Lava and JBits: From HDL to Bitstream in Seconds

Satnam Singh and Phil James-Roxby
Xilinx Inc.
San Jose, California 95124, U.S.A.
{Satnam.Singh, Phil.James-Roxby}@xilinx.com

Abstract

The paper reports a design methodology that allows FPGA programming bitstreams to be generated in seconds starting from a very high level circuit description. High speed bitstream generation and manipulation is particularly important for reconfigurable computing systems that can not wait for the typical run times incurred by conventional flows. The preliminary version of this system can generate bitstreams from HDL source 12 times faster than the conventional flow and future work may offer significantly larger speed improvements.

1 Introduction

This paper describes how two separate technologies have been combined to produce a system suitable for producing dynamically reconfigurable computing systems. The Lava [2] hardware description language allows circuits to be described using powerful features like higher order functions and polymorphism and permits layout to be effectively captured at the HDL level using *layout combinators* [7][8][19]. However, Lava does not provide any feature for performing routing. The JBits system contains a high speed router that can take a totally placed design and generate a Xilinx FPGA programming bitstream. The JBits system has been extended to accepted the output of the totally placed circuits generated by Lava and turn them into programming bitstreams. This results in a complete flow from high level description all the way to programming bitstream in seconds.

The high performance of this system allows run-time reconfiguration to be captured at the language level e.g. dynamic specialisation (constant propagation at the circuit level) by partial evaluation (in Lava) [14]. This provides much needed support for automating certain types of dynamic reconfiguration required by custom computing machines.

The ability to describe reconfigurable systems in a high level without expending a huge amount of effort on tedious layout greatly facilitates the development of custom computing machines. As specialised tools and techniques like those presented in the paper mature we expect the process of designing for custom computing machines to become less painstakingly slow.

The paper motivates the need for the a high performance HDL-based flow for reconfiguration. Then the Lava HDL is used to illustrate how total layout can be expressed in a high level manner resulting in a fully placed EDIF netlist. The next stage describes the JBits system with a Lava/EDIF interface which completes the routing and generates the bitstream. An example of a design that fills a Xilinx Virtex™ XCV300 chip is subjected to both the EDIF/JBits flow and the conventional flow and the relative performance is evaluated.

2 Motivation

Xilinx already offers a mainstream back-end flow that accepts EDIF as input and produces a configuration datastream as output. Over the years, this flow has become increasingly sophisticated, producing ever smaller and faster circuits. It is worth pointing out that this work is in no way intended to replace this back-end flow: there is simply no way to compete. However, there are a number of areas lacking in the mainstream flow, which are important for the development of custom computing machines.

The main area lacking is support for partial configuration. Currently, there is no way to produce the partial configuration frames for the small changes required to change circuit *A* into circuit *B* using the mainstream tools. The mainstream tools are designed and optimised to produce single static circuits.

The second area lacking is the ability to very quickly produce configurations given an EDIF netlist. Though the mainstream tools have become increasingly faster, there is still considerable overhead in producing the configurations. If the dream of moving from a HDL to configuration in seconds is to be achieved, it is clearly not going to involve a conventional flow.

In order to make the construction of a replacement back-end flow feasible, we restrict the type of EDIF accepted by the flow. The EDIF must be fully mapped, that is can only consist of the primitives encountered in the Virtex CLB, which is shown in Figure 1. This means that EDIF primitives such as LUT4, FD and MUXCY are permitted, whilst AND4 and the like will be rejected. The second restriction is that the EDIF must be fully placed. Each primitive must have an associated RLOC property that specifies its position.

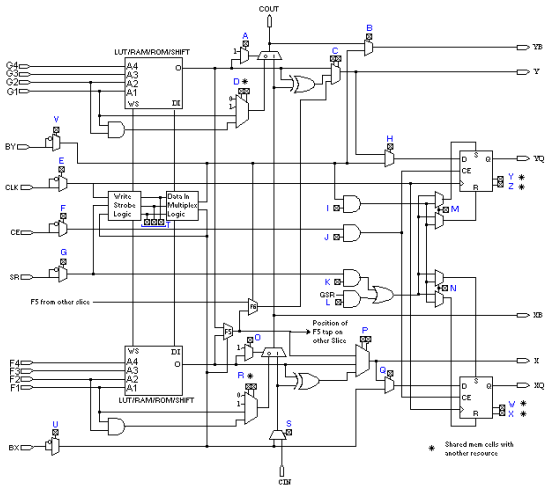


Figure 1 The Virtex CLB

An example of the type of EDIF accepted by the tools is shown in shown below:

```
(instance lut2_21330
   (viewRef prim
      (cellRef lut2 (libraryRef lava_virtex_lib)))
      (property INIT (string "6"))
      (property RLOC (string "R-24C24.S1")))
(instance xorcy_21331
   (viewRef prim
      (cellRef xorcy (libraryRef lava_virtex_lib)))
      (property RLOC (string "R-24C24.S1")))
(instance muxcy_21332
   (viewRef prim
      (cellRef muxcy (libraryRef lava_virtex_lib)))
      (property RLOC (string "R-24C24.S1"))
```

EDIF represents the hand-over point between the front-end and the back-end tools. As discussed earlier, the intention is to produce a run-time system, where Lava would initially produce the EDIF, and then the new back-end flow would produce the partial configurations required to implement this EDIF, offering a high-level language interface to run-time reconfiguration. As such, the back-end flow must produce the configurations much quicker than the mainstream tools. In the second stage of this project Lava will be modified to directly call the JBits interface.

