# JRoute: A Run-Time Routing API for FPGA Hardware

Eric Keller

Xilinx Inc.
2300 55th Street
Boulder, CO 80301
Eric.Keller@xilinx.com

**Abstract.** JRoute is a set of Java classes that provide an application programming interface (API) for routing of Xilinx FPGA devices. The interface allows various levels of control from connecting two routing resources to automated routing of a net with fanout. This API also handles ports, which are useful when designing object oriented macro circuits or cores. Each core can define its own ports, which can then be used in calls to the router. Debug support for circuits is also available. Finally, the routing API has an option to unroute a circuit. Built on JBits, the JRoute API provides access to routing resources in a Xilinx FPGA architecture. Currently the Virtex$^{\mathrm{TM}}$ family is supported.

## 1 Introduction

JRoute is an API to route Xilinx FPGA devices. The API allows the user to have various levels of control. Using this API along with JBits, the user can create hierarchical and reusable designs through a library of cores. The JRoute API allows a user to perform run-time reconfiguration (RTR) of the routing resources by preserving the elements of RTR that are present in its underlying JBits[1] foundation. RTR systems are different from traditional design flows in that circuit customization and routing are performed at run-time. Since the placement of cores is one of the parameters that can be configured at run-time, the routing is not predefined. This means that auto routing can be very useful, especially when connecting ports from two different cores. Furthering the development of RTR computing designs, JRoute enables the implementation of nontrivial run-time parameterizable designs.

Since JRoute is an API, it allows users to build tools based on it. These can range from debugging tools to extensions that increase functionality. It is important to note that the JRoute API is independent of the algorithms used to implement it. The algorithms discussed in this paper are the initial implementations to further explain the API. This paper is meant to present features and benefits of the API, not the algorithms.

## 2 Overview of the Virtex Routing Architecture

The Virtex architecture has local, general purpose, and global routing resources. Local resources include direct connections between horizontally adjacent configurable logic blocks (CLBs) and feedback to inputs in the same logic block. Each provides high-speed connections bypassing the routing matrix, as seen in Figure 1. General-purpose routing resources include long lines, hex lines, and single lines. Each logic block connects to a general routing matrix (GRM). From the GRM, connections can be made to other GRMs along vertical and horizontal channels. There are 24 single length lines in each of the four directions. There are 96 hex length lines in each of the four directions that connect to a GRM six blocks away. Only 12 in each direction can be accessed by any given logic block. Some hexes are bi-directional, meaning they can be driven from either endpoint. There are also 12 long lines that run horizontal, or vertical for the length of the chip. Long lines are buffered, bi-directional lines that distribute the signals across the chip quickly. Long lines can be accessed every 6 blocks. Each type of general routing resource can only drive certain types of wires. Logic block outputs drive all length interconnects, longs can drive hexes only, hexes drive singles and other hexes, and singles drive logic block inputs, vertical long lines, and other singles. There are also global resources that distribute high-fanout signals with minimal skew. This includes four dedicated global nets with dedicated pins to distribute high-fanout clock signals. The array sizes for Virtex range from 16x24 CLBs to 64x96 CLBs. For a complete description of the Virtex architecture, see [3].

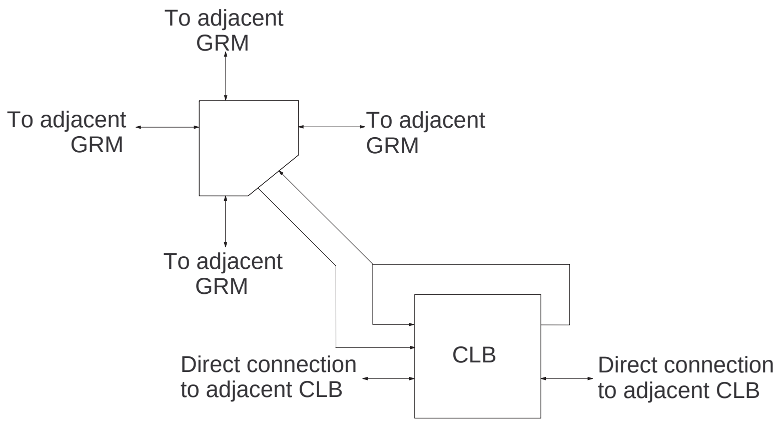
To adjacent GRM
To adjacent GRM
To adjacent GRM
To adjacent GRM
CLB
Direct connection to adjacent CLB
Direct connection to adjacent CLB

