Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configurations

Philip James-Roxby
School of Electronic and Electrical Engineering
University of Birmingham
Edgbaston, Birmingham, B15 2TT
United Kingdom
P.B.James-Roxby@bham.ac.uk

Steven A. Guccione
Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124
United States
Steven.Guccione@xilinx.com

Abstract

As FPGA devices have increased in density, the demand for pre-designed logic modules or cores has increased correspondingly. These cores permit reuse of portions of existing designs, reducing the overall design effort. Currently, nearly all cores are specified in some static netlist-oriented format. Such specifications are not well suited for use in a run-time reconfigurable or run-time customizable environment. This paper describes JBitsDiff, *a tool constructed using Xilinx's* JBits™ *software, which extracts circuit information directly from configuration bitstreams and produces pre-routed and pre-placed cores suitable for use at run time. Further work to produce parameterizable and reconfigurable cores using* JBitsDiff, *and some of the pitfalls encountered are also discussed.*

1. Introduction

As FPGA devices have increased in size and complexity, so has the demand for pre-designed logic modules, or cores. With FPGA devices currently in the million gate range, commercially available cores have increased in both number and size. Pre-constructed and pre-tested functions consisting of tens of thousands of gates or more are commonplace. Reuse of such intellectual property has become necessary to permit larger FPGA design to be implemented efficiently.

One recent innovation in the packaging of intellectual property is the *Run-Time Parameterizable (RTP) Core* [1]. These cores are used in the *JBits*™ [2] software environment and exist as compiled Java classes which directly modify FPGA resources. While this approach provides support for run-time reconfiguration and permits on-the-fly customization of logic, existing cores are usually defined in formats which are not readily convertible for use at run-time.

Today, nearly all existing FPGA tools are based on netlist descriptions of circuits. While this provides a common format for exchanging and manipulating cores, it is not directly useful in the *JBits* approach. This makes it difficult to use the existing body of cores available for FPGA designs.

This paper describes *JBitsDiff*, a tool which uses *JBits* to extract circuit information from cores at the configuration bitstream level. Using this approach, existing cores may be translated into a form useful to *JBits*, independent of the original design method Also addressed are barriers to run time reconfiguration such as address line swapping in Look-Up Tables (LUTs).

2. Parameterized Core Library Development

A framework for developing parameterised core libraries is described in [3], with the XC6200 series as the target FPGA. A declarative language Ruby is used to allow an initial exploration of the design space. The Ruby description of the core is then (manually) translated into parameterised VHDL. This VHDL contains placement attributes, which are used for the subsequent automatic synthesis and translation to configuration bitstream. These attributes ensure efficient implementations of the cores over a range of parameters. The framework was further developed and described in [4]. The block language Pebble is used for the initial description, and is automatically translated into parameterised VHDL.

Commercial toolsets also now include sophisticated core generation systems. Traditionally, the library components supplied with a design environment have been quite low level, encompassing registers, basic arithmetic units, and replacements for popular logic families. Tools such as the Xilinx Core Generator supplied as part of the Xilinx Foundation design environment offer cores such as large dual port memories or entire PCI interfaces. In order to promote re-use, these cores can often be parameterised by users when designing their overall system.

In both the cases outlined above, the role of the cores in the design flow is similar. They are intended to be used early in the design flow, producing netlists in the case of Core Generator, and synthesisable VHDL in the case of Luk's framework. The place of each of the core types in a hypothetical design flow is shown in figure 1.

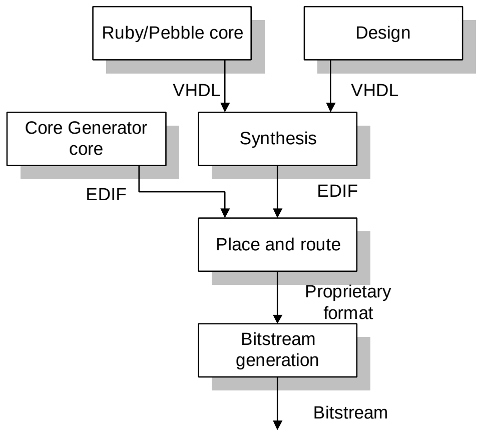

Figure 1. Cores in a standard VHDL design flow

An alternative approach to cores is to insert them later in the design flow. Other core generation systems produce information which is processed by vendor tools: typically, synthesis and place and route, or in certain cases, place and route only. This paper presents a method of generating cores which can be inserted at the very last part of the design flow: after the configuration bitstream has been generated. Information is generated which can be directly integrated into existing device configurations or into existing configuration datastreams. In fact, since this generation could theoretically take place at application run-time, these cores are known as *Run-Time Parameterisable (RTP) Cores* [1].

