Run-Time Parameterizable Cores

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. As FPGAs have increased in density, the demand for predefined intellectual property has risen. Rather than re-invent commonly used circuitry, libraries of standard parts have become available from a variety of sources. Currently, all of these offerings are based on the standard ASIC design flow and are used to produce fixed designs. This paper discusses *Run-Time Parameterizable* or *RTP* Cores which are an extension of the traditional static core model. Written in the Java (tm) programming language, RTP Cores are created at run-time and may be used to dynamically modify existing circuitry. In addition to providing support for run-time reconfigurable computing, RTP Cores permit run-time parameterization of designs. This adds flexibility and portablilty unavailable in existing design environments.

1 Introduction

In most FPGA designs, engineers make extensive use of preconstructed libraries. Rather than implement an entire design in low-level detail, these library elements or *Cores* are commonly used to build complex designs. These cores supply high level building blocks which can greatly simplify the design task for users. While a variety of core libraries have existed from both commercial vendors and independent sources, the increase in density in FPGAs has led to an increasing demand for cores. The complexity of these cores has also increased correspondingly.

The most basic type of core is a *fixed core*, which has a pre-defined size and cannot be modified by the designer. This type of core has been the basis of most FPGA design tools for the past decade. These cores typically supply standard functions familiar to board level designers, including components such as TTL 7400 series.

As larger cores have been added to libraries, the trend has been away from fixed cores to *parameterizable* cores. Parameterizable cores permit the user to enter information about the desired core, and a customized circuit conforming to the information supplied by the user is constructed.

One example of a parameterizable core would be an adder circuit. When requesting an adder core, a user could be asked to specify the bit width of the adder. This permits any size adder to be generated by a user. With fixed cores, the library would supply standard size adders, typically 4 bit, 8 bit and 16 bit adders. The user would select the pre-constructed circuit which best fit the needs of the circuit being designed.

Parameterization can extend beyond simple bit widths. Adders, for example, can be parameterized to provide various speed versus area tradeoffs. The ability to customize these cores is limited only by the skill of the core designer. With this ability to select from a larger number of cores, circuits can be better tailored to the particular design at hand, with less wasted circuitry and higher performance.

Recently, researchers and commercial vendors have focused attention on these parameterizable core libraries [3] [4] [6]. While these core libraries increase flexibility and provide better solutions to designers, their use is limited to what information can be provided at the time the circuit is designed.

This paper discusses a substantially new type of parameterizable core circuit, the *Run-Time Parameterizable* or *RTP* Core. This type of core is used exclusively in a *Run-Time Reconfigurable* or *RTR* system, where FPGA logic and routing are dynamically modified at run-time. These RTP Cores permit circuits to be constructed and instantiated at run-time, while the system is executing. This permits a new degree of flexibility in design, particularly in RTR systems. Designs may now react to information supplied in real-time, either by system software, user input or by real-time sensor data. This provides a high-level mechanism for performing true RTR using FPGAs. With this capability, systems can be designed which can interact with real-time data and host software.

2 Run Time Parameterization

Unlike compile-time parameterization, designs which are run-time parameterizable have an added degree of flexibility. The ability to create or modify circuits at run-time creates new design options that are unavailable in designs parameterized at compile-time. In general, RTP Cores provide a simple mechanism for producing designs which can vary in functionality based on various types of input. While this list is not exhaustive, some types of input and their use to parameterize circuits is discussed below:

Command line: Using *Command line* parameters, a command-line flag, or other forms of user input can be used to select a particular configuration. This is useful in applications where very similar circuits providing different functionality are supplied in a single RTR design. An example is an encode / decode circuit. A command line flag may be passed to the RTR software to permit either an encoder or a decoder circuit to be configured, depending on the desired mode of use.

Device type: In addition to reacting to command line parameters, it is also possible to parameterize circuits at run-time based on the actual *Device type* being used. For instance, if a small device is being used, a smaller, perhaps less accurate circut may be configured. If a larger device (or portion of a device) is available, a larger, perhaps more accurate circuit may be configured. For instance, in a DSP application, an 8-bit filter may be configured in a small device, where a 16 or even 32 bit instantiation may be used in a larger device. This not only allows one single compiled design to be used on many different sized devices across a family, but allows functionality such as accuracy or speed to change appropriately, depending on the hardware being used.

