APU Floating-Point Unit v3.1

March 11, 2008 **Product Specification**

Introduction

The Xilinx Auxiliary Processor Unit (APU) Floating-Point Unit LogiCORE™ is a single-precision floating-point unit designed for the PowerPC™ 405 embedded microprocessor of the Virtex™-4 FX FPGA family. It is tightly coupled to the PowerPC™ core with the APU interface. The FPU is not Power-ISA compliant and provides support for floating-point arithmetic operations in single precision only. With compiler modifications provided by Xilinx, single-precision floating point instructions can be executed to achieve increased performance over software emulation.

Features

- Compatible with the *IEEE-754* standard for single-precision floating-point arithmetic, with minor and documented exceptions
- Decodes and executes standard single-precision PowerPC floating-point instructions
- Uses autonomous instruction issue to hide arithmetic latency and decrease cycles per instruction
- Provides optional divide and square-root operators for increased flexibility
- Optimized implementation leverages Virtex-4 high-performance DSP features
- Integrated into Xilinx Embedded Development Kit (EDK) design flow
- Built upon intellectual property licensed from QinetiQ ltd.

QinetiQ

LogiCORE Facts			
Core Specifics			
Supported Device Family	Virtex-4 FX		
Resources Used	Slices	Xtreme-DSP Blocks	Block RAMs
Lite (no div/sqrt)	1300	4	2
Full (with div/sqrt)	1600	4	2
Clock Speeds	-10 part	233 MHz	
	-11 part	275 MHz	
	-12 part	333 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		
Reference Design	http://www.xilinx.com/products/ipcenter/DO-DI-FPU-SP.htm		
Design Tool Requirements			
Xilinx Implementation Tools	ISE™ 9.1i		
Verification	ModelSim® PE 5.4e		
Simulation	ModelSim PE 5.4e		
Synthesis	XST		
Support			
Provided by Xilinx, Inc @ www.xilinx.com			

Functional Overview

The APU Floating-Point Unit comprises a number of 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 FPU architecture.

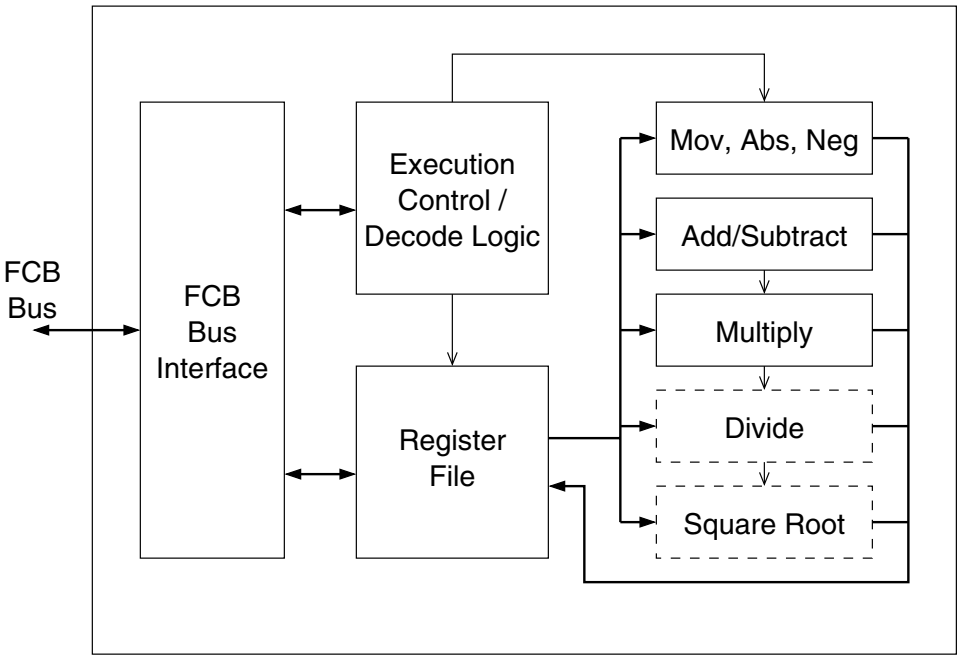

Figure 1: **Top-level APU Floating-Point Coprocessor Architecture**

Applications

The APU Floating-Point Unit augments the capabilities of the PowerPC 405 processor core with support for floating-point instructions. Many software applications make use of floating-point (or real) arithmetic, whether for occasional calculations or for intensive computation kernels. Following are some examples of application areas where floating-point arithmetic can be useful.