Parameters to modify RTP Cores can thus be supplied either at system design-time, as with standard cores, or at application run-time. As far as core design is concerned, no distinction is made between these two cases. A further point is that no further processing of the design takes place after the RTP core is inserted into a bitstream: therefore, any logic minimization or other design customizations have to be performed by the designer of the core.

Previously, parties other than the device vendors have been able to produce core libraries because the cores were output in a standard format such as VHDL or EDIF. If figure 1 is examined, it can be seen that vendor tools take over once these cores have been assimilated into the design. The vendor-supplied place and route tools produce a proprietary format, which is subsequently processed by a further vendor-supplied tool to produce a proprietary configuration datastream. One device series that allowed an open format up to and including configuration was the XC6200 series. This allowed a complete design environment to be constructed [5], as well as allowing the run-time parameterisation of cores.

3. The *JBits* Design Environment

If open device architectures are not supported, the alternative is to produce tools that open the proprietary configuration datastream, whilst maintaining a level of security to protect intellectual property. *JBits* is an API supporting the reading, manipulating and writing configuration bitstreams for Xilinx XC4000™ and Virtex™ series devices [2]. It can be used in a number of ways. Firstly, it can be used for the low-level analysis of existing designs produced by standard design methodologies. This analysis is entirely passive, and in no way affects the functionality of the design. Secondly, a designer can create new designs using the API, directly constructing cores within a bitstream. Thirdly, *JBits* can be used by a designer to manipulate existing designs, making changes through the API to the original design, once the internal structure is known.

The use of software for the construction of logic circuits and cores is not new. It is interesting to trace the development of JBits from its earliest form. The Java Environment for Reconfigurable Computing (JERC) is a software environment which allows logic and routing to be configured and macros to be constructed at run time [6]. JERC targets the XC6200 series, and is designed in two layers. The bottom layer (Level 0) is intended to abstract away the underlying MUX-based hardware implementation. The top layer (Level 1) offers a series of logic cell primitives, both Boolean functions and bit storage that are built from level 0 facilities. Both level 0 and level 1 primitives can be used from within a Java program, configuring both logic and routing at run time.

The SPODE circuit specification library operates in a similar manner to JERC, but is not designed in layers [7]. Rather, SPODE allows full access to all configurable resources within the XC6200 series, whilst still providing primitives to configure at a logic cell level. The library functions within SPODE are called from a C program, which produced a configuration file. This file can subsequently be used to configure an XC6200 series device. JERC and SPODE were finally combined to form JERCng (JERC next generation), which is a full Java environment allowing the complete functionality of the XC6200 series devices to be utilized [8].

The cell architecture of the XC6200 series devices naturally supported constructing circuits from simple primitives, with relatively simple wiring. *JBits* is an API that allows access to the programmable resources of Xilinx Virtex devices, and builds on the original version that operated on the XC4000 series, described in [2]. At the heart of *JBits* are four functions. The first two allow configuration datastreams to be read and written. The third function allows the state of a programmable resource to be queried, whilst the fourth function allows the state of a programmable resource to be set to a defined value. The rest of the *JBits* API is a series of constants defining each of the programmable resources within the device, and the values they can be set to. *JBits* thus hides the proprietary nature of the configuration datastream whilst still allowing full read and write access to all programmable resources.

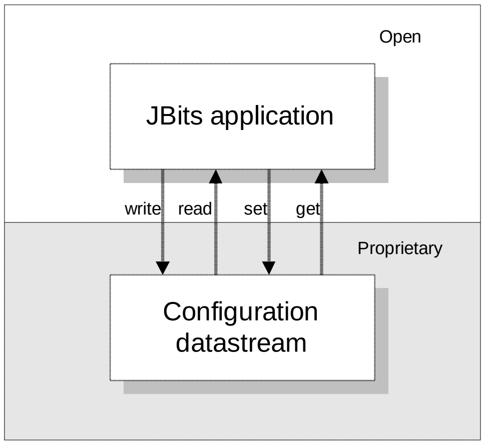

Figure 2. The JBits API.

