Virtex-5 APU Floating-Point Unit v1.0

April 16, 2008 **Product Specification**

Introduction

The Xilinx LogiCORE™ IP Auxiliary Processor Unit (APU) Floating-Point Unit is an optimized floating-point unit designed for the PowerPC™ 440 embedded microprocessor of the Virtex™-5 FXT FPGA family. It provides support for *IEEE-754* floating-point arithmetic operations in single or double precision. The APU is tightly coupled to the PowerPC core with the APU interface. Software applications can use native PowerPC floating-point instructions to achieve typical speed-ups of 6x over software emulation for floating-point intensive code. Code optimization can provide up to 30x speed-up, which equates to a sustained performance of approximately 200MFLOPS.

Features

- Compliant with the *IEEE-754* standard for single- and double-precision floating-point arithmetic, with minor and documented exceptions
- Decodes and executes standard PowerPC floating-point instructions
- Optimized for 2:1 APU:CPU clock ratio, allowing PowerPC to operate at maximum frequency
- Uses autonomous instruction issue to hide arithmetic latency and decrease cycles per instruction
- Optimized implementation leverages Virtex-5 high-performance DSP features
- Integrated into Xilinx Embedded Development Kit (EDK) design flow

LogiCORE IP Facts			
Core Specifics			
Supported Device Family	Virtex-5 FXT		
Resources Used	LUT-Reg Pairs	DSP Blocks	Block RAMs
Single (full)	2520	3	0
Double (full)	4950	13	0
Clock Speeds	-1	200 MHz	
	-2	225 MHz	
Provided with Core			
Documentation	Product Specification •		
Design File Formats	VHDL		
Constraints File	UCF (user constraints file)		
Verification	VHDL Test Bench		
Instantiation Template	VHDL Wrapper		
Design Tool Requirements			
Xilinx Implementation Tools	ISE™ 10.1		
Support			
Provided by Xilinx, Inc @ www.xilinx.com			

Functional Overview

The APU Floating-Point Unit comprises execution units, a register file, bus interface and all the control logic necessary to manage the execution of floating-point instructions. Figure 1 provides an overview of the Floating-Point Unit (FPU) architecture.

