PREPRINT / To appear in the Proceedings of FCCM'07

A Library and Platform for FPGA Bitstream Manipulation

Adam Megacz[1]
Computer Science Division
UC Berkeley
megacz@cs.berkeley.edu

Abstract— **Since 1998, no commercially available FPGA has been accompanied by public documentation of its native machine code (or *bitstream*) format. Consequently, research in reconfigurable hardware has been confined to areas which are specifically supported by manufacturer-supplied tools.**

Recently, detailed documentation of the bitstream format for the Atmel FPSLIC series of FPGAs appeared on the usenet group `comp.arch.fpga`[11]. This information has been used to create `abits`, a Java library for direct manipulation of FPSLIC bitstreams and partial reconfiguration. The `abits` library is accompanied by the `slipway` reference design, a low-cost USB bus-powered board carrying an FPSLIC.

This paper describes the `abits` library and `slipway` platform, as well as a few applications which they make possible. Both the `abits` source code and `slipway` board layout are publicly available under the terms of the BSD license. It is our hope that these tools will enable further research in reconfigurable hardware which would not otherwise be possible.

I. Introduction

IN the world of software running on microprocessors, device vendors generally publish their product's bitstream format in an architecture manual for use by third-party compiler writers. The roles of chip designer and compiler writer are effectively decoupled.

In the world of FPGAs, the situation is quite different. Since the discontinuation of the XC6200 series in 1998[17], the trend has been overwhelmingly in the direction of bitstream secrecy. Currently no major vendor discloses the bitstream format of their device. This has had the effect of stunting research in several areas, including partial reconfiguration, evolvable hardware, and fault recovery. Additionally, alternative design methodologies such as self-timed circuitry or pausible clocks[18] become difficult to implement properly if the manufacturer's tools do not support them.

In August of 2005, a document describing the format of most of the Atmel FPSLIC bitstream format was posted to the `comp.arch.fpga` newsgroup[11]. This information has been used to create the `abits` library and its companion platform `slipway`, a simple two-layer FPSLIC PCB which can be inexpensively manufactured and assembled without special equipment.

The rest of this paper briefly reviews the FPSLIC fabric (section II), introduces the `abits` library (section III) and the `slipway` platform (section IV), and finally demonstrates three example applications (section V) which require bitstream access: live debugging, self-timed circuits, and frequency division of layout-sensitive signals.

II. The FPSLIC Reconfigurable Fabric

The FPSLIC reconfigurable fabric consists of an array of up to 48x48 configurable logic blocks (CLBs). The diagram for a single block is shown in Figure 2.

Each CLB has four inputs (`W`, `Xi`, `Yi`, and `Z`), shown as the four muxes near the top of the diagram. Five "sector routing" wires (`L0`...`L4`) enter each logic block, and each of the four muxes can take its input from any of these five wires. Additionally, the `Xi` input may come directly from any of the four nearest diagonal neighbors, and the `Yi` input may come directly form any of the four orthogonal neighbors; these inputs are shown as vertical dashed lines at the top of the figure.

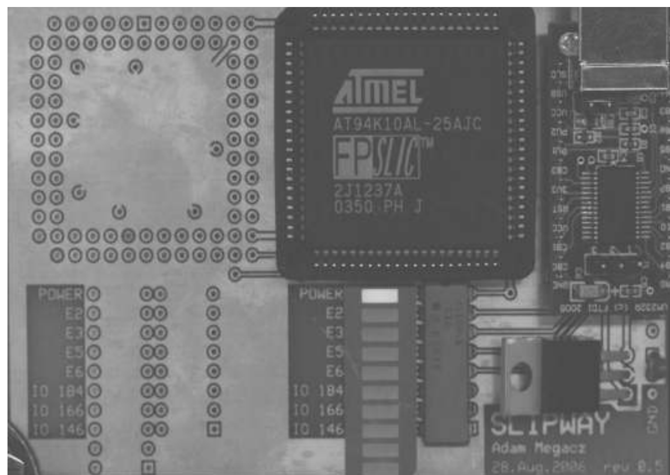

Fig. 1. The `slipway` printed circuit board

[1] This work was supported by a Graduate Fellowship from the National Science Foundation

Two look-up tables (LUTs), shown as squares in the diagram, are present in each configurable logic block. Each LUT is capable of computing any function of three one-bit inputs. The outputs of these two LUTs form two of the three *intermediate values* of the cell. The third intermediate value is the output of a mux fed by the two LUTs (`C`); it is effectively an arbitrary function of four one-bit inputs (the three one-bit LUT inputs and the `Z`-input).

