Tune Multicore Hardware for Software

Teja FP and Xilinx FPGAs give you more control over hardware configuration.

by Bryon Moyer
VP, Product Marketing
Teja Technologies, Inc.
bmoyer@teja.com

One of the key preferences of a typical embedded designer is stability in the hardware platforms on which they program; poorly defined hardware results in recoding hassles more often than not. But a completely firm, fixed hardware platform also comes with a set of constraints that programmers must live with. These constraints – whether simply design decisions or outright bugs – can force inelegant coding workarounds or rework that can be cumbersome and time-consuming to implement.

By combining an FPGA platform with a well-defined multicore methodology, you can implement high-performance packet processing applications in a manner that gives software engineers some control over the structure of the computing platform, potentially saving weeks or months of coding time and reducing the risk of delays.

Much of the hardware design process goes into defining a board. Elements such as the type of memory, bus protocol, and I/O are defined up front. If a fixed processor is used, then that is also defined early on. But for gigabit performance on packet processing algorithms, for example, a single processor typically won't cut it, and multiple processors are necessary.

The best way to build a processing fabric depends on the software being run. By using an FPGA to host the processing, you can defer making specific decisions on the exact implementation until you know more about the needs of the code. The new Teja FP platform provides a methodology and multicore infrastructure on Xilinx® Virtex™-4 FPGAs, allowing you to decide on the exact configuration of the multicore fabric after the code has been written.

When Software Engineers Design Hardware

Hardware and software design are two fundamentally different beasts. No matter how much hardware design languages are made to look like software, it is still hardware design. It is the definition of structure, and processes ultimately get instantiated as structure. However, it is clear that software engineers are designing more and more system functionality using C programming skills; tools now make it possible for this functionality to target software or hardware.

The software approach is much more process-oriented. It is the "how to do it" more than the "what to build" because traditionally there has been nothing to build –

Copyright © 2006 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

the hardware was already built. A truly software-based design approach includes key functionality that is not built into the structure but is executed by the structure in a deployed system. The benefit of software-based functionality is flexibility: you can make changes quickly and easily up until and even after the system has shipped. FPGAs can also be changed in the field, but software design turns can be handled much more quickly than hardware builds.

Because hardware and software design are distinct, designers of one or the other will think differently. A hardware engineer will not become a software engineer just by changing the syntax of the language. Conversely, a software engineer will not become a hardware engineer simply by disguising the hardware design to resemble software. So allowing a software engineer to participate in the design of a processing fabric cannot be done blithely. In addition, turning a hardware-oriented design over to a software engineer is not likely to be met with cheers by hardware designers, software designers, or project managers. Hardware decisions made by software engineers must be possible using a methodology that resonates with a software engineer in a language familiar to the software engineer.

The key topology of the processing fabric for a multicore packet processing engine is the parallel pipeline, shown in Figure 1. Such an engine comprises an array of processors, along with possible hardware accelerators. Solving the problem means asking the following questions:

- How many processors do I need?
- How should they be arranged?
- How much code and local data store does each processor require?
- What parts of the code need hardware acceleration?

Let's take these questions case by case and assemble a methodology that works for software engineers.

Processor Count and Configuration

The number of processors required is more or less a simple arithmetic calculation based on the cycle budget and the number of cycles required to execute code. The cycle budget is a key parameter for applications when you have a limited amount of time to do the work. With packet processing, for example, the packets keep coming, so you only have so many cycles to finish your work before the next packet arrives. If your code takes longer than that, you need to add more processors. For example, if the cycle budget is 100 cycles and the code requires 520 cycles, then you need 6 processors (520 divided by 100 and rounded up). You can fine-tune this, but it works for budgetary purposes. The cycle count is determined by profiling, which you can perform using Teja's tools.

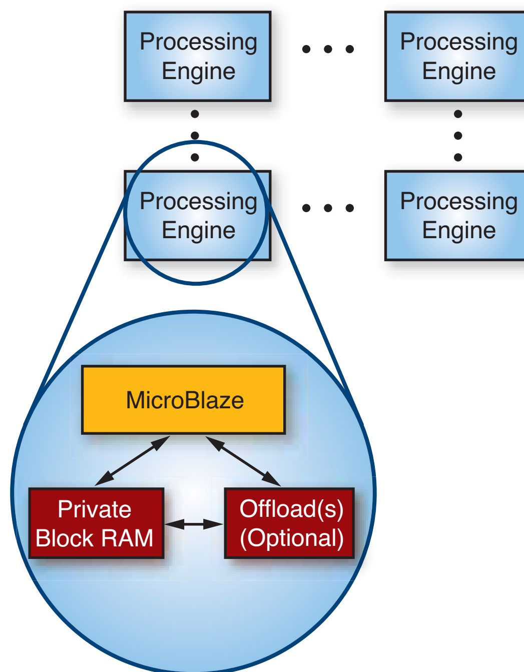

Figure 1 – A parallel pipeline where each engine comprises a MicroBlaze processor, private memory, and optional offloads.

The next question is how to arrange the processors. The simplest way to handle this is to decide if you want to partition your code to create a pipeline. A pipeline uses less resources but adds latency. If you want to partition your code, then pick an obvious place – something natural that works intuitively. There's no need to figure out where the exact cycle midway point is.

Let's say that with six processors you will have a two-stage pipeline. Now you need to figure out how many processors go in each stage; you can do this by profiling the cycle counts of each of the partitions and dividing again by the cycle budget. So if the first partition took 380 cycles, it will require 4 processors; that leaves 140 cycles for the second stage, which will require 2 processors. (The cycle counts of the two partitions won't actually add neatly to the cycle count of the unpartitioned program, but it is close enough for our purposes.) So this two-stage pipeline will have four processors in the first stage and two in the second stage. By using Xilinx MicroBlaze™ soft cores, you can instantiate any such pipeline given sufficient logic resources.

