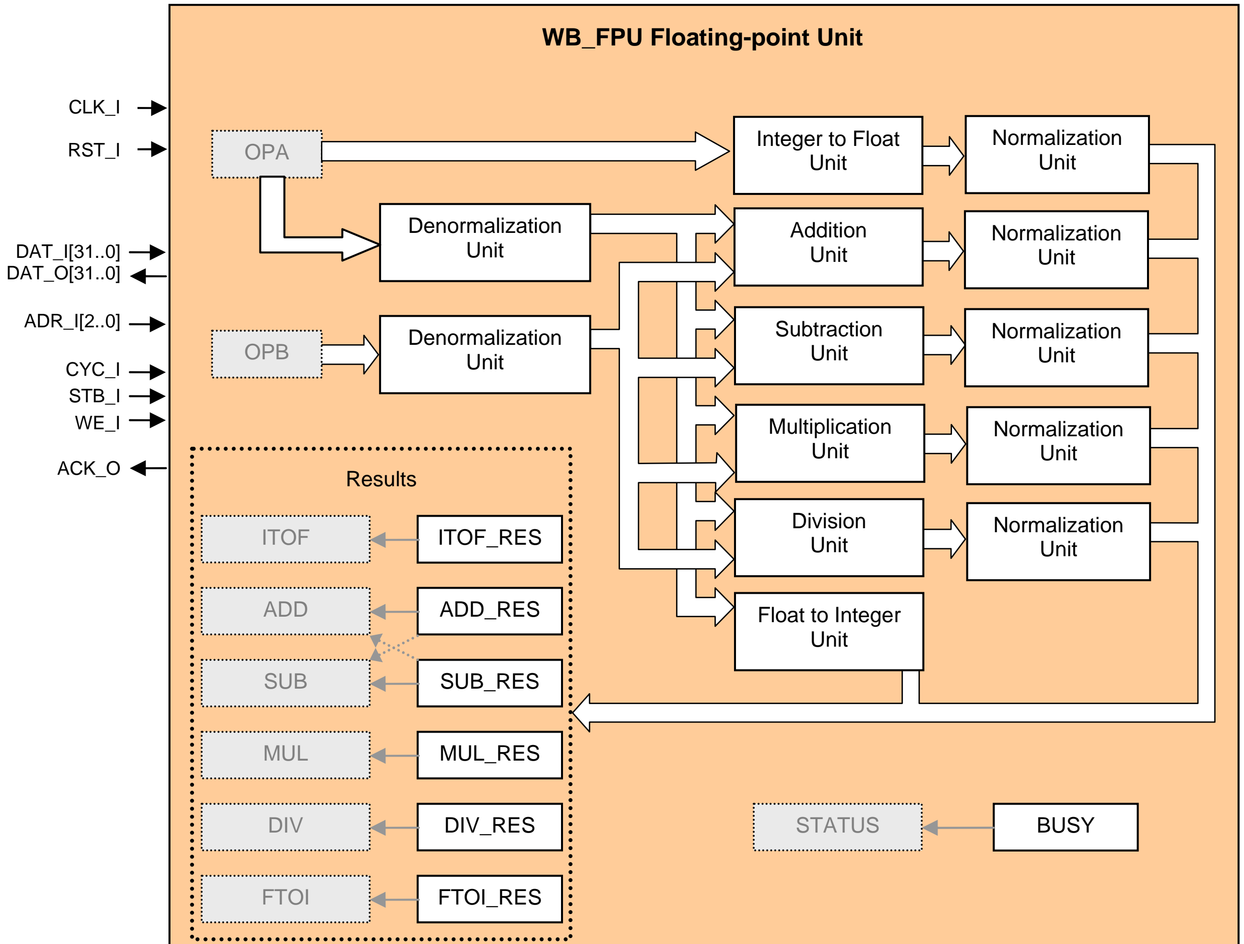

Figure 3. WB_FPU block diagram.

Internal Units

The following sections detail the various units that constitute the WB_FPU.

Denormalization Units

Denormalized numbers themselves are not supported in the WB_FPU. However, for the purposes of calculation, it is necessary to internally perform a light form of denormalization. The Denormalization

Unit takes an IEEE 754-formatted floating-point number and splits it into a "float" typed signal containing the following information bits:

- NaN, Infinity and Zero bits – indicating special values
- 1-bit Sign
- 10-bit Exponent
- 24-bit Significand – consisting the implicit or hidden bit as the MSB, followed by the fraction
- Guard and Sticky bits – used as an aid in rounding.