User input: *Command line* selection typically permits a simple configuration choice to be made at the beginning of execution. It is also possible with RTR to provide a user interface which permits circuit modification at the user's command. An example of *User input* configuration would be an image processing application which uses an FPGA to do coprocessing. A GUI which takes user inputs for processing parameters such as gain, offset, and coloring could be used to directly generate circuit parameters. These parameters may then be used to construct the RTP Cores used in the image processing circuit.

Real-time input: Where *User input* configuration requires a human to manually control RTR, it is also possible to query real-time data, typically from sensors, to drive RTR without human intervention. This permits capabilities such as adaptive digital filtering, where filtering is modified depending on various real-time system conditions.

Circuit state: Where *User and Real-time input* configuration provide for external control of RTR, it is also possible to query data within the currently configured circuit, typically by reading registers from software. The system could then use these values to perform RTR. While similar to using real-time inputs, the ability to probe the internal state of the device permits the possibility of simpler system integration. One example would be an internally configured counter circuit which keeps track of an external event count. Probing the state of such a counter permits software to react to internal values, and to use these internal values to perform circuit parameterization in real-time.

These represent broad classes of input data which can be used to drive RTR using RTP Cores. Using such data to construct circuit configuration parameters, RTP Cores can be instantiated and used to configure FPGAs at run-time. In addition to producing smaller, faster circuits, this approach permits the construction of circuits which would not be feasible using compile-time parameterization of circuits. Reacting to command line parameters, device type, user input, real-time input and circuit state all provide capabilities for circuit designers far beyond what is available using static circuit design tools such as schematic capture and hardware description languages.

3 The *JBits* System

Because RTP Cores are parameterizable at run-time, it is not practical to implement them using standard design tools. These tools, including schematic capture and hardware description languages, were originally designed to produce fixed, static circuits. Using these existing tools to support RTR and RTP Cores is currently not possible.

For this reason, RTP Cores are implemented using the Xilinx *JBits* interface [5]. This design environment is implemented completely in the Java (tm) programming language and currently provides RTR support for the Xilinx

XC4000EX (tm) and *XC4000XL* (tm) series of FPGA devices. *JBits* provides an *Application Program Interface (API)* into the device configuration bitstream, permitting logic and routing to be modified at run-time.

It should be noted that *JBits* is based on earlier work on the Xilinx *XC6200* (tm) reconfigurable device. This work was know as the *Java Environemnt for Reconfigurable Computing* or *JERC6K* [7]. While a very small number of RTP Cores were supplied with *JERC6K*, the emphasis was on other aspects of RTR. Following the experience gained in development and use of *JERC6K*, *JBits* was implemented and focus quickly moved from low-level to high-level design details. This has led to more of an emphasis on cores and other support for high-level design activities.

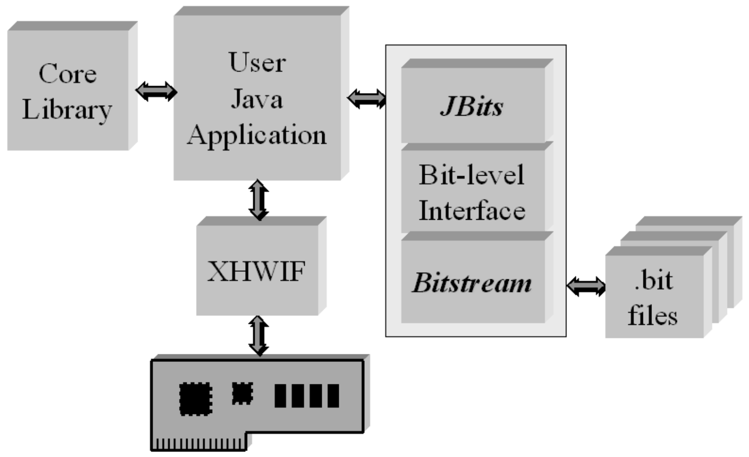

Fig. 1. The *JBits* system.

The *JBits* system views all devices in a given FPGA family as tiled arrays of *Configurable Logic Blocks* or *CLBs*. Associated with each CLB tile is some amount of routing. Partitioning the device in this way permits small sections of the device to be programmed, and the programming to be replicated in loops or implemented conditionally. This programming language support for repitition and conditionality is what provides the basic and necessary support for all paramtereizable cores, but particularly for RTP Cores.