By contrast, a fixed pipeline structure would have a pre-determined number of processors in each stage. This would mandate a specific partition, which can take a fair bit of time to implement. Instead, the pipeline is made to order and can be irregular, with a different number of processors in each stage. So it doesn't really matter where the partition occurs. The hardware will be designed around the partitioning decision instead of vice versa.

The question of how a software engineer can implement this is key. Teja has assembled a set of APIs and a processing tool that allows the definition of the hardware platform in ANSI C; the tool executes that program to create the processing platform definition in a manner that the Xilinx embedded tools can process. This set of APIs is very rich and can be manipulated at a very low hardware level, but most software engineers will not want to do this. Thus, Teja has put together a "typical" pipeline definition in a parameterized fashion. All that is required to implement the pipeline in the example is to modify the two simple #define statements in a configuration header file.

The following statements define a two-stage pipeline, with four engines in the first stage and two in the second stage:

```
#define PIPELINE_LENGTH 2
#define PIPELINE_CONFIG {4,2};
```

From this and the predefined configuration program, the TejaCC program can build the pipeline. Of course, if for whatever reason the pipeline configuration changes along the way, similar edits can easily take care of those changes.

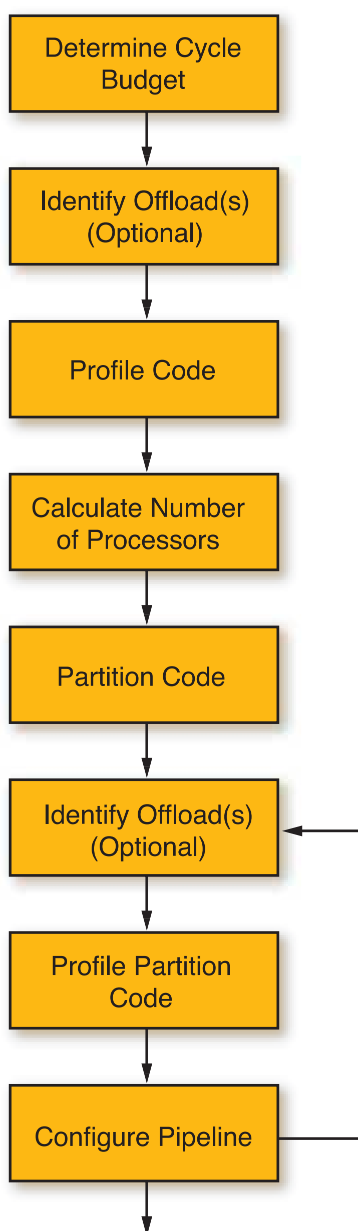

Figure 2 – Process for configuring a parallel pipeline

Memory

The third question is about the amount of memory required. In a typical system, you are allocated a fixed amount of code and data store. If you miss that target, a significant amount of work can go into squeezing things in. With FPGAs, however, as long as the amount of memory needed is within the scope of what's available in the chip, it is possible to allocate a tailored memory size to each processor. More typically, all would have the same size (since there is a 2K minimum allocation block, limiting the amount of fine tuning possible). Code compilation provides the size required, and the configuration header file can be further edited with the following statements, which in this example allocates 8 KB of memory for both code and data store:

```
#define CPE_CODE_MEM_SIZE_KB 8
#define CPE_DATA_MEM_SIZE_KB 8
```

Offloads for Acceleration

The fourth question deals with the creation of hardware accelerators. There may be parts of the code that take too many cycles. Those cycles translate into more processors, and a hardware accelerator can eliminate some of those processors. As long as the hardware accelerator uses fewer gates than the processors it is replacing, this will reduce the size of the overall implementation.

Teja has a utility for creating such accelerators, or offloads, directly from code. By annotating the program, the utility will create:

- Logic that implements the code
- A system interface that allows the accelerator to plug into the processor infrastructure
- An invocation prototype that replaces the original code in the program
- A test bench for validating the offload before integrating it into the system

Once an offload is created, the cycle count is reduced, so you will have to re-evaluate the processor arrangement. But because it is so easy to redefine the pipeline structure, this is a very straightforward task.

A Straightforward Methodology

Putting all of this together yields the process shown in Figure 2. You can define offloads up-front (for obvious tasks) or after the pipeline has been configured (if the existing code uses too many processors).

The controls in the hands of the software engineer are parameters that are natural or straightforward for a software engineer and expressed in a natural software language – ANSI C. All of the details of instantiating hardware are handled by the TejaCC program, which generates a project for the Xilinx Embedded Development Kit (EDK). EDK takes care of the rest of the work of compiling/synthesis/placing/routing and generating bitstreams and code images.

In this manner, hardware engineers can design the boards, but by using FPGAs they can leave critical portions of the hardware configuration to the software designer for final implementation. This methodology also lends itself well to handling last-minute board changes (for example, if the memory type or arrangement changes for performance reasons). Because the Teja tool creates the FPGA hardware definition, including memory controllers and other peripherals, board changes can be easily accommodated. The net result is that the hardware target can be adapted to the software so that the software designer doesn't have to spend lots of time coding around a fixed hardware configuration.

The Teja FP environment and infrastructure make all of this possible by taking advantage of the flexible Virtex-4 FPGA and the MicroBlaze core. Armed with this, you can shave weeks and even months off of the development cycle.

Teja also provides high-level applications; these applications can act as starting points for launching a project and reducing the amount of work required. By combining a time-saving flexible methodology with pre-defined applications, creators of networking equipment can complete their designs much more quickly.

For more information on Teja FP, contact *bmoyer@teja.com.*