> The exponent in the denormalized value is still biased. This avoids the need for extra hardware to perform bias handling in both Denormalization and Normalization units.

The internal representation of the exponent has two extra bits – MSB and MSB-1 – both of which are initially set to '0'. These bits are used to facilitate easy detection of exponent underflow and overflow in the calculation units (see *Normalization Units*).

When the Denormalization Unit has finished constructing the internal float value it will set a ready flag, to indicate that this value is ready for further processing.

Two Denormalization Units are included, for operand A and operand B respectively. Each unit takes its input directly from the Wishbone interface (on the DAT_I bus), provided a write to the corresponding operand address (1h for OPA and 2h for OPB) is being performed.

Conversion Units

The WB_FPU contains two conversion units – one for floating-point-to-integer (FTOI) conversion and one for integer-to-floating-point (ITOF) conversion.

The ITOF Unit takes its input (a 32-bit integer value) directly from the Wishbone interface (on the DAT_I bus), provided a write to the OPA address (1h) is being performed. The unit generates an internal float-typed signal and a ready signal – to signal the subsequent Normalization Unit that the converted value is ready for processing.

The FTOI Unit takes its input (an internal float-typed value) from the Denormalization Unit associated with OPA, which in turn takes its input (an IEEE 754-formatted floating-point value) directly from the Wishbone interface (on the DAT_I bus), provided a write to the OPA address (1h) is being performed. The unit converts the float-typed value to a 32-bit integer and generates a ready flag – to signal that the converted value is ready for transfer over the Wishbone interface.

Addition Unit

The Addition Unit is able to add two IEEE 754 floating-point values (OPA and OPB) without taking signs into account.

The unit takes its operand inputs from the Denormalization Units. The result of the addition may have Guard and Sticky bits set for rounding purposes, and the special value bits (NaN, Infinity, Zero) will be updated if required. Once the calculation is complete a ready flag will be set, to signal the subsequent Normalization Unit that a new value is ready.

The resulting Sign bit will be that of OPA.

Since addition is performed without regard for sign, operands with different signs should be subtracted rather than added. When reading the result of an addition (reading address 2h) featuring operands of equal sign, the normalized result from the Addition Unit will be obtained. When the signs differ, the normalized result from the Subtraction Unit will be obtained instead.

Subtraction Unit

The Subtraction Unit is able to subtract two IEEE 754 floating-point values (OPA and OPB) without taking signs into account. The smaller value is always subtracted from the larger.

The unit takes its operand inputs from the Denormalization Units. The result of the subtraction may have Guard and Sticky bits set for rounding purposes, and the special value bits (NaN, Infinity, Zero) will be updated if required. Once the calculation is complete a ready flag will be set, to signal the subsequent Normalization Unit that a new value is ready.

The resulting Sign bit will depend on the initial operand values:

- If OPA < OPB, then the resulting Sign bit will be the inverse of the original Sign bit for OPA
- If OPA > OPB, then the resulting Sign bit will be that of OPA.

Since subtraction is performed without regard for sign, operands with different signs should be added rather than subtracted. When reading the result of a subtraction (reading address 3h) featuring operands of equal sign, the normalized result from the Subtraction Unit will be obtained. When the signs differ, the normalized result from the Addition Unit will be obtained instead.

Multiplication Unit

The Multiplication Unit multiplies two IEEE 754 floating-point values (OPA and OPB).

The unit takes its operand inputs from the Denormalization Units. The result of the multiplication may have Guard and Sticky bits set for rounding purposes, and the special value bits (NaN, Infinity, Zero) will be updated if required. Once the calculation is complete a ready flag will be set, to signal the subsequent Normalization Unit that a new value is ready.

Division Unit

The Division Unit divides two IEEE 754 floating-point values (OPA by OPB).

The unit takes its operand inputs from the Denormalization Units. The result of the division may have Guard and Sticky bits set for rounding purposes, and the special value bits (NaN, Infinity, Zero) will be updated if required. Once the calculation is complete a ready flag will be set, to signal the subsequent Normalization Unit that a new value is ready.

Division is not a ready-to-use operational block in VHDL. Therefore a successive approximation algorithm has been implemented that determines one bit of the significand per cycle of CLK_I. As a result, the floating-point division operator is slower in comparison to the other operators.

Normalization Units

The Normalization Unit takes as input an internally-formatted float-typed number and constructs the corresponding IEEE 754-formatted floating-point value. The value will be rounded in accordance with the significand LSB, the Guard and Sticky bits. Once construction of the IEEE 754 floating-point value is complete, the unit generates a ready flag – to signal that the value is ready for transfer over the Wishbone interface.