4. Implementing Cores Using *JBits*

JBits cores can either be fixed or parameterized. Fixed cores cannot be modified by the end designer, whilst parameterized cores allow the user to enter information about the required core, and a custom circuit is constructed which embodies the user's information, such as the operand width for an adder. Initially, fixed cores will be considered.

Fixed *JBits* cores are Java class files, with the following interface. Firstly, the location of the core's origin on the programmable device is defined in terms of CLB coordinates, with (0,0) corresponding to the lower, left corner of a device. The origin is the lower, left corner of the bounding box that wholly contains the cores functionality and routing, and can be both set and read. The extent of the bounding box is also defined, and can be set and read. A number of identification mechanisms exist, which interface with the *BoardScope* software [9] and are not considered here. *JBits* cores can also be protected, preventing interrogation of the programmable resources within the bounding box. Finally, the user is able to instantiate the whole of the core in an existing datastream by a set() method. The set() method contains the sequence of JBits.set() calls required to set all the programmable resources within the bounding box to the desired values.

Thus, the effort of designing a *JBits* core resides in constructing the set() method. All other functionality is easily added. As noted previously, there is no processing step after the generation of the *JBits* core: therefore, it is crucial that the core is efficient and complete.

Modern logic synthesis tools represent many person-millennia of human design effort. On first inspection, it appears that the designer of a *JBits* core library is not able to take advantage of this effort. Furthermore, individuals and organisations are likely to have a great deal of intellectual property scattered over various designs. However, this IP is likely to have been produced by a variety of design capture tools, and is not in a form to be directly incorporated into a *JBits* core library.

This situation is similar to the situation discussed by Brebner, when considering swappable logic units (SLU) on the XC6200 series [10]. It was argued that due to the relative immaturity of the SLU concept, SLU awareness is unlikely to be built into design tools. Therefore, a tool was presented which could detect SLUs within existing XC6200 circuitry through a combination of automatic detection and user knowledge.

5. Automated Core Generation

In order to incorporate IP cores in a *JBits* core library, it is necessary to be able to translate the IP cores into *JBits* cores, in a manner that is independent of the original methodology used to construct them. One thing all IP cores have in common is that ultimately they are transformed into configuration datastreams, or to be more accurate, instantiations of the cores are transformed. Since *JBits* allows configuration datastreams to be queried, it is theoretically possible to construct a tool that can transform the configuration datastream constructed from an IP core into a *JBits* core.

A similar tool is presented in [13] though it is used for a different purpose. The tool *ConfigDiff* accepted two XC6200 series configuration files, and produced the incremental configuration that allowed the first configuration to be transformed into the second configuration. *ConfigDiff* was possible due to the open architecture of the XC6200 series, and was economical in practice because of the random access that was permitted to the memory map storing the configuration.

ConfigDiff produces all configuration data items required to transform one configuration (**current**) into another (**next**). Parts of the logic that are in next but not in current are configured, whilst parts of the logic that are in **current** but not **next** are set to unused logic. Heron and Woods [11] presented a proposed modification to *ConfigDiff*, allowing rapid configuration in two directions, both from current to next, and back from next to current. In the modified version, parts of the logic that are in **current** but not in **next**, are disconnected from the rest of the circuitry but left for the most part intact. If reconfiguration is then required back from **next** to **current**, a smaller number of changes are required overall than in the original *ConfigDiff*, which would require the logic changed to unused logic in the first transformation to be changed back to its original value.

It is pointed out by Heron and Woods that the technique can be applied to device families other than the XC6200: such a tool has now been produced called *JBitsDiff*, and versions have been produced to work with both the XC4000 series and the Virtex series. *JBitsDiff* accepts two configuration datastreams as input: again, these are referred to as current and next. In addition, a rectangular bounding box is defined. *JBitsDiff* automatically produces a *JBits* core as output. This *JBits* core, which is a series of *JBits* calls, can then be inserted into an existing configuration datastream (usually **current**).

