WP348 (v1.1) August 30, 2008

MicroBlaze System Performance Tuning

By: Richard Griffith and Felix Pang

A previous article, "Designing a Custom Processor Peripheral Using Xilinx EDK," explored methods of creating custom processor peripherals to be attached to the Processor Local Bus (PLB v4.6). Adding custom hardware to a processor system is a powerful way of increasing its power, but adding that hardware as a peripheral on a bus is not always the best way to solve a given problem. Busses are designed to be flexible and to permit many different functional blocks to share a common communications path. This flexibility is achieved at the expense of maximum performance; because all bus transactions require arbitration and handshaking to occur, this can be wasteful in time-critical applications. This White Paper examines ways to add custom hardware to a processor to achieve hardware acceleration without sacrificing performance of the processor or the bus to which it is attached.

The Mission

I was recently asked to analyze a customer's processor system to identify possible performance bottlenecks. The design in question used a MicroBlaze^{TM} soft processor core to perform, amongst other tasks, vector mathematics on a series of grid coordinates to calculate the distance between two points on an X/Y grid.

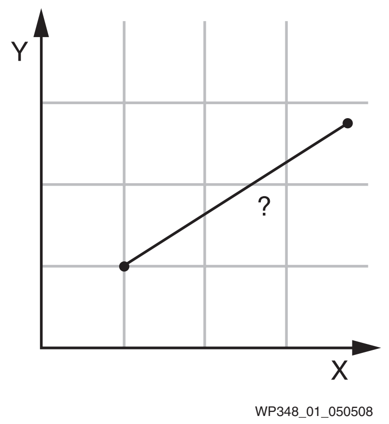


Figure 1: **The Distance Between Two Points on an X/Y Grid**

The design was running too slowly to meet the performance requirements of the system, and the customer requested that I find ways to accelerate the design in the most efficient way.

Closer examination showed that the design was fundamentally using Pythagoras' theorem to perform each calculation; many thousands of vectors were being continuously calculated from the source data containing grid coordinates that were stored in very large arrays.

Most of you are familiar with Pythagoras' theorem: In a right-angled triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides. If we consider the following triangle as an example, the line on the hypotenuse "C" represents the vector with a length that we wish to calculate. The horizontal "A" and vertical "B" sides of the triangle represent the grid lines, which can be calculated using simple subtraction (X2-X1, and Y2-Y1).

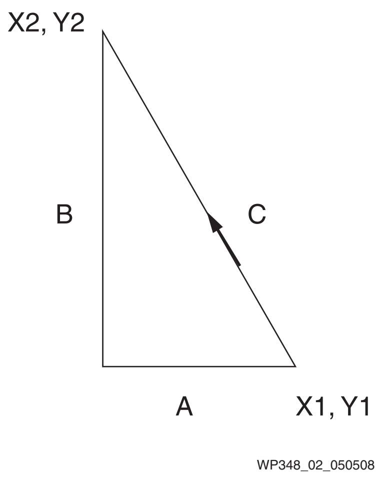


Figure 2: **Pythagoras' Theorem**

Using Pythagoras' theorem, we can calculate the squares of the sides, as illustrated below; this in turn allows us to calculate the length of the vector of the hypotenuse of the original triangle:

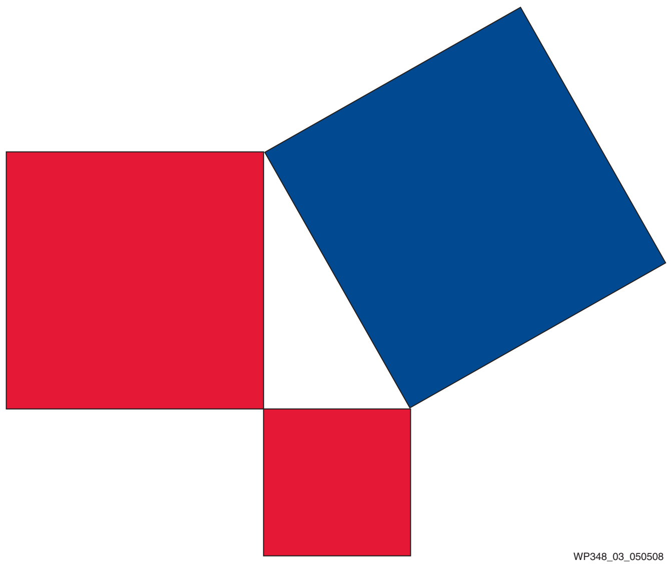


Figure 3: **Calculating the Squares of the Sides**

The Science Behind the Theory

Implementing this calculation in software is fairly simple using the equation $C^2 = A^2 + B^2$; however, we must make a detailed examination about how this would be coded in software for a processor.

Using a simple triangle as an example, the easiest way would be to declare two variables "A" and "B" to represent the length of the horizontal and vertical grid lines calculated from the values of the coordinates of points X1, Y1 and X2,Y2. Using these values, we can then calculate the whole thing in one simple line of C code using the square root function called "sqrt" from the "math.h" library:

```
result = sqrt ((a*a) + (b*b));
```

It looks easy, doesn't it? Problem solved? Not quite. By using this one line of code in C, we have successfully performed an accurate calculation using Pythagoras' theorem. It is a working solution, it produces consistently correct results, and it is precisely what was written in the piece of code that I received from the customer.

However, the performance of this solution leaves much to be desired. In the customer's example, the above line of code was used inside a loop, which performed not just one calculation using Pythagoras' theorem, but many thousands of calculations for thousands of triangles. The lengths of the sides of the triangles were stored in huge arrays in the C code, and the results were calculated as one small part of the overall software application.

Identifying the Bottlenecks

Fixing a performance bottleneck in a system is one challenge, but the problem that frequently befalls us as engineers is one of finding that bottleneck in the first place. One tried and tested method is to use a hardware timer to count clock cycles and thus measure time during execution. The theory is quite simple; add the timer to the design, find a piece of code you suspect is causing the performance bottleneck, start the timer, execute the code, stop the timer, and read the value on the timer. The example below uses this technique. (The calls to start, stop, and read the hardware timer would require the use of specific driver function calls that would be dependant on the address map of the system. These have therefore been intentionally simplified in this example.)

```
// Constants and Variables declarations
volatile unsigned int const a = 353;
volatile unsigned int const b = 476;
volatile unsigned int result = 0;

start_timer();
result = sqrt((a*a) + (b*b));
stop_timer();

xil_printf("result = %d \r\n", result);
read_timer();
```