Using this device model and programming interface, RTP Cores are defined as Java objects which can be constructed and then written as device configuration data. Because the device model is a simple two dimensional array of CLBs, the only difference between various devices in a family is the size of the array. This permits cores to be relocatable and device-independent. A *JBits* core used on the smallest XC4000EX family part will also work, unmodified, on the largest XC4000EX family part. Of course, the size of a core may limit the devices with which it can be used. A core whose smallest instantiation is 20 x 20 CLBs clearly will never fit in a device which contains only 10 x 10 CLBs.

Finally, note that the details of the *JBits* interface are not discussed in detail in this paper. Suffice it to say that all device resources may be configured via *JBits*. This capability is used to build Java objects which construct higher level circuits which remove the need for the user to understand the low-level details of *JBits* and the underlying device architecture. For more a more detailed description of *JBits*, see [5]. Note that this paper refers to the *JBits* software as *"XBI"*. This was the internal development name of *JBits*.

4 Stitcher Cores

While RTP Cores implemented using *JBits* permit fast, run-time parameterization of circuits, one crucial piece is still missing. It is easy to construct and instantiate RTP Cores, but the issues of interconnecting these cores has until now been avoided, even in previous RTR efforts. Clearly, one possibility is using a general purpose router such as the one used in traditional placement and routing tools. Such a router would have to be more intelligent than current routers to keep track of the dynamically changing circuit within the device. In addition, all RTR code, including cores, would have to keep the router informed in some way of the resources being used. This system would be analogous to dynamic memory allocation in software. Because the usage of memory is varying at run-time, some mechanism, be it either an explicit relinquishing of resource, or some form of run-time garbage collection, must be performed to keep track of used and free resources.

While it is possible to implement a general-purpose router which works at run-time, there are two immediately obvious problems with this approach. First, indeterminate circuit generation times could be a problem for many systems, particularly those with real time constraints. And since it is assumed that the FPGA is being used as a coprocessor of some sort to off-load work from the main processor, it may be difficult to justify re-loading the processor with the complex task of performing routing. It is likely that any processor capable of performing real-time FPGA routing would have little need for a high performance FPGA coprocessor.

Finally, while router performance is an issue, completion is an even more severe problem. What should a system do if the router fails to find a solution within the given constraints? It is not likely that many systems would be able to function with even the possibility of such a failure.

One possible solution is to define a special type of RTP core called a *Stitcher*. This core is the same as any other RTP core, except that it has no logic and only modifies routing resources.

The operation of a Stitcher Core is very simple. The Stitcher Core abstracts away the underlying routing architecture in the same way that standard RTP Cores abstract away the underlying logic architecture. Inputs of an RTP core are connected in some structured fashion to the outputs of another RTP core.

```
/* Set up the JBits interface */
jbits = new JBits(deviceType);

/* Instanitate a 16-bit counter core at CLB(10,10) */
Counter counter = new Counter(16, 5);
counter.set(jbits, 10, 10);

/* Instanitate a 16-bit +7 constant adder core at CLB(10,11) */
ConstAdder constAdder = new ConstAdder(16, 7);
constAdder.set(jbits, 10, 11);

/* Stitch the counter and adder together */
Stitcher stitcher = new Stitcher(Stitcher.F1_IN, Stitcher.YQ_OUT, 16);
stitcher.set(jbits, 10, 11);
```

Fig. 2. A *JBits* RTP Core and Stitcher example.

The Stitcher must be aware of the geometry of the core inputs and outputs in question, but this has not been a problem. All cores in the library have so far fallen into a small number of input and output styles, the largest variation being the *stride* relative to the CLB array. In any case, if other more unusual RTP Cores are built in the future, it should be no problem to write new Stitcher Cores which work with these cores.

Using an RTP Stitcher Core is very simple. First, the two cores to be stitched are instantiated using their Java constructors. Then they are written to particular locations in the device using the *JBits set()* function. Once the RTP Cores are in place, the Stitcher may be used to connect them. The Stitcher is also instantiated, then written at the juncture of the two cores, again using the *JBits set()* function. This connects the cores.

