Programmable Asynchronous Pipeline Arrays

John Teifel and Rajit Manohar

Computer Systems Laboratory
Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853, U.S.A.
{teifel,rajit}@csl.cornell.edu
http://vlsi.cornell.edu

Abstract. We discuss high-performance programmable asynchronous pipeline arrays (PAPAs). These pipeline arrays are coarse-grain field programmable gate arrays (FPGAs) that realize high data throughput with fine-grain pipelined asynchronous circuits. We show how the PAPA architecture maintains most of the speed and energy benefits of a custom asynchronous design, while also providing post-fabrication logic reconfigurability. We report results for a prototype PAPA design in a $0.25\mu m$ CMOS process that has a peak pipeline throughput of 395MHz for asynchronous logic.

1 Introduction

We present programmable asynchronous pipeline arrays (PAPAs) as a high-performance FPGA architecture for implementing asynchronous circuits. Asynchronous design methodologies seek to address the design complexity, energy consumption, and timing issues affecting modern VLSI design [10]. Since most experimental high-performance asynchronous designs (cf. [1, 13]) have been designed with labor-intensive custom layout, we propose the PAPA architecture as an alternative method for prototyping these asynchronous systems.

Previously proposed asynchronous FPGAs have shown that it is possible to port a clocked FGPA architecture to an asynchronous circuit implementation (cf. [2, 14]). However, in an asynchronous system, logic computations are not artificially synchronized to a global clock signal and hence we can explore a larger programmable design space. In this paper we present one such exploration into the design of high-performance pipelines suitable for programmable asynchronous systems.

The PAPA architecture is inspired by high-performance, full-custom asynchronous designs [1, 13] that use very fine-grain pipelines. Each pipeline stage contains only a small amount of logic (e.g., a 1-bit full-adder) and combines computation with data latching, such that explicit output latches are absent from the pipeline. This pipeline style achieves high data throughput and can also be used to design energy-efficient systems [15]. As a result, we use fine-grain asynchronous pipelines as the basis for our high-performance FPGA architecture.

Existing work in programmable asynchronous circuits has concentrated on three design approaches: (1) mapping asynchronous logic to clocked FPGAs (cf. [3, 5]), (2) asynchronous FPGA architectures for clocked logic (cf. [4, 16]), and (3) asynchronous FPGA architectures for asynchronous logic [2, 6, 8, 14]. The first approach suffers from an inherent performance penalty because of the circuit overhead in making a hazard-prone clocked FPGA operate in a hazard-free (the absence of glitches on wires) manner, which is necessary for correct asynchronous logic operation. Likewise, the second approach is not ideal because clocked logic does not behave like asynchronous logic and need not efficiently map to asynchronous circuits. The third approach runs asynchronous logic natively on asynchronous FPGA architectures. The work in this area has largely been modeled from existing clocked FPGA architectures, with the most recent running at an unencouraging 20MHz in $0.35\mu m$ CMOS [6].

In this paper we introduce the PAPA architecture as a new asynchronous FPGA that is designed to run asynchronous logic, yet differs from existing work because it is based on high-performance custom asynchronous circuits and is not a port of an existing clocked FPGA. The result is a programmable asynchronous architecture that is an order-of-magnitude improvement over [6]. Section 2 describes the asynchronous pipelines that our FPGA targets. In Section 3 we present the programmable asynchronous pipeline array architecture and in Section 4 describe its circuit implementation. Section 5 analyzes the performance of the PAPA architecture and Section 6 discusses logic synthesis results.

2 Asynchronous Pipelines

We design the logic that runs on PAPAs and other asynchronous systems as a collection of concurrent hardware processes that communicate with each other through message-passing channels [11]. Asynchronous pipelines can be constructed using such processes by connecting their channels in a FIFO configuration, where each pipeline stage consists of a single process. We refer to data items in a pipeline as *tokens* (i.e., the messages passed on channels).

Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens on channels. All PAPA channels use three wires, two data wires and one acknowledge wire, to implement a four-phase handshake protocol. The data wires encode bits using a dual-rail code, such that setting "wire-0" transmits a "logic-0" and setting "wire-1" transmits a "logic-1". The four-phase protocol operates as follows: the sender sets one of the data wires, the receiver latches the data and raises the acknowledge wire, the sender lowers both data wires, and finally the receiver lowers the acknowledge wire. The *cycle time* of a pipeline stage is the time required to complete one four-phase handshake.

In PAPA logic designs we enforce the following constraints on channels and processes: (1) no shared variables, (2) no shared channels, (3) no arbiters, and (4) the ability to add an arbitrary number of pipeline stages on a channel without changing the logical correctness of the original system. These system restrictions are reasonable for many high-performance asynchronous systems, including entire microprocessors [13], and in the rest of this paper we restrict our attention to asynchronous pipelines and circuits satisfying them.