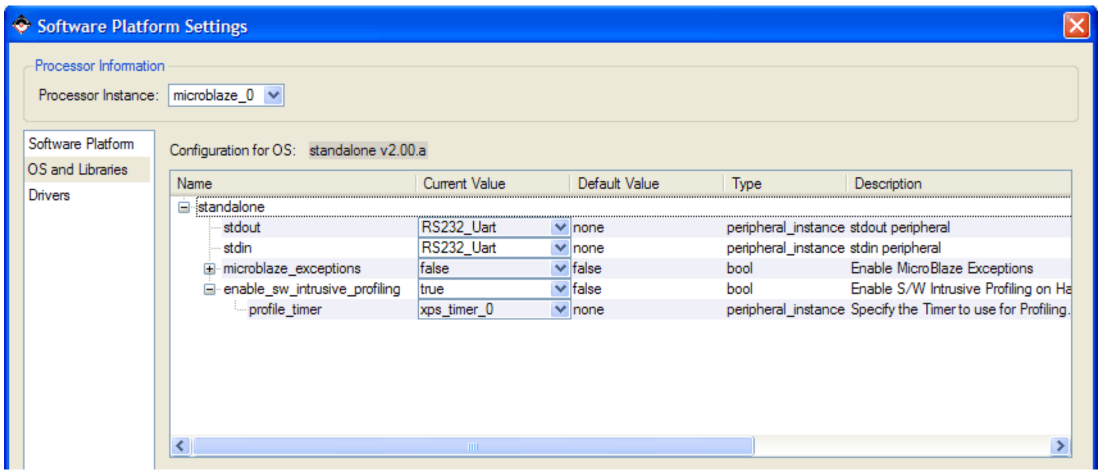
In the supplied reference design, we can see that this particular calculation requires 186525 clock cycles to execute, which equates to approximately 1865 ms of real time when executed at 100 MHz. This method of software intrusive analysis can be used throughout any given piece of software to determine its performance; however, the manual editing of the code in this manner is cumbersome, time-consuming, and prone to accidental error. Fortunately, the Embedded Development Kit provides an automated method with which to perform this type of analysis, commonly known as "Code Profiling".

Code Profiling in the EDK

Code profiling is a very important and powerful feature of the Embedded Development Kit, and it has been designed to provide users with performance analysis information without requiring much time and effort. It is important to note that code profiling will not help users find functional bugs in their code, because that method of testing is provided by standard debugging tools such as GDB. However, if, as an engineer, you have ever asked yourself "Where in my code is my processor spending all of its time?", then code profiling is for you.

Just as we saw in the previous simple example, a code-profiling tool uses a hardware timer to measure the number of clock cycles taken to execute a given software function. The power of the code-profiling tool comes from the automation of this process, and it does not require the user to add timer function calls all over the code. The timer value is captured whenever a software function is called and whenever it terminates; by using the values captured from the hardware timer, the code-profiling tools produce a report showing the time taken to execute each function.

A number of items are required to enable this functionality; the first is that a hardware timer must be added to the embedded processor design. Timers are available as pre-supplied IP blocks in the Embedded Development Kit, so this is a very simple addition to make using the Platform Studio GUI. The timer is used in interrupt mode during code profiling, so the interrupt output of the timer must be connected to the interrupt pin on the processor. You must also set the "enable_sw_instrusive_profiling" to True and specify the timer instance to "profile_timer" option for the Standalone OS by using "Software Platform Settings" GUI in the Platform Studio:



Software Platform Settings

Processor Information

Processor Instance: microblaze_0

Software Platform
OS and Libraries
Drivers

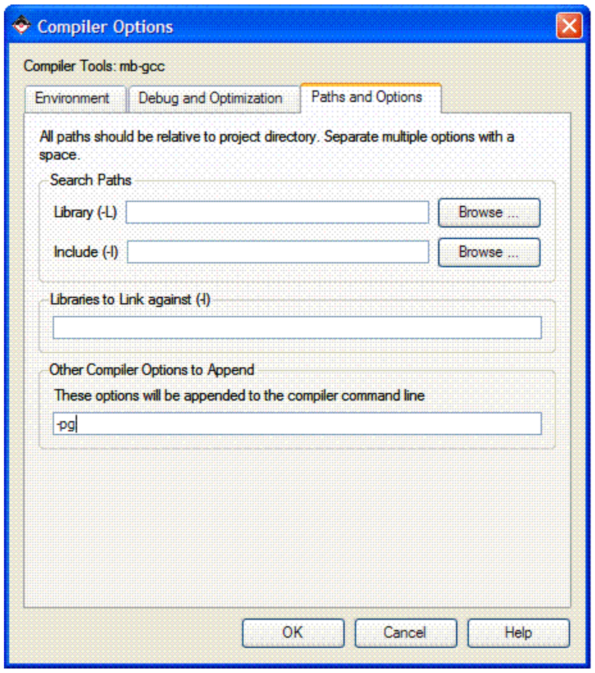
Configuration for OS: standalone v2.00.a

Name	Current Value	Default Value	Type	Description
standalone				
stdout	RS232_Uart	none	peripheral_instance	stdout peripheral
stdin	RS232_Uart	none	peripheral_instance	stdin peripheral
microblaze_exceptions	false	false	bool	Enable MicroBlaze Exceptions
enable_sw_intrusive_profiling	true	false	bool	Enable S/W Intrusive Profiling on Ha
profile_timer	xps_timer_0	none	peripheral_instance	Specify the Timer to use for Profiling.

Figure 4: **Setting the "enable_sw_instrusive_profiling" and "profile_timer" Options by Using "Software Platform Settings" GUI**

As the code-profiling tools run at real-time speed, we must set aside some memory for code-profiling information to be stored at run time. The contents of this memory will then be read back in an XMD debugging session and used to create a report file. The amount of memory required for this purpose varies depending on the software application to be profiled, but the EDK tools will report the required quantity of memory automatically. I will discuss this procedure in detail later in the article.

Last, the "-pg" switch must be added to the GNU compiler command-line for the application that is to be analyzed using the code-profiling feature of the tools. This can be done manually from the makefile, or by using the "Compiler Options" GUI in the Platform Studio.



Compiler Options

Compiler Tools: mb-gcc

Environment | Debug and Optimization | Paths and Options

All paths should be relative to project directory. Separate multiple options with a space.