The cell generates three *output values* by selecting various subsets of the intermediate values, and possibly feeding them through a synchronous register. These three outputs go to the cell's nearest orthogonal neighbors (`Yo`), nearest diagonal neighbors (`Xo`), and sector routing wires (`L`) at top right. The `F` mux output can also optionally serve as a *combinational feedback* by routing its output through the `T` mux.

The FPSLIC device consists of a homogeneous array of these cells, partitioned into 4x4 groups called "sectors". Wires within a given sector may be connected by means of horizontal and vertical wires spanning the sector. Sector wires of adjacent sectors may drive each other by means of configurable unidirectional buffers. Further details can be found in [12].

A large SRAM distributed throughout the device holds the *configuration bits*, which control the LUT equations, routing connections, and the selector inputs to the labeled muxes in Figure 2.

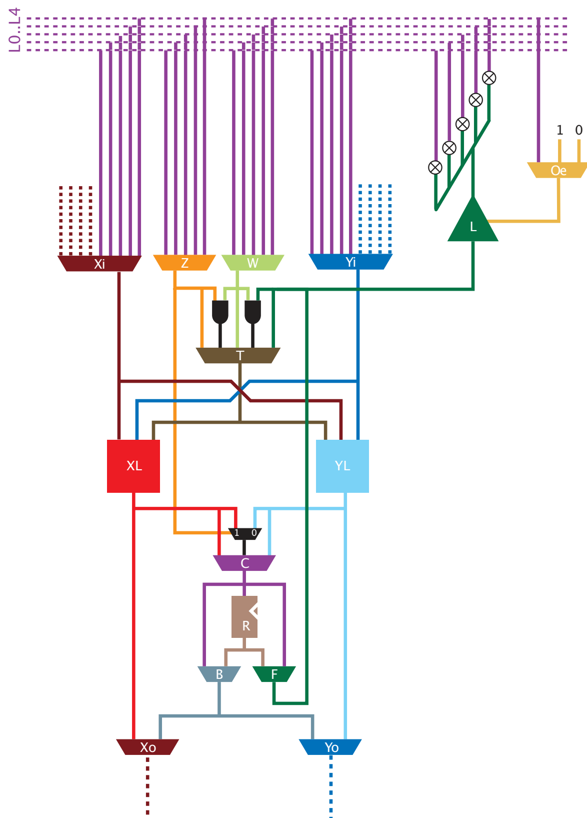

Fig. 2. Diagram of the FPSLIC Configurable Logic Block

Every configurable resource on an FPSLIC is assigned an X,Y coordinate in the cartesian plane. The position of a CLB in the fabric determines its own X,Y coordinate, and most other resources assume the coordinate of the nearest CLB. Each coordinate is specified with an 8-bit value, and an additional 8-bit value (called the "Z coordinate" by extrapolation) differentiates between the many resources at a particular X,Y location.

Configuration data for the FPSLIC is partitioned into bytes; each configuration byte has its own X,Y,Z address. Together, the three-byte address and one byte of configuration data form a *configuration word*, which is typically specified in Z,Y,X,D format.

While the device is powered up, the FPSLIC's on-board AVR processor can program the fabric by issuing complete configuration words via a set of four special registers. This partial reconfiguration can be performed at an extremely fine granularity; in particular, a LUT truth table can be atomically reconfigured without affecting any of the other resources.

III. The `abits` Library

The `abits` library is designed to configure any device which knows its own dimensions and accepts configuration words as described in the previous section. This lets the library operate identically on bitstream files, in-memory images, and live fabrics whose AVR microcontroller has some way of receiving configuration words.

Any device exporting the following interface can be programmed:

```
public interface FpslicInterface {

  /** device width, in cells */
  public int getWidth();

  /** device height, in cells */
  public int getHeight();

  /** writes data byte "d" at coordinates x,y,z */
  public int mode4(int z, int y, int x, int d);

}
```

The underlying target modified by these configuration words might be a file on a disk, a `byte[]`, or a `slipway` board connected to the host via a USB cable.

