Practical approach to designing controllers using FPGA with embedded processors

By Samek Mokryn

Abstract

There is a real danger that the most advanced FPGAs with embedded hard processors are facing extinction. The reasons can be traced to difficulties associated with designing, debugging and support of complex systems. The wrong positioning and a lack of dedicated, specialized tools greatly contributed to this outcome.

In this paper, we describe alternative tools, methodologies and procedures we used to design FC and SAS controllers and protocol converters, which we successfully implemented utilizing FPGAs with embedded processors.

The recommended approach is to relegate the embedded processor to the role of a micro-programmed controller of hardware functions; implement dedicated tools for ease of debugging and support; and finally to avoid using encrypted IP blocks as much as possible.

Introduction

When in 2001 the two largest FPGA companies, Altera and Xilinx introduced FPGAs with hard embedded processors, my first impression was that this was the most important development since the invention of the microprocessor. The hardware design was elevated to a new level: implementation of Systems on a Chip, with an ever increasing level of intelligence and complexity.

By then, I was already designing intelligent controllers based on embedded microprocessors and FPGAs for a number of years. The advantages of combining the processors and the logic on a single chip were obvious to me, as well as the new set of problems this technology introduced.

But by 2003 Altera started to phase out their ARM®-based Excalibur devices. When I asked their VP of marketing their reasons for doing so, the answer was that they didn't see a way how to position the devices. What is more probable, that there was a lot of interest but not enough sales to justify the large effort required to support the devices and design tools needed.

In their latest Virtex 6 line Xilinx too failed to introduce devices with embedded hard processors. Instead they opted for an announcement of some sort of collaboration with ARM, but it will probably take number of years till operational FPGAs with embedded hard ARM processors will be available.

There is a tendency to equate hard processors with soft processors, and some people are trying to promote soft processors instead of dedicated hard processors. However, there is

really no comparison between these two, both performance and complexity wise. And in applications we are dealing with - multi Gb communication controllers, there is no substitution. The reason is simple – we have to deal with both complex checking and error recovery processes, too difficult to implement in hardware alone, and extremely fast data paths.

The Challenge

Designing hardware only FPGAs involves implementation of exactly defined, rather simple functions, their placement, routing, loading, power dissipation, simulation etc. This process is well understood and there are excellent tools to support it.

The introduction of a processor into FPGA changes the picture dramatically. Now we are dealing with whole system on a chip, with its architecture, programming, human interfaces and support requirements. There is a huge set of new tools involved: compilers, debuggers, operating systems, Intellectual Property (IP).

In order to manage this process, the manufacturers provide a set of predefined IPs and procedures under a common framework (EDK), treating the final project as a Lego type assembly. Well, Lego it is not.
What makes the development process so difficult is that the debugging of the final product is highly integrated and involves all tools, both hardware and software. This process is very hard to partition and it requires a very good system, programming and hardware design skills on the part of the designer.

It is not that the design is difficult; it is the debugging that is a nightmare. (For more information on the debugging process please see my other essay: The Art of Debugging – – see http://www.halstor.com/ArtofDebugging.pdf).

General Computing versus Embedded Computing

For some reason vendors positioned the imbedded processors as general computing platforms, with Linux as a targeted operating system. Xilinx even elected to switch from PowerPC405 to 440, processor with virtual addressing mode exclusively. This approach makes a little sense to me, since such architecture requires an extensive I/O, which is very costly to achieve in the resource limited FPGA environment. Standard, external processors do much better job, hence I cannot see a reason why to duplicate them in the FPGA environment.

What makes sense to me is to treat the internal processors as dedicated embedded engines, in essence an extension of hardware. In other words, the processors perform more complex hardware functions, like, for example, a verification of communication frame validity, pointer manipulations, events scheduling and logging. However, these functions are very specialized, they must be performed extremely fast and there is little or no overhead available to support standard, universal operating systems. In other words the embedded code is a set of very small routines, residing in the processor's cache when

possible, and handcrafted to read registers, perform operand manipulation in a strictly controlled order and activate various hardware registers.

Hence all my designs are build on the assumption that there is some external processor running some standard OS with all its facilities and interfaces, while the internal processors are running a dedicated code to support the hardware functions implemented in the device.

Before you start

The first thing anybody has to do is to acquire a set of tools required to implement projects of such magnitude.