Search Paths

Library (-L)

Include (-I)

Libraries to Link against (-l)

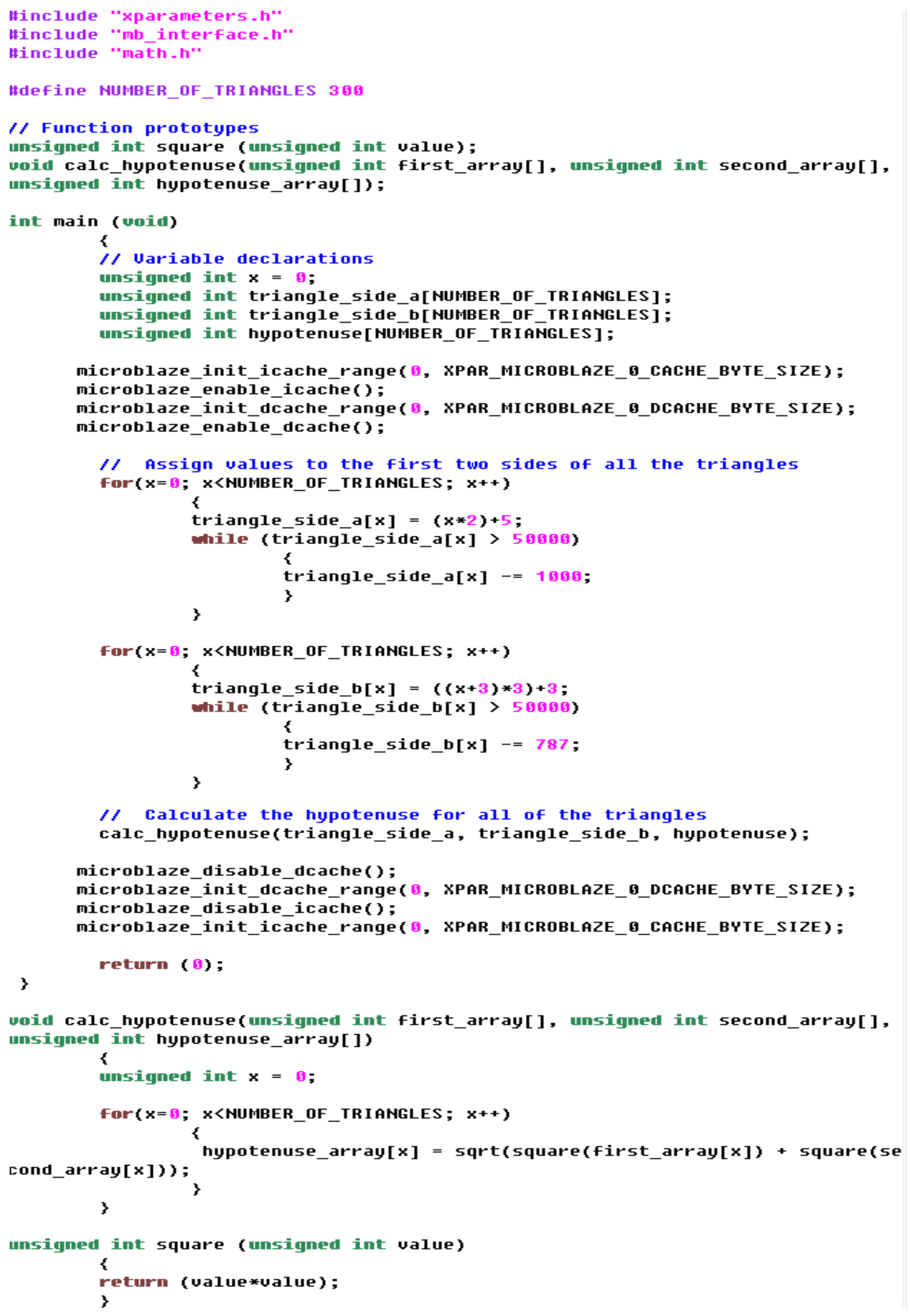
Other Compiler Options to Append

These options will be appended to the compiler command line

-pg

Figure 5: **Adding the "-pg" Switch by Using the "Compiler Options" GUI**

In the next example, we will modify the code to perform Pythagoras' theorem calculations on 300 triangles. The sides of the triangles will be pseudo-randomly generated and then stored in two of three arrays of data type "unsigned integer". We will then use Pythagoras' theorem to calculate the hypotenuse on each of the 300 triangles, and store the results in the third array.



```
#include "xparameters.h"
#include "mb_interface.h"
#include "math.h"

#define NUMBER_OF_TRIANGLES 300

// Function prototypes
unsigned int square (unsigned int value);
void calc_hypotenuse(unsigned int first_array[], unsigned int second_array[],
unsigned int hypotenuse_array[]);

int main (void)
        {
        // Variable declarations
        unsigned int x = 0;
        unsigned int triangle_side_a[NUMBER_OF_TRIANGLES];
        unsigned int triangle_side_b[NUMBER_OF_TRIANGLES];
        unsigned int hypotenuse[NUMBER_OF_TRIANGLES];

      microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
      microblaze_enable_icache();
      microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
      microblaze_enable_dcache();

        //  Assign values to the first two sides of all the triangles
        for(x=0; x<NUMBER_OF_TRIANGLES; x++)
                {
                triangle_side_a[x] = (x*2)+5;
                while (triangle_side_a[x] > 50000)
                        {
                        triangle_side_a[x] -= 1000;
                        }
                }

        for(x=0; x<NUMBER_OF_TRIANGLES; x++)
                {
                triangle_side_b[x] = ((x+3)*3)+3;
                while (triangle_side_b[x] > 50000)
                        {
                        triangle_side_b[x] -= 787;
                        }
                }

        //  Calculate the hypotenuse for all of the triangles
        calc_hypotenuse(triangle_side_a, triangle_side_b, hypotenuse);

      microblaze_disable_dcache();
      microblaze_init_dcache_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
      microblaze_disable_icache();
      microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);

        return (0);
 }

void calc_hypotenuse(unsigned int first_array[], unsigned int second_array[],
unsigned int hypotenuse_array[])
        {
        unsigned int x = 0;

        for(x=0; x<NUMBER_OF_TRIANGLES; x++)
                {
                 hypotenuse_array[x] = sqrt(square(first_array[x]) + square(se
cond_array[x]));
                }
        }

unsigned int square (unsigned int value)
        {
        return (value*value);
        }
```

Figure 6: **Code for Calculating the 300 Triangles**

We will add the “-pg” switch to the compiler command line and then compile the code in the usual way, creating a design upon which we can begin to use the code-profiling tools. The FPGA bitstream is downloaded to the board normally, readying the hardware for use. We can then use the “XMD” tool to open a debug connection to the embedded processor system via JTAG.

When the prompt appears in the XMD session, we must put the tools into profiling mode. This is done by using the “profile” command with several configuration options. The code-profiling tools can be configured to take a sample at varying frequencies, giving us the option to increase or decrease the granularity of the profiling report to be suitable for fast or slow software applications. We can also adjust various other advanced options such as the “binsize”, but we will leave this alone for our evaluations. In this example, we will configure the profiling tools to use external memory at the address 0x8d000000 with a default binsize of 4 words; we then set the sampling rate to 10000 samples per second. We do this by means of the following command at the XMD prompt:

```
XMD% profile -config sampling_freq_hz 10000 binsize 4 profile_mem
0x8d000000
```

