The Advantages of Run-Time Reconfiguration

Steven A. Guccione and Delon Levi

Xilinx Inc.
2100 Logic Drive
San Jose, CA 95124 (USA)
Steven.Guccione@xilinx.com
Delon.Levi@xilinx.com

Abstract. FPGAs have been sucessfully used to accelerate a wide variety of applications on a large number of systems. The FPGA devices in these systems are typically configured once by the application and seldom perform any sort of reconfiguration during execution. With the advent of new device architectures and new software tools, the interest in *Run-Time Reconfiguration* or *RTR* has increased. As with previous efforts, the focus of RTR has primarily been either in purely theoretical work or in demonstrating performance improvements over software-based solutions. In this paper we explore some of the more practical design issues surrounding RTR systems, and evaluate the advantages of RTR in terms of savings in hardware and software complexity. Preliminary results indicate that RTR can dramaticall reduce the amount of FPGA logic and software support necessary for even simple coprocessing applications.

1 Introduction

The field of Reconfigurable Computing has advanced steadily for the past decade, using FPGAs as the basis for high-performance reprogrammable systems [2]. Many of these systems have achieved very high levels of performance and have demonstrated their applicability to a wide variety of problems. While these systems describe themselves as reconfigurable, they are typically configured once at run-time and seldom, if ever, modified by applications at run time.

Run-Time Reconfigurable or *RTR* systems distinguish themselves by performing circuit logic and routing customization at run-time. While RTR has received some attention from the research community [7], [6], [1], it has lagged other areas in reconfigurable computing research. For the most part, this has been due to a lack of software supporting RTR. With new software development tools arriving [3] [4], it is likely that research and eventually commercial use of RTR systems will begin to accelerate.

While RTR is an exciting research area, very little exploration of the tangible cost and benefits of RTR on system design has been made. This paper seeks to examine the new design issues surrounding design of RTR systems. Early indications are that RTR systems require less hardware, less software and less IO than other FPGA-based systems. This should combine to produce smaller, cheaper systems which are easier and faster to design.

2 The ASIC Approach

Defining RTR itself poses some difficulty. While the definition could artificially be limited to FPGA-based hardware, it is not clear that other systems, including ASICs and other custom hardware, do not make use of RTR. In the most general sense, a system can be said to be reconfigurable if it uses *run-time data to alter the function of hardware*. This is a somewhat broad definition, but should help to define more specifically what is unique about FPGA reconfigurability, and how, if at all, can it be used profitably by system designers.

Using the general definition of RTR, most hardware makes some specialized use of RTR. For the most part, RTR is supported by writable registers which change the operating *mode* of part or all of the hardware. A simple example is a serial interface controller. Writing specific bit patterns to registers inside of this device alter the transmission speed, and even the communication protocol. While this type of circuit may be implemented in an FPGA, there is nothing that requires FPGA technology to produce this type of RTR circuit. For this reason, we will refer to this register-based style of RTR as *the ASIC approach* to RTR.

The circuit in Figure 1 shows a block-level diagram of a datapath circuit used to add a value M to some input data and multiply the result by some other value N, supplying the function $D_{out} = (D_{in} + M) * N$. In this implementation of the circuit the two values are held in two registers which may be written by software at run time. While a simple datapath, this can be viewed as an *offset* and *gain* control of some digital signal.

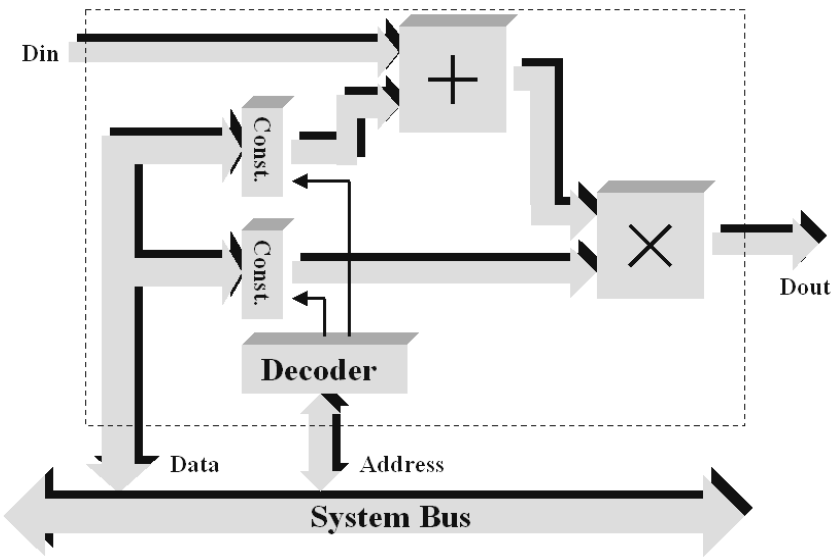

Fig. 1. The ASIC appraoch.

