Reconfigurable Asynchronous Logic

Rajit Manohar
Computer Systems Laboratory
Cornell University, Ithaca, NY 14853.

Abstract— **Challenges in mapping asynchronous logic to a flexible substrate include developing a balance between circuit-level flexibility, mapping complexity, and logic overhead. We have developed a reconfigurable dataflow architecture that addresses these challenges, and have also created the necessary synthesis flow required to map designs to the architecture. The architecture exploits some of the unique features of asynchronous logic, and attains a performance that significantly exceeds previous asynchronous FPGAs.**

I. Introduction

Post-silicon reconfigurability is becoming an increasingly attractive method for designing VLSI systems. Verification and validation of modern VLSI systems that contain hundreds of millions or even billions of transistors is a daunting task. The ability to change a design post-fabrication improves the chances of obtaining a working design on first silicon.

A field-programmable gate array (FPGA) takes this idea to the extreme. It typically consists of an arrayed set of reconfigurable blocks that are carefully designed so that any design can be mapped to the architecture by an appropriate choice of configuration. The configuration typically corresponds to the state of a set of switches that are used to control connectivity and logic function. Developing such a flexible substrate is challenging because it involves the co-design of both the underlying hardware as well as a method for mapping designs to the substrate.

The regular nature of an FPGA is also appealing from the standpoint of manufacturability. The complexities of modern lithographic techniques at 90nm and below require careful physical design to ensure good yield. A regular design reduces the overhead involved in satisfying manufacturability guidelines. Also, the inherent flexibility of the FPGA substrate allows the architecture to tolerate defects, because the system can be reconfigured to not use the defective regions of the die.

FPGAs have a number of drawbacks compared to designing a customized chip, because there is a cost to be paid for reconfigurability. In particular, this cost can be measured in terms of reduced performance, higher area for the same function, and increased power consumption [9]. An easy way to realize why this is the case is to think of an FPGA as having circuits that can implement *any* logic function, with the FPGA configuration selecting the appropriate subset of the architecture that corresponds to the logic being implemented. The increased area of an FPGA makes the problem of distributing a clock over the entire FPGA harder, because the same logic function requires significantly more area. Also, the increased power consumption of FPGAs exacerbates the power problems plaguing designers today.

Asynchronous logic is a way to design digital systems without clocks. The approach abandons the notion of global synchrony in digital design and replaces it with local synchronization among parts of the design that exchange information [17]. Asynchronous design is also commonly viewed as a low-power design method. The elimination of a global clock combined with potential power benefits makes asynchronous logic an appealing method for the design of reconfigurable systems.

Early work on reconfigurable asynchronous logic focused on developing gate-level programmable logic [4]. Other architectures were based on "porting" a clocked FPGA architecture, and the result was relatively low throughput [6], [15], [8]. More recently, there has been work that is similar in spirit to the design described here, where an asynchronous FPGA architecture based on programmable pipeline stages was proposed [24]. There has also been a some work on prototyping asynchronous logic using commercially available synchronous FPGAs (e.g. [5], [22]).

We present an asynchronous FPGA architecture that is capable of implementing high-performance asynchronous logic. The architecture is developed by examining the common components used by previously designed complex asynchronous chips—in particular, fine-grained bit-level pipelined asynchronous designs (Section II). Reconfigurability is introduced as a method to enable the construction of arbitrary asynchronous pipeline topologies. We pick a level of abstraction for reconfiguration that enables a designer to focus on the functionality of the asynchronous logic, without having to worry about low-level details such as hazard-free logic synthesis (III). In fact, mapping designs to the architecture benefits from most (if not all) the common logic optimization techniques developed for synchronous FPGAs. We discuss the

performance of benchmark applications, commenting on the benefits and limitations of the architecture (Section IV).

II. Asynchronous Logic

The term "asynchronous logic" (also called "self-timed" logic) applies to a wide variety of circuit families that do not use clock signals for their operation [17]. While the logic families differ in the nature of the delay assumptions made for correct operation, the high-level description of asynchronous logic is similar across all circuit families.