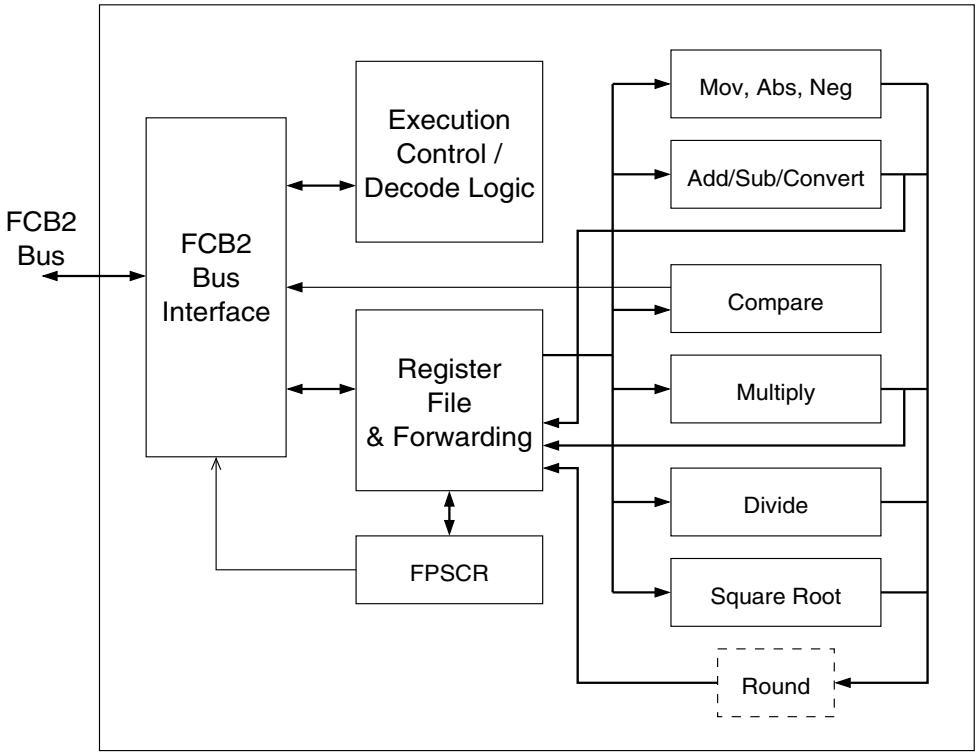

Figure 1: **Top-level APU Floating-Point Coprocessor Architecture**

Applications

The APU Floating-Point Unit augments the capabilities of the PowerPC 440 processor core with support for floating-point instructions. Many software applications make use of floating-point arithmetic, whether for occasional calculations or for intensive computation kernels. Some examples of application areas where floating-point arithmetic can be useful are:

- Digital signal processing of high-quality audio or video signals where a very large dynamic range is needed to retain fidelity.
- Matrix inversion in wireless communications and radar where algorithms such as QR decomposition and singular value decomposition are numerically unstable without sufficient dynamic range.
- Interpolation and extrapolation where quantization errors can lead to sub-optimal results.
- Digital signal processing tasks, particularly spectral methods such as FFT, in which the required range and precision of data samples may be difficult to predict at design time.
- Statistical processing and *ad hoc* calculations, where floating-point is often the simplest way to avoid integer overflow and rounding errors.

Increased Processing Capacity

The APU Floating-Point Unit increases the processing capacity of a PowerPC-based embedded system in the following ways:

1. Hardware floating-point operations complete faster than the equivalent software emulation routines, making floating-point arithmetic faster.
2. In software, only one floating-point operation can be in progress at a time. The floating-point operators within the FPU are pipelined so that multiple floating-point calculations can proceed in parallel. This parallelism in a floating-point algorithm can lead to dramatic speedups.
3. The FPU is autonomous; therefore, the PowerPC internal pipeline can continue to execute integer instructions while floating-point operations are handled by the FPU in parallel.

Functional Description

Fabric Coprocessor Bus (FCB) Interface

The FPU is connected to the PowerPC processor via the Auxilliary Processor Unit (APU) interface. This dedicated co-processor port is tightly coupled with the PowerPC's internal instruction pipeline and memory subsystem. This makes it ideal for connecting co-processors that execute instructions from the PowerPC's instruction stream, such as an FPU.

A block of logic known as the APU controller mediates between the PowerPC and the fabric co-processor. The bus connecting the FPU to the APU controller is known as the Fabric Coprocessor Bus, or FCB.

The data buses within the FCB which carry data to and from the memory system are 128 bits wide. The instruction bus is 32 bits wide, as are the operand data buses which carry data to and from the PowerPC register file (these are not used by the FPU). There are a number of other control signals involved in an FCB transaction. Full details of the APU/FCB interface can be found in the *Virtex-5 Embedded Processor Block for PowerPC 440 Designs Reference Manual* [Ref 1].

It is possible to connect multiple co-processors to the FCB. However, due to the additional multiplexing logic required, doing so will adversely affect the maximum clock frequency of the system. For the highest possible performance, it is recommended that the FPU is the only co-processor attached to the APU controller.

The APU controller allows the FCB (and thus the FPU) to run at a different speed from the CPU itself. Any integer FCB:CPU clock ratio between 1:1 and 1:16 is a valid configuration. The FPU is optimized for operation at 200 MHz on speed grade 1 devices, which gives a 1:2 clock ratio when the PowerPC is operated at its maximum frequency (400 MHz on speed grade 1).

When running in Exceptions Disabled mode the FPU can accept an instruction on every APU clock cycle, provided that the APU controller can sustain this transfer rate and that FPU execution does not need to stall due to data dependencies. All instructions can be acknowledged within a single clock cycle with the exception of store operations which take at least three clock cycles.

Register File

The register file contains 32 floating-point registers. The width of these registers and of the internal data path depends on the precision selected—either 32 bits (single) or 64 bits (double). Regardless of the precision chosen, the APU/FCM interface ports remain 128 bits wide, so data transfers of any size require only a single clock cycle.

Figure 2 shows the layout of the fields that compose a floating-point number within the binary word(s) for each precision. The most significant bit is numbered as bit zero.