From the diagram in Figure 1, it is clear that the datapath for this circuit will consist of more than a simple adder and multiplier *Core* macro. In addition, two writable registers must be used to hold the values for M and N. But to support these registers, a relatively large amount of complex support hardware must be designed into the system.

First, the circuit must interface to the system bus, so that the values of M and N can be written and (optionally) read. Minimally, this interface will consist of hardware to decode the addresses of the two registers from the system address bus, and routing to send the data from the system data bus to the registers.

This interface circuitry, while fairly common in system design, still has many features which add complexity to the design in general and to FPGA circuits more specifically. The extra hardware required is:

Address Decoder: This takes an address from the system address bus and decodes the address of the desired register. Decoders are somewhat complicated circuits and require relatively large amounts of routing and logic. For this reason many FPGAs offer special hardware to help cope with decoders. Note that each register will require its own address decoding.

Data Bus: The data bus to and from the registers is also problematic. This internal bus typically has a high fan-out, and requires the use of tri-state lines if the registers are readable. This not only requires fixed and limited FPGA resources, but introduces internal timing issues that require detailed analysis and simulation.

IOBs: The address and data bus require IOBs to send and recieve data to and from the circuit. Again, this uses limited and fixed resources and further constrains the circuit.

In addition to adding this extra logic and routing, issues of bus interface timing arise. This may require anything from adherence to a simple timing specification to implementation of a full bus interface, complying with all bus timing and protocols. This portion of the design can quite easily consume the majority of the designer's time.

Finally, once all of the hardware is built and tested, much software remains to be written. In general the software required to support writable configuration resgisters include:

Device Drivers: In most systems some type of device driver must be written to communicate with the hardware. In this case, a driver must be written to read and write the registers. Writing such a driver, even though it usually represents a very small piece of code, is typically a significant portion of the cost of the system design and requires intimate knowledge of the hardware and the system software.

Libraries: While the driver typically supplies direct access to the hardware, some sort of library interface to the driver is usually required. This library provides function calls to users which provide a higher level software interface to the underlying hardware.

Finally, if the hardware is to be used on more than one platform, drivers and libraries must be re-written. This is often the case not just if the hardware

platform is changes, but if any significant change is made to the underlying system software.

3 The FPGA Approach

The ASIC approach to reconfiguration is both widely used and well understood. While it is not specific to FPGA design, this approach is frequently used to support reconfiguration or reconfigurable behavior. While this is adequate for most tasks, many researchers have realized that it is possible to take advantage of the specific characteristics of FPGAs to better support RTR.

Figure 2 shows the same circuit from Figure 1, but using RTR to reprogram the registers. Unlike the previous implementation this circuit is specific to RTR FPGA hardware, and could not be implemented in custom static hardware.

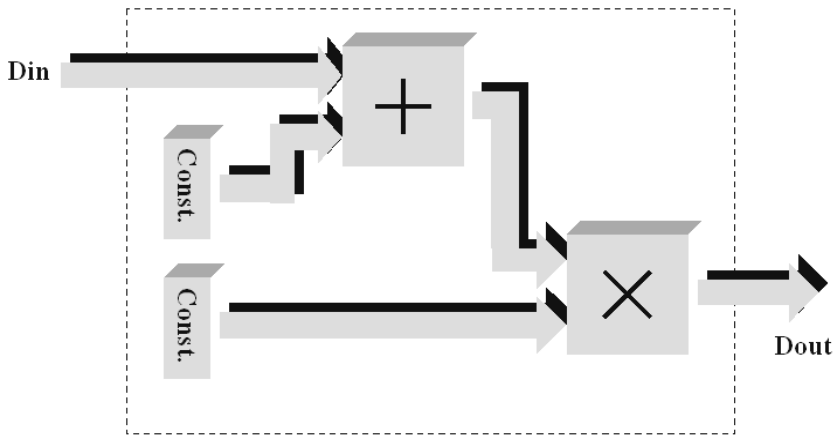

Fig. 2. The FPGA approach.

The differences between the two circuits are obvious at a glance. All of the bus interface logic is eliminated, leaving only the adder and multiplier datapath elements and the M and N registers. The registers are now constant values defined using RTR, rather than written through the system bus at run-time.

The first and most obvious advantage is the elimination of much of the system interface circuitry. Rather than interface to the FPGA in an application-specific or even system-specific manner, the configuration download interface is used to supply the constant values in the registers. This elimination of large amounts of logic and routing results in a smaller circuit than the ASIC approach. This leads to decreased system costs by using a smaller FPGA device, or in increased performance by supporting larger portions of the algorithm in hardware. But

there are other advantages which are even more significant to designers than cost savings or increased performance.