Previous back-end replacement tools include the Trianus system that targeted the XC6200 series [9], and acted as a replacement for the XACT Step 6000 back-end flow [16]. Trianus offered a total flow from specification in the HDL Lola through to an XC6200 series configuration datastream. Trianus was assisted in part by the open architecture of the XC6200 series devices. To the best of our knowledge, no other back end replacement flow has been designed for the proprietary Virtex format.

3 Circuit Layout in Lava

This section provides a brief overview of the features available in the Lava HDL for describing and totally laying out circuits which can then be used by the JBits system for bitstream generation. Lava is embedded in the functional programming language Haskell and all the Lava code presented is legal Haskell code.

3.1 Lava Co-ordinates

Lava provides a set of primitive components that can be instanced and placed on a Cartesian grid which has the origin at the bottom left hand corner. When an EDIF netlist is produced Lava converts the Cartesian LUT-based co-ordinates into the form expected by the Xilinx design tools (row and column based on slices with the origin at the top right and with multiple LUTs in a slice depending on the architecture).

The general version of Lava allows circuits to be described with blocks which contain laid out components but which do not have any relationship to each other. This is supported by using the HUSET attributes to identify distinct coordinate spaces. The JBits back-end for Lava insists that all circuit elements are in the same HUSET i.e. every cell is placed relative to every other cell. Without this restriction the JBits system would not have a fully placed design to work on.

3.2 Primitive Components

Lava provides a commonly used set of primitive components. Most of these components have a Lava dimension of (1,1) i.e. one unit wide and one unit high. Depending on the architecture several components can reside at the same location e.g. a LUT (look-up table), XORCY (a special XOR gate used to implement the result of an addition quickly) and MUXCY (a special high speed carry propagation multiplexor) can all reside at the same location if they are appropriately wired together (see Figure 1).

Some languages require every possible type of combinational gate function to be uniquely named (with an entity and component declaration in the case of VHDL). Lava provides a more flexible way for specifying the behaviour of 1, 2, 3 and 4 input and one output combinational functions (i.e. those that can be implemented in a single LUT). Since functions are first class objects in Lava, the lut2 circuit is really a circuit *constructor* (or higher order function) which takes a function of two inputs and one output at the Lava/Haskell level and returns a circuit of two inputs and one output which implements that function.

For example, assuming that we have defined the following boolean function in Lava that computes the exclusive OR function:

```
exor :: Bool -> Bool -> Bool
exor a b = a /= b
```

we can then define the Lava gate the performs the exor function by passing this function to the lut2 function:

```
xor2 :: (Bit, Bit) -> Bit
xor2 = lut2 exor
```

This is how all the combinational functions in Lava are implemented. The Lava system exhaustively evaluates the function which is passed as a parameter to compute its truth table which is then transcribed into a 2-input LUT with an appropriate initialisation attribute. The lut2 combinator has a higher order type as show below i.e. (Bool -> Bool -> Bool) represents the type of a *function* or *circuit*.

```
lut2 :: (Bool -> Bool -> Bool) ->
        (Bit, Bit) -> Bit
```

Note that lut2 takes a *curried* boolean function (which allows partial application) but returns a circuit that requires a tupled input (to facilitate composition). A curried function is a function that is only supplied some of its arguments resulting in a residual function of fewer parameters. For example, for the function add x y = x + y, of type Int -> Int -> Int, the curried function add 72 has type Int -> Int and is a function that takes one argument and returns a result which is 72 plus this argument.

The lut1, lut3 and lut4 combinators work in a similar manner. It is even possible to directly write in a function as the first argument to lut2 using lambda expression which represent functions (or circuits) as *expressions*:

```
or2 :: (Bit, Bit) -> Bit
or2 = lut2 (\a b -> a || b)
```

defines a two-input OR-gate using a lambda expression of two bound variables and the built in Haskell or-operator ||. Here are some other circuits:

```
inv :: Bit -> Bit
inv i = lut1 not

and2 :: (Bit, Bit) -> Bit
and2 = lut2 (&&)

mux :: (Bit, (Bit, Bit)) -> Bit
mux = lut3 (\s a b -> if s then b else a)
```

This feature allows users to describe any 1 to 4 input and one output combinational function using the full expressive power of the language. The Lava system also provides combinators for realising larger combinational functions by composing LUTs together and connecting them with fast resources like the F5/F6 multiplexers available in Virtex.

3.3 Serial Composition

In the JBits variant of Lava one can not simply instance several primitive gates and wire up their ports and get an implementation (like in VHDL). Doing this does not say anything about how to lay the circuits out. One could attach attributes to each primitive element specifying its Cartesian co-ordinate. However, except for the simplest circuits, this is a very tricky and error prone task. For circuits with recursive structure and layout it is almost impossible.