A component in an asynchronous circuit is a block with input and output ports. Ports can be connected to each other to form point-to-point communication links called *channels*. Data are exchanged between asynchronous blocks by message-passing over these channels. Communication occurs between pairs of components, where one participant sends a message while the other receives a message. Channels do not store information, and therefore the send and receive actions also result in rendezvous synchronization—a send operation blocks until the receive is ready, and vice versa. If a designer would like a channel to store information, explicit first-in first-out (FIFO) buffers are introduced.

Channels are implemented by sets of wires. The communication semantics can be implemented using a variety of handshake protocols. The simplest method is the four-phase handshake protocol with dual-rail data encoding. In this method, the channel is implemented using three wires—two to encode one bit of data and one for the acknowledge. Figure 1 illustrates the protocol. The protocol steps through a sequence of states: (1) The appropriate data wire is sent high (the choice of wire determines the bit being transmitted); (2) Once the receiver has read the bit, the acknowledge is set high; (3) The data wires are reset; (4) The acknowledge is reset. Once this is complete, the next bit of data can be transmitted along the channel. In a protocol where the data wires (or "rails") initiate the handshake (as illustrated in Figure 1), the data wires are sometimes also called *request* signals since they initiate the handshake protocol.

A. Quasi Delay Insensitive Logic

Quasi delay-insensitive (QDI) logic is a commonly used asynchronous circuit family. It is the most conservative circuit family in terms of timing assumptions. A QDI circuit is one whose correct operation does not depend on gate delays or wire delays, except for certain branches known as isochronic forks [13], [11]. This minimal requirement from the underlying implementation technology means that QDI circuits are robust to process variations, voltage fluctuations, and temperature changes.

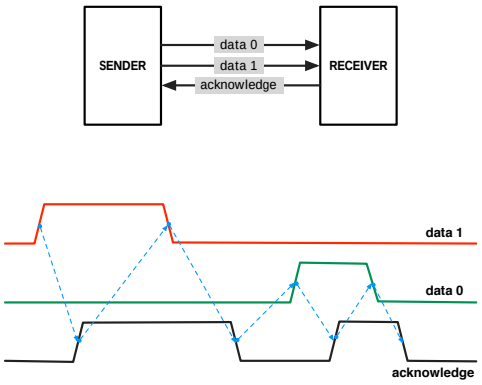

Fig. 1. A four-phase handshake protocol with dual-rail data. The arrows indicate causality—the target transition cannot occur until the source transition is complete.

The robustness of QDI circuits has been demonstrated in a variety of complex chips. Asynchronous microprocessor designs have shown that QDI asynchronous circuits are capable of both high-performance operation [14], as well as ultra low power consumption [3]. For the purposes of reconfigurable logic, we focus on QDI circuit families that demonstrate high-performance operation. In particular, we examine the circuits that were used to implement the high-performance pipelines in the MiniMIPS asynchronous processor [14].

Instead of adopting a fully general synthesis approach, the MiniMIPS processor design adopted a set of parameterized circuit *templates* that captured most of the components needed to implement the processor. The most common circuit template was the "pre-charge half-buffer" circuit (PCHB), illustrated in Figure 2 [14]. The gates labeled "C" are C-elements or consensus elements commonly found in asynchronous logic [17]. The core computation logic is implemented using n-transistors (labeled "computation"), and in the buffer example the computation is simply the identity function. In general, the n-transistor stack can be complex and implement arbitrary logic functions just like conventional synchronous dual-rail domino logic.

The region of the circuit labeled "completion logic" corresponds to the additional circuits needed to implement the handshaking protocol for an asynchronous implementation. The inverted acknowledge signals implement flow control and prevent data races. Various optimizations of this template are possible to change the trade-off between area, performance, and power. For example, it is possible to replace the NAND gate with an OR gate whose inputs are taken from the output of the $R0(i+1)$ inverter. This OR gate can then be shared with *next* stage of logic, thereby eliminating a significant fraction of the completion circuitry. This template was used for about 90% of the logic in the MiniMIPS asynchronous processor.