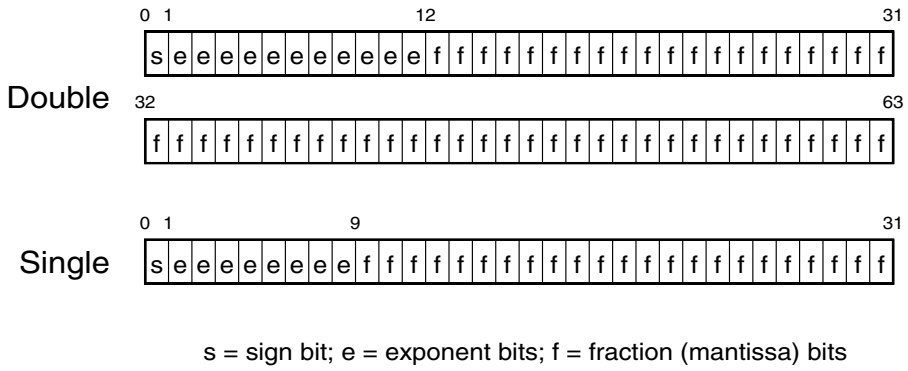

Figure 2: **Supported Floating-Point Format Layout**

PowerPC Instruction Set Support

The FPU supports both single and double precision. Table 1 details which instructions are supported by each configuration. For more information about the PowerPC floating-point instruction set, see *Book E: Enhanced Power PC Architecture Version 1.0* [Ref 2].

In general, the entire floating-point instruction set is supported, with the following exceptions:

- No FPU configurations support the instructions *fres*, *frsqrte* and *fsel*. Most compilers do not use these instructions unless specifically requested.
- The single-precision FPU:
  - Does not support conversions from floating-point to integer double-word formats, like *fctid* and *fctidz*, nor the "round double to single" operation *frsp*.
  - Interprets the *fcfid* instruction (convert from integer double-word) as a non-standard "convert from integer word." This is because the source register is only 32 bits wide.
  - Accepts double-precision arithmetic operations but will execute them in single precision.

Both single and double-precision FPUs support both single and double-precision load and store operations. When transferring double-precision data into or out of a single-precision FPU, the values are silently expanded or truncated as necessary (without rounding or renormalization).

Table 1: **PowerPC FP Instruction Set Support**

Instruction	Description	Single Precision	Double Precision
lfs(u)(x)(e)	Load floating-point single	Yes	Yes
stfs(u)(x)(e)	Store floating-point single	Yes	Yes
lfd(u)(x)(e)	Load floating-point double	Yes[1]	Yes
stfd(u)(x)(e)	Store floating-point double	Yes[1]	Yes
stfiwx(e)	Store float as integer word	Yes	Yes
fabs	Absolute value	Yes	Yes
fmr	Move	Yes	Yes
fnabs	Negative absolute value	Yes	Yes

Key: Yes = supported; No = not supported;
SP = operation performed in single-precision; NS = non-standard[2]

Table 1: **PowerPC FP Instruction Set Support *(Continued)***

Instruction	Description	Single Precision	Double Precision
fneg	Negate	Yes	Yes
fadd	Add	SP	Yes
fadds	Add (single)	Yes	Yes
fdiv	Divide	SP	Yes
fdivs	Divide (single)	Yes	Yes
fmul	Multiply	SP	Yes
fmuls	Multiply (single)	Yes	Yes
fsqrt	Square root	SP	Yes
fsqrts	Square root (single)	Yes	Yes
fsub	Subtract	SP	Yes
fsubs	Subtract (single)	Yes	Yes
fmadd	Multiply-add	SP	Yes
fmadds	Multiply-add (single)	Yes	Yes
fmsub	Multiply-subtract	SP	Yes
fmsubs	Multiply-subtract (single)	Yes	Yes
fnmadd	Negative multiply-add	SP	Yes
fnmadds	Negative multiply-add (single)	Yes	Yes
fnmsub	Negative multiply-subtract	SP	Yes
fnmsubs	Negative multiply-subtract (single)	Yes	Yes
fcfid	Convert from integer double-word	NS[2]	Yes
fctid	Convert to integer double-word	No	Yes
fctidz	As fctid, but round to zero	No	Yes
fctiw	Convert to integer word	Yes	Yes
fctiwz	As fctiw, but round to zero	Yes	Yes
frsp	Round to single precision	No	Yes
fcmpo	Compare (ordered)	Yes	Yes
fcmpu	Compare (unordered)	Yes	Yes
fres	Reciprocal estimate	No	No
frsqrte	Recip. sqrt. estimate	No	No
fsel	Select (ternary operator)	No	No
mcrfs	Status/control register to condition register	Yes	Yes
mffs	Move from status/control register	Yes	Yes
mtfsb0	Move to status/control register	Yes	Yes
mtfsb1	Move to status/control register bit 1	Yes	Yes
mtfsf	Move to status/control register fields	Yes	Yes
mtfsfi	Move to status/control register immediate	Yes	Yes

