Abstract
System-on-a-chip FPGAs including embedded processors (hard or soft), busses, memory and hardware accelerators provide an opportunity for system designers to develop high performance, optimal systems. However, to realize the promise of this vision, a complete tool chain from concept to implementation is required. We describe an automated design framework that enables development of hardware / software FPGA systems starting with a pure, ANSI-C design specification. Profiling and analysis assist the developer in determining the hardware / software partition while a suite of verifications technologies including functional, cycle accurate, timing accurate and hardware-in-the-loop assist in system verification. Finally, advanced compilation technologies, including an optimized C to hardware compiler, provide a full push-button implementation flow.

Introduction / Background
The constant drive to provide more functionality, performance and flexibility for embedded applications is stressing the limits of traditional design and development approaches. As we attempt to squeeze more performance out of embedded processors, we run into limitations on throughput and performance based on system bottlenecks, architectural constraints and memory or interface bandwidth.

Figure 1: Typical embedded computational system: microprocessor, memory, IO

Assuming that the IO is sufficient to meet throughput requirements, then a number of items typically become system bottlenecks: processor speed, memory bandwidth, bus architecture.
Current approaches to addressing these issues are to use computational elements that are more targeted for the application domain. For example, specialized processors such as DSPs may be used for certain types of applications. Alternately, Application Specific Standard Products (ASSPs) may be used if appropriate ones can be found. For extremely high volume applications, a design team may decide to develop a specialized processing chip such as an ASIC to solve a problem. More commonly, FPGAs have been used to accelerate a wide variety of applications. However, moving from a stand-alone embedded processor to a system that contains multiple processing elements (DSPs, ASICs, ASSPs or FPGAs) immediately increases the complexity of the design task by a significant margin. This opens up the architectural choices and often requires the use of multiple discrete design flows to program each element. In addition, the use of an FPGA has historically required significant hardware expertise to design. An ASIC requires perhaps even more hardware expertise as well as significant up-front NRE costs and a significant investment in a large verification team and associated tools.

However, the use of these other processing elements may be the most effective manner to address the system cost/performance constraints if the design challenge can be overcome. Traditional hardware / software co-design attempts to manage the development of the hardware and software together, but still fundamentally requires the use of multiple discrete design flows at the implementation level – the system is specified and analyzed, then the hardware specification is “thrown over the wall” to the hardware designers while the software specification is passed to the software designers. Not only does this introduce the distinct possibility for errors of interpretation, it also requires a design team with a broad skill set to be successful.

This paper will describe a software-centric development methodology for platform FPGAs that addresses embedded system design. FPGAs are arguably the most flexible of the processing alternatives and have the ability to incorporate traditional processors (both hard processors and soft processors), on-chip busses, memory, a wide variety of IO and interface standards, customized hardware acceleration processors as well as mixed, custom communication (bus-based, peer to peer) topologies to improve system performance.

Hard processors are microprocessors that have been diffused in the silicon of an FPGA. For example, the Virtex II Pro™ chip from Xilinx includes a PowerPC™ processor. Soft processors are microprocessors that are created out of the FPGA gate array and can be configured to suite a particular application. An example is the MicroBlaze™ 32-bit RISC processor available from Xilinx.
FPGAs have historically been used as the “glue” to connect various other elements within a system. However, the last couple of generations of FPGA devices have provided such increases in capability that they have been moving into the ‘center’ of the system as the fundamental computational element. As FPGAs have grown larger and developed the capability to include processors and more flexible IO, FPGAs can be considered processing platforms in their own right. The challenge accompanying this potential capability is that design complexity has risen. In addition, FPGAs have historically been a hardware design problem. That was fine when they were simply used to consolidate random logic on a board design – the people who could make the most effective use of FPGAs had the hardware design skill set necessary to be successful with them. The addition of significant software content to FPGAs in the form of the code that runs on the on-chip embedded processors means that there is a shift in the design expertise necessary to effectively use all of the elements on the FPGA. Both software and hardware expertise is now necessary. Furthermore, the addition of multiple, distinct architectural elements coupled with on-chip busses and memory drives the need for system design and system architecture experience. In essence, the design challenge has grown not only in complexity of the individual elements, but also with respect to the skill sets necessary to fully utilize the capabilities provided by each type of available Platform FPGA element.