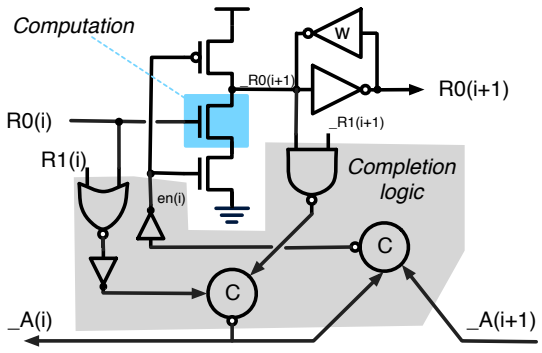

Fig. 2. A pre-charge half-buffer FIFO. The completion logic is shared among the two data rails, while the computation logic is replicated.

The entire microprocessor operated at a cycle time between 16 and 18 FO4 delays, even though it used QDI logic [14].

Figure 2 also illustrates the overhead associated with QDI asynchronous logic. The area used by the completion detection circuitry in the PCHB template is significant, and care must be taken to ensure that the additional area in fact translates to improvements in other metrics (e.g. performance). For instance, the low power SNAP/LE asynchronous processor primarily used other QDI circuits rather than the PCHB template, as performance was not a primary goal [7].

Circuits used to implement asynchronous logic contain both standard combinational gates and state-holding gates as illustrated in Figure 2. High-performance asynchronous circuits can contain complex pull-up and pull-down stacks. However, mapping these circuits to a reconfigurable fabric creates challenges beyond those present in conventional clocked circuits. In particular, the absence of a clock means that control signals in asynchronous logic must be hazard-free. Any mapping to a gate-level configurable fabric requires a method that preserves the hazard-free nature of the critical control signals. There has been some work that has developed hazard-free synthesis methods to map asynchronous logic to both standard FPGAs (e.g. [5], [22]) as well as FPGAs with support for asynchronous logic (e.g. [4]). However, the resulting mapping is not as efficient as conventional synchronous flows due to the overhead of supporting hazard-free mapping.

B. Pipelined Logic

The computation in asynchronous circuits is data-driven. When data arrives at the input ports of a component, the component activates and processes the data, possibly sending messages to other components. When such a sequence of components is connected in a linear array, the result is an asynchronous pipeline as shown in Figure 3.

An asynchronous pipeline has several intriguing properties that makes them differ from their synchronous counterparts. A pipeline containing a sequence of PCHB stages can implicitly latch data. For example, as soon as data arrives on the input and the output has been computed, "$\_A(i)$" can go low permitting the input data rails to reset. However, the output will not change until "$\_A(i+1)$" goes low (i.e. until the output data has been acknowledged). This is why such pipelines are considered *fine-grained*, because data items being operated on in the pipeline can be separated by very few gates (for a PCHB, four gate delays that include two inverters; a modification to the completion logic can enable data to be separated by as little as two gate delays). The data items flowing through an asynchronous pipeline are referred to as "tokens," as shown in Figure 3.

There is a difference between *physical* or circuit-level pipelining, and *logical* or architectural pipelining in an asynchronous system. By physical pipelining, we mean the introduction of additional fine-grained pipeline stages. Logical pipelining, on the other hand, is the introduction of a new data token in the pipeline. To illustrate this difference, consider a ring topology that constitutes an iterative computation. Even if the number of circuit-level pipeline stages are changed, this does not change the number of data tokens in the ring. In the synchronous case, adding a pipeline stage also adds a new data token (unless data is explicitly tagged with valid bits). In general, there is no obvious way to simply add an additional pipeline stage in a synchronous ring.

Changing the physical pipelining of an asynchronous system might modify the behavior of the system. However, it has been shown that under a very general set of conditions, adding physical pipelining does not affect the result of an asynchronous computation [12]; it only impacts its performance [23], [10]. These conditions were used to reason about the correctness of the MiniMIPS asynchronous processor [14].

Our approach to developing an architecture for asynchronous reconfigurable logic is to construct configurable bit-level pipelines. By introducing configuration memory, we can develop configurable pipeline blocks where the configuration memory controls the computation being performed. More importantly, the configuration memory cannot specify the precise set of gates used to implement the logic. This allows us to think of the reconfigurable logic in terms of asynchronous pipelines, rather than the detailed asynchronous

Fig. 3. A linear asynchronous pipeline, with the arrows denoting the data and acknowledge rails. The solid circles are data tokens, and the labels represent the function being computed. The pipeline computes $h(g(f(x))$, where x is the input data.