The “dow” command can then be used to download the compiled code (called “executable.elf” in this example) to the processor’s memory. The tools will automatically detect that we have enabled code profiling mode and will report the quantity of memory required to store the profiling data, which in this example is 4524 bytes

```
XMD% dow fsl_sqrt/executable.elf
_gmonparam start addr: 0x8d000000
System Reset .... DONE
Downloading Program -- fsl_sqrt/executable.elf
        section, .vectors.reset: 0x00000000-0x00000007
        section, .vectors.sw_exception: 0x00000008-0x0000000f
        section, .vectors.interrupt: 0x00000010-0x00000017
        section, .vectors.hw_exception: 0x00000020-0x00000027
        section, .text: 0x8c000000-0x8c003453
        section, .init: 0x8c003454-0x8c003477
        section, .fini: 0x8c003478-0x8c003493
        section, .rodata: 0x8c003494-0x8c0034bb
        section, .sdata2: 0x8c0034bc-0x8c0034bf
        section, .data: 0x8c0034c0-0x8c0035ef
        section, .ctors: 0x8c0035f0-0x8c0035f7
        section, .dtors: 0x8c0035f8-0x8c0035ff
        section, .eh_frame: 0x8c003600-0x8c003603
        section, .jcr: 0x8c003604-0x8c003607
        section, .bss: 0x8c003608-0x8c00362f
        section, .heap: 0x8c003630-0x8c00562f
        section, .stack: 0x8c005630-0x8c005a2f
Program fsl_sqrt/executable.elf being Profiled on Hardware
Initialized Profile Configurations for the Program :
---------------------------------------------------
Sampling Frequency..............10000 Hz
Histogram Bin Size..............4 Words
Memory for Profiling used from...0x8d000000
Memory Used for Profiling Data...4524 Bytes
Setting PC with Program Start Address 0x00000000

XMD%
```

XMD will detect that code profiling mode has been selected during the compile stage and automatically check that the profile settings are suitable for the application. We now want to execute the software in order to collect the profiling data, but we want the debug tools to return debug control to us when the program has executed to completion. We will therefore set a breakpoint at the end of the code which is conveniently and automatically labeled as “exit” by the linker. To do this, we use the XMD command “bps exit” to set the breakpoint, and then use the “con” command to tell the processor to execute the code.

```
XMD% bps exit
Setting breakpoint at 0x8c001dd8

XMD% con
Info:Processor started. Type "stop" to stop processor

RUNNING> XMD% Info:Software Breakpoint 0 Hit, Processor Stopped at
0x8c001dd8

XMD%
```

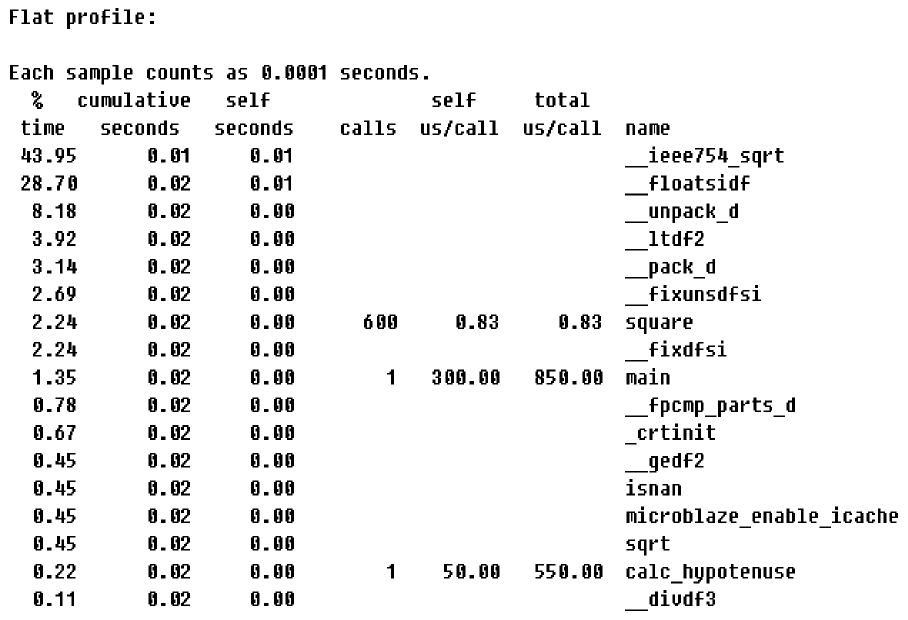
As can be seen here, the software application executed as expected on the board, and then automatically returned control to us via the XMD prompt when the “exit” breakpoint was reached. We can now tell XMD to collect the profile data that has been stored in the external memory by using the “profile” command. This causes the profile data to be written to the disk on the host PC into a file called “gmon.out”.

```
XMD% profile
Profile data written to gmon.out
XMD%
```

Now that we have collected this profiling data, we can analyze it using a tool called “gprof”, which is another of the tools supplied as part of the GNU series. This tool takes the collected data and then analyzes it into a human-readable report in ASCII format. The tool requires that the user provide the “.ELF” file from the compiler, and the “gmon.out” file that was generated by the code profiling tools. The output is usually displayed at the command prompt, but for ease of use we shall pipe the output to a text file called “profile_info.txt” using the following command:

```
XMD% mb-gprof fsl_sqrt/executable.elf gmon.out > profile_info.txt
XMD%
```

Code profiling is now complete, and all that remains is to interpret the data so that we can use it to improve our design. The first part of the “profile_info.txt” file is shown below:



```
Flat profile:

Each sample counts as 0.0001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 43.95      0.01     0.01                             __ieee754_sqrt
 28.70      0.02     0.01                             __floatsidf
  8.18      0.02     0.00                             __unpack_d
  3.92      0.02     0.00                             __ltdf2
  3.14      0.02     0.00                             __pack_d
  2.69      0.02     0.00                             __fixunsdfsi
  2.24      0.02     0.00      600     0.83     0.83  square
  2.24      0.02     0.00                             __fixdfsi
  1.35      0.02     0.00        1   300.00   850.00  main
  0.78      0.02     0.00                             __fpcmp_parts_d
  0.67      0.02     0.00                             _crtinit
  0.45      0.02     0.00                             __gedf2
  0.45      0.02     0.00                             isnan
  0.45      0.02     0.00                             microblaze_enable_icache
  0.45      0.02     0.00                             sqrt
  0.22      0.02     0.00        1    50.00   550.00  calc_hypotenuse
  0.11      0.02     0.00                             __divdf3
```

Figure 7: **Snipt of the Dumped Profiling Output File for "sqrt" Function**

A full explanation of this data can be found in the "gprof" documentation, but for clarity we shall examine a few of the columns of data shown to understand the meaning of the output. The "gprof" tool lists the software functions executed by the processor according to how much processor time was taken to execute them. The functions requiring most processor time are shown at the top of the list, and the ones requiring very little processor time are shown at the bottom. The name of each software function is shown in the column on the right, and the percentage of processor time required to execute that function appears on the left.