**Embedded Design Challenges**

Adding to the difficulties for an embedded / system designer who would like to use an FPGA is the fact that the design flow and tool chain for FPGAs have historically been based on a hardware-centric flow. This makes good sense from a certain perspective. FPGAs have been a predominately hardware design task and current design flows have evolved accordingly. In addition, an FPGA is an array of configurable logic elements, so it fundamentally was hardware design. However, FPGAs are moving into application spaces and high complexity arenas that drive the need for a new type of design flow.

One challenge is simply design complexity. Hardware design flows using hardware description languages (HDLs) such as Verilog or VHDL have been largely unchanged for more than a decade. In that time, Moore’s Law has seen available hardware resources on FPGAs increase by more than 100x. This has given rise to an increasingly large design productivity gap that is well documented within the hardware design community. HDLs
are relatively low-level design capture languages. They require a designer to capture not only the functionality of the design, but also important details about the architecture and implementation of the hardware at the same time. This mixing of functional capture with implementation details is one of the reasons that HDLs do not scale well to very large and complex designs.

A time honored approach to dealing with increased design complexity is through the use of appropriate abstraction. The use of a language that separates the functional description from the implementation can be a very powerful tool in dealing with complexity. We advocate the use of software languages such as C to be used in this regard, but also realize that they have their own challenges that must be overcome, such as with advanced compiler analysis. For example, software languages such as C are effective at capturing design functionality but they are inherently sequential. Since much of the advantage of hardware acceleration is realized through the exploitation of parallelism, either the language must be modified / extended to encompass this, the compiler must provide analysis to extract parallelism or some combination of both approaches must be taken. Modification / extension of the language may mitigate the advantages of using C as a capture language as it has the undesired effect to muddy the clean separation between functionality and implementation as well as impact abstraction levels.

A second consideration is that current system design methodologies often use an executable C specification as the ‘golden’ description of functionality, even including those that will ultimately end up as hardware. Sometimes, the C specification is developed directly as the system specification. In many cases, a higher-level analysis tool, such as Matlab will be used initially and the results of that analysis will then be used to create the C specification. So, an executable C specification is a common representation of system functionality in current system design methodologies. As such it is a reasonable place to start for any comprehensive system design methodology for Platform FPGAs.

Another consideration is simply that virtually all modern embedded systems now include some level of software content running on a traditional microprocessor. Having the system specification captured in C simplifies the process of implementing that specification on embedded processors since C is the implementation language of choice for the majority of embedded systems.

Finally, there are market acceptance considerations in proposing a new design methodology. A design methodology that leverages familiar mental models, languages and ways of working is more likely to garner acceptance. This is because it helps reduce the learning curve and perceived risk from the developer’s perspective. We want to meet the embedded developers on their terms. Our ultimate goal in developing this design methodology is for embedded / system developers to be able to view Platform FPGAs as simply a different embedded compiler target much as they view the difference between compiling to a PowerPC, ARM, MIPS or other traditional processor. The perceptible differences to the developer will be only in terms of improved capabilities and performance. Design flows will look and feel the same even though ‘under the hood’ the
implementation technologies are quite different and may include customized hardware, multiple busses, memories, etc. We call this new style of design methodologies for Platform FPGAs “software-centric.”

**Approach / Philosophy / Design Drivers**

**Software-Centricity**

What is a software-centric flow? Software-centric means that we must meet software developers on their own terms. The idea is to provide a flow that leverages the skill set available to the software engineer and mitigates the need for hardware specific expertise. As mentioned above, we must provide familiar mental models. Start at the embedded processor, leverage C syntax and semantics, and provide a modern integrated development environment for software. New users must accept the risk of a new processor (a Platform FPGA), so therefore we cannot also ask them to risk a new development paradigm and develop a new skill set as well.

This software-centric orientation drives a large number of design decisions with respect to developing our methodology. First, we must provide a total end-to-end flow. Note that with the emergence of Platform FPGAs that can include both soft and hard embedded processors, comprehensive end-to-end tool suites are available for implementing these systems. Typically, they require a mixed team of engineers that includes hardware and software expertise. Once the hardware platform is defined, the software designers can use these tool suites to develop the code for the embedded processors using traditional embedded development methodologies. Still, as an overall system design methodology, it requires significant hardware expertise and therein lies the distinction with the methodology described in this paper.