- 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.
- DSP 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 three 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 operations.

Figure 2 illustrates a typical embedded system containing an APU Floating-Point Unit.

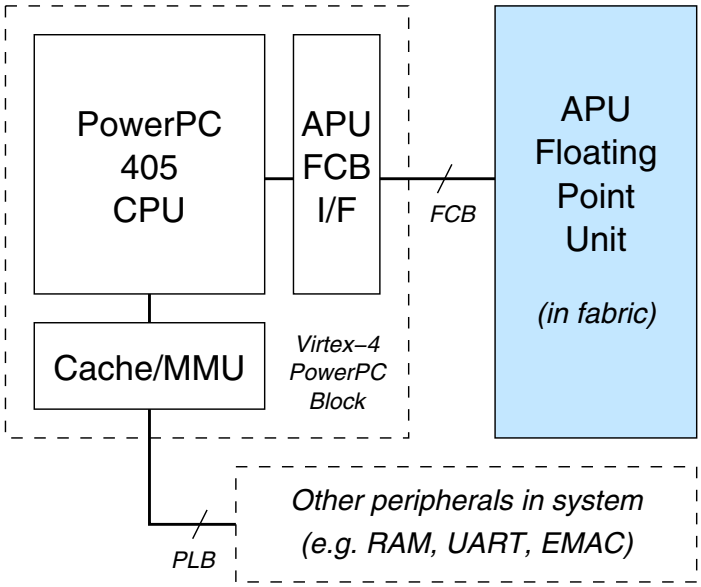

Figure 2: **Embedded System Containing an APU-FPU Core**

Functional Description

Register File

The Register file contains 32 floating-point registers. The width of these registers is 32 bits (single precision). The APU/FCM interface ports are also 32 bits wide.

The APU/FCM interface between the Floating-Point Coprocessor and the PowerPC 405 core is described in detail in the *PowerPC 405 Processor Block Reference Guide* (reference 1 in this document).

PowerPC Instruction Set Support

The FPU supports two operator configurations (full and lite). 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,* (reference 2).

In the full configuration, the addition, subtraction, multiplication, division, and square root operations are supported, along with fused multiply-add and its sign-modified variants. The lite mode is similar, but divide and square root operators are not included. The reciprocal estimate and reciprocal square root estimate functions are not supported in either configuration.

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

Instruction	Description	Full	Lite
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	No	No
stfd(u)(x)(e)	Store floating-point double	No	No
stfiwx(e)	Store float as integer word	Yes	Yes
fabs	Absolute value	Yes	Yes
fmr	Move	Yes	Yes
fnabs	Negative absolute value	Yes	Yes
fneg	Negate	Yes	Yes
fadd	Add	SP	SP
fadds	Add (single)	Yes	Yes
fdiv	Divide	SP	No
fdivs	Divide (single)	Yes	No
fmul	Multiply	SP	SP
fmuls	Multiply (single)	Yes	Yes
fsqrt	Square root	SP	No
fsqrts	Square root (single)	Yes	No
fsub	Subtract	SP	SP
fsubs	Subtract (single)	Yes	Yes
fmadd	Multiply-add	SP	SP
fmadds	Multiply-add (single)	Yes	Yes
fmsub	Multiply-subtract	SP	SP
fmsubs	Multiply-subtract (single)	Yes	Yes
fnmadd	Negative multiply-add	SP	SP
fnmadds	Negative multiply-add (single)	Yes	Yes
fnmsub	Negative multiply-subtract	SP	SP
fnmsubs	Negative multiply-subtract (single)	Yes	Yes
fcfid	Convert from integer double-word	NS	NS
fctid	Convert to integer double-word	No	No
fctidz	As fctid, but round to zero	No	No
fctiw	Convert to integer word	Yes	Yes
fctiwz	As fctiw, but round to zero	Yes	Yes
frsp	Round to single precision	No	No
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	No	No
mffs	Move from status/control register	Yes	Yes

Key: Yes = supported; No = not supported; SP = operation performed in single-precision; NS = non-standard (see note)

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

Instruction	Description	Full	Lite
mtfsb0	Move to status/control register bit 0	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 (see note)			

Notes:

1. The 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 behaviour is non-standard, but allows hardware acceleration of format conversions that would not otherwise be possible in a single-precision unit.