We can see that the function called "__ieee754_sqrt" is consuming over 43.95% of the processor's time during the execution of the software application. It is worth noting that the percentage column treats each function individually, rather than measuring a function and its hierarchical "children". If this were not the case, we would see the "main" function listed as consuming 100% of the processor's time, which would be meaningless for our performance analysis. The third column shows how much processor time was spent executing each function, and we can see from the results that the "__ieee754_sqrt" function consumed 0.01 seconds of processor time during execution. While this may not seem like a large amount of time for our test application, it is important to remember that we are only calculating the results for 300 triangles rather than for the many thousands in the real application.

Using the code-profiling information, we have discovered why the design is running so slowly. Perhaps unsurprisingly for this simple design, the slowest operation in the code is the Square Root function "__ieee754_sqrt". The most interesting part of this discovery is in the name of the function, which appears to be from the IEEE754 library. Those of you familiar with DSP applications will recognize this IEEE specification as the one relating to binary floating-point mathematics.

Further investigation reveals that the other software functions at the top of the report ("__floatsidf", "__unpack_d", "__ltdf2", "__pack_d", "__fixunsdfsi" and "__fixdfsi") are all functions associated with the manipulation of floating-point numbers. If we add up the time taken for all of these functions to execute, we will discover that a massive 90.58% of the processor's time is being consumed by floating-point calculations!

The second column in the table shows us a cumulative quantity of time, and it illustrates that the top six software functions in the list cumulatively took 0.02 seconds to execute. Lastly, in the sixth column, we can see that the function containing the square root calculation loop "calc_hypotenuse" takes 550 us of processor time to execute.

If we now look back at the source code, we find that we are not actually using any floating-point numbers; all of our data is stored using the "unsigned integer" data type. Mysterious? Perhaps. Incorrect? Not necessarily…

It is an unfortunate reality of the world that software compilers and other similar automated design tools have some very nasty habits, perhaps the most sinful of which is their almost unrivaled ability to arrogantly assume they know what the user wishes to do. It is all too easy for engineers to write some code and then have faith that the tools will do their jobs and present us with the best solution. To give these tools the credit they deserve, we must state that they will invariably arrive at a functionally correct solution to a given problem; we can see from our square root example that this is indeed the case.

The problem with these tools is that although they deliver a functional solution, it may not necessarily be an optimal solution. Results can be sub-optimal for many reasons: cost, performance, and complexity to name but a few. This example illustrates a solution that is functionally correct but suffers from performance problems. To fully understand the unexpected use of the floating-point libraries for the square root operation, we must look deeper into the IEEE 754 mathematics library to find out what is happening—which we undertake in Part 2 of this White Paper.

Performance Tuning

The "sqrt" operator that we used in the C code is part of the floating-point mathematics library "math.h". If we examine the function declaration for "sqrt" in the math.h library, we see that it will only accept input data values of data type "double", which are double-precision floating-point numbers. Given that we want to perform a square root operation on an unsigned integer, the compiler will recognize this data type limitation and silently perform automatic data type conversion for us.

What actually happens is that each of our unsigned integer values is converted to type "double", the square root operation is completed using the math.h library in floating-point mode, and the result is then converted back to the "unsigned integer" data type before being stored in the array. This is functionally correct, but slow, thus affecting the overall performance of the solution.

When we work with embedded systems inside FPGAs, it is quite common for data types to be of the fixed-point "integer" or "unsigned integer" data type, so some type conversion will generally be necessary for all functions using the floating-point libraries. This automatic conversion is fine in theory because everything is converted automatically without the designer having to think about it too hard; however, double-precision floating-point calculations can be especially slow in a processor without a double-precision floating-point execution unit, MicroBlaze being one such processor.

One of the options for curing this problem is to enable the EDK tools option to add a floating-point execution unit to the MicroBlaze processor instance. This will accelerate the computational performance of floating-point mathematical functions, but it requires that a huge amount of additional FPGA resources be used to implement the extra logic.

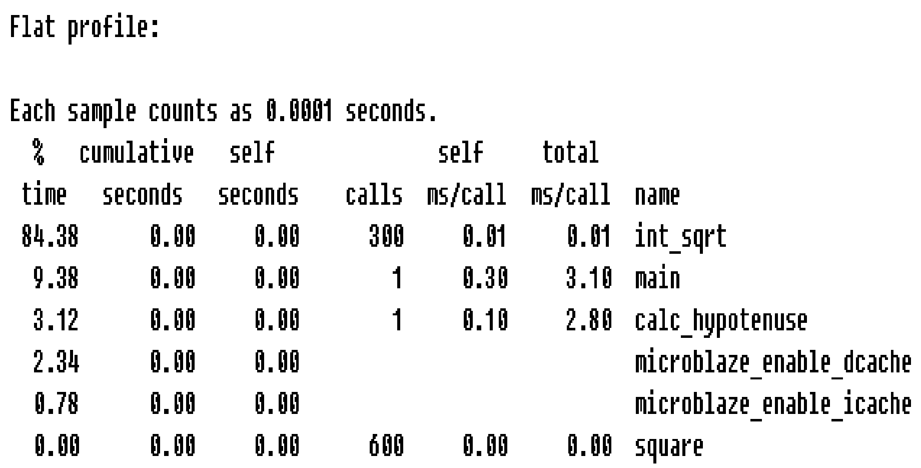
As engineers, we see that we are trying to solve a problem by curing the symptom, rather than eliminating the original problem. The ideal solution is to avoid using floating-point mathematics in the first place, which effectively means that we must avoid using the "sqrt" software function. The next option is to use a carefully written function in C code to perform a square root operation directly on fixed-point numbers, thus eliminating the need for any data type conversion.

One such function is shown below:

```
#define UPPERBITS(value) (value>>30)

unsigned int int_sqrt (unsigned int value)
{
   int 1;
   unsigned int a = 0, b = 0, c = 0;

   for (i=0; i < (32 >> 1); i++)
   {
        c<<= 2;
        c += UPPERBITS(value);
        value <<= 2;

        a <<= 1;
        b = (a<<1) | 1;

        if (c >= b)
             {
             c -= b;
             a++;
             }
   }

  return a;
}
```

To test the performance of this solution, we can simply replace the "sqrt" operator in the C code with the call to "int_sqrt" shown above. Re-compiling the code and testing on the board reveals that correct functional results are obtained, proving that we have correctly implemented an alternative for the "sqrt" function. Once again, we can follow the same flow to run the code-profiling tools to determine whether our adjustment to the code has improved the performance.