First, the portion of the logic eliminated is the portion which makes use of special purpose FPGA resources such as decode logic and tri-state buffers. This increases the flexibility available to the design software in implementing the circuit. Perhaps most importantly from a hardware savings point of view, a large number of IOBs are saved. In the case of a 32 bit bus, at least 32 dedicated IOBs are are saved. Depending on the implementation this number could increase to 64 or more IOBs. This translates into either system cost savings in packaging, or the ability to provide increased bandwidth to other parts of the system using these freed IOBs. Of course, the dedicated configuration pins are still required, but these are required in any system, regardless of the interface style.

Finally, the design of this circuit no longer requires extensive analysis to perform bus interfacing. With these timing issues removed, the circuit resembles more closely the original algorithm, $D_{out} = (D_{in} + M) * N$. With these timing issues eliminated, the design of this circuit becomes more a case of simple datapath construction, which may be accomplished quickly and efficiently using special datapath construction software tools. Whatever the technique used, the design of the circuit is greatly simplified.

Finally, all of the register interface software is also eliminated. Since there are no registers to be read or written there are no new device drivers or libraries to be written. Note that support software to configure the FPGA is still necessary, but that this software is required whenever an FPGA circuit is used. This software may still take the form of device drivers and libraries, but they will not be application-specific; their function will be identical across platforms.

Also note that software tools which support this reconfiguration will also be necessary. In this case, all that is required is software which sets the appropriate constants in the circuit configuration data. While this support is not directly available in most circuit design tools, more recently developed RTR tools support this directly (cite JavaBits).

4 The Run-Time Reconfiguration Approach

The circuit in Figure 2 makes use of RTR to greatly reduce the original circuit. But it is possible to take further advantage of FPGA capabilities to reduce the circuit even more dramatically. The *FPGA approach* to design only changed the path by which data is written to registers. No actual logic or routing in the FPGA was modified at run time.

The circuit in Figure 3 shows the same circuit, with the constants folded into the arithmetic units. Instead of using a general purpose multiplier and adder and register inputs, constant coefficient arithmetic units are used. These units are necessarily generated and configured at *run-time*. This is necessary because the constant values may not be known at compile-time, and generating all possible circuits for all possible constant values will quickly become prohibitively large.

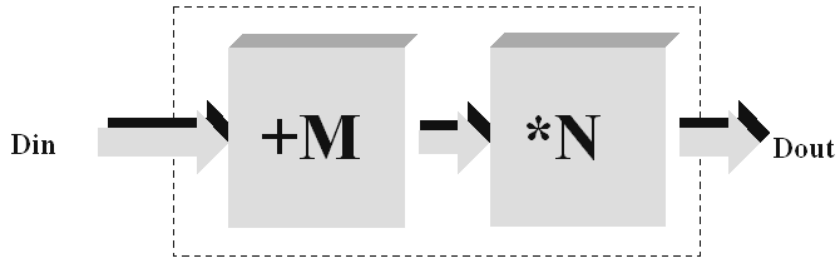

Fig. 3. The FPGA RTR approach.

Compared to the circuit in Figure 2, this use of constant coefficient arithmetic units further reduces circuit size and complexity. Not only are the sizes of the arithmetic units themselves reduced, but the way in which they are interconnected is greatly simplified. In the block diagram, inputs from the registers and from the data inputs are drawn as segregated into two buses. Typically, the routing will be more complicated than is indicated by this stylized circuit. In the adder, for instance, bit zero of each bus will typically enter *bit 0* of the adder, bit one of each bus will enter *bit 1* of the adder, etc. This results in an *interleaving* of the buses, which greatly increases routing complexity.

In the RTR example, note that each arithmetic unit has a single bus input and a single bus output. Routing the circuit, in this case requires only connecting bus outputs to corresponding bus inputs. This task can easily be accomplished, even at run-time, using simple routing algorithms.

One potential drawback of this approach is the amount of configuration data which must be written to the FPGA device. In both of the previous register-based examples, only small amounts of data were involved in customizing the behavior of the circuit. Using the FPGA RTR approach, fully customized circuits must be completely constructed. Depending on the FPGA architecture, there could be a substantial cost to performing this configuration. Whether or not this overhead is excessive will depend not only on the FPGA device being used, but on the application being implemented.