Effect of special bits

The special value bits of the input number (NaN, Infinity, Zero) are checked, with the following influence on the resulting IEEE 754 floating-point value:

- If the NaN bit of the input value is '1', then the Sign of the resulting IEEE 754 float will be set to '0' and all other bits (30..0) set to '1'.

Sign	Exponent	Significand
0	11111111	11111111111111111111111

- If the Infinity bit is '1', then the Sign of the IEEE 754 float will be set to that of the input number, the Exponent will be set to all '1's and the Significand set to all '0's.

Sign	Exponent	Significand
Input Sign	11111111	00000000000000000000000

- If the Zero bit of the input value is '1', then the Sign of the IEEE 754 float will be set to that of the input number and all other bits (30..0) set to '0'.

Sign	Exponent	Significand
Input Sign	00000000	00000000000000000000000

Exponent underflow and overflow detection

As mentioned previously, the internal representation of the exponent has two extra bits – MSB and MSB-1 – both of which are used to facilitate easy detection of exponent underflow and overflow in a calculation unit. Detection is as follows:

- If MSB of exponent is '1', the resulting exponent underflows and the IEEE 754 floating-point number returned will be rounded to zero by the Normalization Unit (i.e. all exponent and significand bits set to '0').
- If MSB of exponent is '0' but MSB-1 is '1', the resulting exponent overflows and the IEEE 754 floating-point number returned will be rounded to infinity by the Normalization Unit (i.e. all exponent bits set to '1' and all significand bits set to '0').

Internal registers

Internal registers within the WB_FPU are not accessed directly. Their content is loaded or accessed by writing to/reading from a particular address (highlighted using a gray background in the block diagram of Figure 3). The following sections summarize each of these addresses.

Status address (STATUS)

Address: 0h

Access: Read only

Internal Register Accessed: BUSY

This address is used to access the BUSY register, which is used to determine the state of the calculation and conversion units – which are still busy and which are ready.

Table 7. The BUSY register

MSB							LSB
31	6	5	4	3	2	1	0
-		ftoi	itof	div	mul	sub	add

Table 8. The BUSY register bit functions

Bit	Symbol	Function
BUSY.31..BUSY.6	-	Not Used.
BUSY.5	ftoi	Float to Integer Unit state. 0 = Ready 1 = Busy
BUSY.4	itof	Integer to Float Unit state. 0 = Ready 1 = Busy
BUSY.3	div	Division Unit state. 0 = Ready 1 = Busy
BUSY.2	mul	Multiplication Unit state 0 = Ready 1 = Busy
BUSY.1	sub	Subtraction Unit state. 0 = Ready 1 = Busy
BUSY.0	add	Addition Unit state. 0 = Ready 1 = Busy

Operand A address (OPA)

Address: 1h

Access: Write only

Write to this address to:

- Load a 32-bit integer value directly into the Integer to Float Unit for processing.
- Load an IEEE 754-formatted value directly into the Denormalization Unit associated with Operand A.

Writing to this address will cause the calculation units, the Float to Integer Unit and the Normalization units to be reset. The BUSY register will be loaded with all '1's during this time. The WB_FPU will send an Acknowledge signal that it has received the loaded data.

Operand B address (OPB)

Address: 2h

Access: Write

Write to this address to load an IEEE 754-formatted value directly into the Denormalization Unit associated with Operand B.

Writing to this address will cause the calculation units, the Float to Integer Unit and the Normalization Units to be reset. The BUSY register will be loaded with all '1's during this time. The WB_FPU will send an Acknowledge signal that it has received the loaded data.

This address is used for two purposes – loading operand B and reading the result of an addition. Provided you are performing a Write (WE_I input High) you will access the Denormalization Unit.

Addition result address (ADD)

Address: 2h

Access: Read

Internal Register Accessed: ADD_RES

This address is used to access the ADD_RES register, containing the normalized result of OPA + OPB. The resulting floating-point value will be set to infinity on overflow.

This address is used for two purposes – loading operand B and reading the result of an addition. Provided you are performing a Read (WE_I input Low) you will access the ADD_RES register. The exception to this is when the original operands (OPA and OPB) are of different sign. In this case, performing a read of this address will obtain the result stored in the SUB_RES register.

Subtraction result address (SUB)

Address: 3h

Access: Read

Internal Register Accessed: SUB_RES

This address is used to access the SUB_RES register, containing the normalized result of OPA - OPB. The resulting floating-point value will be set to infinity on overflow.