The "dotted" instruction forms, which return exception summary information to PowerPC condition register 1 on completion, are not supported in any configuration.

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 PowerPC GNU Compiler Reference Manual for details.

IEEE 754-1985 Standard Compliance

The Floating-Point Unit complies with the IEEE-754 standard for binary floating-point arithmetic, with the following exceptions.

- **Rounding Modes**. Except for those instructions that explicitly round their result towards zero, the only supported rounding mode is round-to-nearest (the default in IEEE-754). An attempt to configure the FPU to operate in any other rounding mode will have no effect.
- **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” of normalized floating-point numbers. These are numbers with magnitude less than 2^{-126}, and the FPU treats such numbers as unrepresentable. If an operation produces such a value, the FPU will indicate an arithmetic underflow. If an operation is presented with such a value, it will be treated as if that value were an equivalently-signed zero.
- **Double-on-Single**. The FPU accepts and executes double-precision arithmetic operations using the single-precision hardware operators. The *fadd* and *fadds* instructions are indistinguishable.

Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) is implemented as described in the PowerPC Book-E specification (reference 2). All instructions for explicit access to this register are supported except for mcrfs (that is: mffs, mtfsb0, mtfsb1, mtfsf and mtfsfi). All FPSCR instructions (other than mffs）take approximately 36 FCB clock cycles to execute.

The following lists some minor deviations from the Book-E-specified behavior. They are mostly concerned with exactly 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 44 - Invalid compare exception.** Always reads as zero.
- **Bit 45 - Fraction Rounded**. This bit always reads as zero.
- **Bit 46 - Fraction Inexact**. This bit always reads as zero.
- **Bits 48-51 - Result flags.** These bits are not set by compare operations.
- **Bit 61 - non-IEEE mode**. This bit is ignored. Only IEEE mode is supported.
- **Bits 62:63 - Rounding Control**. Round-to-Nearest mode is always used; setting these bits has no effect.

The only Floating-Point exception modes currently supported is “exceptions ignored” mode.

Adding an FPU to an EDK Project

As of EDK 9.1i the Base System Builder wizard allows the APU FPU to be included in a project by means of a simple check-box in the PowerPC configuration dialog. Provided that the clocking architecture that was chosen is compatible with the FPU, the wizard will automatically add the FPU and FCB cores and make the appropriate connections between them. The only requirement is that the system (bus) clock be running at half the frequency of the processor clock.

If there is no half-rate clock, or it is necessary to add the APU FPU to an existing PowerPC-based EDK project, then two basic steps are required. These steps are described in more detail in this section.

1. Add the FPU and the associated Fabric Coprocessor Bus (FCB) interface to the project.
2. Wire up the appropriate clock and reset signals to ensure correct FCB/FPU operation.

There are two clock domains in a PowerPC-FPU system—the PowerPC (core) clock and the FPU clock. The FPU logic runs internally at half the speed of the PowerPC. The FCB interface must utilize the same clock as the PowerPC (i.e. a 1:1 ratio). If your project does not already have a suitable clock signal for the FPU’s internal half-rate clock, consult the platform studio documentation for how to add an appropriate DCM module, or modify an existing one to produce an appropriate clock signal. Details of the achievable clock speeds for the FPU can be found in the LogiCORE Facts table on page 1.

To quickly get started using the FPU, use the example EDK project from the reference design provided as a starting point. The ML403 reference design supplied with the FPU is configured with the PowerPC at 200 MHz, and the FPU and system logic at 100 MHz. For more information, see "Example Design Source Files" on page 16.

Using the FPU in EDK

This section contains the procedure for adding the FPU to an EDK project. The steps illustrated are for EDK v9.1i, but the general procedure also applies to later versions of EDK.

Adding FPU and FCB Cores

1. Click the IP catalog tab on the Platform Studio window.

   The apu_fpu core appears under the arithmetic category.
2. Double-click apu_fpu to add the FPU to your project.

   The fcb_v1_0 core appears under the bus category.
3. Double-click fcb_v1_0 to add the FCB bus to your project.
4. Go to the System Assembly view and select Bus Interface from the Filters radio selector at the top of the page.