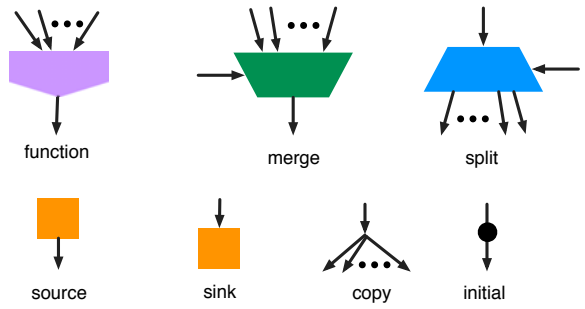

Fig. 4. List of dataflow building blocks for constructing asynchronous pipelines.

circuit implementation. A designer need not be concerned with whether or not the circuits are hazard-free—they will be hazard-free by construction. Instead of having control over routing individual wires, our architecture will route channels—wire bundles.

III. Asynchronous Dataflow Architecture

The asynchronous FPGA (AFPGA) architecture we developed is based on the notion of configurable asynchronous pipelines. As discussed, asynchronous pipelines can be thought of as blocks of logic connected in various topologies, where data tokens are transformed as they flow through the logic. Thus, the AFPGA can be thought of as implementing a reconfigurable static dataflow machine [2], [19].

A. Static Dataflow

There are a number of possible building blocks for static dataflow computations. The ones we chose for the AFPGA architecture are based on the commonly occurring blocks in finely pipelined asynchronous designs. Figure 4 shows the complete list of dataflow blocks, and we describe each of them below.

Function. The function block has N inputs and one output. This is the basic logic computation element. It receives a data token from each of its inputs, computes a function of the received input data, and produces the value as an output token.

Source. A source produces a stream of constant tokens on its output.

Sink. A sink consumes any tokens it may receive on its input.

Copy. A copy is used to implement the equivalent of signal fanout. It replicates every input token it receives on all of its outputs.

Initial. An initial block begins by producing a token on its output, and then after that simply copies any input token it receives to its output.

Merge. The merge block is a conditional block. It receives a data token from its control input (shown as a horizontal arrow in Figure 4). The value of this data token is used to select an input port. The input data on the selected input port (vertical arrows) will be sent to the output. No other input tokens are consumed.

Split. The split block is the dual of a merge. It receives a data token from its control input (shown as a horizontal arrow in Figure 4). The value of this data token is used to select an output port. The input data value (vertical arrow) will be sent to the selected output port.

Arbitrary computations can be constructed from these basic building blocks. As an example, consider an iterative multiply-accumulate dataflow graph that has an input port and an output port and some internal state x. The value produced on the output is $x + ab$, where a and b are the new inputs received. Finally, x is updated with the last value produced on the output. An asynchronous dataflow graph that implements this is shown in Figure 5(a). We can augment this with an additional input c that is used to reset x, by saying that if the value received on c is zero then the result is ab, and if the value is a one the result is $x + ab$. Figure 5(b) shows a modified dataflow graph that contains this functionality, illustrating the use of initial tokens, splits, merges, sources, and sinks. We have developed a systematic method for mapping computations to such dataflow graphs [21].

B. AFPGA Design

To evaluate these ideas, we developed two asynchronous dataflow FPGA architectures [18], [20]. We fabricated a small prototype AFPGA as well, and we discuss the architecture of the fabricated design.

The AFPGA architecture is an "island-style" architecture, consisting of an array of configurable logic blocks surrounded by programmable routing tracks. The routing tracks intersect at switch boxes that contain programmable connections among the tracks to enable configurable connectivity. The logic blocks connect to the routing tracks at connection boxes. Figure 6 shows a block diagram of an island style FPGA architecture.

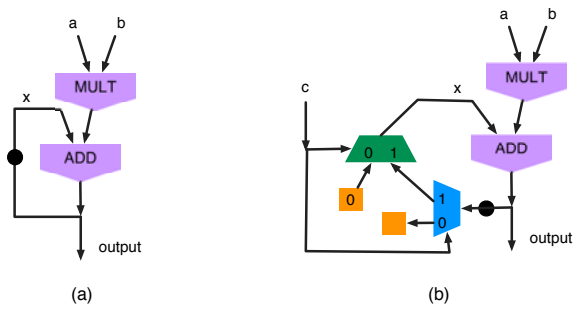

Fig. 5. Asynchronous dataflow graph implementing a multiply-accumulate function.