Since this current FPGA development environment is hardware-centric, adding in a software centric point tool adds value for the embedded system developer only in specific instances. For example, software point tools that support code-compile-debug on an embedded FPGA microprocessor add value in the context of enabling a software developer to write code that will run on an embedded microprocessor. It does not help the embedded software developer with the overall system design challenge.

On the other hand, examine a technology that can compile software code into hardware implementations. This clearly is a software-centric technology. It starts with software and abstracts the details of the hardware implementation. However, it is only a point tool. A hardware developer could use such a tool to increase design productivity because, coupled with their hardware expertise, they could run through a complete design flow. However such a tool provides almost no value to a software developer because even though they now can create the hardware block that represents the software functionality, they do not have the skill set to complete the rest of the flow. How will they interface this block with the rest of their design? How will they download the design into the FPGA, etc? We are driving towards a software-centric methodology that provides a complete end-to-end framework which enables developers with minimal hardware design expertise to successfully implement a design for a Platform FPGA target.
Pure C language
The entry point for our design methodology is an executable functional specification written in ANSI C. While higher level analysis tools may be used, C is already commonly used for this purpose in traditional embedded development. This also plays into the concept of meeting software developers on their own terms. Developers do not need to learn a new language to utilize the flow. Even dialects of C that are close-but-not-quite C have been stumbling blocks for acceptance. This is because these seemingly innocuous extensions and modifications cause, among a host of other problems, complications for moving code between hardware and software implementation targets. Our approach is to allow the use of standard C syntax. Advanced compiler analysis takes the place of requiring the developer to learn and use constructs that are required for efficient hardware implementation.

System View
Our methodology clearly espouses a system FPGA view of the design challenge. The executable C ‘application’ encompasses everything that will be included on the Platform FPGA including the physical interfaces. It is therefore fundamentally a top-down design methodology, but as the process is iterative and incremental, bottom up and middle-out methodologies are supported as well. The process of compiling a design generates two important components:
1. An architecture which is a structured composition of processing elements (microprocessors, IP cores, hardware accelerators), interconnect (on-chip busses and peer-to-peer.) and external interfaces (IO such as uart, Ethernet, etc.)
2. Executable code, which runs on the appropriate architectural elements.

Application acceleration
Our focus is on implementing a custom system that meets a developer’s system cost/performance constraints. There is an assumption inherent that improved performance is a fundamental goal. If performance could easily be achieved with traditional solutions, then developers would not be motivated to look at a new methodology and a new implementation technology. This means that the methodology starts with the application / problem to be solved and the goal of the compilation process is to accelerate this software application through the construction of an optimal HW / SW system.

Implementation focus
While the methodology encompasses a range of technologies to address profiling, analysis, debug, verification, etc, a key differentiating value of this methodology is the ability to move an application written in C to a Platform FPGA that includes one or more embedded processors, various IO devices and optimized hardware acceleration. This value is enhanced by the ability to model and verify the system, but our focus is on getting quickly to the FPGA, not extensive, formalized modeling. Detailed modeling and extensive verification make sense for ASIC flows where the cost of making a change is very high in terms of both time and money. For FPGA flows, the cost is simply the recompile time. As such, FPGA flows can roughly emulate software development processes that focus on quickly achieving executable code on the target and using tight code-compile-debug cycles to achieve optimal results. Historically, system design efforts
had a gaping hole in the area of implementation flows. Up front analysis and modeling was possible, but when that was complete, there was no path from the model to implementation short of tedious and error-prone re-writing by hand.