The FCB appears as a bus connecting the PowerPC-405 core (master) to the APU FPU core (slave).

5. Expand these two instances to reveal the MFCB and SFCB interfaces inside.
6. Click on the empty square and the empty triangle on the FCB bus line to connect the master and slave interfaces together.

   An example of a correct FCB configuration is shown in Figure 3.

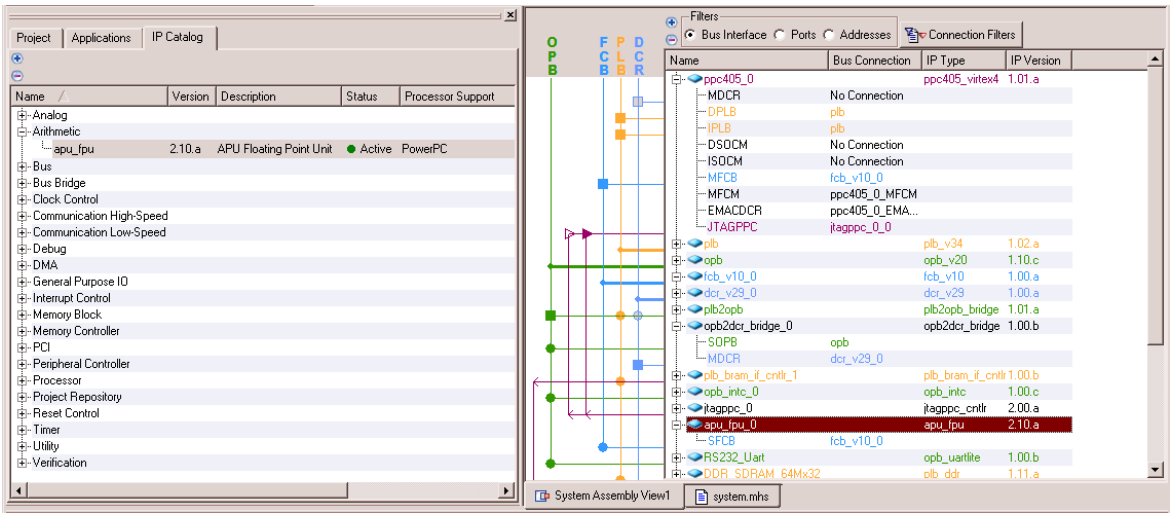

Figure 3: **FCB Bus Connection Between PowerPC and APU-FPU**

7. Right-click the FPU instance in the System Assembly view, and select Configure IP.
8. Set the feature-set of the FPU according to your requirements.
9. Right-click the PowerPC CPU instance in the System Assembly view, and select Configure IP.
10. Click the tab marked APU, and modify the APU Controller Configuration Register Initial Value to *0b0000000000000001* to enable the APU interface.

Wiring the FCB Clock Signals

The example in this section assumes that the PowerPC core is using a clock called `PROC_CLK_S`, and the FCB is running at half this frequency, using a clock called `SYS_CLK_S`. These are the default system clock names provided by the Base System Builder tool. You can replace these names with the appropriate clock signals in your system.

1. Go to the System Assembly view and select the Ports filter.
2. Expand the FCB instance to see its ports.
3. To specify the signal connected to a port, select the port and click in the Net column, then select or type the name of the appropriate signal name.

   If desired, you can create a new internal signal by typing a name that does not currently exist within the system.
4. The `FCB_CLK` port must be connected to the PowerPC processor clock (`PROC_CLK_S` in this example). The `SYS_RST` port must be connected to the system reset net, usually called `SYS_BUS_RESET`.
5. Now expand the FPU instance to see its ports.
6. The `FCB_CLK` port must be connected to the PowerPC processor clock (`PROC_CLK_S` in this example). The `FPU_CLK` port must be connected to the half-rate clock (`SYS_CLK_S` in this example).

Software Support

The only target platform supported is the standalone board support package. The EDK compiler system, based on GCC, provides support for the APU Floating-Point Unit compliant with the PowerPC EABI. 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 in Table 2.

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

FPU Variant	Compiler flag	Floating-point representation	
		float	***double***
Single precision without sqrt, div	-mfpu=sp_lite	32-bit	64-bit (SW)
Single precision with sqrt, div	-mfpu=sp_full	32-bit	64-bit (SW)

All double-precision operations are emulated in software. 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 very large (often 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.