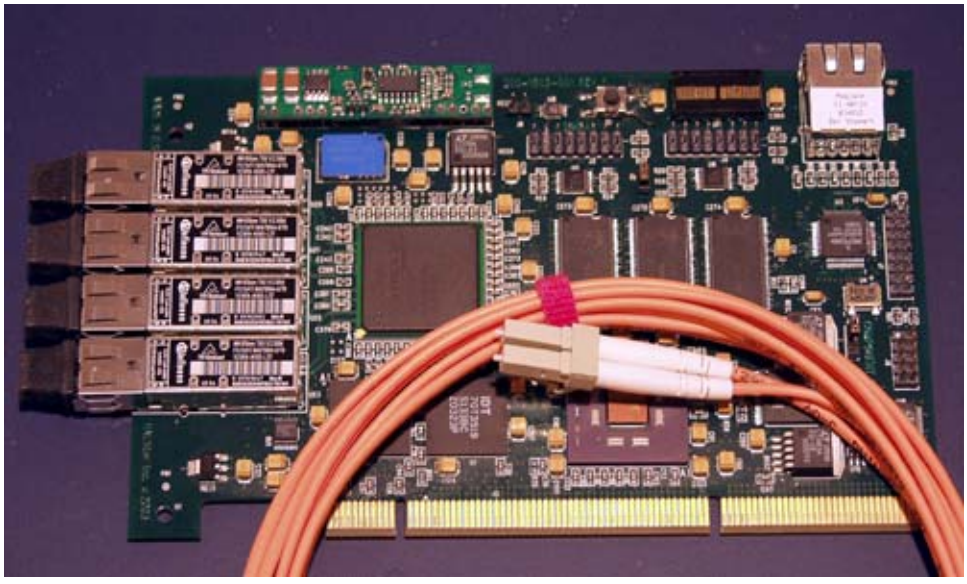
Without proper tools the probability of a successful project completion is next to nil!!!

Assuming again, that the most complex part of this project is the debugging phase, hence the tools have to facilitate an efficient and integrated (both hardware and software) debugging environment.

The first problem arises, when the available tools don't fulfill all your needs. And the chance is that they will not.

Development platform

Since no tools provided by Xilinx and other parties were fulfilling my criteria for an efficient development platform, I had no choice, but to design and implement our own development platform (SIMBA – see www.halstor.com/DvlpSystem.htm).



The approach that I am using in our FPGA development platform employs an external processor which is the operational base for the debugging tools. This processor is tightly

coupled with the embedded (internal to FPGA) processors, it can control their operations and has an access to common, memory-based debugging information. In this way, the embedded processor is free from the tools overhead and their special requirements (like a specific operating system), while the external processor can be adopted for any set of tools.

Since the internal processors are PowerPC405 (V2Pro and V4) or PowerPC440 (V5) the external processor is, by choice, also a PowerPC440, to allow for a common set of software tools and programs.

The external processor is running Linux, which provides a wide range of applications, compilers and communication facilities. All these programs make the development and debugging process easier and provide for fast and efficient human interaction.

The internal embedded processors are running dedicated and specialized code for the functions they have to perform. There is no standard operating system of any sort, which provides for a full transparency of the code – it allows me to control any aspect of the code, optimizing it for the performance and functionality. For an example, I can easily pin some small, critical piece of code in the cache (like polling for some expected event in the existing state) and respond immediately when the event occurs.

The embedded processor communicates with the external processor using operations like xput(), xget() and xprint(). These operations can be easily inserted into the operating program, providing the external processor with information about the system operations. In addition, placing the program stack and data regions into the common memory provides the external processor with debugging information even when the embedded processor is hanging due to some severe error. No user interface is required to be implemented by the embedded processor, since this role has been shifted to the external processor. As a result the embedded code is greatly simplified, making it much more efficient and the testing easier.

To EDK or not to EDK

Very early in the V2Pro introduction process somebody in Xilinx decided that the Intellectual Property (IP) associated with the designs can be a handsome source of revenue.

However, in order to provide revenue, it has to be encoded (not transparent), otherwise it will be too easy to “steal”. The problem with such an approach is that an opaque piece of hardware or software makes it often impossible to debug the design, and may cause the whole project to ultimately fail. Hence the dilemma: is it better to sell more chips and less IP, or vice versa?

Since Xilinx is essentially a hardware and FPGA tools company, I would have expected them, from an end user point of view, to go after the chip sales, but they obviously decided otherwise.

Given that essentially Xilinx's EDK (Embedded Development Kit) is a framework for both general purpose as well as a specialized IP with a complete disregard to the debugging process, the EDK did not fulfill my requirements.

Instead I opted for an implementation of my own specialized development environment, optimized for its intended role. The resulting design is simpler, faster, better integrated with the rest of the project and by far easier to debug than the solutions, if any, provided with the EDK.

One may argue that the design of a circuitry already developed by others is a waste of time and money, however if this circuitry is very difficult to debug, the cost is merely passed on to the integration and debugging phases. All my design experience proves to me, that it is much easier to develop simple transparent circuitry, than to debug somebody else's opaque design.

Architecture

Each set of applications and common underlying technology calls for a dedicated architecture. In our case all designs are based on our proprietary Advanced Serial Controller Architecture (ASCA) which allows a unified treatment of various serial communication protocols. This architecture, combined with our dedicated Serial Controller Development System, provides a significant reduction in the development effort and supports the creation of high performance, cost-effective connectivity for internal and external storage and storage area networks.