**Productivity**
The second area of focus with respect to our philosophy in developing this methodology was to focus on productivity. To some extent, this is highly complementary with our software-centric focus. Indeed the software-centric focus can be thought of as one way to improve productivity. Generally, if designers have the time and expertise, they could (and probably should) use low level tools to manually optimize all system elements, but there is never enough time and HW expertise. In addition, as systems get more complex, hand optimization may never find an optimal solution. Optimization at the local level does not guarantee a globally optimal solution and large complex systems may be not be amenable to comprehensive analysis without specialized tools. Our methodology aims to provide tools to satisfy the complex interaction of development time versus system optimization. The idea is to make it easy to quickly get a complete working system, and then the developer can allocate the remaining time to optimize the cost/performance limiters for that particular design. Things that can be modified include: system architecture, compiler directives, source code, libraries, hardware optimizations, etc. The use of a higher level system view provides insight into system optimizations that may be overlooked with lower levels of abstraction. This enables a developer to make most effective use of the design cycle time by focusing on the most important bottlenecks. In addition, this focus on productivity drives a number of other methodology considerations described below.

**Orthogonality of concerns**
Functionality, architecture and performance are three fundamental components of system design. To some extent, one can think of providing a functional specification and mapping that to an architecture to derive performance. On the other hand, one could specify a functionality and performance and then derive an architecture to support those constraints. So, while they have interrelations, our approach is to separate them to the extent possible. This positively impacts complexity and enables developers to focus on particular aspects of their system discretely thus improving clarity and productivity.

In our methodology, functionality is captured in C code and architecture can be constrained and/or inferred. That is, the architecture of the implemented system will consist of a number of elements including: microprocessors, interconnect (busses and peer-to-peer), interfaces (IO), and hardware accelerators (IP core library or automatically-generated). If the application is targeting a specific board that is already available, then the architecture would be constrained at least by the interfaces available on that particular board. In addition, the developer could define other constraints or enable the compilation process to infer the other elements.

Performance will be inherent in any implementation. That is, once the application is mapped to a target architecture, some level of performance will be achieved. If the resulting performance is not adequate, then the performance goals are achieved through
modification of the architecture, compile-time optimizations, etc. One common optimization would be to move a complex processing block to a hardware accelerator.

Layered complexity
A primary method to manage complexity is to hide it to the extent possible through the use of appropriate abstractions and layering. One of our goals from a software-centric standpoint is to minimize the need for hardware expertise in completing the design flow. This also has the effect of layering the complexity. Clearly, the use of a Platform FPGA will ultimately require the use of the low-level implementation tools that perform the placement and routing of the various hardware resources. However, within our methodology, these tools are called automatically behind the scenes to perform their functions without any user interaction. There is however, the ability to push down to lower levels of detail as time, need and expertise allow.

Concurrency
Whereas C is an inherently sequential programming language, hardware implementations derive much of their advantage through the use of parallelism, both temporal and spatial. Traditional programming models have limited ability to express this concurrency, especially fine-grained. The optimal solution from both a software-centric view and with respect to productivity is that the compiler must leverage concurrency automatically through advanced analysis. There is a distinction between fine grained concurrency and coarse grained concurrency. Fine grained concurrency is exploited automatically by compiler. This includes the use of advanced data and control dependency analysis to enable computing to occur as soon as possible by default as well as temporal concurrency through the use of automatic pipelining which effectively allows software functions to be called multiple times before returning up to the limit of their pipeline depth. On the other hand, coarse grained concurrency is best specified and leveraged by the designer. Again, this goes back to adopting familiar mental models and programming models. The models are readily used already in traditional programming paradigms. For example, threads are a very common model for specifying and emulating concurrency in a sequential processor. However, within a Platform FPGA, we can achieve true concurrency as opposed to pseudo-concurrency.

Automation
A key component of our development methodology is extensive automation. Fundamentally, designing a Platform FPGA involves highly complex tools. Detailed interaction with these tools is a barrier to entry for software developers. Advanced automation builds the necessary hardware expertise for programming FPGAs into the tools. In addition, this automation helps to layer complexity because a designer only has to dig in deeper and use lower level tools if automation doesn’t provide adequate results. Furthermore, automation has significant advantages with respect to system design because it facilitates the easier exploration of the larger portion of the solution space. For example, one of the tasks for automation is to provide automatic interface generation when a portion of the application is moved to a hardware accelerator. Assuming even that the hardware accelerator exists (automating the process of taking C code and turning it into a hardware accelerator is a enabling technology for the overall flow!) a designer
wishing to incorporate this acceleration into their system is left with the challenge of removing the function from their software specification and then replacing it with code to access the accelerator including all initialization operations and moving the data to and from the accelerator, monitoring for error conditions, etc. Without automation, this is a tedious and error prone process and inhibits the ability to explore the use of multiple accelerators and multiple arrangements of accelerators because of the logistics of making this happen. In an environment where this is as simple as a few mouse clicks, designers will have a dramatically increased ability to improve their system design through exploration.