5 Run-Time Reconfiguration Implementations

Until recently, this ability to dynamically reconfigure logic and routing in FPGAs was not available to users. The first significant breakthrough in this area was the Xilinx XC6200 (tm) Reconfigurable Processing Unit [8]. This device featured a novel bus interface which permited fast partial reconfiguration. Perhaps more importantly, the configuration data on this device was published in a public specification. This meant that anyone who desired could perform run-time

reconfiguration of the XC6200 device. Unfortunately, high-level tools to support this capability were slow in arriving.

One exception was the *The Java (tm) Environment for Reconfigurable Computing (JERC6K) (tm)* [4]. This package supports full run-time reconfiguration and comes with a simple run-time parameterizable core library. While the XC6200 device and the *JERC6K* software received little exposure outside of the research community, it led to some ideas which permitted more mainstream FPGA architectures to support run-time reconfiguration.

Based on the *JERC6K* work, the *Xilinx Bitstream Interface (XBI) (tm)* for the XC4000 (tm) family of devices [3] now permits simple reconfiguration of registers, as well as complete run-time reconfiguration of logic and routing resources. This allows users to write high-level software which directly supports RTR on the Xilinx XC4000 family of devices. In addition, a portable hardware interface has been implemented for *XBI* applications [5]. This portable interface permits applications to literally be moved from one platform to another *without recompilation.*

Finally, *XBI* comes with a *Run-Time Parameterizable Core* library. Much like the core library available with *JERC6K*, circuits may be defined and constructed at run-time, possibly using data unavailable at compile time. Early experiences with applications using this library indicate that this core-based approach provides a simple, powerful and portable model for RTR design. For more detailed information on *XBI* and its portable hardware interface, see [3] [5].

6 Conclusions

Some form of RTR is necessary to provide flexibility in hardware functionality. Without the ability to reconfigure hardware at run-time, only the most basic fixed circuits would be possible. The style of reconfiguration typically used by custom hardware designers today is via writable registers to change operational *modes.*

Using FPGA technology, this register-based method of reconfiguration can be dramatically simplified. Instead of writing data directly to registers embedded in the hardware, the circuit is reconfigured to supply the new values. Using reconfiguration in this manner offers several tangible savings to the system designer.

On the hardware side, application specific bus interface logic is eliminated. This dramatically reduces the size of the circuit and frees up Input / Output Buffers (IOBs). In addition, the various timing issues involved in bus interfacing are eliminated. This can greatly reduce the burden on the hardware designer.

On the software side, application-specific low-level interfaces to the application including device drivers and libraries are eliminated. All communication with the FPGA is done through the configuration interface. This not only reduces the design effort in one of the most difficult portion of the system, the hardware / software interface, but it enhances system portability. New interface software does not have to be written if the design is moved to a new system.

Finally, further use of FPGA capabilities to modifiy logic and routing at run-time allow custom synthesized logic to be generated and configured at run-time. This allows registers to be eliminated and the constant values to be folded into the circuitry. This results in an even smaller, simpler and faster circuit than any existing approaches.

It is expected that the recent availability of RTR software will accelerate research in RTR systems. And because RTR offers enhanced performance combined with savings in both hardware and software complexity, it is likely that this interest will quickly move from research to commercial use. Our experiences using tools like *XBI* on real applications, using real hardware, indicate that it is possible to produce true RTR applications using existing mainstream architectures.

References

1. Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dynamic reconfiguration run-time system. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 66–75, Los Alamitos, CA, April 1997. IEEE Computer Society Press.
2. Steven A. Guccione. List of FPGA-based computing machines. World Wide Web page http://www.io.com/~guccione/HW_list.html, 1997.
3. Steven A. Guccione and Delon Levi. XBI: A java-based interface to FPGA hardware. In John Schewel, editor, *Configurable Computing Technology and its use in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East*, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.
4. Eric Lechner and Steven A. Guccione. The Java environment for reconfigurable computing. In Wayne Luk and Peter Y. K. Cheung, editors, *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997. Lecture Notes in Computer Science 1304*, pages 284–293. Springer-Verlag, Berlin, September 1997.
5. Delon Levi and Steven A. Guccione. BoardScope: A debug tool for reconfigurable systems. In John Schewel, editor, *Configurable Computing Technology and its use in High Performance Computing, DSP and Systems Engineering, Proc. SPIE Photonics East*, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.
6. Wayne Luk, Nabeel Shirazi, and Peter Y. K. Cheung. Compilation tools for run-time reconfigurable designs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, Los Alamitos, CA, April 1997. IEEE Computer Society Press.
7. Herman Schmit. Incremental reconfiguration for pipelined applications. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, Los Alamitos, CA, April 1997. IEEE Computer Society Press.
8. Xilinx, Inc. *The Programmable Logic Data Book*, 1996.

This article was processed using the LaTeX macro package with LLNCS style