A system that satisfies the aforementioned constraints is an example of a *slack-elastic* system [9] and has the nice property that a designer can locally add pipelining anywhere in the system without having to adjust the global pipeline structure. This property allows PAPA logic cells to be implemented with a variable number of pipeline stages and enables channels with long routes to be pipelined to improve performance. Any non-trivial clocked design will *not* be slack elastic, since changing local pipeline depths in a clocked system may require global retiming of the entire system. Adding high-speed retiming hardware support to a clocked FPGA incurs a significant register overhead [17], which the PAPA architecture can avoid because its logic cells are inherently pipelined and its channels are slack elastic.

Asynchronous (fine-grain) pipeline stages perform one or more of the following dataflow operations: (1) compute arbitrary logical functions, (2) store state, (3) conditionally receive tokens on input channels, (4) conditionally send tokens on output channels, and (5) copy tokens to multiple output channels. While strategies for implementing these pipeline operations in custom circuitry have been described in [7], the goal of the PAPA architecture is to implement these operations in a programmable manner.

Techniques for implementing operations 1 and 2 are well-known in both the clocked and asynchronous FPGA circuit literature (cf. [2, 14]). PAPAs have a *Function* unit to compute arbitrary functions and use feedback loops to store state. However, because operations 3, 4, and 5 involve tokens they are inherently asynchronous pipeline structures. The PAPA architecture provides a *Merge* unit to conditionally receive tokens, a *Split* unit to conditionally send tokens, and an *Output-Copy* unit to copy tokens. Since a clocked FPGA circuit has no concept of a token, it uses multiplexers, demultiplexers, and wire fanout to implement structures similar to operations 3, 4, and 5, respectively. The main difference is that these clocked circuits are destructive (i.e., wire values not used are ignored and overwritten on the next cycle), whereas an asynchronous circuit is non-destructive (i.e., tokens remain on channels until they are used).

3 The PAPA Architecture

The PAPA architecture is a RAM-based, coarse-grain FPGA design and consists of *Logic Cells* surrounded by *Channel Routers*. Figure 1a shows the basic PAPA logic cell and channel router configuration that is used in this paper. Logic cells communicate through 1-bit wide, dual-rail encoded channels that have programmable connections configured by the channel routers.

Logic Cell. The pipeline structure of a PAPA logic cell is shown in Figure 1b. The *Input-Router* routes channels from the physical input ports (Nin, Ein, Sin, Win) to the three internal logical input channels (A, B, C). This router is implemented as a switch matrix and is unpipelined. If an internal input channel is

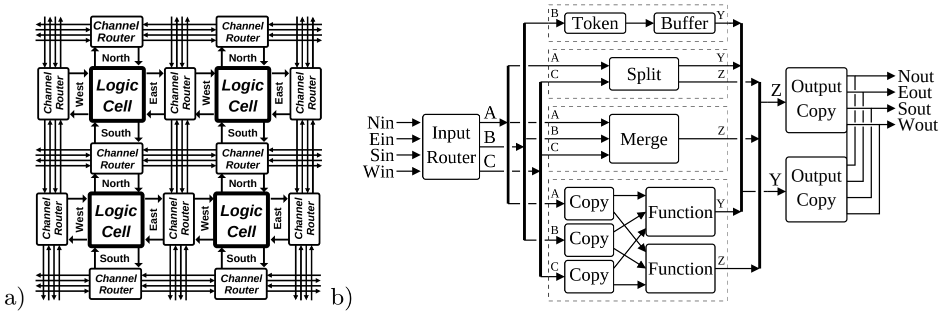


Fig. 1. PAPA architecture: (a) logic cell and channel router configuration, (b) pipeline structure of logic cell.

not driven from a physical input port, a token with a "logic-1" value is internally sourced on the channel (not shown in the figure). The internal input channels are shared between four logical units, of which only one unit can be enabled.

The logical units are as follows:

- *Function* Unit (2 pipeline stages): Two arbitrary functions of three variables. Receives tokens on channels (A, B, C) and sends function results on output channels (Y, Z). (e.g., this unit efficiently implements a 1-bit full-adder).
- *Merge* Unit (1 pipeline stage): Two-way controlled merge. Receives a control token on channel C. If the control token equals "logic-0" it reads a data token from channel A, otherwise it reads a data token from channel B. Finally, the data token is sent on channel Z.
- *Split* Unit (1 pipeline stage): Two-way controlled split. Receives a control token on channel C and a data token on channel A. If the control token equals "logic-0" it sends the data token on channel Y, otherwise it sends the data token on channel Z.
- *Token* Unit (2 pipeline stages): Initializes with a token on its output. Upon system reset a token (with a programmable value) is sent on channel Y. Afterwards the unit acts as a normal pipeline (i.e., it receives a token on channel B and sends it on channel Y). Unit is used for state initialization.

The *Output-Copy* pipeline stage copies result tokens from channels Y and Z to one or more of the physical output ports $(Nout, Eout, Sout, Wout)$ or sinks the result tokens before they reach any output port.