**A Proposed Methodology**

A comprehensive methodology meeting the requirements described above necessitates an integrated suite of tools to address the multiple aspects of effective system development. These include specification capture, debug / verification (functional, cycle accurate, HW-in-loop), profiling, analysis, implementation, architectural design, tradeoff analysis and interface capture.

Given the software-centric focus, this methodology should strive to look and feel very similar to a typical software development flow even if the results include completely different products. The overarching goal is to take a functional specification captured in C code and map it onto a set of computational and interface elements instantiated on the Platform FPGA. Ideally, this set of computational and interface elements can each be customized and connected in an arbitrary fashion to ideally fit the application. However, while that is a longer term goal of this effort, it is beyond the scope of this paper. Instead, we will focus on two architectural archetypes:

1. A traditional embedded microprocessor system
2. A co-processor architecture that includes a traditional embedded microprocessor coupled with one or more specialized hardware accelerators

![Figure 2: Traditional embedded microprocessor system as implemented on a platform FPGA](image)

9
A traditional embedded system development flow would target the first architectural option. Our goal is to use an analogous flow to target the second architectural option. As a matter of fact, our flow will start with developing for the first architecture and then through the use of advanced compilers and analysis tools will automatically generate the second architecture under the control of the user. Throughout this section, we use the Mpeg 1 layer 3 decoder, commonly known as mp3 application as a concrete design example to show how this methodology applied to an application. We use the MAD open source project code as our pure C specification.
Figure 4: A simplified flow diagram of the steps from system concept to implementation on target system

Functional capture
The first step in the process is to choose the target that one wishes to develop for and capture the functionality that one would like to run on that target. The target is essentially the environment that one will write the code to run in. So this will specify the embedded processor (in this case a MicroBlaze soft processor) and the interfaces used to move the data into and out of the processor (in this case an AC97 controller). Note that this is essentially our baseline system as shown in Figure 2. Our first goal will be to get the application running in this architecture, and then we will add in accelerators to meet our performance requirements. The C code we used for the mp3 decoder is provided in the
appendix. We have chosen to use and extend the Eclipse C Development Toolkit (CDT) as the environment for our development tools. This provides a comprehensive software IDE for code development and debug. The ‘compile to host’ option allows us to run and debug the mp3 code on our workstation to quickly and easily verify its correctness.

**Analysis**

Once we are confident that the code is working correctly, our next step is to see if it meets our performance requirements. We have multiple choices at this juncture. We can either execute a cycle accurate model of the system running on the host. This will include an Instruction Set Simulator (ISS) of the processor and cycle accurate models of the interfaces. Or if the actual target board is available, we can run directly on the target. If we determine that we are not meeting our performance requirements, then we next need to determine where the bottlenecks are and how to remove them so that we can achieve our desired performance. An easy first step is to profile our code on the host to start to gain an understanding of which functions in the code consume the most execution time. While this will not provide absolute performance numbers since the host is quite likely a significantly different architecture than the actual target, it is an easy step to perform and should provide some insight into the relative amounts of computing resources that each function consumes.

For the mp3 application, we first undertook to run the application on the target board and determined that it did not meet real-time performance requirements. We then profiled the mp3 application on the host and profiling results revealed that the application spends a significant amount of time in executing a handful of computationally intensive functions. These functions are:

- DCT32
- IMDCT36
- MAD_MAC
- MAC16
- MAC28

DCT32 and IMDCT36 perform the discrete cosine transform and inverse discrete cosine transform respectively. The other functions are multiply-accumulate functions of various sizes. These are the major number crunching parts of the code and can be accelerated by implementing them in hardware. The chart below shows the profiling results for these functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Host Profile: % of total execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT32</td>
<td>34.79 %</td>
</tr>
<tr>
<td>IMDCT36</td>
<td>32.3 %</td>
</tr>
<tr>
<td>MAD_MAC</td>
<td>11.13 %</td>
</tr>
<tr>
<td>MAC16</td>
<td>9.1%</td>
</tr>
<tr>
<td>MAC28</td>
<td>4.6%</td>
</tr>
<tr>
<td>Total</td>
<td>91.92%</td>
</tr>
</tbody>
</table>


As can be seen from the table, these functions comprise over 90% of the total application execution time on the host.