Instead of separating the concerns of composing behaviour (i.e. connecting ports together) and layout (calculating Cartesian co-ordinates) Lava provides circuit combinators that compose behaviour *and* layout. This design decision has allowed us to capture many kind of complex layout patterns which in turn allows us to totally lay out circuits for the JBits phase.

As a specific example of a layout combinator consider left to right serial composition, written as the infix operator >-> whose type is shown below:

```
(>->) :: (a->b) -> (b->c) -> (a->c)
(>->) f g x = <definition omitted>
```

This combinator takes a circuit f, supplies it with its input x and then connects its output to the input of circuit g. It also records the layout tile needed for circuit f and circuit g and to ensure that circuit f is laid out to the left of circuit g. The result of circuit g is returned as the result of the composite serial composition.

As a concrete example, consider the definition of a NAND gate shown below:

```
nandGate :: (Bit, Bit) -> Bit
nandGate = and2 >-> inv
```

The serial composition combinator causes the output of the AND-gate to be connected to the input of the invertor: note the left to right information flow. Figure 2 illustrates the basic tiles and how they are connected and laid out by the serial composition combinator which causes its first circuit to be laid out to the left of the second circuit. If this circuit were implemented on a Virtex part the AND-gate would be realised in one slice and the invertor in the slice immediately adjacent on the right.

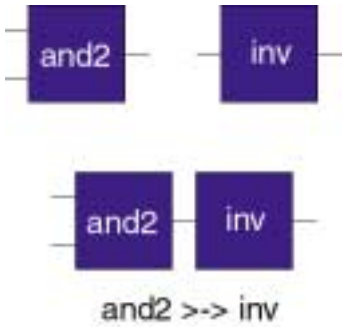


Figure 2 Serial Composition

3.4 Parallel Composition

Another very useful combinator is parallel composition which allows us to compose two circuits that do not communicate so that they are laid out vertically. The type of the parallel composition combinator is shown below:

```
par :: [(a->b)] -> [a] -> [b]
par circuits inputs = ...
```

This type says that the par combinator takes a list of circuits and another list of inputs and applies to inputs respectively to the circuits. Furthermore, the circuits are laid out vertically. An example of parallel composition is shown in Figure 3

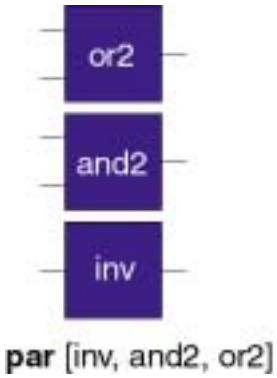


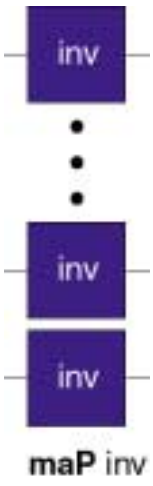
Figure 3 Parallel composition combinator

3.5 Map

Yet another useful combinator is map which replicates a circuit vertically across an input bus:

```
maP :: (a -> b) -> [a] -> [b]
```

and this is illustrated in Figure 4. This combinator is spelt with a capital 'P' at the end to avoid a name clash with the Haskell prelude function 'map'.



3.6 Four Sided Tiles

The tiles presented so far have horizontal information flow and interfaces only on the left and right hand sides. Lava also supports four sided tiles which allow information to flow horizontally and vertically. Four sided circuits are understood to be pair to pair circuits where each element of each pair describes the signal on a particular face of the tile. For example, the **below** combinator takes one tile and places it below another tile, connecting them on their common face, as shown in Figure 4. There is also a **beside** combinator.

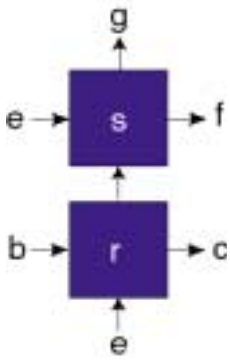


Figure 4 The below combinator

The type of the **below** combinator is:

```
below :: (((a, b) -> (c, d)) ->      -- circuit r
         ((d, e) -> f, g))   ->      -- circuit s
         ((a, (b,e)) ->((c,f), g))) -- result
```

3.7 Rows and Columns

This below combinator composes just two tiles that agree on the type on their interface. Another powerful combinator is col, which will compose any number of tiles. The source for this combinator is shown below and an illustrative diagram is shown in Figure 5. There is also a **row** combinator.

```
col ::  Int ->                -- Number of tiles in column
        ((a, b) -> (c, a)) -> -- circuit to replicate
        (a, [b]) ->           -- Type of input for col
        ([c], a)              -- Type of output for col
```

4 An Example: A Vertical Adder

To illustrate the idiomatic use of some of the combinators presented in the previous section we present the definition of a ripple carry adder. First, we present the definition of the basic full adder circuit:

```
fullAdder :: (Bit, (Bit, Bit)) -> (Bit, Bit)
fullAdder (cin, (a,b))
  =   (sum, cout)
```

```
where
part_sum = xor2 (a, b)
sum = xorcy (part_sum, cin)
cout = muxcy (part_sum, (a, cin))
```

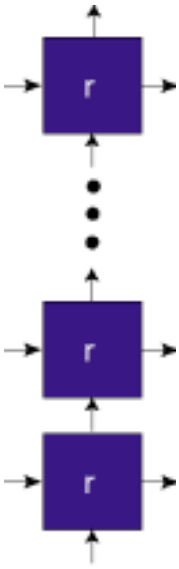


Figure 5 A column of 4-sided circuits

This description simply instances three primitive elements that can jointly reside in one element of the FPGA. The xor2 gate performs the exclusive-OR function, the muxcy circuit computes the carry, and the xorcy gate is a special exclusive-OR gate for computing sums from a partial sum and a carry.