Figure 2 shows actual *JBits* code to which contains a 16-bit counter (which is initialized to a value of "5") and a 16-bit constant adder which adds "7" to its input. The Stitcher core in this code connects the *YQ* CLB outputs of the counter to the *F1* LUT inputs of the adder. Note that since the stitcher has no width, it is *set()* to the same location as the constant adder. Also note that the Stitcher Core is treated just like any other RTP core.

While Stitcher Cores typically contain only routing resources, they are often represented as one-dimensional cores with no width. This indicates the absence of logic resources, and permits automatic placement and relative placement software to function in the presence of Stitchers. Figure 3 shows a diagram of such a Stitcher Core. However, it should be mentioned that it is not necessary that cores be abutted and Stitchers have zero width. Because *JBits* is a software solution, it is possible to define stitchers which operate in any mode and connect arbitrary cores at arbitrary locations. In this sense, Stitchers may be viewed as small, special purpose *auto-routers*. This permits them to execute very quickly, with approximately one interconnect performed per line of Java code. Perhaps

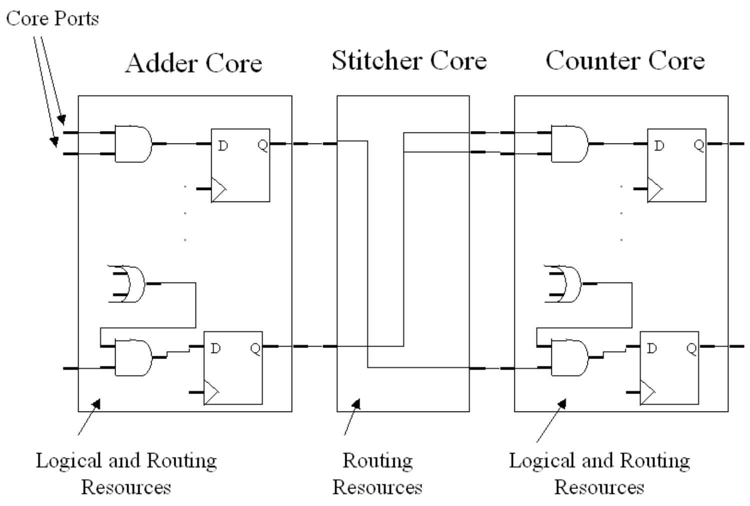

Fig. 3. A Stitcher Core.

more importantly, RTP Stitcher Cores guarantee completion of routing.

5 Conclusions

Run-Time Parameterizable Cores or *RTP Cores* are high-level circuit components which can be configured at run-time. Unlike other existing systems which supply fixed circuit parameters at compile time, new levels of flexibility are possible using RTP Cores. Circuits which adapt based on user input, real-time input, the device being used or even internal FPGA state are possible with RTP Cores.

RTP Cores are currently implemented using the in the Xilinx *JBits*. This system currently supports run-time reconfiguration on the Xilinx XC4000EX and XC4000XL families of devices. Using this RTP core based design approach, fairly complex circuits using *Run-Time Reconfiguration* or *RTR* can be built.

In addition, the introduction of *Stitcher* cores to perform special-purpose routing tasks supplies the final piece of the design environment. While *JBits* and *RTP* cores have been in use by our group for a little over a year, the results have been very positive. A small handful of demonstration applications have been constructed and run on actual hardware. In most cases, the applications have been run on different hardware platforms with different FPGA devices, without re-compilation.

References

1. Gordon Brebner. The swappable logic unit: A paradigm for virtual hardware. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for*

Custom Computing Machines, pages 77–86, Los Alamitos, CA, April 1997. IEEE Computer Society Press.

2. Gordon Brebner. Circlets: Circuits as applets. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 300–301, Los Alamitos, CA, April 1998. IEEE Computer Society Press.
3. Michael Chu, Nicholas Weaver, Kolja Sulimma, Andre DeHon, and John Wawrzynek. Object oriented circuit generators in Java. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, April 1998. IEEE Computer Society Press.
4. PAM-Blox: High Performancs FPGA Design for Adaptive Computing. Oskar mencer and martin morf and michael j. flynn. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, Los Alamitos, CA, April 1998. IEEE Computer Society Press.
5. 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.
6. James Hwang, Cameron Patterson, S. Mohan, Eric Dellinger, Sujoy Mitra, and Ralph Wittig. Generating layouts for self-implementing modules. 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.
7. 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.

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