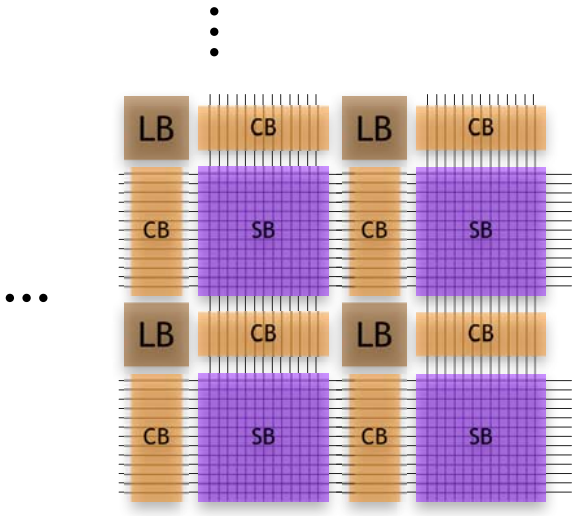

Fig. 6. An island-style FPGA architecture. "LB" are the logic blocks, "SB" are the switch boxes, and "CB" are connection boxes.

We use a very simple logic block that has four inputs and four outputs distributed on the north, south, east, and west edge of the block. As noted earlier, we route dual-rail channels rather than individual wires, so a single routing track corresponds to three wires.

The configuration logic for the AFPGA is conventional. The configuration bits are stored in a distributed SRAM; in fact we adopted a completely synchronous approach to the configuration logic so as to reduce the area overhead. For the small prototype, we used a shift-register approach for simplicity, even though this is not ideal in terms of area.

The asynchronous logic is initialized with a global reset signal. This reset signal is held high while the configuration bits are loaded into the AFPGA. Once the configuration bits have been set, the global reset is lowered and computation proceeds normally.

The logic block contains each of the dataflow building blocks shown in Figure 4. Each block is augmented with configuration bits that controls its functionality. The source block can be configured to either produce a zero or a one on its output. The initial block can specify the initial value of the token on its output. The copy block has a configurable number of destinations (ranging from one to four). The split and merge blocks are combined into a single configurable split/merge called a conditional unit. The conditional unit can be configured as a two-way split or a two-way merge. The function block is implemented as a configurable four-input lookup table. The inputs and outputs to each of the configurable dataflow blocks can be connected to either the north, south, east, or west inputs or outputs respectively. A block diagram of the logic block is shown in Figure 8. As is common in synchronous FPGA architectures, we augment the function computation to include a dedicated north-south carry chain in the AFPGA architecture so as to enable fast adder support. Also, the function unit has a programmable AND gate embedded internally so as to improve the performance of multipliers implemented using the logic block array. The entire AFPGA logic block is implemented using pipelined asynchronous logic. As an example, Figure 7 shows the logic pull-down stack of the programmable four-input lookup table. The inputs are pre-processed into a one-hot code that selects one of the sixteen possible output values. The enable signal (here labeled $\_pchg$) is generated by the completion logic.

The connection boxes connect the north, south, east, and west inputs and outputs to the routing tracks. These connections are made using pass-transistors, and is similar to existing synchronous architectures.

Pass transistors in the switch boxes control connectivity between various routing tracks. These switch boxes are a source of performance loss in a conventional FPGA architecture if signal paths are routed through a large number of switch boxes. In an asynchronous dataflow architecture, it is easy to establish that we can add circuit-level pipelining without affecting the result of the computation [12]. Therefore, to exploit this property, we introduce buffer stages in the switch boxes to pipeline the routing of data tokens along the interconnect.

Pipelining the interconnect two major consequences. First, the architecture has no long circuit-level signal paths. This helps with signal integrity even in the presence of long routes because the signals are always buffered. Second, the performance of the reconfigurable logic is enhanced in certain cases because we have local handshakes among circuit components that are near each other rather than through very long signal paths that propagate through multiple switch boxes.