The bounding box is required to define the extent of the logic. The tool makebits that generates configuration information for the XC6200 could be made to only generate data items for a section of the device, since the device supported partial configuration. The equivalent tool bitgen for the Virtex and XC4000 series currently produces data items for the whole of the device. In the case of the XC4000 series, this is due to the serial nature of configuration. The Virtex series, however, supports partial configuration: this is not currently implemented in bitgen. Therefore, it is necessary to define the bounding box of the core manually.

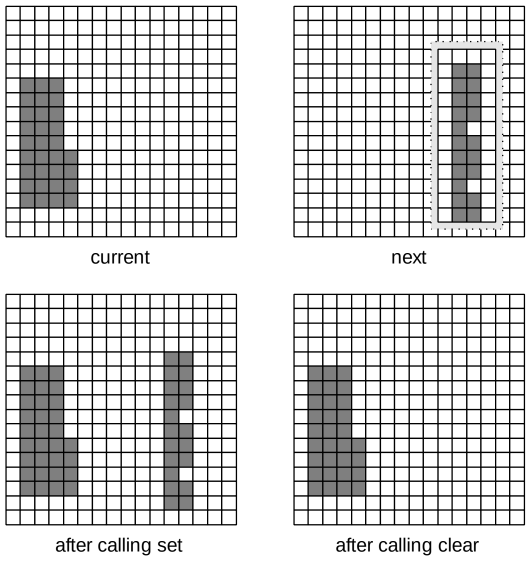

Figure 3: Operation of JBitsDiff.

The core contains three main functions. The first function is the set() method described previously. The set() method inserts all the logic from next within the bounding box into current. The clear() method inserts all the logic from current within the bounding box into next. Finally, the default() method sets all logic within the bounding box to the default values set when the device is first powered up. The operation of *JBitsDiff* is shown diagrammatically in figure 3. Other functions allowing logic to be powered down and powered back up by removing and adding their clock signals are also produced, but are not considered here.

It is important to realise that the **current** configuration need not be the base configuration for calling set(). Set() can be called using any configuration, and will insert the area of logic enclosed in the bounding box into the new configuration. The area of logic enclosed in the bounding box can be considered as a core. This core can be instantiated at any position within the new configuration, providing the bounding box can be wholly enclosed at the new position.

Figure 3 shows the two cores well separated on the device. This need not be the case. However, the nature of Virtex routing means that care must be taken that cores do not share routing resources, unless this is intentional. JBitsDiff reports any clashes of resources between **current** and **next**, rather than prohibiting them, since currently, this is the way in which cores are connected together. Similarly, the fact that two cores share a clock source may be intentional. However, care must be taken with bi-directional lines to ensure they are only driven from one source.

6. Case Study : The Constant Coefficient Multiplier

6.1. Core Description

The constant coefficient multiplier under consideration here is the KCM [12], which simplifies the task of multiplying a variable by a constant *k*. The KCM holds a number of copies of the *k* times table, each of which contains 16 entries. When multiplying an *n*-bit variable by a constant, *n*/4 copies of the table are required. Each *n*/4 bit nibble of the variable addresses a table, and the partial products produced are summed to provide the required product.

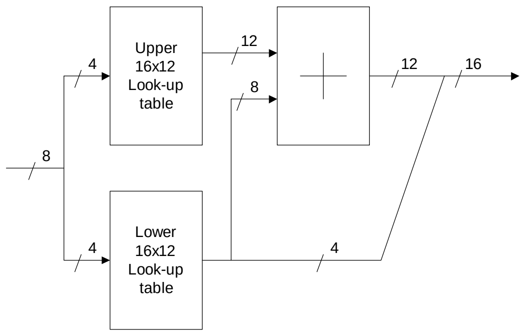

Figure 4: The KCM schematic diagram.

A constant coefficient multiplier which multiplies an eight bit variable by an eight bit constant will be considered. The schematic is shown in Figure 4. The circuit was described using VHDL, initially without any placement constraints or floorplan, in order to investigate the core size. The core took its input from device pins, and drove its output to device pins. Whilst this is the case for some cores, it is by no means a pre-requisite, hence registers were placed around the core to emulate the direct connection of the cores to other logic or routing cores.