This address is used for two purposes – loading operands A and B simultaneously (see next section) and reading the result of a subtraction. Provided you are performing a Read (WE_I input Low) you will access the SUB_RES register. The exception to this is when the original operands (OPA and OPB) are of different sign. In this case, performing a read of this address will obtain the result stored in the ADD_RES register.

Dual Operand Loading

Provided you are performing a Write (WE_I input High) to address 3h, you can load the same floating-point value for both operands (OPA and OPB) simultaneously. You can then read the ADD_RES register (address 2h) or MUL_RES register (address 4h) to obtain a cheap $2x$ or x^2 result.

Multiplication result address (MUL)

Address: 4h

Access: Read only

Internal Register Accessed: MUL_RES

This address is used to access the MUL_RES register, containing the normalized result of OPA * OPB. The resulting floating-point value will be set to infinity on overflow and zero on underflow.

Division result address (DIV)

Address: 5h

Access: Read only

Internal Register Accessed: DIV_RES

This address is used to access the DIV_RES register, containing the normalized result of OPA / OPB. The resulting floating-point value will be set to infinity on overflow or divide-by-zero, zero on underflow, and NaN for zero/zero.

Integer to Float result address (ITOF)

Address: 6h

Access: Read only

Internal Register Accessed: ITOF_RES

This address is used to access the ITOF_RES register, containing the normalized result of the conversion from 32-bit integer value into IEEE 754-formatted floating-point value.

Float to Integer result address (FTOI)

Address: 7h

Access: Read only

Internal Register Accessed: ITOF_RES

This address is used to access the FTOI_RES register, containing the result of the conversion from IEEE 754-formatted floating-point value into 32-bit integer.

Interfacing to a 32-bit Processor

Figure 4 shows an example of how a WB_FPU device can be wired into a design that uses a 32-bit processor – in this case a TSK3000A. A configurable Wishbone Interconnect device (WB_INTERCON) is used to simplify connection and also handle the word addressing – taking the 24-bit address line from the processor and mapping it to the 3-bit address line used to drive the WB_FPU.

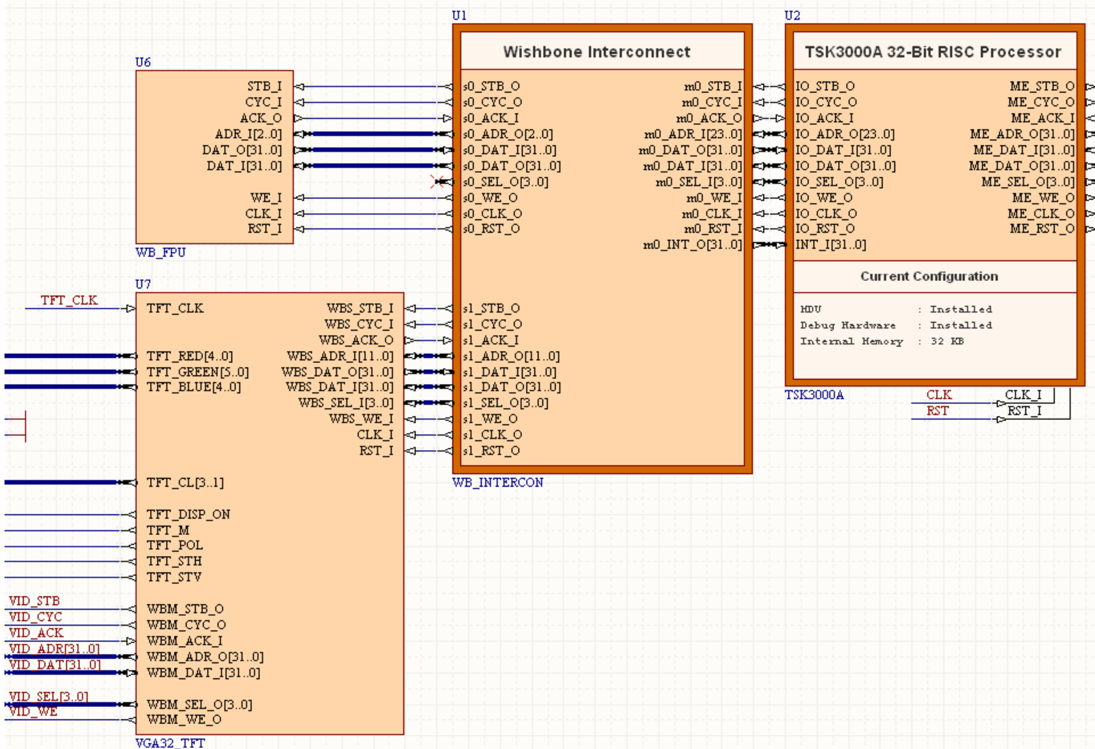

Figure 4. Example interfacing between a 32-bit processor (TSK3000A) and a WB_FPU.