**Fig. 1.** Virtex routing architecture.

## 3 JRoute Features

The JRoute API makes routing easier to perform and helps in the development of large systems with reusable libraries. Unlike the standard Xilinx tools, JRoute can perform the routing at run-time. It also provides debugging facilities.

Before describing each of the calls, the architecture description file must first be described. There is a Java class in which all of the architecture information is held. In this class each wire is defined by a unique integer. Also in this class the possible template values are defined, along with which template value each wire can be classified under. A template value is defined as a value describing a direction and a resource type. For example, a template value of NORTH6 describes any hex wire in the north direction, a template value of NORTH1 describes any single wire in the north direction. Similar values are defined for each resource type in each direction that it can go. Also in this Java class is a description of each wire, including how long it is, its direction, which wires can drive it, and which wires it can drive.

### 3.1 Various Levels of Control

The JRoute API was designed with the goal of providing various levels of control. The calls range from turning on or off a single connection to auto-routing a bus connection.

`route (int row, int col, int from_wire, int to_wire):`
This call allows the user to make a single connection (i.e. the user decides the path). This can be useful in cases where there is a real time constraint on the amount of time spent configuring the device. However, the user must know what each wire connects to, and which wires are used. This call turns on the connection between from_wire and to_wire in CLB (row,col). The following example shows how to create a route connecting S1_YQ in CLB (5,7) to S0F3 in CLB (6,8) going through Out[1], SingleEast[5], and SingleNorth[0].

```
router.route(5, 7, S1_YQ, Out[1]);
router.route(5, 7, Out[1], SingleEast[5]);
router.route(5, 8, SingleWest[5], SingleNorth[0]);
router.route(6, 8, SingleSouth[0], S0F3);
```

`route (Path path):`
This call allows the user to define a path. A path is an array of specific resources, for example HexNorth[4], that are to be connected. The path also requires a starting location, defined by a row and column. The router turns on all of the connections defined in the path. The following example shows how to construct and route a path for the same route as in the previous example.

```
int[] p = {S1_YQ, Out[1], SingleEast[5], SingleNorth[0], S0F3};
Path path = new Path(5,7,p);
router.route(path);
```

`route (Pin start_pin, int end_wire, Template template):`
This call allows the user to specify a template and the router picks the wires. A template is defined as an array of template values, previously defined. The user does not have to know the wire connections and the resources in use. Using a template can also take advantage of regularity which would occur, for example, when connecting each output bit of an adder to an input of another core. The cost is longer execution time, and there is no guarantee that an unused path even exists. For this method a starting pin, defined as a wire at a specific row and column, needs to be defined. As well, the ending wire and the template to follow is specified. The router begins at the start wire, then goes through each wire that it drives, as defined in the architecture class, and checks first if the wire's template value matches the template value specified by the user. If so, then it checks to make sure the wire is not already in use. A recursive call is made with the new wire as the starting point and the first element of the template removed. The call would fail if there is no combination of resources that are available that follow the template. In this case a user action is required. The following example shows how to construct a template and route using it. The source and destination are the same as in the previous two examples. However, the specific resources may differ.

```
int[] t = {OUTMUX, EAST1, NORTH1, CLBIN};
Template template = new Template(t);
Pin src  = new Pin(5, 7, S1_YQ);
router.route(src, S0F3, template);
```

Finally, there are the auto-routing calls. This involves source to sink, source to many sinks, and a bus connection of many sources to an equal number of sinks.