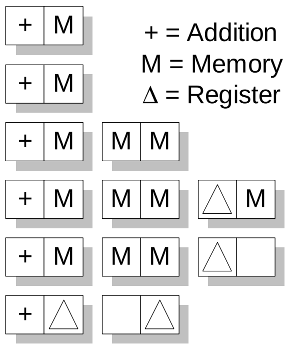

Figure 5: The KCM synthesized layout.

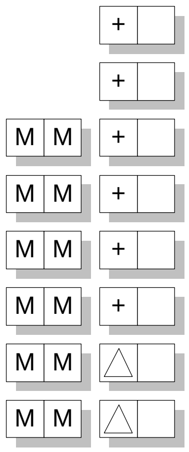

Figure 6: The KCM floorplanned layout.

The constant coefficient multiplier could rapidly be designed and tested using mainstream tools. The VHDL was designed and tested using Active VHDL, and synthesised using Synplicity. The resulting EDIF netlist was then passed through the Xilinx M1 tools to produce the configuration datastream. Initially M1 produced the circuit shown in Figure 5. The floorplanner was then used to align the two memories in the same column, and also to tidy up the placement of the delay registers. The floorplanned version is shown in Figure 6.

6.2. Producing an initial core

An initial core can automatically be produced by running the *JBitsDiff* tool using the bitstream generated from the floorplanned layout as the **next** configuration. Initially, it will be assumed that the core is to be instantiated into empty space on a device: therefore, it is only necessary to specify the non-default resources. The simplest method of doing this is to specify an empty bitstream as the **current** configuration. *JBitsDiff* returns a set method for the core, allowing it to be instantiated into a new bitstream. The KCM core covers 16 slices: the set method for this core is some 520 lines of code. 80% of this code deals with routing, both the routing of signals into the CLBs, and the setting of switches in the general routing matrix.

This core is quite limited: it represents a point solution of the KCM. It is of fixed bit-width, the level of pipelining is fixed, and most drastically, the constant encoded within it is fixed. Methods of extending the scope of the core are now investigated.

6.3. Varying the constant

Initially, it is important to understand how the constant is encoded in a KCM. The constant is used to produce a times-table that can be addressed by the multiplier, as shown in Figure 4. For an 8 bit unsigned multiplicand, a maximum of 12 bits of storage is required for each entry in the times table. The look-up tables in the KCM are implemented as twelve 16-bit storage elements, which map simply to the internal look-up tables of the Virtex slices. Each of these 16 bit-LUTs corresponds to a bit position within the overall table. By using the floorplan, it is possible to arrange these physically in ascending order, such that the LUT corresponding to the least significant bit of the overall table is in the bottom LUT. Therefore, since the contents of the original table are known, it is possible to determine what the contents of each of the twelve 16-bit LUTs should be.

It would be tedious and time consuming to repeat the process of synthesizing a KCM core for each of the possible constants, producing for an 8 bit version, 256 versions of the set method. Other run-time reconfiguration methodologies based on static design tools implicitly or explicitly take this approach. In this approach, the sections of the core requiring run-time modification and the type of modification performed are supplied by the core designer as part of the core API.

6.4. Address lines

The values stored in the F and G LUTs of the Virtex slice can be read by using the relevant *JBits* constant. For example, to read the value stored in the F LUT of slice 0 of CLB(0,0) the following code is used:

```
stored=JBits.get(0, 0, LUT.SLICE0_F);
```

Similarly, to set the value stored in the F LUT to a new value, the following code is used

```
JBits.set(0,0, LUT.SLICE0_F, newValue);
```

Theoretically, it would be a simple matter to produce a Java method allowing the mapping of a KCM look-up table to each of the primitive LUTs that implement the overall table. From the floorplan, it can be determined which bit position of the look-up table is mapped into which LUT: 24 JBits.set() calls would be required, 12 for each look-up table.

However, when the router assigns ROM address lines to the lines of the F and G LUT slices, it does so according to an overall cost function : simplifying the life of *JBits* core designers is not one of the terms of the cost function. As far as the router is concerned, for a 16x1 ROM, there is no real reason to map the ROM address lines in order to the address lines of the LUT. They can be assigned in any order, and the ROM initialisation value scrambled to produce the correct output when addressed. This is shown schematically in Figure 7.