The report file from the "gprof" tools shows us that the use of the "int_sqrt" function has made a difference:



```
Flat profile:

Each sample counts as 0.0001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.38      0.00     0.00      300     0.01     0.01  int_sqrt
  9.38      0.00     0.00        1     0.30     3.10  main
  3.12      0.00     0.00        1     0.10     2.80  calc_hypotenuse
  2.34      0.00     0.00                             microblaze_enable_dcache
  0.78      0.00     0.00                             microblaze_enable_icache
  0.00      0.00     0.00      600     0.00     0.00  square
```

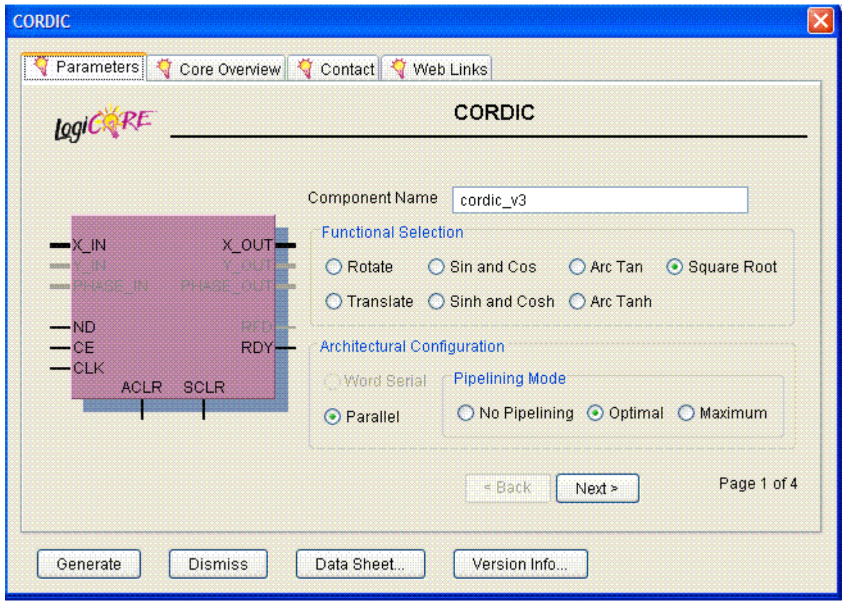
Figure 8: **Snipt of the Dumped Profiling Output File for "int_sqrt" Function**

Note that the number of functions executed by the processor has been dramatically reduced. None of the floating-point math functions are present, simply because all of the complex data type conversion that was previously necessary is no longer needed. Unfortunately, the processor is spending 84.38% of its time performing square root calculations, which is actually worse than before! The complexity of the application has been reduced by removing the floating-point calculations, but we can see that the square root calculations are still demanding a very high percentage of the processor's time, which could be better used by other software tasks.

Sharing the Work

We previously discussed the use of an optional floating-point unit to accelerate the slow functions in this design, but dismissed the idea as being inefficient in terms of silicon area (and therefore cost). A full floating-point unit is a large piece of hardware, but a more specific square root hardware acceleration block may hold the solution to our problems. One such hardware solution is available through Xilinx CORE Generator™, which is supplied with the standard Xilinx ISE® tools. A brief look through the selection of DSP IP blocks reveals the "CORDIC" block. CORDIC is an acronym for COrdinate Rotation DIgital Computer, and it is widely used in a variety of DSP applications to implement shift-add algorithms for rotating vectors in a plane. Fortunately for us, CORDIC can also be used to perform square root operations.

We shall therefore use the CORE Generator tool to generate a dedicated hardware block for square root operations using the CORDIC algorithm. A number of configuration options are available when building this block, but we shall configure the CORDIC algorithm to operate in pipelined mode, process unsigned integers, round the results up or down to the nearest even, and deliver the result 11 cycles after the input is captured.



Component Name cordic_v3

Functional Selection
Rotate Sin and Cos Arc Tan [✓ Square Root]
Translate Sinh and Cosh Arc Tanh

Architectural Configuration
Word Serial
[✓ Parallel]

Pipelining Mode
No Pipelining [✓ Optimal] Maximum

Figure 9: **Configuring CORDIC By Using CORE Generator GUI**

The CORE Generator tool produces a hardware netlist for the CORDIC block and generates a HDL template for ease of integration. The "Designing a Custom Processor Peripheral Using Xilinx EDK" article showed that it was quite simple to create custom processor peripherals to be attached to the PLB bus, but a bus connection is not the best option for this application.

Busses are ideal if one wishes to make a block of hardware accessible to a number of bus masters, because they ease the problems associated with arbitration and increase flexibility in the design. This flexibility comes with a catch, and in most cases this is represented by a drop in performance. The whole idea of adding the CORDIC block is to increase the performance of a software-based algorithm, so we do not wish to waste valuable clock cycles by arbitrating for the bus and then performing multiple memory-mapped bus transactions to read and write values to registers in a custom peripheral.

By placing the CORDIC block on a bus, we would require the processor to write the value to be square-rooted to a register in the peripheral, then send a command to a second register telling the CORDIC block that there is a valid value present on the inputs. The processor would then have to wait for the CORDIC block to complete the square root operation before making a third bus transaction to read the result from an output register. Each bus transaction would take five cycles, assuming that the bus arbitration was not required and that there were no other transactions currently in progress on the bus. It would be a functional solution, but hardly an efficient one!

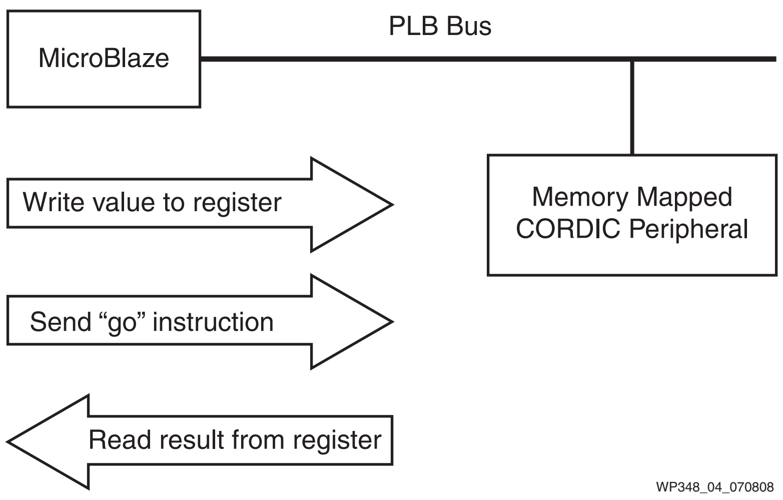


Figure 10: **Attaching CORDIC Block via PLB Bus**

Fortunately, the MicroBlaze processor provides us with a better alternative in the form of the FSL connections. FSLs are Fast Simplex Link ports on the processor that are intended specifically for high-speed uni-directional transfers to and from the processor. No arbitration is required on these connections because they do not support multiple masters.

The theory behind FSLs is very simple: the MicroBlaze processor has eight input ports and eight output ports which can be accessed using special instructions in the processor's memory map ("write to FSL" and "read from FSL"). In the case of a write transaction, data values are moved from the general purpose registers on the processor and placed onto the FSL connection in only two clock cycles. Read transactions operate in a similar fashion, also requiring only two cycles for the operation to complete.

The FSL links can be enhanced by adding optional FIFO buffers to the FSL ports, enabling a user to send "bursts" of data to the external hardware block before reading a similar "burst" of results. This improves efficiency, which substantially increases performance.