The shape of the tuples used in this circuit description is important. The first element of the input tuple identifies the input signal entering the bottom of the tile (in this case the carry in). The second element of the input tuple identifies the two bits to be added (which come in through the left hand side of the tile). The first element of the result identifies the output signal at appears on the right hand side of the tile (in this case the sum) and the second element identifies the output signal at the top of the tile (the carry output). This is illustrated in Figure 6.

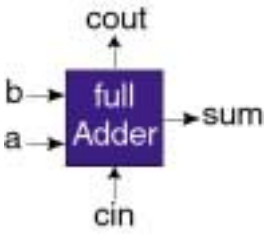


Figure 6 A Full Adder ready for use in a column

To make a ripple carry adder all we have to do is to compose several of these full adders into a column of full adders:

```
adder :: Int -> ([bit], [bit]) -> ([bit], bit)
adder size (a, b) = col size fullAdder (zero, zip a b)
```

This description takes a pair of bit-vectors, zips them together so corresponding bits are paired, and then supplies this to a column of full adders (using zero as the carry input) and returns a pair which contains the sum and the carry output. An adder of a specific size is easily specified:

```
adder4 = adder 4
```

and the resulting circuit is shown in Figure 7 which also shows the actual FPGA layout of a 16-bit adder produced from Lava.

It is useful to have an adder that does not take a carry input and which adds the carry output to the sum to accommodate bit growth. It is also useful to have an adder whose size is determined by the length of the input bit-vectors and there should be no requirement on the input bit-vectors to be the same length. We call such an adder verticalAdder and we also require that this adder to throw away the carry to produce an output the same size as the inputs.

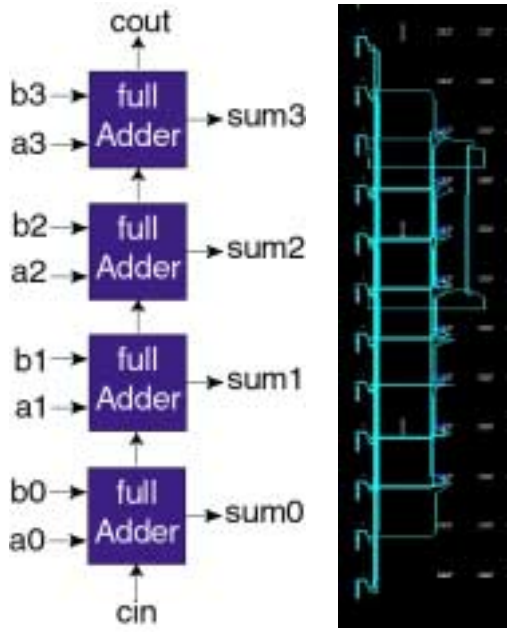


Figure 7 A 4-bit Adder and the actual layout of a 16-bit adder

5 Adder Trees

The experimental circuit that we shall use for analysing the performance of the Lava/JBits system is an pipelined adder tree laid out horizontally with no bit-growth which results in a rectangular footprint.

We assume the availability of a layout combinator called middle which composes and lays out three circuits as shown in Figure 8.

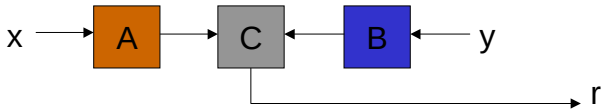


Figure 8 The **middle** combinator

Using this combinator we can define a recursive tree combinator which makes a tree of 2-input 1-output components laid out as shown in Figure 9. The definition of **tree** is:

```
tree :: ((a,a) -> a) -> [a] -> a
tree circuit [x] = x
tree circuit [x, y] = circuit (x, y)
tree circuit input
  = (middle (tree circuit) circuit (tree circuit)) (halveList input)
```

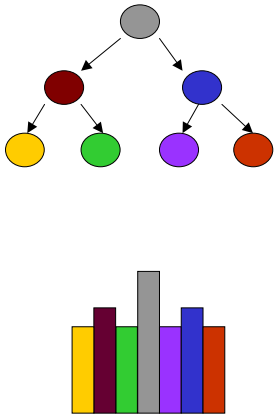


Figure 9 A recursive binary tree layout

and using this combinator we can make an adder tree:

```
adderTree = tree verticalAdder
```

Four instances of a tree that adds 96 inputs have been placed on top of each other to totally pack a XCV300 Virtex FPGA. This design uses every slice of the chip and every component has a layout specified. The circuit that we actually use is a slight variant of this version because we register each result to produce a pipelined adder tree. This is the test circuit used to evaluate the JBits flow.