One very interesting aspect of designing projects based on FPGAs with embedded processors is the possibility to move the functionality from software to hardware and vice versa. This capability, which was used extensively in my designs, allows starting a design with minimum hardware logic and adding performance sensitive hardware functions as project was progressing. This approach allows cutting man-years from the development effort, while optimizing both resources and functionality.

On Simulation

Due to the "too much data" syndrome, simulation of the entire design doesn't make too much sense. It is very difficult to properly initialize the whole design to the condition being simulated, and it is simply too much detail that the designer has to deal with. This is the main difficulty with ASIC designs, where there is no other way, however with FPGA, due to their inherent flexibility, the "trial and error" method is by far much more productive.

The way I conduct the development process is to first build the function of interest and enough of the additional logic needed to interface the design to the external I/O (like disks and controllers in my case) and using my tools download the design and observe its behavior in operation. The main idea is to collect as much information as possible, and. if

needed, to simulate only a subset of the design in some particular state. Such approach requires a lot of creativity and tools that allow easy modifications of the design and excellent capabilities to collect the information required. The end results will speak well enough for themselves.

Debugging FPGAs with embedded processors

Debugging FPGAs with embedded processors is extensively discussed in the "Art of Debugging" essay and will not be repeated here. Instead I will concentrate on my favorite tools and methodologies that I use for debugging.

Frame Sniffer

For debugging purposes it is mandatory to determine the state of the system (both hardware and software) at the moment the error occurred. Since my target designs deal with frames, hence, as part of the design, I included a built-in rudimentary frame sniffer showing the last n frames passed thru the system. This tool is especially valuable for multi-processor protocol converters, where the knowledge of the interaction between the processors and the external world is essential.

For other types of design, different, comparable tools can be envisioned. In any case tool like this shows how the system interacts with the outside world, and should be as passive as possible. They also should present the interactions in the highest level of abstraction possible, in order to facilitate easy understanding of the system behavior.
Simple traces are simply not good enough.

Event Logging

The event logging is another tool that registers program execution. Essentially, it is based on processors writing messages to some common memory, accessible even when processors hang-up. The examples of such messages are: Start I/O, status received, exceptions encountered, timeouts etc. Such messages impose an additional load on processors and as such have an operational impact. Hence this tool should be used with care, while messages and their updates are frequently modified during the development and debugging processes.

Frankly, without such tools I would not have been able to implement the complex designs as specified.

Final Thoughts

Halstor tested its approach and platform in our SAS and Fibre Channel designs. The result was that we were capable of implementing these complex designs within record time using very limited resources, while most of the debugging was done online.

The debugging process is iterative in nature, with the number of iterations unknown from the start. Finding some errors can take days, weeks, even months. As a result, this is the

most unpredictable and possibly the most expensive part of the design process. Even worse, this is a never-ending process, since problems can be found once the product is manufactured in quantities and distributed to thousands of users. More than this, some users will invariably put the product to work under conditions not expected or not tested by the manufacturer.

The testing considerations should be part of the system specifications. Even as these considerations are limited, covering only a subset of possible error sources and operational behavior only under limited and specific conditions, the flexibility of making quick design changes (both hardware and software) can be a sufficient substitute.

The design and debugging of Systems on Chip requires dedicated tools, tailored to the target project. It also requires plenty of talent, experience, imagination and discipline on the part of the designers. The thinking “out of the box” is a rule, not an exception. Otherwise it is very easy to get bogged down and to inordinately stretch the development period (if there is enough money to support it). It also requires plenty of patience, since, due to the inherent complexity, projects like these are very difficult to schedule.

With all this said, designing SOCs on FPGAs with embedded processors is by far easier than designing equivalent ASICs due to the inherent flexibility and speed of modifications.

About the author:

Samek Mokryn is the founder and president of HalStor, Inc. (www.halstor.com), a design and consulting company specializing in storage and communication interfaces and architectures. He founded HalStor, Inc., after more than 30 years of experience in product development, design and project management.

In 1996, Samek founded C-Star Corp., a company that focused on the development of new data technologies. C-Star later became SANgate Systems - a storage appliance company - where he invented SANgate Systems' breakthrough technologies for managing stored data. SANgate Systems later became Sepaton, Inc., (http://www.sepaton.com) currently located in Marlborough, MA.

Samek previously held senior-level technical positions with EMC Corp., including leading the development team for ESCON connectivity within EMC's Symmetrix Information Storage Systems. He also influenced EMC’s product and marketing policies. Before joining EMC, he served in various development roles related to I/O controllers and computer systems.

Samek holds an MSEE degree from Columbia University and earned a BSEE degree from Technion of Haifa, Israel.

He holds two patents in the area of a data replication.