The C definitions of these functions are as follows:

**DCT32:**

```c
#include "dct32_constants_1d.h"
#define DCT32MUL(X, Y)  (signed int) (((signed long long) (X))*((signed long long) (Y))) >> 28)
#define mad_fixed_t signed int

void dct32(mad_fixed_t const in[32], mad_fixed_t lohi[32])
{
    int i, j, k;
    mad_fixed_t temp1, temp2;
    k = 0;
    for (i = 0; i < 16; i++) {
        temp1 = temp2 = 0;
        for (j = 0; j < 32; j++) {
            temp1    += DCT32MUL(in[j],dct32_lo[j*16 + i]);
            temp2    += DCT32MUL(in[j],dct32_hi[j*16 + i]);
        }
        lohi[k++] = temp1;
        lohi[k++] = temp2;
    }
}
```

**IMDCT36:**

```c
#include "imdct_constants.h"
#include "MAD_F_MUL.c"

void imdct36(mad_fixed_t const X1[18], mad_fixed_t x2[36])
{
    int i, j;
    for (i = 0; i < 36; i++) {
        x2[i] = 0;
        for (j = 0; j < 18; j++) {
            x2[i] += MAD_F_MUL(X1[j], imdct_constants[j][i]);
        }
    }
}
```

**MAD_MAC:**

```c
mad_fixed_t mad_mac(mad_fixed_t in1[8], mad_fixed_t in2[16]){
    int i;
    mad_fixed_t sum = 0;
    for(i=0; i<8;i++){
        /* sum += mad_f_mul(in1[i], in2[(16-i*2)&0xf]); */
        sum += mad16(in1[i],in2[(16-i*2)&0xf]);
    }
    return sum;
}
```
MAC16:

```c
mad_fixed_t mad16(mad_fixed_t x, mad_fixed_t y) {
    int A, B, C, D;
    int prod_AD, prod_BC, prod_AC, prod_BD;
    int Sum1, Sum2, Sum3;
    int Result;
    A = (x >> 16);
    if (x < 0) {
        A = A | 0xFFFF0000;
    }
    B = (x & 0xFFFF);
    C = (y >> 16);
    if (y < 0) {
        C = C | 0xFFFF0000;
    }
    D = (y & 0xFFFF);
    prod_AD = A * D;
    prod_BC = B * C;
    prod_AC = A * C;
    prod_BD = B * D;
    Sum1 = prod_AD + (prod_BD >> 16);
    /* Sum2 is the low half word of long long mul op (top 16 bits are not valid) */
    Sum2 = Sum1 + prod_BC;
    /* Sum3 is high word of long long mul op */
    Sum3 = (Sum2 >> 16) + prod_AC;
    /* Shift Result by 16 bits */
    Result = (Sum3 << 16) + (Sum2 & 0xFFFF);
    return (Result);
}
```

```c
mad_fixed_t mac16(mad_fixed_t x, mad_fixed_t y, mad_fixed_t ret){
    return ret += mad16(x,y);
}
```
MAC28:

mad_fixed_t mad28(mad_fixed_t x, mad_fixed_t y)
{
  int A, B, C, D;
  int prod_AD, prod_BC, prod_AC, prod_BD;
  int Sum1, Sum2, Sum3;
  int Result;

  A = (x >> 16);
  if (x < 0)
    { A = A | 0xFFFF0000; }
  B = (x & 0xFFFF);
  C = (y >> 16);
  if (y < 0)
    { C = C | 0xFFFF0000; }
  D = (y & 0xFFFF);

  prod_AD = A * D;
  prod_BC = B * C;
  prod_AC = A * C;
  prod_BD = B * D;