Programming Considerations

The performance of floating-point code running on the PowerPC with an APU FPU depends on the dataflow properties of the algorithm in question, and on the latency and throughput properties of the FPU itself. Both of these factors are examined briefly in this section.

Operator Latencies and Initiation Intervals

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 FP register file.

Table 3: **FPU Operator Latencies**

Instruction	FPU Clock Cycles Required
Add, Subtract	6
Multiply	5
Divide	17
Square Root	17
Convert	6
Fused Multiply-Add/Sub	10
Move, Abs, Neg, *etc.*	2
Compare	3

The FPU clock runs at half the speed of the CPU clock. In the current implementation, two FPU clock cycles are required to issue a floating-point instruction. Thus, an instruction can be issued every four CPU clock cycles.

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 takes at least three FPU clock cycles. Note that the current APU controller cannot process more than one outstanding load instruction, so this latency occurs on each load. Floating-point store operations take four FPU clock cycles (assuming that there is no data dependency on a previous instruction whose result is still outstanding).

When performing a sequence of multiply-accumulate operations using fused multiply-add instructions, note that the initiation interval is only 6 cycles (time taken for an addition) rather than the full 10 cycles. The FPU achieves this by deferring the addend read in multiply-add instructions until the latest possible point, thus increasing the performance of key DSP algorithms.

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. When 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! */
```

Note that 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-constants`.

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.

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. Note that 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 is made (which cannot use the FPU) and then a truncation back to float is performed.

The solution is to use the non-ANSI 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.

Also remember that arrays and pointers can often limit the compiler's ability to allocate variables to registers. If you have small arrays of floating-point values, better performance may be possible if you declare a small number of individual variables instead (*i.e.* `float a0, a1, a2` instead of `float a[3]`), and unroll any loops that index into them.

Algorithm Optimization Example - FIR Filter

The theoretical peak performance is determined by the maximum issue rate. With the CPU running at 233 MHz, 58 million floating-point instructions can be issued per second. Because the FPU supports fused multiply-add instructions (which perform two floating point operations for one issue) the peak performance figure is 116 MFLOPS. In practice, this figure will not be attainable due to load and store instructions overhead, loop control, and stalls due to data hazards.

It is clear from the data that floating-point operations performed by the FPU have considerably higher latencies than integer operations carried out within the PowerPC core. To make best use of the FPU resources, the pipelined operators must be kept supplied with useful work. For example, consider an algorithm whose inner loop consists solely of single-precision multiply–add instructions. As noted in Table 3, this operation takes 10 CPU clock cycles to execute. With an issue rate of 1/4, three instructions could be issued before the first one has completed. For this to be achievable, there must not be any dependencies between these three instructions. If one instruction depends on the result of another that is still executing, the later instruction must be stalled until the earlier one completes. Such stalls cause a decrease in performance levels.

There are three ways to discover independent instructions:

- by hardware at run time
- by the compiler at compile time
- by the programmer at design time

Hardware discovery usually requires support for out-of-order execution—not provided by the current implementation of the FPU. Discovery by the compiler requires an optimizing compiler. The current implementation of gcc is capable of performing some basic instruction scheduling functions, but does not perform advanced loop transformations. Therefore, to obtain highest performance, the programmer must sometimes be prepared to rephrase the algorithm.

Example: Parallelism in a FIR Filter

Consider the following simple piece of code to implement a FIR filter:

```
for (j = 0; j < nsamples; j++)
{ x = input++;
  accum = 0.0;
  for (i = 0; i < ntaps; i++)
    accum += coeffs[i] * *x--;
  *output++ = accum;
}
```

The value of the accumulator resulting from iteration *N* of the inner loop will be required again as an input to iteration *N+1*. This means that execution of the multiply–add instructions cannot be allowed to overlap. To improve matters, it is possible to use two accumulators: one for odd taps and one for even taps. The code will then look similar to this:

```
for (j=0; j < nsamples; j++)
{ x = input++;
  accum1 = accum2 = 0.0;
  for (i = 0; i < ntaps; i+=2)
  { accum1 += coeffs[i] * *x--;
    accum2 += coeffs[i+1] * *x--;
  }
  *output++ = accum1+accum2;
}
```

The two multiply–add operations within the inner loop can now be performed in parallel, and the FPU allows them to proceed through the pipeline together. Performance rates will be better, but still lower than the maximum available. Note that these two are equivalent only if the number of taps is even. The two functions may also give subtly different results because the order in which they perform the additions differs.

Optimization of data movement

The parallel-accumulator approach described above can be extended to use a number of accumulators to provide enough independent operations to fill the pipeline. However, there is another problem with this method. Each multiply-add instruction requires two new pieces of data to be loaded into floating-point registers, further limiting the number of parallel operations. Only one in three of the issued instructions is doing useful work.

To remedy this, look at the outer loop as well as the inner loop. If multiple samples can be processed in parallel, then each input sample can be reused once it has been read into the floating-point register file. Consider the equations below, which give the calculations required for 8 samples of an 8-tap FIR filter:

$$
\begin{aligned}
r_0 &= \mathbf{h_0 x_0} + h_1 x_{-1} + h_2 x_{-2} + \ldots + h_7 x_{-7} \\
r_1 &= h_0 x_1 + \mathbf{h_1 x_0} + h_2 x_{-1} + \ldots + h_7 x_{-6} \\
r_2 &= h_0 x_2 + h_1 x_1 + \mathbf{h_2 x_0} + \ldots + h_7 x_{-5} \\
&\ldots \\
r_7 &= h_0 x_7 + h_1 x_6 + h_2 x_5 + \ldots + \mathbf{h_7 x_0}
\end{aligned}
$$

The terms hilighted in bold all use the first input sample (index 0), and can all be performed in parallel as they contribute the different accumulators. By allowing computation to proceed diagonal by diagonal, the reuse of input (and coefficient) values can be greatly improved. Of course, any incomplete diagonals require special handling. Usually the number of taps will be larger than the number of samples to be computed in parallel, so a three-phase computation strategy (spin up, compute, spin down) can be employed. All of these optimizations can be expressed using standard ANSI C, without recourse to non-portable assembly language.

Understanding the parallelism and data locality inherent in an algorithm is the key to constructing a high-performance implementation. However, keep in mind that optimizations such as loop unrolling and software pipelining will always increase code size. Clearly, the code optimization process is not always a simple one. For a comprehensive treatment of the subject of instruction-level parallelism, refer to *Computer Architecture: A Quantitative Approach* (reference 3).

Example Design

The APU FPU is supplied with a demonstration design that contains several applications that are heavily dependent on floating-point. All the applications are graphical, displaying their output on a standard ANSI terminal using RS232. A facility is also provided for profiling the various routines. Full C source code is included.

The applications currently included in the demonstration suite are:

- **Mandelbrot and Feigenbaum fractal generation**
  These fractals arise from simple iterative conformal mappings of complex and real numbers respectively. The inner loops consist of multiplication, addition and comparison instructions. Some divisions and integer conversions are also performed.
- **Gaussian random number tester**
  This application generates normally-distributed random numbers by the Box-Muller transformation method, and compares the resulting distribution with a perfectly computed Gaussian bell curve. This involves several add, square root and divide operations, as well as numerous calls to the math library for exponentials and logarithms.
- **Solution of the traveling salesman problem by simulated annealing**
  The Metropolis algorithm searches for a good (but not provably optimal) solution to the NP-complete Travelling Salesman problem. The path metric routines rely heavily on addition, multiplication and square root to find the Euclidean distance between two points in the itinerary. The cooling schedule requires the calculation of exponentials.
- **Fast Fourier transform.**
  This application contains a straightforward routine to calculate the Fourier transform of a data series. The inner FFT loop is of the Cooley-Tukey (decimation-in-time) variety. The application generates a variety of real-valued waveforms (noise, square waves, sine waves), computes their Fourier transforms, then plots both the signals and their spectra.

Building and Running the ML403 Demonstration

Extract the example design files to a working directory, and start Platform Studio. Open the project file system.xmp in the top-level directory. You can now browse through the hardware and software elements of the example design.

The project contains a PowerPC processor core, along with a single-precision FPU, some block RAM for program and data storage, and an RS232 UART for user interaction. The default configuration has a full-featured FPU with divide and square-root operations. This can easily be changed from the Add/Edit Cores dialog. However, if the goal is to experiment with the relative performance of different FPU configurations or software emulation, it is more convenient to vary the FPU type compiler flag instead. That way, there is no need to rebuild the hardware repeatedly (just the software, which is much quicker to compile).

The build procedure for this project is exactly the same as for any other Platform Studio project. There is no external memory configured, so the FPGA bitstream contains the full executable for the design and can be downloaded "all at once" over JTAG.

Table 4: **RS-232 Terminal Settings for Demo Application**

Parameter	Value
Baud Rate	115200
Data bits, Parity, Stop bits	8-N-1
Flow Control	None
Terminal Emulation	ANSI
Font	TERMINAL
Force to 7-bit ASCII	Off
Wrap long lines	Off

To interact with the demo application, connect the ML403 to your PC using a standard null-modem serial cable, and attach a terminal emulator to the appropriate COM port. The parameters you should

set up are shown in Table 4. A pre-configured terminal settings file for Hyper terminal (*.ht)* is included in the top-level directory.

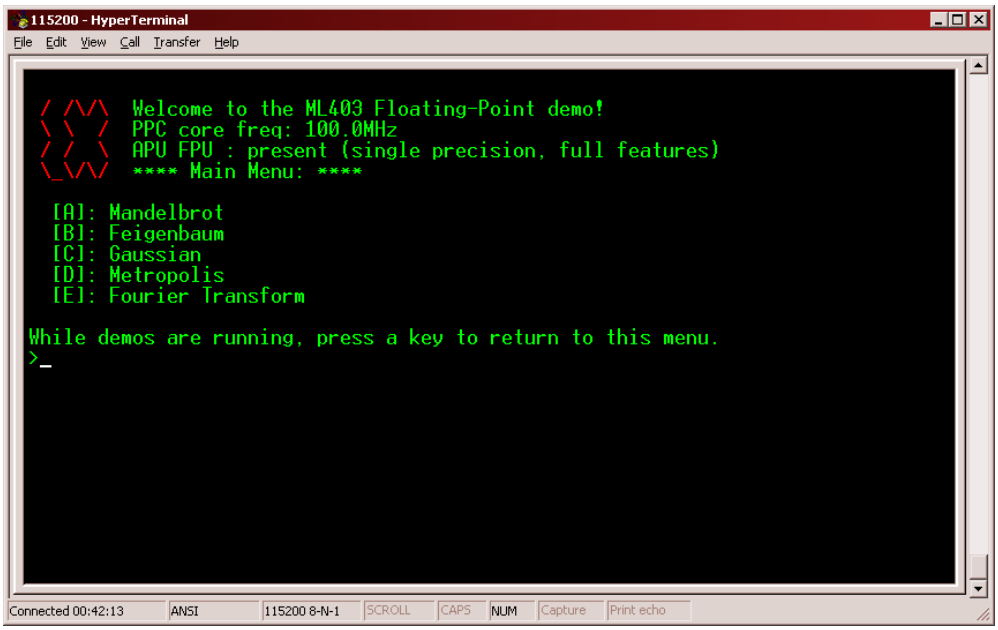

Figure 4: **ML403 Demo Application - Main Menu**