6 JBits

JBits was first presented under the name XBI in [10]. JBits is derived from earlier work, which targeted the XC6200™ series called the Java Environment for Reconfigurable Computing or JERC [17]. JERC offered a programming API that allowed manipulation of the low-level configuration bits through a symbolic interface. Each programmable resource on the XC6200 was assigned a human-friendly name, and interaction was allowed in both directions: user programs could both read and write through these human-friendly names. JBits is a similar configuration API, which currently targets all members of the Virtex family, from the XCV50 to the XCV1000. Again, users are given human-friendly names for each programmable resource on the Virtex device, and are given full, unrestricted read and write access to all resources.

JBits began as basically a configuration opener, opening up a proprietary data format. However, under a DARPA-funded project, JBits has added much more functionality, making it a truly useful tool for more than just low-level PIP manipulation. The current functionality is shown in Figure 10.

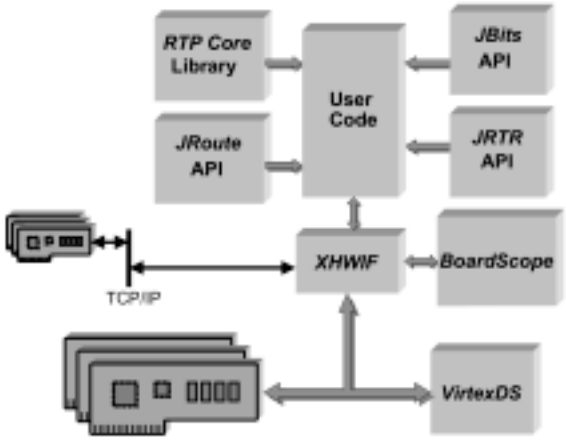


Figure 10 The JBits flow

The BoardScope tool, which is similar to the old WebScope tool for the XC6200 series, now targets Virtex. A similar tool, ChipScope is now shipped as part of the mainstream Xilinx tool distribution. BoardScope offers a variety of ways of viewing the circuit, including a power view, showing areas consuming the most power on the device. BoardScope operates in a variety of ways, either using local hardware, hardware that is connected via a network, or more recently, by using the DeviceSimulator application [11]. DeviceSimulator performs a direct simulation of the hardware based on the configuration settings, and can be used when no hardware is available. In certain cases, the DeviceSimulator will actually be faster than using hardware directly, since there is no need to perform a readback operation in the simulator.

Since Virtex routing is so much more complex than the XC6200 series, possibly the most important difference

between JERC and JBits is the JRoute API [13]. JRoute allows a designer to specify a route in terms of a source and any number of sinks. These are both defined as symbolic pins on the device, so for example a route could be requested between the X output of slice 0 at row 23 column 50 to the F1 input of slice 1 at row 14 column 22. JRoute will then determine the PIP settings required to make this route, and will either set the PIPs to actually make the route (run-time routing) or print out the PIP settings, allowing the designer to cut and paste this code into their main application (design-time routing). Routing accounts for the vast majority of programmable resources on the device: making these routes manually would be tedious and be prone to errors.

Support for core generation has now been added to JBits through the CoreTemplate. This allows designers to construct circuits by generating a hierarchy finishing at low-level primitives, implemented by manipulations to the configuration directly. EDIF and XDL can also be output by cores adhering to the CoreTemplate. Finally, the JRTR [12] API is an extension of the JBits API to take advantage of the partial reconfiguration support provided by Virtex devices. This interface provides a caching model that automatically tracks changes to configuration data. As a result, only dirty frames requiring reconfiguration are amalgamated into a partial configuration packet. This is all transparent to the user.

The old view of JBits performing low-level manipulations listed in a user's application is becoming increasing rare, outside of the CLB at least. However, the fact that 'PIP-poking' is still permitted opens up whole areas of research from evolvable hardware, to the application under consideration in this paper, the construction of an entire back-end implementation flow

7 Parsing the input

In order to get the EDIF into a form that is more easily manipulated, it is necessary to parse the input file, and produce a series of symbol tables, storing the required instances for example. The EDIF was parsed using the Antlr translator generator [15], which constructs a full parser given a grammar. Using this tool, it is simply a matter of hanging actions off tokens, so for example when instances are encountered by the parser, the relevant information such as the instance name and type can be stored. This parsing is by far the most time-consuming part of the replacement back-end flow.

8 Mapping

Mapping normally covers the mapping of arbitrary logic components such as AND gates onto the architecture specific components such as LUTs and the carry chain logic. Since the EDIF accepted by the back-end replacement is fully mapped EDIF, there is no real mapping to do. However, one task that remains is to pack the logic into one specific area of the CLB. The placement constraints that are applied only define which CLB to pack the logic into, they don't actually specify which of the two LUTs to use for example. Referring back to Figure 1, a logical LUT can be mapped either to the F or the G physical LUT. Normally, it doesn't matter which of these physical LUTs is used, if the logical LUT is simply being used as memory or for logic implementation. Asymmetry arises when the LUTs must interact with either the carry logic, or the logic expansion multiplexor F5. In the case of F5, there is an optional inversion on the select line, so theoretically, logic could be placed in either LUT. In the case of the carry chain however, the order of the LUTs is important, since the carry out produced as a result of the addition in LUT F is rippled upwards to contribute to the carry out produced as a result of the addition in LUT G. Therefore, the order of packing is important.