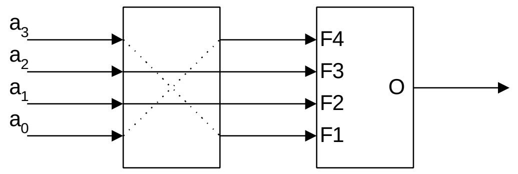

Figure 7: Router assignment of address lines.

Consider the case for storing the value 0000 0001 0000 0000 in the 16x1 LUT shown. If the router decides that the address lines a_3 and a_0 have to effectively be swapped over, the initialisation value must change. The case where the LUT would output a 1 when a_3 is 1 and a_2,a_1 and a_0 are all zero, now becomes the case where the LUT outputs a 1 when F1 is 1, and F2, F3 and F4 are all zero. Therefore, the initialisation value becomes 0000 0000 0000 0010. This will produce the correct output when the address lines a_3..a_0 are used.

When *JBits* queries the value stored in a look-up table, the initialisation value is returned. Therefore, if this initialisation value has been mapped to compensate for the router assignment of address lines, the value returned will not be the value specified in the original circuit description. Similarly, if *JBits* sets a value stored in a look-up table without regard for the router assignment of address lines, the LUT will not behave as planned.

Therefore, a method is required to automatically determine the address mapping used by the router. There are two methods available. The first is to use *JBits* to trace address lines around the device by querying the state of routing resources. This would determine the interconnections between the inputs of the LUTs and other logic dealing with the address lines: for example, an input register, or device pins. With the complex routing structure of Virtex, this is quite a difficult task.

A simpler method is to utilise an interesting feature of a set of the 65536 possible initialisation values for a 16x1 LUT. The router can choose 4! = 24 possible ways of mapping the 4 address lines, which is the set M. An initialisation value I is chosen, and mapped by each M_i from M, producing a set of 24 new initialisation values W_i. For a certain set of numbers, the only way to retrieve I from the value W_i is by applying the original mapping M_i.

Clearly, 0x0000 is not a member of this set: any of the 24 mappings Mi would retrieve 0x0000 no matter which of the mappings was originally used. Similarly, 0x0001 is not able to discriminate between the swapping of the three upper address lines. The smallest value that satisfies the criteria is 0x001A.

The design flow is now as follows. The initialisation values of the LUTs are set to one of these special values. For each LUT, the router then manipulates the address lines, producing a new initialisation value if appropriate. Once the configuration datastream is produced, *JBits* is used to read back the values stored in each of the LUTs. Each of the 24 possible mappings is then applied to the original initialisation value, and the mapped value is compared to the value read back from the LUT. Only one of these mapped values will match the value read back, and hence the address line mapping for each LUT can be determined.

The mapping of address lines is performed by the router, and is embodied in the settings of the routing resources surrounding the various LUTs. It therefore falls outside of the scope of the reconfigurable section of the circuit. This means that determining the mapping of address lines can be performed at core design-time, since these mappings remain fixed. This is however, the point at which the automatic extraction is complete: it is now up to the core designer to manually add this functionality. Once the mappings are automatically determined, it is a simple matter to produce a Java method allowing the mapping of a KCM look-up table to each of the primitive LUTs that implement the overall table. An example method is shown in Figure 8.

```
   public void
   setConstant(int  iConstant,jBits  jBits,  int  clbRow,  int
clbColumn) {
     int [][]iKCMtable = new int [12][16];
     int iBitPos,iBit;
     int [] iInit = new int [16];
     int [] iWarped = new int [16];

     iKCMtable = getKCMtable(iConstant);
     for (iBitPos=0; iBitPos<12; i++) {
       for (iBit=0;iBit<16;iBit++)
         iInit[i] = iKCMtable[iBitPos][i];
         iWarped = addressWarp(iInit,table0warp[iBitPos]);
         if ((iBitPos%2)==0)
   jBits.set(clb_row+(iBitPos/2),clb_col+1,LUT.SLICE0_F,iWarped);
         else
   jBits.set(clb_row+(iBitPos/2),clb_col+1,LUT.SLICE0_G,iWarped);
         iWarped = addressWarp(iInit,table1warp[iBitPos]);
         if ((iBitPos%2)==0)
   jBits.set(clb_row+(iBitPos/2),clb_col+1,LUT.SLICE1_F,iWarped);
         else
   jBits.set(clb_row+(iBitPos/2),clb_col+1,LUT.SLICE1_G,iWarped);
     }  /* end for() */
   }   /* end setConstant() */
```

Figure 8: Example method allowing parameterisation by a constant

6.5. Changing pipelining

Pipelining is often present in FPGA designs due to the register-rich architecture. At a slice level, it is clear how pipelining operates: the LUTs implementing the logic function drive a register and a wire directly. To get a pipelined version of a signal, it is necessary to use the output of the register rather than the output of the LUT to drive the following circuitry. Since pipelining registers in multiple parts of the device will operate at the same frequency, the clock to each register will normally be supplied by one of the dedicated global clock signals.