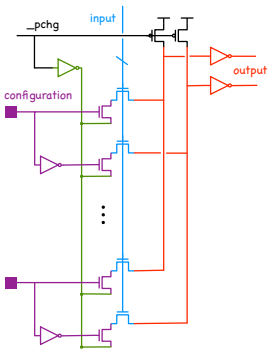

Fig. 7. Configurable pull-down stack for the four-input lookup table.

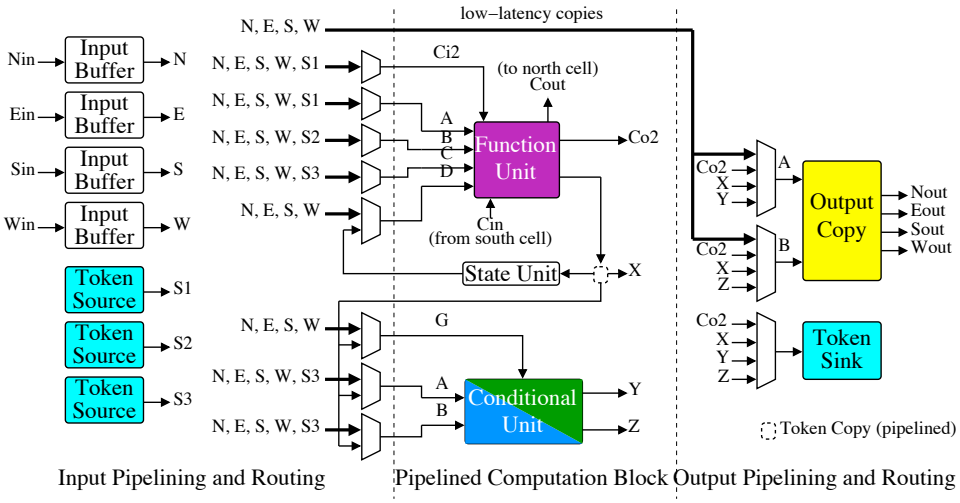

Fig. 8. Design of the logic block, showing the details of the connectivity possible between the inputs, outputs, and the dataflow blocks.

IV. Results and Discussion

We implemented a prototype AFPGA in TSMC's 0.18μm CMOS process through the MOSIS VLSI service. We also implemented a tool flow that enabled us to map computations to the AFPGA automatically, including logic synthesis, optimization, clustering and logic packing, and finally place-and-route. For most of these steps we used conventional algorithms [16].

The prototype AFPGA was a small 5×5 array due to limited die space. We were able to make measurements from the AFPGA array and calibrate our HSPICE simulations as well as back-annotate our switch-level simulations. We first report the measured peak performance of the prototype AFPGA, and then benchmark results using back-annotated simulations.

The AFPGA configuration used to measure the performance was the slowest local handshake cycle we found using HSPICE simulations. This "critical path" corresponded to two adjacent logic blocks communicating through the channel box (i.e. no intervening pipelining), with the two blocks both configured as lookup tables. The AFPGA had an on-chip frequency divider to simplify the measurement.

A. Measurement Data

Figure 9 shows the results of our measurements. At room temperature with a supply voltage of 1.8 V (nominal), we measured a throughput of 674 MHz. To our knowledge, the only other published configurable asynchronous circuit that was fabricated is the PCA-1 architecture, and they reported a peak throughput of 20 MHz in a $0.35\mu m$ CMOS process [8]. Even adjusting for feature size, our results show an order of magnitude improvement.

At nominal temperature (294K), the AFPGA was found to be functional when we varied the voltage continuously from 130 mV to 2.3 V. (We did not attempt to exceed 2.3 V.) The throughput ranged from 1.7 KHz at 130 mV, and increased with voltage to 870 MHz at 2.3 V. HSPICE simulations had led us to expect 700 MHz throughput at 1.8 V, and our measured performance of 674 MHz is in good agreement with simulation data.

We also made numerous measurements at high temperature (400K) and low temperature (77K, liquid nitrogen). Both high and low temperature measurements were made by mounting the AFPGA in a cryostat. The results are summarized in Figure 9, with the highest performance point of 1.12 GHz at 2.3 V at a temperature of 77K. As expected, the AFPGA stops operating at a higher voltage at low temperature due to a shift in the threshold voltage.