Key: Yes = supported; No = not supported;
SP = operation performed in single-precision; NS = non-standard[2]

1. Double-precision store instructions can be issued to a single-precision FPU. Data will be converted on the fly between double- and single-precision formats as necessary.
2. A single-precision FPU will treat the *fcfid* (convert from signed integer double-word to FP double) as if it were *fcfiw* (convert from signed integer word to FP single). This behavior is non-standard, but allows hardware acceleration of format conversions that would not otherwise be possible in a single-precision unit.

The "dotted" form of an instruction, which returns exception summary information to PowerPC condition register 1 on completion, is supported wherever the non-dotted form is supported.

If a program attempts to execute an unsupported floating-point instruction, there are two possible outcomes. If the instruction belongs to one of the groups that can be disabled by the APU controller and this group has been disabled, then an exception will be raised. Otherwise, the result is boundedly undefined.

Use the appropriate compiler flags to ensure that unsupported instructions are not generated by the compiler. See the *Xilinx Embedded System Tools Reference Manual* for details.

For information about how to disable decoding of unsupported instructions, see the *Virtex-5 Embedded Processor Block for PowerPC 440 Designs Reference Manual* [Ref 1].

IEEE 754-1985 / Book-E Standard Compliance

The double-precision Floating-Point Unit complies with the majority of the IEEE-754 and Book-E requirements for binary floating-point arithmetic, including support for both single and double precision and all four standard rounding modes. The following list details the FPU's deviations from these standards:

- **Denormalized Numbers.** The standard defines a means of representing very small numbers by allowing significands of the form "0.x" in addition to the usual "1.x" used by normalized floating-point numbers. These are numbers with magnitude less than 2^{-126} (single precision) or 2^{-1022} (double precision). The FPU treats such numbers as zero. If an operation is presented with such a tiny value, it will be treated as an equivalently-signed zero. If an operation would produce such a tiny value, the FPU will indicate an arithmetic underflow and produce a zero result instead.
- **Multiply-Add.** Book-E asks that the multiply part of a multiply-add operation should not round its result before supplying it to the addition part, but the FPU does not operate in this way. The combined operation is simply the equivalent of performing individual multiply and add instructions. This may have implications for certain numerical algorithms that rely on the extra accuracy of the intermediate result in this case. Additionally, the optional negation of the addition result available in the *fnmadd(s)/fnmsub(s)* instructions is applied to the operands at the input of the addition instead. The resulting behavior is identical, except that a zero result will always have a positive sign.
- **Default Results.** The "default IEEE result" is always delivered to the destination register regardless of any exception resulting from the instruction. The following are not supported:
  - The suppression of execution for enabled Invalid Operation Exceptions and enabled Zero Divide exceptions
  - The production of adjusted intermediate results in the case of enabled Overflow/Underflow exceptions
- **NaN handling.** The FPU treats all "not-a-number" (NaN) values as quiet NaNs, which do not cause exceptions. When a floating-point operation results in a NaN because one of the inputs was a NaN, the input NaN is not propagated to the output; the default quiet NaN value is provided. This value is 0x7ff8000000000000 in double precision, and 0x7f800000 in single precision.

All of the above deviations apply to both the single-precision and double-precision variants of the FPU. See "PowerPC Instruction Set Support" on page 4 for a description of those deviations from the Book-E model that apply only to the single-precision FPU.

Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) is implemented as described in the PowerPC Book-E specification [Ref 2]. All instructions for explicit access to this register are supported in all configurations. All FPSCR instructions take approximately 10 FCB clock cycles to execute.

The following lists some minor deviations from the Book-E-specified behavior. Most of these items relate to how the FPSCR bits are set as a by-product of executing arithmetic instructions.