In the Virtex devices, the part of the internal structure responsible for propagating signals from the slices onto routing wires is a series of multiplexers, modelled in *JBits* as an output bus. Each multiplexer driving a signal on the output bus takes 12 inputs from the inside of the slice, and passes at most one of these signals. The output bus can then be connected to external routing. Amongst the inputs to the multiplexer are the outputs of each internal look-up table, and the output of the registers.

Therefore, to allow pipelining to be specified as a parameter, it is necessary to first find the sections of the core that correspond to the outputs to be pipelined, and modifying the relevant bit of the output bus to take its input from the registered output rather than the output directly. In addition, a clock signal must be provided.

One complication is with the use of skewing registers, which are required to ensure that the correct operands appear at a functional unit at the same time. In the case of the KCM, if the adder shown in Figure 4 is pipelined, a skewing register is required for the lower four bits to ensure the correct output is generated when the lower four bits of one the LUTs is recombined with the output of the adder. It is easier to construct the initial core using the maximum amount of pipelining with skewing registers if appropriate. Then the sections of the *JBits* core that deal with pipelining registers can manually be removed to a separate method, replacing pipelined LUTs with LUTs driving the output bus directly, and replacing sole registers intended as skewing registers with buffers again driving the output bus directly.

6.6. Changing clock sources

Determining the source of the clock signal for a core is performed by the place and route software. It may however, be necessary to modify the assigned clock at design time, or indeed at run-time, selecting a different clock. Consider the case of a device containing a series of non-cooperating *JBits* cores designed to operate at different clock frequencies. At design time, they may all have been assigned the same clock signal, as the router was not aware they would be used simultaneously. It is precisely this level of fine control over system resources that is required for effective run-time reconfiguration.

In the case under consideration, the clock drives each of the pipeline registers, and was selected by the place and route software as Global Clock 1. The following line shows how the clock source is set in the body of the core.

```
jBits.set(r,c,S0Clk.S0Clk, S0Clk.GCLK1);
```

It is possible to allow the clock source to be supplied as a parameter to the core. Each call that sets the source of a clock signal is manually copied from the body of the set method, and is placed in a separate method, which accepts a constant defining the clock source. This constant is then used to select which of the global clocks should be used to drive the clocked sections of the core.

6.7. Changing size of operands

Currently, the KCM core uses 8 bit operands. In order to increase the size of the operands, extra look-up tables are required, and extra adders are required to deal with the output of the new look-up tables. Currently with this technique, using the size of operands as a run-time parameter would be difficult. Therefore, separate cores would be produced to deal with common operand sizes: for example, in [12], multipliers working with 8, 10 and 16 bit operands are implemented.

6.8. Final core structure

The final core has the following functionality. Firstly, a set method is provided, which constructs a “default” KCM. This default KCM uses the constant ‘0’, has two levels of pipelining (the ROMs and the adder are pipelined), and uses **gclk1**. Three methods of parameterisation are provided: Firstly, the constant can be set to any unsigned 8 bit value. Secondly, pipelining can be specified, to either remove all pipeline registers, pipeline just the ROMs, pipeline just the adder, or pipeline the ROMs and the adder. Finally, the source of the clock signal can be specified, to any of the four inbuilt clocks. In each case, this parameterisation can be performed at run-time: configuration datastreams are constructed directly, without any further passes through design tools.

Parameter	Values	Example
Constant	0..255	MyCore.setConstant(123,jBits,0,0);
Pipeline	0..3	MyCore.setPipeline(kcm.justAdder,jBits,0,0);
Clock	1..4	MyCore.setClock(kcm.gclk1,jBits,0,0);

Table 1: Final core interface

7. Conclusions and Further Work

A method of constructing *JBits* cores has been presented, which allows cores to be produced by a wide range of design methodologies before being automatically transformed into *JBits* cores. Tool support is provided to produce an initial core from an existing configuration datastream. Generally, this initial core will be insufficient, as it represents a single parameterised version of a core. Therefore, a method of manually adding parameters has been described.