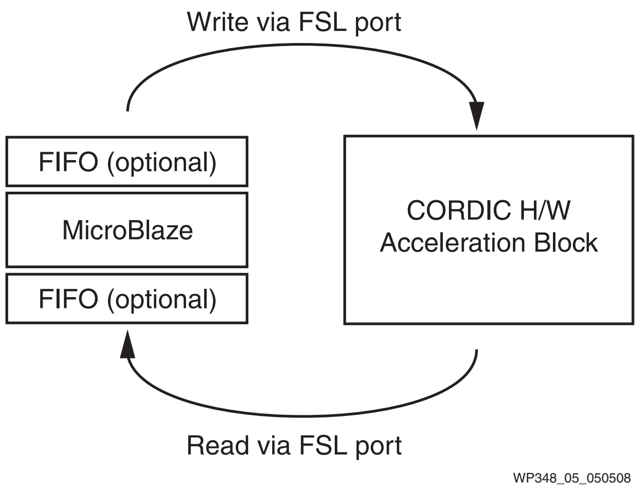


Figure 11: **Connecting CORDIC Block to MicroBlaze via FSL Link**

To achieve maximum performance from our CORDIC acceleration block, we shall use the FSL ports on MicroBlaze and implement the optional FIFOs using a depth of 16 words.

In the next and final part of this White Paper, we will delve into the creation of FSL peripherals in EDK and see how to accelerate the hardware.

Creating FSL Peripherals in EDK

Although the MicroBlaze FSL ports are not complex, the EDK tools provide a utility called the “Create/Import Peripheral Wizard”. This automated wizard allows users to build custom hardware modules and connect them to the processor core quickly and easily. This utility allows a user to specify the number of input and output ports required on the FSL, and it will then produce an HDL template into which the CORDIC block can be added. Once this has been done, the template will look something like the following:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----------------------------------------------------------------------
entity fsl_sqrt is
        port
        (
                -- DO NOT EDIT BELOW THIS LINE ---------------------
                -- Bus protocol ports, do not add or delete.
                FSL_Clk : in    std_logic;
                FSL_Rst : in    std_logic;
                FSL_S_Clk       : out   std_logic;
                FSL_S_Read      : out   std_logic;
                FSL_S_Data      : in    std_logic_vector(0 to 31);
                FSL_S_Control   : in    std_logic;
                FSL_S_Exists    : in    std_logic;
                FSL_M_Clk       : out   std_logic;
                FSL_M_Write     : out   std_logic;
                FSL_M_Data      : out   std_logic_vector(0 to 31);
                FSL_M_Control   : out   std_logic;
                FSL_M_Full      : in    std_logic
                -- DO NOT EDIT ABOVE THIS LINE ---------------------
        );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end fsl_sqrt;
```

Figure 12: **FSL Peripheral VHDL Template**

```
-------------------------------------------------------------------------------
-- Architecture Section
-------------------------------------------------------------------------------
architecture EXAMPLE of fsl_sqrt is

component cordic_wrapper is
    Port
                (
                clk : in std_logic;
                reset : in std_logic;
                data_in : in std_logic_vector(31 downto 0);
                data_out : out std_logic_vector(16 downto 0);
                capture_input_data : in std_logic;
                result_ready : out std_logic
                );
end component;

-- Signal declarations
signal data_from_sqrt : std_logic_vector (0 to 16);

begin

cordic_block : cordic_wrapper
      port map
              (
              clk => FSL_CLK,
              reset => FSL_Rst,
              data_in => FSL_S_DATA,
              data_out => data_from_sqrt,
              capture_input_data => FSL_S_EXISTS,
              result_ready => FSL_M_WRITE
              );


FSL_S_READ <= FSL_S_EXISTS;
FSL_M_DATA <= "000000000000000" & data_from_sqrt;  -- SQRT result is only 17 bits wide

--FSL_S_CLK <= CLK;
--FSL_M_CLK <= CLK;
FSL_M_CONTROL <= '0';