  Sum1 = prod_AD + (prod_BD >> 16);
  /* Sum2 is the low half word of long long mul op (top 16 bits are not valid) */
  Sum2 = Sum1 + prod_BC;
  /* Sum3 is high word of long long mul op */
  Sum3 = (Sum2 >> 16) + prod_AC;
  /* Shift Result by 16 bits */
  Result = (Sum3 << 4) + ((Sum2 >> 12) & 0xF);

  return Result;
}

mad_fixed_t mac28(mad_fixed_t x, mad_fixed_t y, mad_fixed_t ret){
  return ret += mad28(x, y);
}
The ability to easily improve the performance of the implementation by automatically creating hardware accelerators is one of the biggest contrasts with traditional embedded development methodologies – we have the option to seamlessly augment the target architecture with accelerators in order to meet our performance requirements, merely by recompiling the application! While we still view this overall process as compiling to the target processors, the compiler has another significant degree of freedom in that it can augment the target architecture by implementing application specific accelerators.

**Target Acceleration**

Having identified the performance bottlenecks we next specify to the tool chain that we wanted to implement co-processor accelerators for those functions to meet our performance requirements. In our methodology, this is as simple as using the Eclipse tagging facilities to mark the functions for hardware acceleration. Once we do this, we can ‘compile for target’ and the tool chain will create an implementation that includes a MicroBlaze processor and interfaces the same as before, but this architecture is augmented with three hardware accelerators that implement the multiplications, DCT and inverse DCT. (See diagram below) The creation of the hardware accelerator blocks is done automatically by:

- the use of an advanced C to hardware compiler optimized for Platform FPGAs.
- the ‘stitching’ of the accelerators into the new co-processing architecture.
- handling the movement of the appropriate data to and from the accelerators.

![Generated co-processor architecture with five hardware accelerators](image-url)
At this point, the designer can once again profile the system to determine performance including the new accelerators. This can be done using the automatically generated cycle accurate system model or by running on the target board. When we do this, we find that the co-processor architecture meets our real-time requirements.

**Benchmark Results**

The performance of the accelerated functions, when implemented on a soft-processor (MicroBlaze) has been studied and compared to an implementation with a co-processor architecture that includes hardware accelerators. Our automatic compilation of the compute intensive functions to hardware provides significantly faster results than implementation solely in code on a microprocessor. The table below shows the comparison on a function-by-function basis and at the overall mp3 application level. The 4.8 times performance improvement as a result of the inclusion of the hardware accelerators enables the mp3 application to run in realtime at a system clock rate of 67.5MHz.

<table>
<thead>
<tr>
<th>Software Function</th>
<th>MicroBlaze Implementation</th>
<th>MicroBlaze plus Hardware Accelerators</th>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Note: all hardware accelerators were automatically compiled directly from C source code</td>
<td></td>
</tr>
<tr>
<td>DCT32</td>
<td>64,391 clock cycles/frame</td>
<td>2,245 clock cycles/frame</td>
<td>28.68x</td>
</tr>
<tr>
<td>IMDCT36</td>
<td>60,817</td>
<td>2,038</td>
<td>29.84x</td>
</tr>
<tr>
<td>MAD_MAC</td>
<td>663</td>
<td>470</td>
<td>1.41x</td>
</tr>
<tr>
<td>MAC16/28</td>
<td>169</td>
<td>159</td>
<td>1.06x</td>
</tr>
<tr>
<td><strong>Overall mp3 Application (1 frame)</strong></td>
<td><strong>7,395,385</strong></td>
<td><strong>1,536,460</strong></td>
<td><strong>4.81x</strong></td>
</tr>
</tbody>
</table>

**Conclusions**

The power and capabilities of current and future FPGA computing platforms, which can contain traditional microprocessors, on-chip busses, a wide variety of IO including high speed interfaces, application specific accelerators, custom communication topologies and all of the glue logic to tie the system together provides a huge opportunity for embedded designers to more effectively address a wide range of applications. However, the complexity of designing, integrating and verifying all of the elements that go into such a system chip can be daunting for even experienced system developers. This paper described a comprehensive methodology based on a software-centric framework to address all aspects of this development. The key insight of this methodology is that in
order to allow embedded developers, who may have strong software skills, but generally have less experience in detailed hardware design, to be successful with a Platform FPGA, we must (to the extent possible) provide an environment that meets them on their terms. The vision is to enable these designers to think of an FPGA as simply another target for their cross-compiler. Code – compile – debug on a new and powerful processor that we call a Platform FPGA!