The main drawback with the tool is the sheer amount of code generated for a core, and the lack of structure. This is because in order to offer the immunity to the original design methodology, *JBitsDiff* operates at the lowest possible level – the configuration datastream. At this level, there is no concept of a design hierarchy, the whole circuit is simply flattened and transformed into the 1's and 0's making up the configuration datastream. Inferring structure from this is very difficult.

It is possible for the tool to determine the usage of a CLB from the settings of the programmable resources. For instance, adders can be found by looking for CLBs using the dedicated carry chain. Currently, the tool uses this information to build a "textual schematic" of the core. The designer can then use this schematic to begin to understand the structure.

As mentioned previously, the bulk of the code deals with the routing resources. Currently, these are simply displayed as they are encountered: no attempt is made to determine the source or destination of the routes. Therefore, the user is presented with a complex series of seemingly unconnected routing assignments. In a future version of the tool, we wish to separate the routing into a series of connections, defining the start and end points of the route as a comment, followed by the series of calls to the programmable resources configuring that connection. The end user would then be able to determine repeated patterns within the routing (e.g. OUT0 of a CLB always drives the F1 input of the CLB above it) and replace a large series of routing resource sets with a single smaller iterated series of calls. This would greatly aid the generation of cores that can be parameterised according to operand widths.

References

[1] Guccione, S.A. and Levi, D.: "Run-Time Parameterizable Cores", *Field-Programmable Logic and Applications (FPL99)*, pp 215-222, 1999

[2] Guccione, S.A., Levi, D. : "XBI: A java-based interface to FPGA hardware," in John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pp 97-102, Bellingham, WA, November 1998

[3] Luk, W, Guo, S., Shirazi, N., Zhuang, N. : "A framework for developing parameterised FPGA libraries", *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL96)*, pp 24-33, 1996

[4] Luk, W., McKeever, S.: "Pebble: A language for parameterized and reconfigurable hardware design", *Field-Programmable Logic: From FPGAs to Computing Paradigm (FPL98)*, pp 9-18, 1998

[5] S. Gehring, S. Ludwig.: "The Trianus system and its application to custom computing", *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers (FPL96)*, pp 176-184, 1996

[6] Lechner, E., Guccione, S.A. : "The Java environment for reconfigurable computing," *Field-Programmable Logic and Applications (FPL97)*, pp 284-293, 1997

[7] Brebner, G. : "CHASTE: A hardware/software co-design testbed for the Xilinx XC6200," *Proc. 4th Reconfigurable Architecture Workshop*, IT Press, Verlag, pp 16-23, 1997

[8] Brebner, G. : "An interactive datasheet for the Xilinx XC6200,", *Field-Programmable Logic: From FPGAs to Computing Paradigm (FPL98)*, pp 401-405, 1998

[9] Levi, D., Guccione, S.. "BoardScope: A debug tool for reconfigurable systems". in John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pp 239-246, Bellingham, WA, November 1998. SPIE -- The International Society for Optical Engineering.

[10] Brebner, G.:. "Automatic identification of swappable logic units in XC6200 circuitry", *Field-Programmable Logic and Applications (FPL97)*, pp 173-182, 1997

[11] Heron, J and Woods, R. 'Accelerating run-time reconfiguration on custom computing machines', *Advanced Signal Processing Algorithms, Architectures and Implementations VIII, invited paper, SPIE Int. Symp. On Optical Science, Engineering and Instrumentation*, July 1998, San Diego, USA

[12] Chapman, K.: "Constant Coefficient Multipliers for XC4000E", Xilinx App. note XAPP054. http://www.xilinx.com/xapp/xapp054.pdf. December, 1996.

[13] Luk, W., Shirazi, N., Cheung, P.: "Compilation tools for run-time reconfigurable designs", *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM97)*, pp 56-65, 1997

[14] Brebner, G.: "Automatic identification of swappable logic units in XC6200 circuitry," *Field-Programmable Logic and Applications (FPL97)*, pp 173-182, 1997