The Java Environment for Reconfigurable Computing

Eric Lechner and Steven A. Guccione

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

Abstract. The *Java Environment for Reconfigurable Computing (JERC)* is a software environment for reconfigurable coprocessor applications. This environment consisting of only a standard Java compiler and a set of libraries. Using JERC, configuration, reconfiguration and host run-time operation is supported. JERC also features design compile times on the order seconds and built-in support for parameterized macros.

1 Introduction

In recent years, there has been an increasing interest in reconfigurable logic based processing. These systems attempt to use reconfigurable logic to implement algorithms directly in hardware, thus increasing performance.

By one count, at least 50 different hardware platforms have been built to investigate this novel approach to computation [5]. Unfortunately, software seems to lag behind hardware in this area. Most systems today employ traditional circuit design techniques, then interface these circuits to a host computer using standard programming languages.

Work done in high-level language support for reconfigurable logic based computing currently falls into two major approaches. The first approach is to use a traditional programming language in place of a hardware description language [2] [6]. This still requires software support on the host processor.

The second major approach is compilation of standard programming languages to reconfigurable logic coprocessors. These typically attempt to detect computationally intensive portions of code and map them to the coprocessor [3] [4] [7] [9] [10] [8]. These compilation tools, however, are usually tied to traditional placement and routing back-ends and have relatively slow compilation times. They also provide little or no run-time support for dynamic reconfiguration.

The *Java Environment for Reconfigurable Computing (JERC)* represents a novel approach to hardware / software codesign for reconfigurable logic based coprocessors. Using the *JERC* libraries and standard Java, configuration, reconfiguration and host interface software for coprocessing applications is supported in a single piece of code. Additionally, since this tool does not make use of the traditional placement and routing approach to circuit synthesis, compilation times are on the order of seconds. This combines to produce a development environment which very closely resembles those used for modern software development.

2 The Design Flow

Design of an application using a reconfigurable logic coprocessor currently requires a combination of two distinct design paths. The first, and perhaps most significant portion of the effort involves circuit design using traditional CAD tools. This design path for these CAD tools typically consists of entering a design using a schematic editor or hardware design language, generating a netlist for this design, importing this netlist into an FPGA placement and routing tool, which finally generates a file used to configure the FPGA logic.

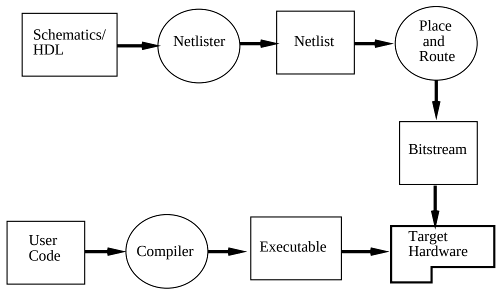

Fig. 1. Traditional design flow.

Once the configuration data has been produced, the next task is to provide software to interface the host system to the reconfigurable logic coprocessor. This task is usually completely decoupled from the task of designing the circuit, and hence is often difficult and error-prone. This dual path design flow is shown in Figure 1.

In addition to the problems of interfacing the hardware and software in this environment, there is also the problem of design cycle time. Any change to the circuit design requires a complete pass through the hardware design tool chain. This process is time consuming, with the place and route portion of the chain typically taking several hours to complete.

Finally, this approach provides no support for reconfiguration. The traditional hardware design tools provide support almost exclusively for static design. It is even difficult to imagine constructs to support run-time reconfiguration in environments based on schematic or HDL design entry.

In contrast, the *JERC* environment consists of a library of functions which permit logic and routing to be specified and configured in a reconfigurable logic device. By making calls to these library functions, circuits may be configured and reconfigured. Additionally, host code may be written to interact with the reconfigurable hardware. This permits all design data to reside in a single system, often in a single *Java* source code file.

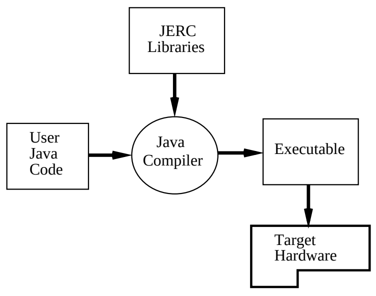

Fig. 2. JERC design flow.

In addition to greatly simplifying the design flow, as shown in Figure 2, the *JERC* approach also tightly couples the hardware and software design processes. Design parameters for both the reconfigurable hardware and the host software are shared. This coupling provides better support for the task of interfacing the logic circuits to the software.

3 The JERC Abstraction

JERC takes a layered approach to abstracting the reconfigurable logic. At the lowest layer, *Level 0*, *JERC* supports all accessible hardware resources in the reconfigurable logic. Extensive use of constants and other symbolic data makes Level 0 usable, in spite of the necessarily low level of abstraction.