`route (EndPoint source, EndPoint sink):`
This single source to single sink call allows for auto-routing of point to point connections. An EndPoint is either a Pin, defined by a row, column, and wire, or a Port, which is described in the next subsection. Many algorithms can be used to implement this call. One possibility is to use a maze router [4] [5]. Another possibility that would potentially be faster is to define a set of unique and predefined templates that would get from the source to the sink and try each one. If all of them fail then the router could fall back on a maze algorithm. The benefit of defining the template would be to reduce the search space. The following example shows how to define the end point (Pins) and connect them. The source and sink are the same as in the previous three examples, for the individual connections, path route, and template route. The template followed and the resources used may not necessarily be the same as it would with the other calls.

```
Pin src  = new Pin(5, 7, S1_YQ);
Pin sink = new Pin(6, 8, S0F3);
router.route(src, sink);
```

`route (EndPoint source, EndPoint[] sink):`
This is the method for a source to several sinks. It decides the best path for the entire collection of sinks. This call should be used instead of connecting each sink individually, since it minimizes the routing resources used. Each sink gets routed in order of increasing distance from the source. For each sink, the router attempts to reuse the previous paths as much as possible. Because it is not timing driven, this algorithm is suitable only for non-critical nets. For critical nets, however, the user would need to specify the routes at a lower level. In an RTR environment traditional routing algorithms require too much time. Currently long lines are not supported; only hexes and singles are used. Using long lines would improve the routing of nets with large bounding boxes.

`route (EndPoint[] source, EndPoint[] sink):`
This is a call for bus connections. In a data flow design, the outputs of one stage go to the inputs of the next stage. As a convenience, the user does not need to write a Java loop to connect each one. If used along with cores, this call can be very useful when connecting ports to other ports. For example, the output ports of a multiplier core could be connected to the input ports of an adder core. Using the bus method, the user would not need to connect each bit of the bus.

Each of the auto-routing calls described above use greedy routing algorithms. This was chosen because of the designs that are targeted. Structured and regular designs often have simple and regular routing. Also, in an RTR environment, global routing followed by detailed routing would not be efficient. Furthermore, RTR designs will be changing during the execution. This leads to an undefined definition of what global routing would mean.

### 3.2 Support for Cores

Another goal when designing the JRoute API was to support a hierarchical and reusable library of run-time parameterizable cores. Before JRoute the user of a core needed to know each pin (an input or output to a logic resource) that needs to be connected. With JRoute, a core can define ports. Ports are virtual pins that provide input or output points to the core. The core can use the ports in calls to the router, instead of specifying the specific pin. To the user there is no distinction between a physical pin, defined as location and wire, and a logical port as they are both derived from the EndPoint class. The core can define a connection from internal pins to ports. It can also specify connections from ports of internal cores to its own ports.

The router knows about ports and when one is encountered, it translates it to the corresponding list of pins. When a port gets routed, the source and sinks connected to the port are saved. This information is useful for the unrouter and the debugging features, which are described later.

There are routing guidelines that need to be followed when designing a core. First, each port needs to be in a group. For example, if there is an adder with an $n$ bit output, each bit is defined as a port and put into the same group.

The group can be of any size greater than zero. Second, the router needs to be called for each port defined. This call defines the connections to the port from pins internal to the core. Finally, a getPorts( ) method must be defined for each group, which returns the array of Ports associated with that group.

## 3.3 Unrouter

Run-time reconfiguration requires an unrouter. There may be situations when a route is no longer needed, or the net endpoints change. Unrouting the nets free up resources. A core may be replaced with the same type of core having different parameters. In this case the user can unroute the core then replace it. The port connections are removed, but are remembered. If the ports are reused, then they will be automatically connected to the new core. For example, consider a constant multiplier. The system connects it to the circuit and later requires a new constant. The core can be removed, unrouted, and replaced with a new constant multiplier without having to specify connections again. Core relocation is handled in a similar way.