On top of the extremely low-level `FpslicInterface` API, the `abits` library layers a medium-level API `Fpslic`:

```
public abstract class Fpslic
  public final class Sector     { /* ... */ }
  public final class SectorWire { /* ... */ }
  public final class Cell       { /* ... */ }
  public final class IOB        { /* ... */ }

  public Cell getCell(int row, int col) {
  /* ... */

  public class Cell {
    public byte xlut()           { /* ... */ }
    public void xlut(byte table) { /* ... */ }
  /* ... */
```

This API lets the user acquire objects representing various resources on the device, such as `Cells` (CLB's), `Sectors`, `IOBs` (I/O Blocks), etc. Methods invoked on these objects are translated into configuration words and dispatched via the lower-level `mode4()` API.

Based on past experience with bitstream tools which attempt to construct an in-memory graph of the entire device [19], this API has been deliberately designed to support manipulating bitstreams of arbitrary size using a fixed amount of memory. In order to achieve this goal, resource objects are *ephemeral*; they are created on-demand and garbage collected when not in use.

IV. The `slipway` Platform

The `slipway` platform is a low-cost vehicle for experimenting with bitstream-level configurability. It can be manufactured cheaply in very small quantities and assembled without special equipment.

The platform consists of an Atmel FPSLIC, status LEDs, power circuitry, and an FTDI 232R USB interface. The USB interface also provides power and a 24Mhz clock to the FPSLIC. A block diagram of the board is shown in Figure 4.

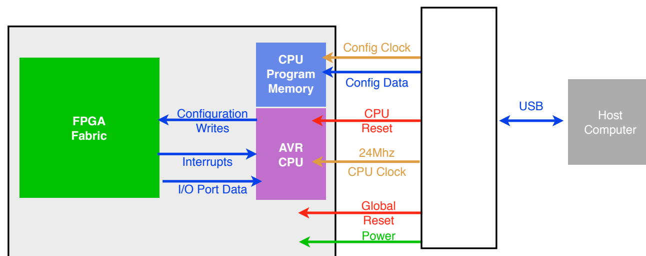

Fig. 4. `slipway` Block Diagram

A host computer attached to the USB interface can remotely reset the CPU or entire device, and can also manipulate the data (`D0`) and configuration-clock (`CCLK`) pins used to load the initial configuration onto the device. Additionally, the AVR's on-chip UART is connected to the USB interface, and is capable of reliably transferring data at 1.5Mbits/sec.

The platform's host software is written in Java and uses the portable `libusb`[15] and `libftdi`[16] libraries to communicate with the device. On initialization, the host software sends a global reset signal to the FPGA and then manually clocks a small software program onto the microcontroller. This software program then listens on the UART for commands from the FTDI device.

After the board has booted, the host software can issue commands to write configuration bits, start/stop timers, count interrupts, and read internal data values from the FPGA fabric. All communication is performed via the USB-to-UART-to-microcontroller data path.

V. Applications

The remainder of the paper describes three applications developed on the `slipway` platform which would not be possible without support for bitstream level access to device configuration. These are: "live" debugging, self-timed circuits, and frequency division of layout-sensitive signals.

A. Live Debugging

Partial reconfiguration and bitstream access can be combined to provide excellent "live" debugging capabilities. This capability is facilitated by the convention of keeping routing plane `L3` unused by the main design, leaving it available for run-time debugging.

A small routine in the `abits` library (`routeProbe()`) uses this plane to route the output of any chosen cell along the vertical and horizontal paths of the `L3` plane to a designated debugging pin on the East edge of the fabric. At this point the output can be either sampled programatically as a IO pin or monitored as an edge-triggered interrupt.

The `slipway` fabric debugger, shown in Figure 3, is a graphical interface capable of visually rendering a design binary (center diagram), displaying the programmed state of any particular cell (table at right), modifying the configuration interactively (keymap, lower left), and sampling the state of each cell (color of diamonds representing LUTs).

B. Self-Timed Circuits

Support for self-timed circuits in manufacturer-supplied FPGA tools is notoriously poor. Many of the tools have difficulty dealing with combinational feedback and few allow a satisfactory level of routing control. With the ability to directly manipulate the configuration state of the Atmel FPSLIC, a number of interesting experiments in self-timed reconfigurable hardware are now possible. In this section, we describe some initial work in creating micropipelines[8] on the FPSLIC device.