This is covered in the back-end flow by examining the placement constraints on elements in the carry chain. When a MUXCY component is found which feeds a MUXCY in a different CLB, then it is clear that the first MUXCY must be mapped to the area around the G LUT. This then means that all logic that interacts with this MUXCY component such as a LUT or an XORCY can also be mapped into the area around the G LUT. This may then in turn constrain the register.

At the end of the map stage, a data structure is produced which contains a fully mapped version of the Virtex device, with all primitives assigned to a particular CLB locations.

9 Placement

Placement normally deals with placing mapped logic to specific CLBs on the device, in order to minimize routing congestion. Since all components are already placed in the new flow, placement is simplified to two tasks. Firstly, it is necessary to convert the relative placement constraints to real locations on the device. This stage also deals with the mismatch between RLOC coordinates, where (0,0) is the top left hand corner, and JBits coordinates where (0,0) is the bottom left hand corner (the same as Lava). Since all RLOCs are relative to an origin, it is necessary to place this origin in a position such that all the logic will fit on the device.

The second task is to actually instantiate the components. In reality, in an FPGA, the components all exist already in silicon. So instantiation is really more a matter of parameterising the fabric of the FPGA. For example, no action is needed to instantiate an XORCY: they are really there. In the case of a LUT, some paramterisation is required, in order to tell the fabric that the LUT is really to be used as a LUT, not a RAM or a shift register, and also to define the initialisation value of the LUT. This is a very simple task.

10 Routing

Routing normally deals with determining the interconnection of primitives on the device. In the case of an FPGA, this will define the settings of the various PIPs on the device making up the nets. Normally this is the most time consuming part of the implementation flow.

Rather like placement, often nets defined in the EDIF do not actually need routing, since they already exist on the device. For example, the net connecting the two MUXCY components in the same CLB is always there w

her it is used or not. Similarly, some nets simply reduce to the setting of internal multiplexors, and do not require an explicit route to be determined. For example, in order to connect the output of the F LUT to the D input of the FD, the multiplexors marked by P and Q in figure 1 must be set accordingly.

Some nets however do require the routing structure of the FPGA to be used, and this is achieved using the JRoute API. When end-points are encountered which are in different CLBs, a call to JRoute is needed, and this is achieved by finding the pin at the source of the net (an example would be the XQ output of slice 0 at a certain location) and then finding all the pins at the destination (an example would be the F1 input of slice 1 at a certain location). JRoute is then called, and will set the programmable resources required to implement that specific net. If JRoute cannot find a route, this will be reported back to the user.

11 Bitstream Generation

In the mainstream back-end flow, the output is produced by the bitgen utility. bitgen does not produce partial configurations (which is one of the major reasons why this replacement flow is required). In the replacement flow, the PartialReconfiguration API of JBits is used, to construct a partial configuration packet. The PartialReconfiguration API is also very fast, as unlike bitgen, it does not perform any DRCs.

12 An Example

This section describes an example circuit which is fully specified and laid out in Lava and then JBits is used to perform the routing and bitstream generation. This illustrates a rapid flow from a HDL description all the way to a bitstream file.

12.1 The Example Circuit

The example circuit that we use is chosen to occupy just about every cell of a XCV300 FPGA which can realise circuits of up to 300,300 gates. By packing the design so tightly into the FPGA we are able to stretch the capability of the routes in both JBits and in the Xilinx tools.

The example test circuit we chose was a stack of four adder trees as shown earlier. Each adder tree is laid out horizontally with enough adders for the width of the device (96). Each adder tree is a quarter of the height of the device and we use four of them to fully occupy the FPGA. It is important to note that although this is a regular design with much repetition, our system in no way is dependent on regularity. The Lava/JBits flow works just as well for heterogeneous circuits.

After mapping with the Xilinx tools the adder tree design occupies 3,040 slices (98% utilisation) and uses 6,080 flip-flops (98% utilisation) and 4,032 LUTs (65% utilisation).

As a result, 329,086 lines of a 9.5MB EDIF are produced consisting of 24,402 instances and 24,338 nets.

12.2 Instantiating components

As mentioned previously, instantiating components is more a matter of parameterising the instances which exist on the actual FPGA. To instantiate a LUT component in slice 0 at position (23,10) in the F LUT, the following JBits calls will be made:

```
jBits.set(23,10,RAM.DUAL_MODE[0],RAM.OFF[0]);
jBits.set(23,10,RAM.RAM_32_X_1[0],RAM.OFF[0]);
jBits.set(23,10,RAM.LUT_MODE[0],RAM.ON[0]);
jBits.set(23,10,RAM.F_LUT_RAM[0],RAM.OFF[0]);
jBits.set(23,10,RAM.F_LUT_SHIFTER[0],RAM.OFF[0]);
jBits.set(23,10,LUT.F[0],LUT4init);
```

To clarify, constants such as RAM.DUAL_MODE[0] refer to the programmable resource which defines whether the LUTs are to be used as dual-ported RAM. Constants such as RAM.OFF[0] mean that this option is not selected.