B. Benchmarking Results

Table I shows the result of several different benchmark applications. Since the AFPGA is aggressively pipelined and designed for high-throughput operation, most of our benchmarks are signal processing/arithmetic kernels. We normalize the performance of the benchmarks to the peak performance of the AFPGA. This peak performance is a function of the supply voltage and temperature.

There were three types of benchmarks considered. The first set of benchmarks (labeled "S") were existing synchronous benchmarks. These synchronous netlists were hand-translated into an asynchronous dataflow netlist. The second set of benchmarks (labeled "A") were from existing asynchronous designs that we had developed. These were also hand-translated to an asynchronous dataflow netlist. The final set of benchmarks

(labeled "Auto") were written in a C-like language and automatically synthesized into an asynchronous dataflow netlist. All the benchmarks were automatically placed and routed using VPR, a public domain place and route tool developed for clocked FPGAs [1].

For benchmarks where the computation is mostly feed-forward (e.g. adders, multipliers) we obtain performance numbers that are close to the peak performance of the AFPGA. This is expected for a custom implementation, but for reconfigurable logic there is the inefficiency that is introduced during place-and-route. The reason these benchmarks result in excellent utilization of the AFPGA is due to the asynchronous nature of the interconnect. As we said earlier, the slowest handshake in the AFPGA includes the four-input lookup table. The interconnect, on the other hand, is simply a FIFO stage with very simple logic. Our simulation results show that the interconnect frequency is significantly higher than what we consider the peak performance of the AFPGA. Therefore, inefficiencies introduced during place-and-route such as pipeline mismatches that could reduce the performance of an asynchronous system [23], [10] do not have a significant impact on throughput. The higher frequency of the interconnect enables it to "absorb" some of the mismatch introduced during place-and-route. Our simulation results show that we need a pipeline mismatch of six switch box stages before any performance loss is observed.

We have included one benchmark taken from the instruction fetch unit of an asynchronous microprocessor. This benchmark shows significantly degraded performance. An analysis of the benchmark showed that the reason for this performance loss was architectural. To illustrate this, consider the simple multiply-accumulate dataflow graph shown in Figure 5(a). Suppose we change the computation by increasing the complexity of the dataflow block that implements the "ADD" function. As the computation becomes more and more complicated, the logic delay along the loop that contains the variable x keeps increasing. As we keep increasing this delay, the throughput of the system is limited by the time it takes for the data token to travel around the loop. This limits the speed at which the computation in the dataflow graph can proceed. More generally, the total logic delay along any cycle divided by the number of initial tokens on the cycle limits the throughput of the pipeline [23], [10]. (The analysis is more complicated when split and merge blocks are part of the cycle.) We refer to this throughput limit as the *algorithmic limit* of the design. A careful analysis of the PC unit showed that it is algorithmically limited to a normalized performance of 0.45, explaining the low performance of the benchmark.

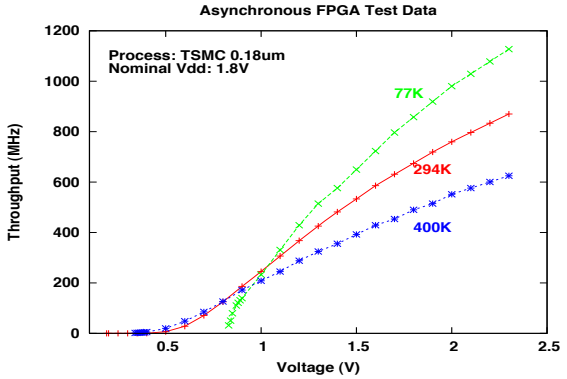

Fig. 9. AFPGA test data showing performance over a wide range of temperatures and voltages.

V. Summary

We described a pipelined asynchronous FPGA architecture that we believe is the highest performing asynchronous FPGA by an order-of-magnitude. The architecture is inherently pipelined, and the use of an asynchronous dataflow model enables interconnect pipelining for increased performance. The results from both measurement and benchmark simulation show that the architecture can operate at high throughput. Measurement results confirm that the architecture is extremely robust to variations in both operating voltage and temperature.