Elementary Self-Timed Gates

One of the most fundamental gates used in self-timed logic is the *Muller C-Element*, a multiple-input gate whose output assumes the value of its inputs when they

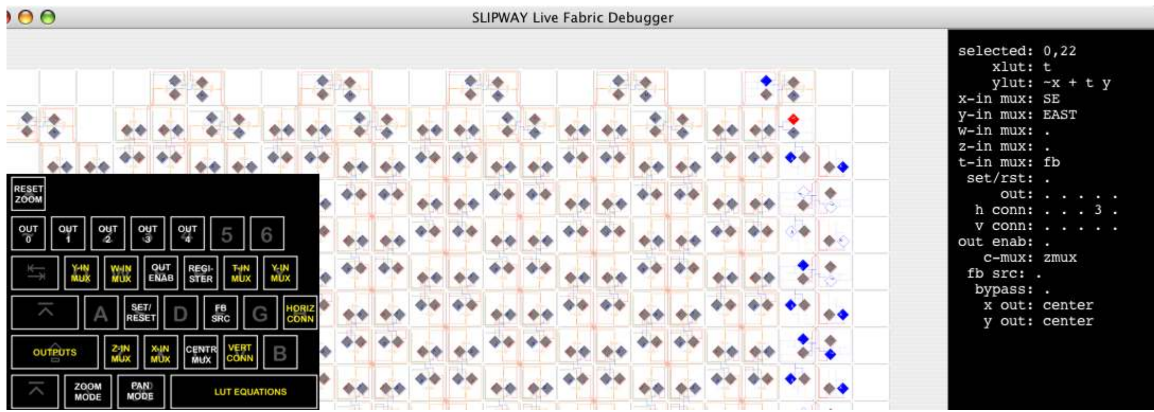

Fig. 3. The `slipway` Fabric Debugger

all agree, but retains its former value when the inputs disagree.

One way to construct an 2-input C-Element is by looping back the output of an 3-input majority gate to provide an additional input (Figure 5).

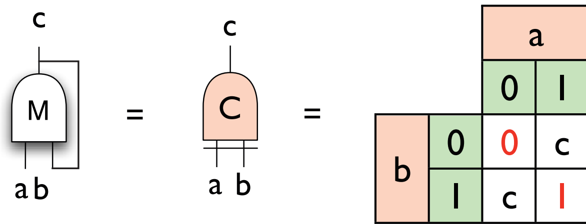

Fig. 5. A majority gate with feedback, Muller C-element, and truth table

In this sort of design, it is important for the loopback connection to have very low latency. The "internal feedback" wire of the FPSLIC (the path through the `C`, `F`, and `T` components in Figure 2) provides just such a low-latency feedback, but unfortunately the vendor's tools are completely incapable of emitting a configuration which takes advantage of the low-latency feedback for this purpose.

Fortunately, `abits` gives us full control of the CLB resources, so we can directly configure one of the LUTs to compute a majority function, and configure the muxes to route its output through the low-latency path back to the `T`-mux input.

Micropipelines

A micropipeline is a structure commonly used to create clockless FIFOs with flow control. By inverting one input to a muller C element, each cell will copy its predecessor's output whenever that output differs from its *successor*'s output. The result is a fifo with backpressure where tokens are represented by adjacent stages which have differing outputs.

Because a LUT can act as *any* function of three inputs, it is trivial to invert one of the inputs. The result is a Micropipeline stage [8] – a C Element with one input inverted. The cell configuration for this gate is shown in Figure 6.

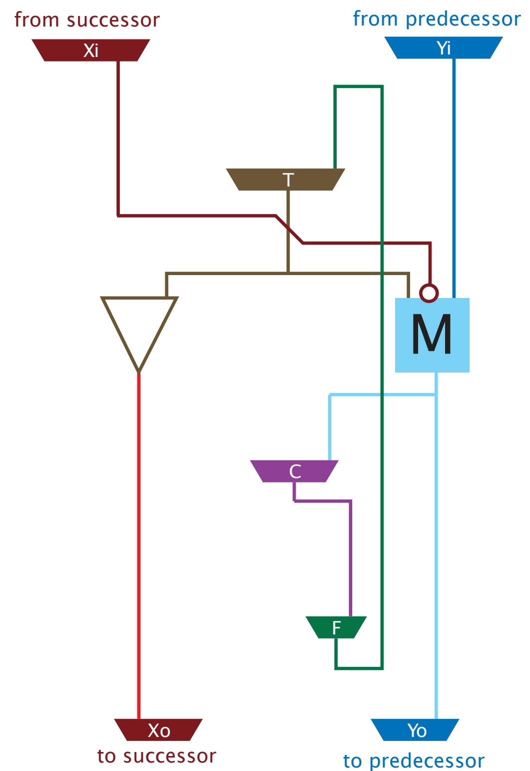

Fig. 6. A CLB Configured as a Micropipeline Stage

The configuration described in the previous section is the most straightforward implementation of a micropipeline stage. However, in order to receive inputs from neighboring stages with the minimum possible delay, those inputs must arrive via the nearest-neighbor inputs rather than the sector wire inputs. Because only

two such inputs are available, and because one must be from an orthogonal neighbor and one from a diagonal neighbor, a regular layout can be difficult to achieve.