12.3 Making the link between internal components

An example of a net linking internal components is shown below:

```
(net (net lava_bit7338 (joined
  (portRef o (instanceRef lut2_7338))
  (portRef s (instanceRef muxcy_7340))
  (portRef li (instanceRef xorcy_7339))))
```

Referring back to Figure 1, it can be seen the output of the LUT already feeds the li input of the XORCY component, and hence no further configuration is needed. In the case of the s input of the MUXCY, the multiplexor shown marked by O in Figure 1 must be configured so a link is made between the F LUT and the select input of the mux. This is done by the following JBits code fragment.

```
jBits.set(23,10,XCarrySelect.XCarrySelect[0],XCar-
rySelect.LUT_CONTROL[0]);
```

12.4 Making the link between external components

In order to make the link between the external components, the JRoute API is used. As described previously, JRoute requires pins to be defined for the source and sinks of the nets, which comprise of the coordinates of the pin and a constant defining which pin for example S0_YQ refers to the output of

the slice 0 Y flip-flop. Once the design has been fully placed, this is a simple look-up operation.

An example of a net requiring a connection through the routing matrix is shown below together with the JBits calls made by JRoute to implement this connection.

```
(net lava_bit2250 (joined
  (portRef q (instanceRef fd_2250))
  (portRef i1 (instanceRef lut2_2282))
 )
)
```

```
jbits.set(36, 18, S1G2.S1G2, S1G2.SINGLE_SOUTH22);
jbits.set(36, 18, OutMuxToSin-
gle.OUT7_TO_SINGLE_SOUTH22, OutMuxToSingle.ON);
jbits.set(36, 18, OUT7.OUT7, OUT7.S0_XQ);
```

In certain cases, a combination of both external and internal routing is required. An example is a net that goes into the ci input of the MUXCY component and the F1 input of the F LUT. This requires the net sinking on the F1 pin be steered through the multiplexor marked with R inFigure 1, after the external connection has been made between the source and the F1 pin.

13 Performance

The back-end flow has been profiled on a 600 MHz PIII NT4SP4 system, 1Gb physical memory, using the 329,086 line EDIF circuit described previously. The target device is an XCV300-BG432. The results are shown in Table 1.

Table 1

Activity	Time
Parse EDIF, construct symbol tables and make JBits and JRoute objects	62.8s
Make JBits JRoute objects	7.23 s
Map	4.6 s
Route and implement instances	15.4 s
Generate a partial configuration as a byte array	94 ms
Generate a partial configuration as a file	47 ms
Total	90.25 s

The size of the partial configuration frame produced is 218,980 bytes. This is to be expected, as the design fills the entire device. The difference can be explained by the slightly different sequence of configuration instructions issued by the PartialConfiguration API and `bitgen`. Using version 3.1.02i of the Xilinx design tools this same fully placed netlist took 1080 second to transform from EDIF input to bitfile generation. For this example the JBits part of our flow performs 12 times faster than the conventional flow.

Most of the time is spent parsing the input file. In the next stage of the project we will change the back-end of Lava to directly call the JBits API rather than generate an intermediate file. This will greatly speed up both the Lava netlist elaboration and the JBits phase resulting in roughly a 50 times speed-up.

In order to verify the correctness of the circuit, currently this is done by a manual inspection of the resources being set. A number of more robust opportunities exist for testing. Firstly, the device simulator could be used, though currently it is impossible to stimulate inputs. Secondly, a prototype EDIFout flow is provided as part of JBits. EDIFout will produce an EDIF netlist given a configuration bitstream, allowing the bitstream to be effectively simulated in a mainstream simulator. This would allow the circuit constructed to be thoroughly exercised by simulation.

Perhaps the most interesting and powerful verification route lies in the formal verification described previously [18]. Under this scheme, two EDIF netlists can be compared, and the differences determined by performing a formal verification. Essentially, the netlists are reduced to a series of logical propositions. These propositions can then be tested for logical equivalence by a proposition prover. An EDIF netlist could be pushed through the back-end flow, producing a configuration bitstream, which is then converted back to EDIF using the EDIFout utility. If the two netlists could be formally verified to be equivalent, this would prove correctness of both the back-end flow and the EDIFout utility.

14 Limitations and Further Work

The treatment of external inputs and outputs is fairly naïve implementation. Each input and output signal is assigned its own IOB and nets that use these as either sources or sinks are routed to the appropriate IOB. The IOBs are assigned in the order they are encountered in the original EDIF. In practice, either locked pin locations should be specified in the EDIF, or the pin locations specified in a side file such as the User Constraints File (.ucf) used in the mainstream Xilinx flow.

Currently the back-end flow does not process Block-RAMs or the different standards supported by the IOBs. Support for both these features has recently been added to JBits, and work is currently underway to provide support for both features in the back-end flow.

Once the Lava back-end can call the JBits interface directly we expect to increase the performance of this system from 12x to 50x faster than the conventional flow.