The current platform for the *JERC* environment is the *XC6200DS* Development System [12]. This system consists of a a PCI board containing a Xilinx *XC6216* FPGA [11]. In the *XC6200*, Level 0 support consists of abstractions for the reconfigurable logic cells and all routing switches, including the clock routing. The code for Level 0 is essentially the bit-level information in the *XC6200 Data Sheet* (cite data sheet) coded into Java.

While Level 0 provides complete support for programming all aspects of the device, it is very low level and may be too tedious and require too much specialized knowledge of the architecture for most users. Although this layer is always available to the programmer, it is expected that level 0 support will function primarily as the basis for the higher layers of abstraction. In this sense, Level 0 is the "assembly language" of the JERC system.

Above the Level 0 abstractions is the Level 1 abstraction. This abstraction permits simpler access to logic definition, clock and clear routing and the host interface.

The most significant portion of the level 1 abstraction is the logic cell definition. This permits cells in the XC6200 to be configured as standard logic operators. Currently, *AND, NAND, OR, NOR, XOR, XNOR, BUFFER* and *INVERTER* combinational logic elements are supported. These may take an optional registered output. Additionally, a *D flip-flop, toggle flip-flop* and a *register* logic cell is defined. All of these logic operators are defined exclusively using *JERC* level 0 operations, and hence are easily extended. Figure 3 gives a diagram of the Level 1 cell abstraction.

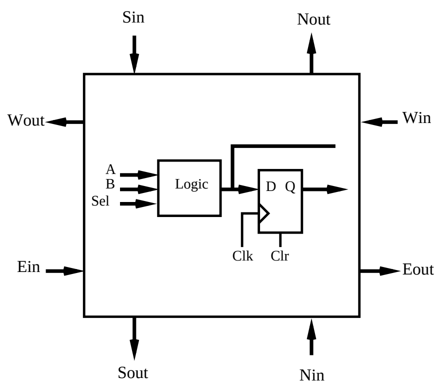

Fig. 3. The JERC Level 1 cell abstraction.

A second portion of the JERC Level 1 abstraction is the *Register* interface. In the XC6200, columns of cells may be read or written via the bus interface. The *Register* interface allows registers to be constructed and accessed symbolically.

In addition to the logic cell and register abstractions, the clock routing is abstracted. Various global and local clock signals may be defined and associated with a given logic cell.

4 A Counter Example

This section describes a simple counter based on toggle flip flops using the Level 1 abstraction. In less than 30 lines of code, the circuit is described and configured and clocking and reading of the counter value is performed. In addition, the structure of this circuit permits it to be easily packaged as a parameterized object. Such and object based approach would permit counters of any size to be specified and placed at any location in the XC6200.

The implementation process is fairly simple. First, the logic elements required by the circuit are defined. These circuit element definitions are abstractions and are not associated with any particular hardware implementation.

Once these logic elements are defined, they may be written to the hardware, configuring the circuit. Once the circuit is configured, run time interfacing of the circuit, usually in the form of reading and writing registers and clocking the circuit, is performed. If the application demands it, the process may be repeated, with the hardware being reconfigured as necessary.

The counter example contains 9 basic logic elements. Two of these are *Registers* which simply interface the circuit to the host software. These two registers are used to read the value of the counter and to toggle a single flip flop, producing the *local clock*.

To support the flip flops in the XC6200, clock and clear inputs must also be defined. The *global clock* is the system clock for the device and must be used as the input to any writable register. In this circuit, the flip flop which provides the software controlled local clock must use the global clock.

The *local clock* is the output of the software controlled clock, and must be routed to the toggle flip flops which make up the counter. Finally, all flip flops in the XC6200 need a *clear* input. In this circuit, the *clear* input to all flip flops is simply set to logic zero.

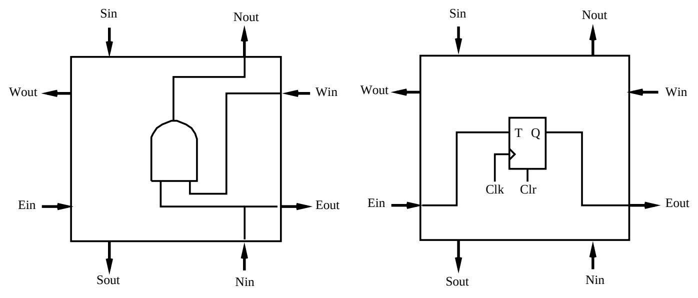

Fig. 4. The carry and toggle flip flop cell definitions.