Initial experiments used the most straightforward possible positioning of the micropipeline stages: each FIFO stage is an orthogonal neighbor of its predecessor and successor stages. An additional row of cells is then used to convert the *orthogonal* output of each stage into the *diagonal* input of one of its neighboring stages; we will refer to such cells as "bridge" cells. A four-stage instance of this layout is shown in Figure 7.

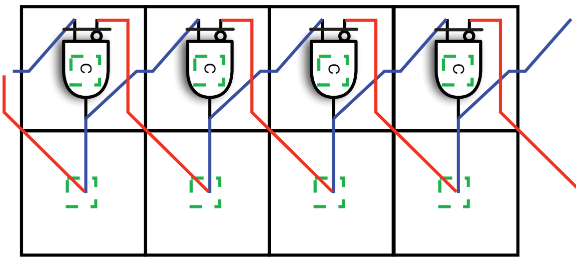

Fig. 7. "Bridged" Layout

An Improved Layout

Improving the layout requires eliminating the extra bridge cells. However, eliminating the bridge cells introduces a fresh constraint: each micropipeline stage must be the *diagonal neighbor of one adjacent stage and the orthogonal neighbor of the other*. The solution to this constraint is to have alternating stages employ alternating assignments of inputs. "Even" stages are orthogonal to their predecessor and diagonal to their successor; "odd" stages are vice versa. This layout is shown in Figure 8.

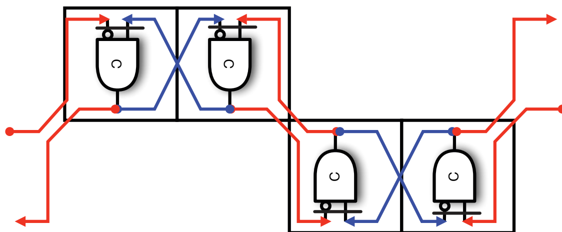

Fig. 8. "Bridgeless" Layout

A pair of FIFOs positioned according to this layout constraint "fold up" nicely into a braided formation, as shown in Figure 9.

The braid can then be "capped" on both ends and "bent" into a pattern that snakes back and forth across the entire fabric (Figure 10). This layout scheme has been programmed as an algorithm to automatically lay out a FIFO along an arbitrary path through the fabric.

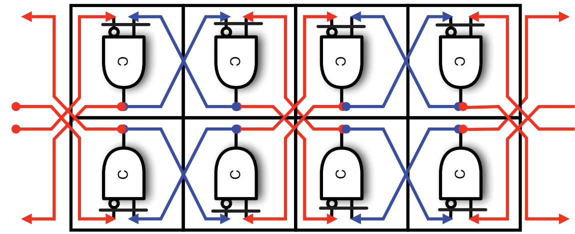

Fig. 9. "Braided Bridgeless" Layout

Ring FIFOs

With a complete implementation of linear micropipeline FIFOs, it is now possible to explore more interesting results by joining the input and output of the FIFO to form a ring. When tokens are placed in this ring, they will spin around at a rate limited only by the switching speeds of the underlying circuitry.

Such self-timed ring FIFOs serve as a canonical model[1] for the throughput and latency behavior of many asynchronous systems. At a fine grain, iterative computations such as division[2] are often implemented as self-timed rings, and at a coarse grain entire processors can be designed as a single self-timed ring[10].

The principal tool for studying these rings is a graph of the *token rate observed at a particular cell* (rate) versus

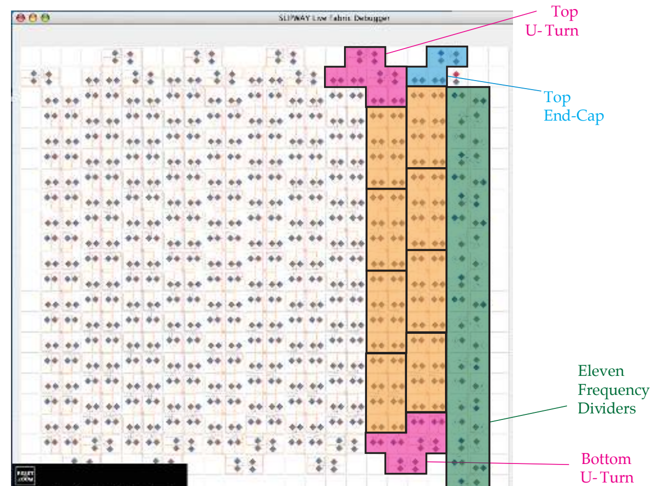

Fig. 10. A 400-Stage Micropipeline Ring with Eleven Frequency Dividers

the ratio of tokens to stages in the ring (occupancy). We will refer to this graph as a "rate/occupancy graph." The following section develops the tools necessary to collect the data for this graph.