Should the programming bitstream format be further modified to facilitate finer grain reconfiguration then this flow provides the capability to very quickly reconfigure filters (by modifying data in memories used to perform distributed serial arithmetic).

The development of the EDIF interface to the JBits system allows front ends other than Lava to be used. Many designers will be more familiar with systems JHDL (a Java based structural HDL) [1] or exotic languages like Pebble [6] or the use of VHDL [5] containing layout attributes either through generic INITs or by attributes Several synthesis flows support this form of entry for VHDL including Synplicity's Synplify system, Synopsys Design Compiler, Leonardo Spectrum and Xilinx's XST. However every cell has to be placed at the HDL level before the synthesised EDIF netlist can be submitted to the JBits system.

For many applications a bitstream design rule check phase will be required to ensure the integrity of the implemented netlist. This process could also be used to estimate timing information.

15 Summary and Conclusions

In summary we have reported preliminary work describing the integration of two experimental research systems to produce a complete flow from HDL to bitstream which performs quickly enough to service many kinds of custom computing applications. This is made possible by (i) using an HDL which has a very high speed layout engine that performs rapid elaboration into EDIF and (ii) a system that can map a fully placed EDIF netlist into a bitstream.

Clearly this flow is not a general one since it is not always convenient or possible to fully lay out a circuit. However we conclude that there are many circumstances where it is useful e.g. in virtual hardware systems where sub-components are swapped into and out of the device from specific locations or when an FPGA has to be fully reconfigured in sequence. In such circumstances an HDL to bitstream system that operates 50 times faster than a conventional flow can make the difference between viable run-time reconfiguration or an unfeasibly slow operation.

16 Acknowledgements

The authors would like to thank Athanas for providing the Antlr grammar used for the EDIF parser. The authors would also like to thank Geraint Jones, Mary Sheeran and Koen Claessen for contributions made to the Ruby and Lava systems which make it possible to describe circuit layout conveniently in a high level language. "Virtex" and "XC6200" are trademarks of Xilinx Inc

References

[1] Peter Bellows and Brad Hutchings. *JHDL - An HDL for Reconfigurable Systems*. FCCM'98. IEEE Computer Society. 1998.

[2] Per Bjesse, Koen Claessen, Mary Sheeran and Satnam Singh. *Lava: Hardware Design in Haskell*. International Conference on Lisp and Functional Programming 98. Springer-Verlag. 1998.

[3] M. Chu, N. Weaver, K. Sulimma, A. DeHon and J. Wawrzynek. *Object Oriented Circuit-Generators in Java*. IEEE Computer Society. 1998

[4] Brad Hutchings, Peter Bellows, Joseph Hawkings, Scott Hemmert, Brent Nelson, Mike Rytting. *A CAD Suite for High-Performance FPGA Design*. FCCM'99. IEEE Computer Society. 1999.

[5] IEEE Std. 1076-1987. *IEEE Standard VHDL Reference Manual*. 1987.

[6] W. Luk and S. McKeever. *Pebble: a language for parameterised and reconfigurable hardware design. Field-Programmable Logic and Applications*. LNCS 1482. Springer-Verlag. 1998.

[7] Satnam Singh. *Architectural Descriptions for FPGA Circuits*. FCCM'95. IEEE Computer Society. 1995.

[8] M. Sheeran, G. Jones. *Circuit Design in Ruby*. Formal Methods for VLSI Design, J. Stanstrup, North Holland, 1992.

[9] Xilinx. *XC6200 FPGA Family Data Sheet*. Xilinx Inc. 1995.

[10] S. 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. SPIE 1998.

[11] S. McMillan, B. Blodget, and S. Guccione. *VirtexDS: A Virtex Device Simulator*. SPIE 2000.

[12] S. McMillan and S. Guccione. *Partial Run-Time Reconfiguration Using JRTR*. Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, LNCS 1896, 2000.

[13] E. Keller. *JRoute: A Run-Time Routing API for FPGA Hardware*. 7th Reconfigurable Architectures Workshop, Lecture Notes in Computer Science 1800, pp 874-881, Cancun, Mexico, May, 2000.

[14] Nicholas McKay and Satnam Singh. *Dynamic Specialisation of XC6200 FPGAs by Partial Evaluation.* Field-Programmable Logic and Applications. Tallinn, Estonia. Springer-Verlag. 1998.

[15] Schaps, G.L. *Compiler Construction with ANTLR and Java*. Dr. Dobb's. Journal, March 1999

[16] R. Woods, S. Ludwig, J. Heron, D. Trainor, and S. Gehring. F*PGA synthesis on the XC6200 using IRIS and Hades*. FCCM'97. IEEE Computer Society. 1997.

[17] Eric Lechner and Steven A. Guccione. T*he Java environment for reconfigurable computing.* In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, Springer-Verlag, Berlin, September 1997. FPL 1997. LNCS 1304.

[18] Satnam Singh and Carl Johan Block. *Formal Verification of Reconfigurable Cores*. FCCM'99. IEEE Computer Society. 1999.

[19] Satnam Singh. *Death of the RLOC?* FCCM'00. IEEE Computer Society. 2000.