When configuring the WB_INTERCON device – in particular the WB_FPU slave interface – ensure that the Address Bus Mode is set to `Word Addressing - ADR_O(0) <= ADR_I(1 or 2)`. As the WB_FPU's data bus width is 32-bit, the two lowest address bits are not connected to the slave device. ADR_I(2) of the master is mapped to ADR_O(0) of the slave, providing sequential word addresses (or addresses at every 4 bytes). Bits 4..2 of the output address line from the host processor (IO_ADR_O) are therefore mapped, through the WB_INTERCON, to bits 2..0 of the WB_FPU's input address line (ADR_I).

The actual 24-bit address sent out from the processor on its IO_ADR_O line is therefore constructed as follows:

WB_FPU Base Address + (Internal Register Address & "00")

The Base Address for the WB_FPU is specified as part of the peripheral's definition when adding it as a slave to the Wishbone Interconnect. For example, if the base address entered for the device is 100000h (mapping it to address FF10_0000h in the processor's address space), and you want to read the result of a multiplication from the MUL_RES register with address 4h, the value entered on the processor's IO_ADR_O line would be:

100000h + 10h = 100010h

For further information on the Wishbone Interconnect, refer to the *WB_INTERCON Configurable Wishbone Interconnect* core reference.

For further information on the TSK3000A processor, refer to the *TSK3000A 32-bit RISC Processor* core reference. Similar references can be found for other 32-bit processors supported by Altium Designer, by using the lower section of the **Knowledge Center** panel and navigating to the *Documentation Library » Embedded Processors and Software Development » FPGA Based and Discrete Processors* section.

Host to Controller Communications

Communications between a 32-bit host processor and the WB_FPU are carried out over a standard Wishbone bus interface. The following sections detail the communication cycles involved between Host and peripheral for writing to/reading from the internal registers.

Writing to an Internal Register

Data is written from the host processor to an internal register in the WB_FPU, in accordance with the standard Wishbone data transfer handshaking protocol. The write operation occurs on the rising edge of the CLK_I signal and can be summarized as follows:

- The host presents the required 24-bit address based on the register to be written on its IO_ADR_O output and valid data on its IO_DAT_O output. It then asserts its IO_WE_O signal, to specify a write cycle
- The WB_FPU receives the 3-bit address on its ADR_I input and, identifying the addressed register, prepares to receive data into that register
- The host asserts its IO_STB_O and IO_CYC_O outputs, indicating that the transfer is to begin. The WB_FPU, which monitors its STB_I and CYC_I inputs on each rising edge of the CLK_I signal, reacts to this assertion by latching the data appearing at its DAT_I input into the target register and asserting its ACK_O signal – to indicate to the host that the data has been received
- The host, which monitors its IO_ACK_I input on each rising edge of the CLK_I signal, responds by negating the IO_STB_O and IO_CYC_O signals. At the same time, the WB_FPU negates the ACK_O signal and the data transfer cycle is naturally terminated.

Remember that when writing to the OPA or OPB address, you are actually loading a value directly into the corresponding Denormalization Unit (and the Integer to Float Unit for OPA).

Reading from an Internal Register

Data is read from an internal register in accordance with the standard Wishbone data transfer handshaking protocol. The read operation, which occurs on the rising edge of the CLK_I signal, can be summarized as follows:

- The host presents the required 24-bit address based on the register to be read on its IO_ADR_O output. It then negates its IO_WE_O signal, to specify a read cycle
- The WB_FPU receives the 3-bit address on its ADR_I input and, identifying the addressed register, prepares to transmit data from the selected register
- The host asserts its IO_STB_O and IO_CYC_O outputs, indicating that the transfer is to begin. The WB_FPU, which monitors its STB_I and CYC_I inputs on each rising edge of the CLK_I signal, reacts to this assertion by presenting the valid data on its DAT_O output. The WB_FPU will assert its ACK_O signal – to indicate to the host that valid data is present – only when the corresponding status flag (in the BUSY register) for the applicable calculation or conversion unit is '0'.
- The host, which monitors its IO_ACK_I input on each rising edge of the CLK_I signal, responds by latching the data appearing at its IO_DAT_I input and negating the IO_STB_O and IO_CYC_O signals. At the same time, the WB_FPU negates the ACK_O signal and the data transfer cycle is naturally terminated.