C. Frequency Division of Layout-Sensitive Signals

Measuring the bit rate turns out to be a rather challenging task. At peak bitrate the individual stages are switching so fast that the signal cannot be brought directly out to the I/O pads of the device.

Worse, a signal at this frequency cannot travel more than two cells using nearest-neighbor connections, or four cells using sector wires. Each layer of logic the signal passes through introduces some variable amount of delay. When the worst-case delay introduced exceeds the oscillation period, a pair of adjacent transitions (one rising and one falling) collide, altering the frequency of the signal. This phenomenon is also observed in *wave pipelined*[21] systems. To avoid this signal corruption, downsampling must be performed using very few stages of logic at a location *physically close to* the pipeline.

In addition, it is desirable to perform all measurements without additional equipment such as an oscilloscope or logic analyzer. This facilitates long runs of unattended data collection in various configurations. For this reason, it is desirable to use the on-die AVR microcontroller for data collection.

The "nearest neighbor" connections to cells along the East edge of the device can be routed to data and interrupt pins on the AVR microcontroller. Connecting the pipeline directly to an interrupt pin is not useful; the microcontroller runs at a peak frequency of 25Mhz, and requires a four-cycle recovery time after an interrupt. In practice the maximum interrupt frequency is even lower since software must be executed with each interrupt.

Thus the problem is to satisfy two constraints: a high-frequency signal that can only travel very short distances on one end, and a fairly low-frequency sampling device on the other. In order to bridge this gap, a frequency divider (one-bit asynchronous counter) was designed. A chain of these dividers was placed along the East edge of the device (Figure 3), bringing a sample from the upper-right corner of the ring down to the interrupt pin in the lower-right hand corner, dividing the signal frequency by a factor of 2^{11} in the process.

Initial Results

Initial measurements using the bridged (Figure 7) layout revealed that the bridge cell had two effects relative to the bridgeless (Figure 8) layout:

- The additional wiring increased the total overall delay around the ring.
- The additional wiring created a very large mismatch between the *forward* and *reverse* delay through the pipeline.

The result of the first effect was a peak token velocity of only 533 Mstages/sec. Regardless, this is encouraging, since Atmel lists the FPGA's nominal operating frequency as 100Mhz[13], meaning that an ideal *synchronous* FIFO ring would achieve a peak velocity of only 100 Mstages/sec.

The second effect – mismatch between the forward and reverse delays – causes the occupancy/rate graph to "lean" sharply in one direction, depending on whether the extra wiring is used to connect stages to their predecessors or to their successors.

Figure 11 shows the original design with the external bridge cell on the wire to the predecessor and the wire to the successor, as well as an additional graph in which one stage of the ring is slower than the rest.

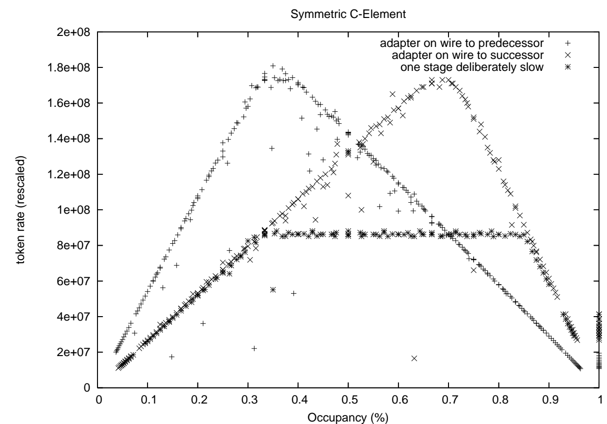

Fig. 11. Original Square-Ring Layout, Reversed, and with Slow Stage

The improved layout and a more robust frequency divider were used to generate Figure 12.