Acknowledgments

The work described in this paper is the result of several years of research by a number of graduate students. The AFPGA architecture was developed primarily by John Teifel. The automated synthesis flow was developed primarily by

TABLE I

Benchmark results. Performance is reported as normalized throughput, with 1.0 being the peak performance (674 MHz at nominal Vdd and room temperature). S = synchronous benchmarks that were converted to asynchronous; A = asynchronous benchmarks; Auto = benchmarks that were automatically synthesized.

Benchmark Name	Type	Perf.
16-bit adder	S/A	0.92
16-bit adder	Auto	0.92
12x12 Booth multiplier	A	0.96
8-bit scaling accumulator	S	0.95
8-tap 8-bit FIR filter	S	0.94
8-bit, 7-lag auto cross-correlator	S	0.89
systolic convolution	S	0.82
MIPS control unit	A	0.95
MIPS control unit	Auto	0.95
16-bit 6-tap LFSR	Auto	0.99
PC unit	Auto	0.42

Song Peng and David Fang. John Teifel was supported in part by an NSF graduate fellowship, and David Fang was supported in part by an NDSEG graduate fellowship. We would like to thank Prof. Clifford Pollock for providing us access to his cryostat.

REFERENCES

[1] V. Betz and J. Rose. VPR: A new packing, placement, and routing tool for FPGA research. In *Proc. International Workshop on Field Programmable Logic and Applications*, 1997.

[2] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[3] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. An ultra low power processor for sensor networks. In *Proceedings of the 11th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 27–36, Boston, MA, 2004.

[4] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.

[5] Q. T. Ho, J-B Rigaud, L. Fesquet, M. Renaudin, and R. Rolland. Implementing asynchronous circuits on LUT based FPGAs. In *Proc. International Conference on Field Programmable Logic and Applications*, 2002.

[6] D. L. How. A self clocked FPGA for general purpose logic emulation. In *Proc. of the IEEE Custom Integrated Circuits Conference*, 1996.

[7] Clinton Kelly IV, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor network asynchronous processor. In *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, pages 24–35, Vancouver, BC, 2003.

[8] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Shiozawa, M. Inamori, and K. Nagami. PCA-1: A fully asynchronous self-reconfigurable LSI. In *Proc. International Symposium on Asynchronous Circuits and Systems*, 2001.

[9] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of FPGA 2006*, 2006.

[10] Andrew Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[11] Rajit Manohar and Alain J. Martin. Quasi delay-insensitive circuits are turing complete. In *Proc. International Symposium on Asynchronous Circuits and Systems*, March 1996.

[12] Rajit Manohar and Alain J. Martin. Slack elasticity in concurrent computing. In *Proc. International Conference on the Mathematics of Program Construction*, 1998.

[13] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proc. Conference on Advanced Research in VLSI*, 1990.

[14] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak-Kwan Lee. The design of an asynchronous MIPS R3000. In *Proc. Conference on Advanced Research in VLSI*, pages 164–181, September 1997.

[15] R. Payne. Asynchronous FPGA architectures. *IEE Computers and Digital Techniques*, 143(5), 1996.

[16] Song Peng, David Fang, John Teifel, and Rajit Manohar. Automated synthesis for asynchronous fpgas. In *Proc. International Symposium on Field Programmable Gate Arrays*, 2005.

[17] Charles L. Seitz. *System Timing*, volume Introduction to VLSI Systems by Carver Mead and Lynn Conway, chapter 7. 1979.

[18] John Teifel and Rajit Manohar. Programmable asynchronous pipeline arrays. In *Proc. International Conference on Field Programmable Logic and Applications*, September 2003.

[19] John Teifel and Rajit Manohar. An asynchronous dataflow fpga architecture. *IEEE Transactions on Computers*, 53(11), 2004.

[20] John Teifel and Rajit Manohar. Highly pipelined asynchronous FPGAs. In *Proc. International Symposium on Field Programmable Gate Arrays*, February 2004.

[21] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proc. International Symposium on Asynchronous Circuits and Systems*, April 2004.

[22] C. Traver, R. B. Reese, and M. A. Thornton. Cell designs for self-timed FPGAs. In *Proc. of ASIC/SOC Conference*, 2001.

[23] Ted E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, 1991.

[24] Catherine G. Wong, Alain J. Martin, and Peter Thomas. An architecture for asynchronous fpgas. In *IEEE International Conference on Field-Programmable Technology*, December 2003.