`unroute (EndPoint source);`
An unrouter can work in either the forward or reverse direction. In the forward direction a source pin is specified. The unrouter then follows each of the wires the pin drives and turns it off. This continues until all of the sinks are found.

`reverseUnroute (EndPoint sink);`
In the reverse direction a sink pin is specified. The entire net, starting from the source, is not removed. Only the branch that leads to the specified pin is turned off, and freed up for reuse. The unrouter starts at the sink pin and works backwards, turning off wires along the way, until it comes to a point where a wire is driving multiple wires. It stops there because only the branch to the given sink is to be unrouted.

## 3.4 Avoiding Contention

`isOn (int row, int col, int wire);`
This call checks to see if the wire in CLB (row,col) is currently in use. The Virtex architecture has bi-directional routing resources. This means that the track can be driven at either end, leading to the possibility of contention. The router makes sure that this situation does not occur, and therefore protects the device. An exception is thrown in cases where the user tries to make connections that create contention. In the auto-routing calls, the router checks to see if a wire is already used, which avoids contention.

## 3.5 Debugging Features

`trace (EndPoint source);`
A JRoute call traces a source to all of its sinks. The entire net is returned for

the trace. Debugging tools, such as BoardScope [2], can use this to view each sink.

`reverseTrace (EndPoint sink);`
A sink is traced back to its source. Only the net that leads to the sink is returned.

## 4 JRoute versus Routing with JBits

JRoute uses the JBits low-level interface to Xilinx FPGA configuration bitstreams, which only provides manual routing. The JRoute API extensions provide automated routing support, while not prohibiting JBits calls. JRoute facilitates the use of run-time relocatable and parameterizable cores.

Using cores and the JRoute API, a user can create designs without knowledge of the routing architecture by using port to port connections. The user only really needs a small set of architecture-specific cores to start with. For example, a counter can be made from a constant adder with the output fed back to one input ports and the other input set to a value of one.

## 5 Portability

Currently, JRoute only supports Virtex devices. However, it can be extended to support future Xilinx architectures. The API would not need to change. However, the architecture description class would need to be created for the new architecture. The algorithms as presented in this paper have some architecture dependencies. For example, when routing a single source to a single sink, defining the set of predefined templates is architecture dependent. However, algorithms can be designed that have no architecture dependencies, and could be used with new architectures. These algorithms would use the architecture class to choose wires, check their lengths, and check the connectivity. The path-based router and template-based router have no knowledge of the architecture outside of what the architecture class provides.

## 6 Future Work

Virtex features such as IOBs and Block RAM will be supported in a future release of JRoute. Also, skew minimization will be addressed. The use of long lines to improve the routing of certain nets will be examined. Finally, different algorithms are being investigated such as [6].

## 7 Conclusions

JRoute is a powerful abstraction of the Xilinx FPGA routing resources. A routing API facilitates the design of object oriented circuits that are configurable at run-time. There are many options that are made available by JRoute such as connecting two points for which the location is determined dynamically.

Hierarchical core-based design using JRoute permits easier management of design complexity than using only JBits. JRoute automates much of the routing and reduces the need to understand the routing architecture of the device. JRoute also provides support for large designs by allowing cores to define ports. RTR features include the unrouter, which allows cores to be removed or replaced at run-time without having to reconfigure the entire design. Auto-routing calls allow connections to be specified, even if the placement is not known until run-time.

## Acknowledgements

Thanks to Cameron Patterson for his guidance and help in understanding routing algorithms. This work was supported by DARPA in the Adaptive Computing Systems (ACS) program under contract DABT63-99-3-0004.

## References

1. S. A. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA, November 1998.
2. D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems," *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA, November 1998.
3. Xilinx, Inc., *The Programmable Logic Data Book*, 1999.
4. Naveed A Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Norwell, Massachusetts, 1993.
5. Stephen D. Brown, Robert J. Francis, Jonathan Rose and Zvonko G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, Norwell, Massachusetts, 1992.
6. J. Swartz, V. Betz and J. Rose, "A Fast Routability-Driven Router for FPGAs," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 1998.