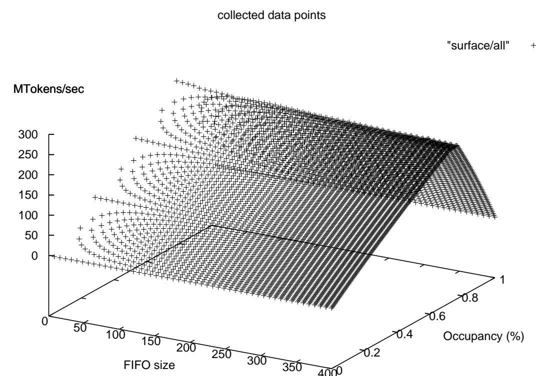

Fig. 12. Token Rates At Every Possible Occupancy for 52 Different Ring Sizes

Internally-Symmetric C Elements

The slope of the "forward limited" and "backward limited" portions of the token-rate graph have unequal slopes, suggesting that the forward and backward delays of the bridged design (Figure 6) are unequal.

In fact, on closer inspection, this can be observed directly from the CLB configuration. The C-Element is placed on the LUT corresponding to the reverse neighbor, and the output to the *other* stage passes through an extra LUT before leaving the cell. This adds at minimum an entire LUT of delay to the reverse stage.

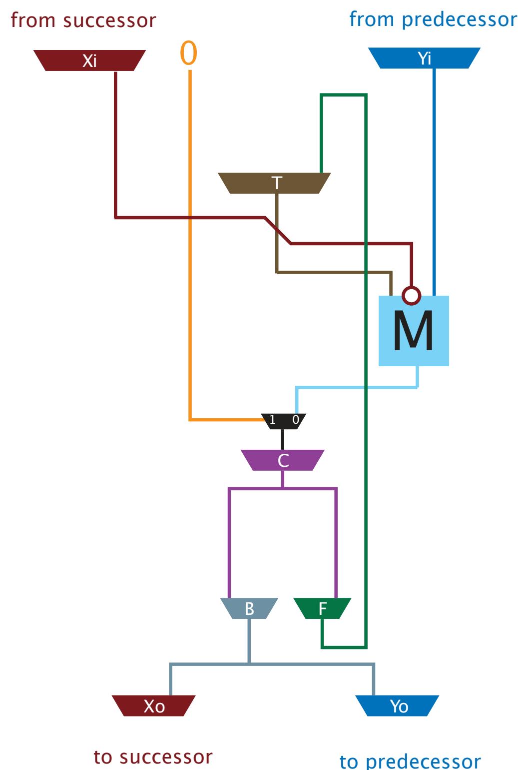

Fig. 13. Symmetric C-Element CLB

A new C-Element design is shown in Figure 13. This new design routes *both* the forward and backward outputs through the `C` and `B` muxes, and then out to the `Xo` and `Yo` outputs, giving the cell an internal symmetry with respect to its outputs.

The occupancy/rate graph for this new C Element design appears in Figure 14. Unfortunately this design is not as robust as the original design; it occasionally loses tokens. However, on average, it is clear that the symmetric design produces a similarly symmetric occupancy/rate graph, with matching "forward limited" and "reverse limited" slopes.

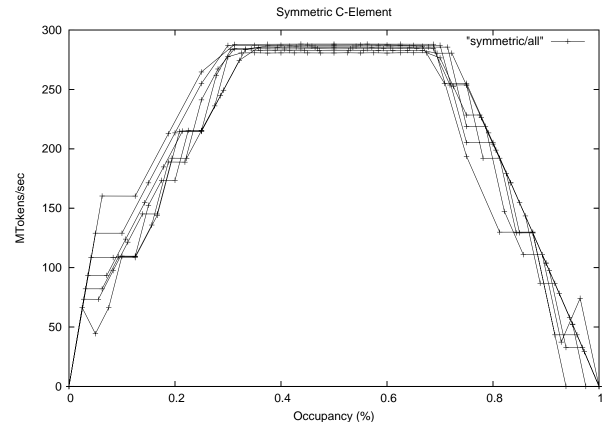

Fig. 14. Symmetric C-Element Data

VI. Related Work

Following the discontinuation of the XC6200, Xilinx briefly distributed a package called "`jbits`" [19] which could manipulate certain aspects of the bitstreams for a few of the early Virtex devices. Unfortunately it never approached a complete solution, lacking basic functionality such as the ability to create new bitstreams.

Although `jbits` was eventually updated to add support for Virtex-II (but not Virtex-II Pro), this internal routing structure of this device has never been documented outside Xilinx, leaving the `jbits` API nearly impossible to use. Since its release, `jbits` has been available only under an extremely restrictive licensing agreement, including clauses such as "You agree not to display the Software on any computer screen"[20].