- **Bit 38 - Inexact exception**. Always reads as zero.
- **Bit 39 - Invalid operation (Signalling NaN)**. All NaNs are currently treated as quiet NaNs. This bit always reads as zero.
- **Bit 45 - Fraction Rounded**. Always reads as zero.
- **Bit 46 - Fraction Inexact**. Always reads as zero.
- **Bit 61 - non-IEEE mode**. This bit is ignored. Only IEEE mode is supported.

The FPU supports all four of the Floating-Point Exception modes defined by Book-E:

- Exceptions Disabled mode provides the highest performance. When exceptional conditions arise, they are recorded in the FPSCR and can be explicitly examined later by the software.
- Imprecise Non-Recoverable mode has a reduced issue rate but still supports the pipelining of instructions. Exceptional conditions will cause a program interrupt to occur, but the instruction at which the interrupt is taken will not necessarily be the instruction that caused the exception. Furthermore, there is no way to tell which instruction did cause the exception.
- Imprecise Recoverable mode is treated the same as Precise mode.
- Precise mode disables instruction pipelining; each instruction is completed before the following instruction is started. This reduces performance substantially, but guarantees that any interrupt arising will be taken at precisely the instruction that caused the exceptional condition. This may be useful for debugging purposes.

Software Support

The EDK compiler system, based on the GNU Compiler Collection (GCC), provides support for the APU Floating-Point Unit. It generates code that is compliant with the standard PowerPC Embedded Applications Binary Interface (EABI). Within the EDK framework, compiler flags are automatically added to the GCC command line based on the type of FPU present in the system. The ANSI C data types *float* and *double* are recognized and interpreted as specified in Table 2.

Table 2: **Compiler Flags for Floating-Point Unit Support**

FPU Variant	Compiler Flag	Floating-point Representation	
		float	***double***
Single precision	-mfpu=sp_full	32-bit	64-bit (SW)
Double precision	-mfpu=dp_full	32-bit	64-bit

For the single-precision FPU, all double-precision operations are emulated in software. For the double-precision FPU, all single-precision operations are carried out using the hardware's double-precision operators, as described above in "IEEE 754-1985 / Book-E Standard Compliance" on page 6.

Also, be aware that the `xil_printf()` function does not support floating-point output. The standard C library `printf()` and related functions do support floating-point output, but are large (sometimes too large for a small-footprint embedded system).

Libraries and Binary Compatibility

There are some subtleties surrounding the linking of object code compiled with the various flags described above. These can be summed up by the following observations:

- If hardware floating-point support for a particular precision is not available, the compiler will ensure that values of that type are passed to emulation routines on the stack.

- Otherwise, if support is present, these values will be passed in floating-point registers.

When compiling a monolithic application, there are no issues since the same compiler flags are used for every file. However, when performing separate compilation, or linking against pre-compiled libraries, these object-code differences can cause problems. A mismatch in parameter-passing semantics can cause functions to receive incorrect values. Issuing unsupported instructions, or issuing any floating-point instructions when the APU interface is not enabled, causes undefined behavior. In general, it is not possible to detect these problems until run-time.

The modified GNU C compiler framework supplied with the FPU includes multiple pre-compiled versions of the C runtime libraries (libxil and others). The linker will choose the appropriate library to link against according to the FPU compiler flag used to build the top-level application. No user intervention is required. For all other cases where separate compilation is used, it is very important that you ensure the consistency of FPU compiler flags throughout the build.

Operating System Support

The double-precision full-featured FPU variant is sufficiently compliant with the standard that it should be directly compatible with off-the-shelf third-party compilers, operating systems and application software that expect a Book-E compliant FPU.

For the single-precision variant of the FPU, some modification of the compiler, operating system and software is likely to be necessary. Xilinx can provide supplementary information, including patches for the GNU tool-chain, on request.

In order for FPU instructions to be decoded by the APU controller and executed by the FPU, the FPU enable bit must be set in the Machine State register. See the *Virtex-5 Embedded Processor Block for PowerPC 440 Designs Reference Manual* [Ref 1] for details of this and other system configuration bits. Many multi-tasking operating systems make use of this feature for "lazy" save and restore of floating-point registers on a context-switch. Attempting to execute an FPU instruction when this bit is not set will result in an unsupported instruction exception.