These four logic elements provide all of the necessary support circuitry to read, write, clock and clear the hardware. The remaining logic elements are used to define the counter circuit itself.

The first logic element in the circuit is the *clock*. This is just a single bit *Register* which is writable by the software. Toggling this register via software control produces the clock for the counter circuit.

The next element is a toggle flip flop, *tff*. This flip flop is defined as having an input coming from the west. This element provides the state storage for the counter. Next, the *carry* logic for the counter is simply an AND gate with inputs from the previous stage and the output of the current stage. This generates the “toggle” signal for the next stage of the counter. Figure 4 gives a graphical representation of these two logic cells.

Finally, a logic *one* cell is implemented for the carry input to the first stage of the counter. Figure 5 give the *JERC* code for describing the basic logic elements. The *pci6200* object passed to each of the logic definitions is the hardware interface to the *XC6200DS* PCI board.

```
Pci6200 pci6200 = new Pci6200N(null); // Hardware interface
pci6200.connect();
Register counterReg =  new Register(COLUMN, counterMap, pci6200);
Register clockReg =    new Register(COLUMN, clockMap, pci6200);
ClockMux localClock =  new ClockMux(ClockMux.CLOCK_IN);
ClockMux globalClock = new ClockMux(ClockMux.GLOBAL_CLOCK);
ClearMux clear =       new ClearMux(ClearMux.ZERO);
Logic tff =            new Logic(Logic.T_FLIP_FLOP, Logic.EAST);
Logic clock =          new Logic(Logic.REGISTER);
Logic one =            new Logic(Logic.ONE);
Logic carry =          new Logic(Logic.AND, Logic.NORTH, Logic.WEST);
carry.setEastOutput(Logic.NORTH); // Set carry output
```

Fig. 5. The logic element definition code.

Once this collection of abstract logic elements is defined, they may be instantiated anywhere in the XC6200 cell array. This is accomplished by making a call to the *write()* function associated with each object. This function takes a *column* and *row* parameter which define the cell in the XC6200 to be configured. Additionally, the hardware interface object is passed as a parameter. In this case, all configuration is done to *pci6200*, a single *XC6200DS* PCI board.

The code in Figure 7 performs all configuration. In the *for()* loop, the *carry* cells go in one column with the *tff* toggle flip flops in the next column. A *local clock* and a *clear* is attached to each *tff* toggle flip flop. A graphical representation of the location of these cells is shown in Figure 6.

Below the *for()* loop, a constant "1" is set as the input to the carry chain. Next the software controlled clock is configured. This is the *clock* object, with its *local Clock* routing attached to the toggle flip flops of the counter. Finally, the *global clock* is used to clock this software controlled clock.

Once the circuit is configured, it is a simple matter to read and write the *Register* objects via the *set()* and *get()* functions. In Figure 8, the clock is toggled by alternatively writing "0" and "1" to the *clock register*. The *counter register* is used to read the value of the counter. Next to the code is an actual trace of the execution of this code running on the *XC6200DS* development system.

While this is a simple example for demonstration purposes, it makes use of all of the features of *JERC*. This includes register reads and writes, as well as features such as software driven local clocking. Other more complex circuits have been developed using JERC, but differ primarily only in the size of the code, not the number of features needed to provide system level support for reconfigurable

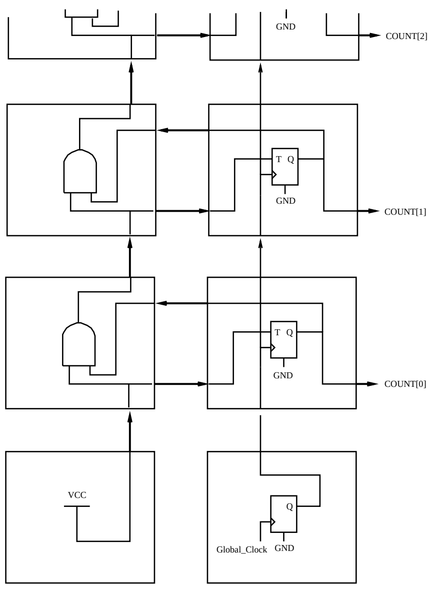

Fig. 6. The relative locations of cells in the counter.

```
/* Configure cells */
for (i=ROW_START; i<ROW_END; i++) { // The counter
   carry.write((COLUMN-1), i, pci6200);
   tff.write(COLUMN, i, pci6200);
   localClock.write(COLUMN, i, pci6200);
   clear.write(COLUMN, i, pci6200);
   } /* end for() */
one.write((COLUMN-1), (ROW_START-1), pci6200); // Carry in
clock.write(COLUMN, (ROW_START-1), pci6200); // Clock
localClock.set(ClockMux.NORTH_OUT);
localClock.write(COLUMN, ROW_START, pci6200);
globalClock.write(COLUMN, (ROW_START-1), pci6200);
```

Fig. 7. The configuration code.

```
for (i=0; i<5; i++) {
   clockReg.set(0); // Toggle clock
   clockReg.set(1);
   System.out.println("Count: " +
      counterReg.get());
   } /* end for() */
```

```
C:\java\JERC>java Counter
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
C:\java\JERC>
```

Fig. 8. The run time code and the execution trace.

processing.