`Jbits` still does not support any device introduced in the last six years.

Despite its shortcomings, the concept and initial promise of `jbits` are admirable, and the name "`abits`" was chosen to acknowledge its influence.

VII. Conclusion

This paper has presented `abits`, a library for manipulating FPSLIC bitstreams and performing partial reconfiguration, and `slipway`, a low-cost platform for experimenting with these technologies.

Source code for all software presented, as well as PCB masks, can be obtained from:

`http://research.cs.berkeley.edu/project/slipway/`

It is our sincere hope that the availability of these tools facilitates a re-emergence of interest in the many areas where progress has been hampered by the lack of a platform with bitstream-level configurability. Obvious

future directions building on this work include evolvable hardware, defect tolerance, self-timed logic, self-modifying hardware.

VIII. Acknowledgements

I would like to thank John Wawrzynek and Ivan Sutherland for their encouragement and advice, and the anonymous referees for their helpful comments.

References

[1] Ted E. Williams, *Analyzing and improving latency and throughput in self-timed pipelines and rings*. In Proc. International Symposium on Circuits and Systems, May 1992.

[2] Ted E. Williams, *Self-Timed Rings and their Application to Division*, Ph.D. Dissertation, Stanford University CSL-TR-91-482, May 1991.

[3] Charles E. Molnar, Ian W. Jones, William S. Coates, Jon K. Lexau, Scott M. Fairbanks, and Ivan E. Sutherland. *Two FIFO ring performance experiments.* Proceedings of the IEEE, 87(2):297-307, February 1999.

[4] Anthony Winstanley and Mark Greenstreet, *Temporal Properties of Self-Timed Rings*. In proceedings of CHARME 2001, Lecture Notes in Computer Science 2144 pp 140-154, 2001.

[5] Winstanley, A.J.; Garivier, A.; Greenstreet, M.R., *An event spacing experiment*. ASYNC 2002. Proceedings. Eighth International Symposium on , vol., no.pp. 47- 56, 8-11 April 2002

[6] Scott Fairbanks and Simon Moore, *Analog Micropipeline Rings for High Precision Timing*. Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on , vol., no.pp. 41- 50, 19-23 April 2004

[7] Zebilis, V.; Sotiriou, C.P., *Controlling event spacing in self-timed rings*. ASYNC 2005.

[8] Ivan Sutherland. *Micropipelines*: Turing award lecture. Communications of the ACM, 32 (6):720–738, June 1989.

[9] Ebergen, J.C.; Fairbanks, S.; Sutherland, I.E. *Predicting performance of micropipelines using Charlie diagrams*. Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on , vol., no.pp.238-246, 30 Mar-2 Apr 1998

[10] Robert F. Sproull, Ivan E. Sutherland, and Charles E. Molnar. *The counterflow pipeline processor architecture.* IEEE Design & Test of Computers, 11(3):48–59, Fall 1994.

[11] Gosset, William Sealey. *Atmel AT40k/94k Configuration Format Documentation*. Posted to `comp.arch.fpga` and archived with message-id `20050812150910.29614.qmail@nym.alias.net`.

[12] Atmel Corporation. *AT94KAL Series Field Programmable System Level Integrated Circuit*. Technical note 1138. `http://www.atmel.com/dyn/resources/prod documents/doc1138.pdf`

[13] Atmel Corporation. *Coprocessor FPGA with FreeRAM*. Technical note 2818. `http://www.atmel.com/dyn/resources/prod documents/doc2818.pdf`

[14] `http://research.cs.berkeley.edu/project/slipway/`

[15] `http://libusb.sourceforge.net/`

[16] `http://www.intra2net.com/de/produkte/opensource/ftdi/`

[17] Xilinx Corporation, *Xilinx XC6200 FPGA-based Reconfigurable Co-Processor Data Sheet* v1.10, 4/97

[18] Seitz, C. *System Timing*. In *Introduction to VLSI Systems*, C. Mead and L. Conway, eds., Addison-Wesley, Reading, Mass., 1980.

[19] Steven A. Guccione, Delon Levi and Prasanna Sundararajan. *JBits: A Java-based Interface for Reconfigurable Computing.* 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).

[20] *Xilinx Jbits SDK Software License*, `http://www.xilinx.com/jbits/agree.pdf`

[21] L. Cotten, *Maximum rate pipelined systems,* in Proceedings of AFIPS Spring Joint Computer Conference, 1969.