C Language Programming

To gain maximum benefit from the FPU without low-level assembly-language programming, it is important to consider how the C compiler will interpret your source code. Very often the same algorithm can be expressed in many different ways, and some are more efficient than others.

Immediate Constants

Floating-point constants in C are double-precision by default. If you are using a single-precision FPU, careless coding may result in double-precision software emulation routines being used instead of the native single-precision instructions. To avoid this, explicitly specify (by cast or suffix) that immediate constants in your arithmetic expressions are single-precision values.

For example:

```
float x=0.0;
…
x += (float)1.0; /* float addition */
x += 1.0F;       /* alternative to above */
x += 1.0;        /* warning - uses double addition! */
```

The GNU C compiler can be instructed to treat all floating-point constants as single-precision (contrary to the ANSI C standard) by supplying the compiler flag `-fsingle-precision-constant`.

Avoid Unnecessary Casting

While conversions between floating-point and integer formats are supported in hardware by the FPU, it is still best to avoid them when possible. Such casts require transfers between the floating-point and

the integer register files, which in the PowerPC architecture always go via memory. These transfers are a bottleneck and can cause performance degradation. This applies to both single- and double-precision FPU variants.

The following "bad" example calculates the sum of squares of the integers from 1 to 10 using floating-point representation:

```
float sum, t;
int i;

sum = 0.0f;
for (i = 1; i <= 10; i++) {
   t = (float)i;
   sum += t * t;
}
```

The above code requires a cast from an integer to a float on each loop iteration. This can be rewritten as:

```
float sum, t;
int i;

t = sum = 0.0f;

for(i = 1; i <= 10; i++) {
   t += 1.0f;
   sum += t * t;
}
```

By making the loop code independent of the integer loop counter, all code inside the loop is carried out using the FPU. The compiler is not at liberty to perform this optimization in general, as the two code fragments above may give different results in some cases (for example, very large *t*).

Runtime Library Functions

The standard C runtime math library functions operate using double-precision arithmetic. When using a single-precision FPU, calls to certain functions (such as `fabs()` and `sqrt()`) result in inefficient emulation routines being used instead of FPU instructions:

```
float x=-1.0F;
…
x = fabs(x);  /* uses double precision */
x = sqrt(x);  /* uses double precision */
```

When used with single-precision data types, the result is a cast to double, a runtime library call (which cannot use the FPU) and then a truncation back to float.

The solution is to use the non-ANSI standard functions `fabsf()` and `sqrtf(x)` instead, which operate using single precision and can be carried out using the FPU. For example:

```
float x=-1.0F;
…
x = fabsf(x); /* uses single precision */
x = sqrtf(x); /* uses single precision */
```

Array Accesses and Pointer Ambiguity

It is difficult for the compiler to detect when two memory references (such as array element accesses) refer to the same location or not. The expected behavior is for the compiler to treat almost all array and pointer accesses as if they conflict. For example, the following code forms the inner loop of a simple Cooley-Tukey FFT algorithm implementation:

```
tr = ar0*Real[k] - ai0*Imag[k];
ti = ar0*Imag[k] + ai0*Real[k];
Real[k] = Real[j] - tr;   /* A */
```

```
Imag[k] = Imag[j] - ti;
Real[j] += tr;              /* B */
Imag[j] += ti;
```

Because the compiler does not know that `Real[k]` and `Real[j]` are never the same element, the addition in statement *B* cannot start until the addition in statement *A* is finished. This spurious dependency limits the amount of parallelism and slows down the computation. One possible solution is to introduce some temporary variables, and separate the memory accesses from the mathematics, like this:

```
r_k = Real[k]; i_k = Imag[k];
r_j = Real[j]; i_j = Imag[j];
tr = ar0*r_k - ai0*i_k;
ti = ar0*i_k + ai0*r_k;
r_k = r_j - tr;
i_k = i_j - ti;
r_j += tr;
i_j += ti;
Real[j] = r_j; Real[k] = r_k;
Imag[j] = i_j; Imag[k] = i_k;
```

While this code is less concise, it gives much better results.