5 Drawbacks of JERC

While *JERC* provides a simple, fast, integrated tool for reconfigurable logic based processing, there are still several drawbacks. First, *JERC* is currently a manual tool. Since it is possible to perform reconfiguration, it is necessary for the programmer to exercise tight control over the placement and routing of circuits. For highly repetitive designs, this is not a problem, but using *JERC* for large, unstructured designs is not recommended.

Also, *JERC* relys on abstractions to simplify the design process. Unfortunately, with this abstraction and simplification comes a loss of flexibility. Many hardware resources are ignored or abstracted away at the higher *JERC* levels. Of course, Level 0 operation is always an option, but this requires detailed knowledge of the underlying architecture, and may be quite tedious.

Finally, *JERC* currently provides no timing analysis of the underlying circuits. This is perhaps a more fundamental problem with reconfigurable systems in general. If true dynamic reconfiguration is possible, analyzing the possible circuits is likely to prove to be a difficult problem. But it should always be possible for the programmer to do simple critical path analysis, which should give a reasonable upper bound on the clock speed.

6 Future Plans

Work on *JERC* is continuing. Three particular directions are being investigated. First, the object oriented nature of *Java* permits libraries of parameterized macrocell-like objects to be built. This could significantly increase the productivity of users of *JERC*.

The second area of investigation is using *JERC* as a basis for a traditional graphical CAD tool. While this would be useful for producing static circuits, it is not clear how temporal reconfiguration would be managed. It would, however, trade very fast compilation times in exchange or the manual design style of *JERC*.

The last area of investigation is higher levels of abstraction. One possibility is to add some limited automatic placement and routing capability.

7 Conclusions

JERC represents a novel approach to reconfigurable computing. Using a single inexpensive, off the shelf development tool, circuits can be constructed, reconfigured and interfaced to host systems.

Perhaps more importantly, the compile times necessary to produce these circuits and run-time support code is on the order of seconds. This is many orders of magnitude faster than the design cycle time of traditional CAD tools.

This permits development in an environment that in nearly all ways operates like a modern intergrated software development environment.

Early experiences has shown *JERC* to be a fast and friendly alternative to existing approaches to algorithm development for reconfigurable computing.

Acknowledgements

Thanks to the engineering staff of the Xilinx Development Corporation for XC6200 and XC6200DS support. And thanks to Eric Dellinger for support of the JERC concept.

References

1. Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
2. David Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 136–144, Los Alamitos, CA, April 1995. IEEE Computer Society Press.
3. Maya Gokhale and Aaron Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pages 399–408, 1996. Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL 95. Lecture Notes in Computer Science 972.
4. Steven A. Guccione. *Programming Fine-Grained Reconfigurable Architectures*. PhD thesis, The University of Texas at Austin, May 1995.
5. Steven A. Guccione. List of FPGA-based computing machines. World Wide Web page http://www.io.com/~guccione/HW_list.html, 1997.
6. Shaori Guo and Wayne Luk. Compiling ruby into FPGAs. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pages 188–197, 1996. Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL 95. Lecture Notes in Computer Science 972.
7. Reiner W. Hartenstein, Alexander G. Hirschbiel, Michael Reidmüller, Karin Schmidt, and Michael Weber. A novel ASIC design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7):975–989, July 1991.
8. Christian Iseli and Eduardo Sanchez. A C++ compiler for FPGA custom execution unit synthesis. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 173–179, Los Alamitos, CA, April 1995. IEEE Computer Society Press.
9. James B. Peterson, R. Brendan O'Connor, and Peter M. Athanas. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 178–187, Los Alamitos, CA, April 1996. IEEE Computer Society Press.
10. Markus Weinhardt. Portable pipeline synthesis for FCCMs. In Reiner W. Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic: Smart Applications,*

New Paradigms and Compilers, pages 1–13, 1996. Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications, FPL 96. Lecture Notes in Computer Science 1142.
11. Xilinx, Inc. *The Programmable Logic Data Book*, 1996.
12. Xilinx, Inc. *XC6200 Development System*, 1997.