end architecture EXAMPLE;
```

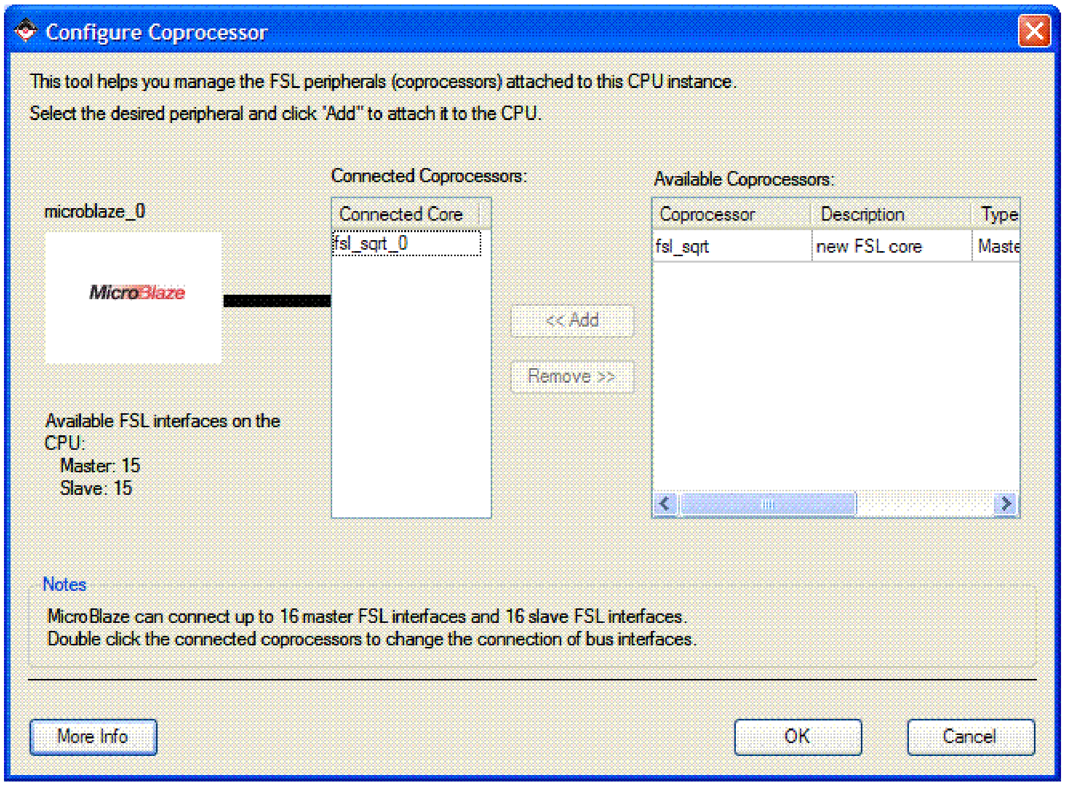
Figure 13: **The CORDIC Block is Instantiated in VHDL Code**

Essentially, this code just instantiates the CORDIC wrapper file that was created using the CORE Generator tool, extends the width of the output bus from 17 bits to 32 bits to match the MicroBlaze processor, and connects the data and control signals to the FSL ports. The control signals simply indicate when new and valid data is present on the FSL output and read the “ready” flag from the CORDIC block to indicate that a result has been processed.

With the hardware acceleration block instantiated in the HDL, we can now re-run the EDK Peripheral Wizard, which will allow us to import the modified peripheral and include the netlist files created by CORE Generator. The peripheral will then be available in the IP list for us to use in processor designs. (Further details regarding the creation of custom IP are available in my previous “Designing a Custom Processor Peripheral Using Xilinx EDK” article; the flow for creating FSL peripherals is the same as for PLB peripherals.)

Hardware Acceleration

Now that we have created the peripheral, we can add it to the existing processor design using the EDK tools. In the EDK 10.1i tools, this option can be found in the Tools menu, labeled “Configure Coprocessor”. Adding the CORDIC acceleration module to the design is simple, requiring just a few mouse clicks.


Configure Coprocessor

This tool helps you manage the FSL peripherals (coprocessors) attached to this CPU instance.
Select the desired peripheral and click 'Add" to attach it to the CPU.

microblaze_0

Connected Coprocessors:

Connected Core
fsl_sqrt_0

Available Coprocessors:

Coprocessor	Description	Type
fsl_sqrt	new FSL core	Maste

Available FSL interfaces on the CPU:
Master: 15
Slave: 15

Notes
MicroBlaze can connect up to 16 master FSL interfaces and 16 slave FSL interfaces.
Double click the connected coprocessors to change the connection of bus interfaces.

Figure 14: **Adding CORDIC Module By Using Configure Coprocessor GUI**

We must now modify the software to use our new hardware acceleration block; fortunately, the EDK LibGen tool automatically creates driver functions, which makes this task very simple indeed. There are many different functions available for our use in the supplied drivers, but we shall look at two of the simplest. The first is called “microblaze_bwrite_datafsl”, and the second is “microblaze_bread_datafsl”. The function names are fairly self-explanatory except for the “b” in “bread” and “bwrite”, which indicates that these functions are “blocking”. The blocking nature of these functions simply indicates that the processor must complete the read or write operation before returning to execute any further code.

It is important to remember that we are reading and writing data values to/from a FIFO buffer; it is therefore conceivable that the FIFOs could be full when the processor attempts to write a value, or empty when the processor attempts to read a value from the FIFO. Our application requires that we experience no data loss under these conditions, so we use the blocking function calls to stall the processor until the FIFOs become accessible. (If we were accelerating a function where the occasional loss of one or two data values was not critical, we would use some equivalent functions called “microblaze_write_datafsl” and “microblaze_read_datafsl”, which are non-blocking in nature and thus allow the processor to continue working.)

Two arguments must be passed into these functions; the first is the variable to be sent to or received from the CORDIC block, and the second is the ID number of the eight FSL channels that is connected to the CORDIC block; “0” in our example.

Our blocking function calls can therefore be arranged into a convenient and modular software function that will perform the square root operation in a way that is consistent with our previously used “sqrt” and “int_sqrt” functions:

```
unsigned int fsl_square_root(unsigned int input_value)
        {
        unsigned int temp = 0;
        microblaze_bwrite_datafsl(input_value, 0);
        microblaze_bread_datafsl(temp, 0);
        return (temp);
        }
```

Figure 15: **C Code for FSL Square Root Function**

We replace the old, slow function calls in the code with the new one to access the FSL for all square root calculations, and the modifications to the code are complete:

```
void calc_hypotenuse(unsigned int first_array[], unsigned int
second_array[], unsigned int hypotenuse_array[])
 {
 unsigned int x = 0;
 for(x=0; x<NUMBER_OF_TRIANGLES; x++)
        {
        hypotenuse_array[x] = fsl_square_root(
 square(first_array[x]) + square(second_array[x]));
                        }
 }
```

Figure 16: **The Code Using “fsl_square_root” Function**

If we now re-compile the code and once again use the code-profiling utility in XMD, we can see if our efforts have solved the performance problems that we previously experienced:

```
Flat profile:

Each sample counts as 0.0001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 50.00      0.00     0.00        1   300.00   450.00  main
 16.67      0.00     0.00      600     0.17     0.17  square
 16.67      0.00     0.00                             microblaze_enable_icache
  8.33      0.00     0.00      300     0.17     0.17  fsl_square_root
  8.33      0.00     0.00                             _crtinit
  0.00      0.00     0.00        1     0.00   150.00  calc_hypotenuse
```

Figure 17: **Snipt of the Dumped Profiling Output File for “fsl_square_root” Function**

As we can see from the profile report, this modification has made a dramatic difference to the execution of the code: the square root operation is now consuming a much smaller percentage of the processor's run time, and the performance of the whole application has increased. These impressive results are seen even though we are using the FSL square root peripheral in the slowest and most inefficient way possible, by sending only one value to the FSL port and then waiting for the result to be calculated before sending the next.

We could modify our code to be more efficient still, by taking advantage of the FIFOs and pipelined nature of the FSL connections. Bursts of up to 16 values could be sent to the square root acceleration block in one go, which the CORDIC block would process in a pipelined fashion. Depending on a user's requirements, it might prove beneficial to increase the size of the FIFOs beyond 16 words, using the EDK tools so that larger bursts of data could be sent and received. Whatever method we choose, it is now possible to use the remainder of the processor's run time to complete other software tasks, as we have eliminated the bottlenecks in the original design

Conclusion

It is clear that accelerating software functions by using dedicated hardware on the FSL connections can be a powerful way of improving the efficiency and processing power of MicroBlaze-based systems. Obtaining the best results from any design can only be achieved by using the careful and considered combination of both custom software and custom hardware, something which would have been completely impossible had we been using an off-the-shelf processor component.

Building embedded processor systems within FPGAs allows us to exploit the best features of each technology; software for flexibility, and custom hardware for raw processing power. Using our simple example, we have examined the techniques required to analyze a software application to identify performance bottlenecks, and we have used the features of the EDK tools to accelerate the slowest parts of the design.

The results speak for themselves; we have transformed the slowest part of the application into one of the fastest, clearly indicating the advantages of these techniques. It is important to remember that the addition of hardware acceleration to a processor system will always consume additional FPGA resources—this may not necessarily be a bad thing, because experience has taught me that spare resources are often available in an FPGA. The intelligent use of the otherwise wasted space can be a powerful, simple, and—most importantly—cost-free solution.

See the reference design for this White Paper.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
08/22/05	1.0	Initial Xilinx release as a TechXclusive. Part 1, 2, and 3.
8/30/08	1.1	Formerly TechXclusive (*The Root of All Evil - Parts 1, 2, 3*); converted to White Paper (*MicroBlaze System Performance Tuning)*.

Notice of Disclaimer