Arrays and pointers can often limit the compiler's ability to allocate variables to registers. If a design has small arrays of floating-point values, better performance may be possible if a small number of individual variables are declared instead (like, `float a0, a1, a2` instead of `float a[3]`), and loops that index into them are unrolled.

Core Parameters

The FPU core has three parameters that influence its implementation, as shown in the table below:

Parameter	Default	Value and Meaning	
		0	***1***
C_DOUBLE_PRECISION	1	Single-precision FPU	Double-precision FPU
C_USE_RLOCS	1	Implemented as random logic	Implemented as RPM
C_LATENCY_CONF	0	High-speed variant	Low-latency variant

To obtain the highest possible performance, `C_USE_RLOCS` should be set to 1 so that the FPU is implemented as a relatively placed macro (RPM). However, in case this pre-placed FPU conflicts with other placement-constrained logic in a design, a version containing no RLOCs is also provided.

When the RPM variant of the FPU is selected, it may sometimes be necessary to ensure that the placement algorithm finds the correct location for the macro by adding a constraint to the design. For designs implemented on FX30T and FX70T devices, this constraint is automatically added when Platgen elaborates the design. For larger devices which contain multiple PowerPC-440 sites, this constraint may need to be added manually. The lines to be included in the project UCF file are shown in the table below. (The text `<fpu_instance>` stands for the name of the `apu_fpu_virtex5` instance in question.)

Device	PowerPC site	Constraint
FX100T	PPC440_X0Y0	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y0;`
	PPC440_X0Y1	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y80;`

Device	PowerPC site	Constraint
FX130T	PPC440_X0Y0	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y20;`
	PPC440_X0Y1	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y100;`
FX200T	PPC440_X0Y0	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y40;`
	PPC440_X0Y1	`INST "<fpu_instance>*.netlist" RLOC_ORIGIN=X0Y120;`

The `C_LATENCY_CONF` parameter controls the latency of the floating point operators. In most systems, where the FPU is running at half the speed of the CPU clock, this parameter should be set to 0 (the default), to obtain the highest possible operating frequency. If the FPU is running at some lower ratio of the CPU clock speed (e.g. one third), or if the CPU itself is operating well below its rated maximum frequency, better overall performance may be obtained by setting this parameter to 1.

Table 3 shows the latencies of the various operations supported by the FPU. The add and multiply operators are fully pipelined, so a new operation can be initiated on each FPU clock cycle. The divide and square-root operators (if implemented) are not pipelined, so only one divide and one square-root operation can be ongoing at any time. The clock cycle figures shown include the time required to read and write the FPU register file.

Table 3: **FPU Operator Latencies and Frequencies**

Instruction	C_LATENCY = 0 (high speed)		C_LATENCY = 1 (low latency)	
	Single	Double	Single	Double
Add, Subtract	5	6	3	4
Multiply	4	6	3	4
Divide	29	60	16	60
Square Root	29	59	16	59
Convert	5	6	3	4
Fused Multiply-Add/Sub	9	12	6	8
Move, Abs, Neg, *etc.*	1	1	1	1
Round	N/A	6	N/A	4
Compare	4	4	4	4
Maximum frequency (-1)	200MHz	200MHz	140MHz	140MHz
Maximum frequency (-2)	225MHz	225MHz	160MHz	160MHz

A floating-point instruction can be issued on every FPU clock cycle (usually every other CPU clock cycle).

The PowerPC architecture does not specify instructions for moving data between CPU registers (GPRs) and floating-point registers (FPRs). All FPU data transfers are therefore between the FPRs and main memory (or data cache, if used). A data load from cache can be performed in a single FPU clock cycle. Note that the APU controller cannot process more than one outstanding load instruction, so this latency occurs on each load. Floating-point store operations take three FPU clock cycles (assuming that there is no data dependency on a previous instruction whose result is still outstanding).

References

1. *Virtex-5 Embedded Processor Block for PowerPC 440 Designs Reference Manual*
2. *Book E: Enhanced PowerPC™ Architecture*. 2002. IBM Corporation.
3. Hennessy, John L. and David A. Patterson. 1996. *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann.

Support

Xilinx provides technical support for this LogiCORE product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled *DO NOT MODIFY*.

Revision History

Date	Version	Revision
02/18/08	0.6	Provisional Xilinx release.
04/16/08	1.0	Initial Xilinx release.

Notice of Disclaimer