Operational Overview

There is no initialization required for the WB_FPU. As soon as you write an operand value, the calculation and conversion units will be reset – clearing any stored values in the result registers.

Example code

The following sections provide example code for each of the calculations and conversions supported by the WB_FPU. The coding assumes connection of the device to a 32-bit processor through a Wishbone Interconnect device (WB_INTERCON), and using the identifier "FPU" when configuring the slave interface therein.

Addition

For addition, simply write the floating-point values for operand A and operand B and read the ADD_RES register (address 2h) for the result. Remember that if the operands are different in sign, reading this register will actually return the contents of the SUB_RES register.

```
#define fpu ((volatile float *) Base_FPU)

#define OP1     fpu[1]
#define OP2     fpu[2]
#define ADD     fpu[2]

// add(): returns a + b, negative or positive infinity on overflow
inline float add( float a, float b )
{
        OP1 = a;
```

```
        OP2 = b;
        return ADD;
}
```

Subtraction

For subtraction, simply write the floating-point values for operand A and operand B and read the SUB_RES register (address 3h) for the result. Remember that if the operands are different in sign, reading this register will actually return the contents of the ADD_RES register.

```
#define fpu ((volatile float *) Base_FPU)

#define OP1     fpu[1]
#define OP2     fpu[2]
#define SUB     fpu[3]

// sub(): returns a - b, negative or positive infinity on overflow
inline float sub( float a, float b )
{
        OP1 = a;
        OP2 = b;
        return SUB;
}
```

Multiplication

For multiplication, simply write the floating-point values for operand A and operand B and read the MUL_RES register (address 4h) for the result.

```
#define fpu ((volatile float *) Base_FPU)

#define OP1     fpu[1]
#define OP2     fpu[2]
#define MUL     fpu[4]

// mul(): returns a * b, negative or positive infinity on overflow,
//        zero on underflow
inline float mul( float a, float b )
{
        OP1 = a;
        OP2 = b;
        return MUL;
}
```

Division

For division, simply write the floating-point values for operand A and operand B and read the DIV_RES register (address 5h) for the result.

```
#define fpu ((volatile float *) Base_FPU)

#define OP1     fpu[1]
#define OP2     fpu[2]
#define DIV     fpu[5]

// div(): return a / b, positive or negative infinity on overflow or
//        divide by zero. 0/0 returns NaN. Returns zero on underflow.
inline float div( float a, float b )
{
       OP1 = a;
       OP2 = b;
       return DIV;
}
```

Involution

To obtain a quick and cheap square of a single floating-point value, simply write the floating-point value to address 3h – effectively loading the same value for OPA and OPB – and read the MUL_RES register (address 4h) for the result.

```
#define fpu ((volatile float *) Base_FPU)

#define OP12    fpu[3]
#define SQR     fpu[4]

// sqr(): returns a * b, positive infinity on overflow or zero on
//        underflow
inline float sqr( float a )
{
       OP12 = a;
       return MUL;
}
```

Integer to Float conversion

To convert a 32-bit integer to an IEEE 754 floating-point value, simply write the integer to the address for operand A and read the ITOF_RES register (address 6h) for the result.

```
#define fpu ((volatile float *) Base_FPU)
#define xfpu ((volatile int *) Base_FPU)

#define OP1     xfpu[1]
#define ITOF    fpu[6]
```

```
// itof(): returns floating-point equivalent of a.
inline float itof( int a )
{
        OP1 = a;
        return ITOF;
}
```

Float to Integer conversion

To convert an IEEE 754 floating-point value to a 32-bit integer value, simply write the floating-point value to the address for operand A and read the FTOI_RES register (address 7h) for the result.

```
#define xfpu ((volatile int *) Base_FPU)
#define fpu ((volatile float *) Base_FPU)

#define OP1     fpu[1]
#define FTOI    xfpu[7]

// ftoi(): returns integer equivalent of a.
inline float ftoi( int a )
{
        OP1 = a;
        return FTOI;
}
```

Revision History

Date	Version No.	Revision
12-Feb-2007	1.0	Initial release.
20-Sep-2007	1.1	Amended #define entries for fpu and xfpu in coding examples.

Software, hardware, documentation and related materials:

Copyright © 2007 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, Altium Designer, Board Insight, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.