When the program starts up, you should see a menu on the terminal emulator screen similar to that shown in Figure 5. Simply press the key corresponding to the demo you want to run. Pressing a key during a demo returns you to this main menu. Figure 6 shows an example screens from the Mandelbrot demo.

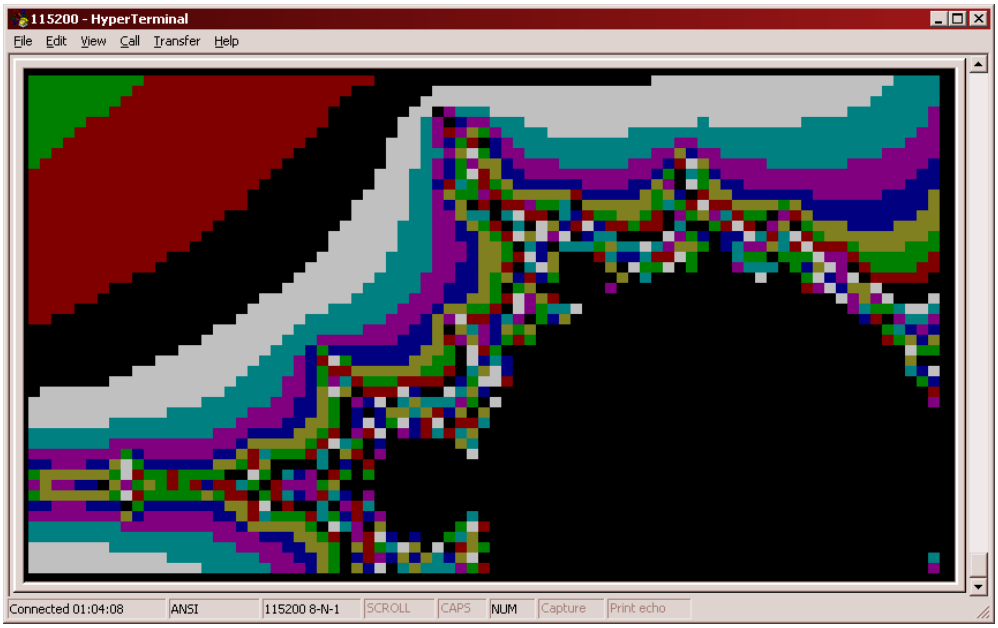

Figure 5: **Mandelbrot Set Demo Screen**

If you have enabled profiling (by defining the symbol PROFILING in the profile.h file), a summary of execution time is shown at the end of each demo. All figures are in PowerPC clock cycles.

Example Design Source Files

Table 5 gives an overview of the source code files included with the ML403 demonstration project, along with a brief description of the contents of each file.

Table 5: **Source code files included with example design**

File(s)	Description
demos.h	Prototypes and configuration definitions for demo functions
mand.c	Draws the Mandelbrot set
fig.c	Draws Feigenbaum's "fig-tree" fractal
gaussian.c	Demonstrates the generation of normally-distributed random numbers
metropolis.c	Solves the Traveling Salesman problem by simulated annealing
fourierf.c	Performs 64-point Fast Fourier Transforms
qrand.c, qrand.h	Random number generation
ttygfx.c, ttygfx.h	Graphics display on an RS232 ANSI terminal
uartio.c, uartio.h	Handles UART input without the overhead of the C standard library
profile.c, profile.h	Utilities for measuring execution times
xmath.h	Wrapper around standard `math.h` include
TestApp_Memory.c	Top level demo application

Resource Utilization and Performance

Table 6 shows silicon resource utilization figures for the various supported FPU configurations. Note that the presence of any fabric co-processor on the APU interface causes the maximum operating frequency of the PowerPC processor to drop. See latest Virtex-4 FX datasheet for exact figures. When the FPU is connected to the APU interface of a PowerPC core through the FCB, operation is possible at up to a 233 MHz FCB clock frequency (-10 part).

Table 6: **Resource Usage of Floating-Point Unit (approximate)**

Coprocessor Variant	Resource Usage		
	Slices	Xtreme-DSP blocks	Block RAMs
Lite (no div/sqrt)	1300	4	2
Full (with div/sqrt)	1600	4	2

Note that minor variations in maximum operating frequencies and slice counts may be seen between different versions of Xilinx implementation tools. Higher performance may be achievable by altering the mapping and/or place and route options, or by selecting a device with a faster speed grade. Remember that the run-time of the implementation tools can often be improved by providing a sufficient timing margin.

References

[1] PowerPC™ 405 Processor Block Reference Guide (v2.0); August 20, 2004; Xilinx ref. UG018

[2] Book E: Enhanced PowerPC™ Architecture Version 1.0; May 7, 2002; IBM Corporation

[3] Computer Architecture: A Quantitative Approach; 2nd Ed, 1996; J. L. Hennessy & D. A. Patterson

Revision History

Date	Version	Revision
3/21/05	1.0	Provisional Xilinx release
9/23/05	1.1	Updated for version 1.1 of FPU hardware
1/26/06	1.2	Updated for version 2.0 of FPU hardware
2/13/06	1.3	Corrected instruction support and size information
5/1/06	2.0	Document updated for early release v2.0
6/21/06	2.5	Updated for v2.1 release.
1/26/07	3.0	Updated for v3.0 release.
3/11/08	3.1	Updated for v3.1 release.