A PAPA logic cell uses 44 configuration bits to program its logic. The configuration bits are distributed as follows: 15 bits for the *Input-Router*, 4 bits for the logical unit enables, 16 bits for the *Function* unit, 1 bit for the *Token* unit, and 8 bits for the *Output-Copy* stages.

Unlike most existing FPGA architectures, PAPA logic cells do not have internal state feedback. Instead, state feedback logic is synthesized with an external feedback loop through an additional logic cell that is configured as a *Token* unit. This ensures that the state feedback loop is pipelined and operates at close to full throughput without adding additional area overhead to the logic cell to support an internal feedback path [7].

Channel Router. A PAPA channel router is an unpipelined switch matrix that *statically* routes channels between logic cells. PAPA channel routers route all channels on point-to-point pathways and all routes are three wires wide (necessary to support the dual-rail channel protocol). Each channel router has 12 channel ports (6 input and 6 output) that can route up to six channels. Four of the ports are reserved for connecting channels to adjacent logic cells and the remaining ports are used to route channels to other channel routers. To keep the configuration overhead manageable, a PAPA channel router does not allow "backward" routes (i.e., changing a channel's route direction by 180 degrees) and requires 26 configuration bits.

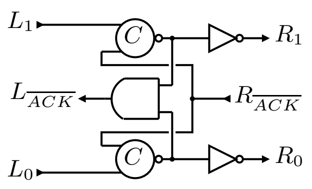
By examining numerous pipelined asynchronous logic examples, we empirically determined the PAPA logic cell and channel router interconnect topology (Fig.1a) as a good tradeoff between performance, routing capability, and cell area. We make no claims that it is the most optimal for this style of programmable asynchronous circuits and in fact it has several limitations. For example, it is not possible to directly route a channel diagonally on a 3x3 or larger PAPA grid using *only* channel routers (routing through one logic cell is required, which will improve performance for long routes). However, since most asynchronous logic processes communicate across short local channels we have not found this long-diagonal route limitation to be overly restrictive. More complicated channel routing configurations (such as those used in clocked FPGAs) could be adapted for the PAPA architecture, with the added cost of more configuration bits and cell area.

4 Pipelined Asynchronous Circuits

The asynchronous circuits we use are quasi-delay-insensitive (QDI). While they operate under the most conservative delay model that assumes gates and most wires have arbitrary delays [12], we believe QDI circuits to be the best asynchronous circuit style in terms of performance, energy, robustness, and area.

Although high-throughput, fine-grain QDI pipelined circuits have been used previously in several full-custom asynchronous designs [1, 13], the PAPA architecture is the first to adapt these circuits for programmable asynchronous logic. A detailed description on the design and behavior of this style of pipelined asynchronous circuits is in [7]. What follows is a summary of their salient features.

- **High throughput** – Minimum pipeline cycle times of ∼10-16 FO4 (fanout-of-4) delays (competitive with clocked domino logic).
- **Low forward latency** – Delay of a token through a pipeline stage is ∼2 FO4 delays (superior to clocked domino logic).
- **Data-dependent pipeline throughput** – Operating frequency depends on arrival rate of input tokens (varies from idle to full throughput).
- **Energy efficient** – Power savings from no extra output latch, no clock tree, and no dynamic power dissipation when the pipeline stage is idle.

a) 

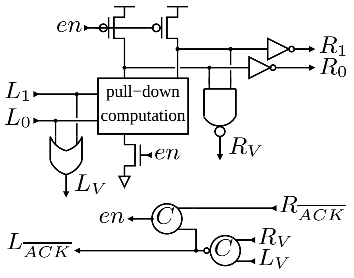
b) 

Fig. 2. Fine-grain pipelined asynchronous circuit templates: (a) weak-condition (dual-rail) pipeline stage, (b) precharge (dual-rail) pipeline stage.

Figure 2 shows the two pipeline circuit templates used in the PAPA architecture. L_0 and L_1 are the dual-rail inputs to the pipeline stage and R_0 and R_1 are the dual-rail outputs. We use inverted-sense acknowledge signals ($L_{\overline{ACK}}$, $R_{\overline{ACK}}$) for circuit efficiency. The weak-condition pipeline stage (Fig.2a) is most useful for token buffering and token copying, while the precharge pipeline stage (Fig.2b) is optimized for performing logic computations (similar to dual-rail clocked domino circuits). Since the weak-condition and precharge pipeline stages both use the dual-rail handshake protocol, they can be freely mixed together in the same pipeline. Weak-condition pipeline stages are used in the *Token* unit, *Output-Copy*, and in the copy processes of the *Function* unit. The *Split* unit, *Merge* unit, and the evaluation part of the *Function* unit use precharge pipeline stages.

A partial circuit used in the evaluation part of the *Function* unit is shown in Figure 3. A, B, and C are the input channels and $S0_d \ldots S7_d$ are the configurations bits that program the function result F_d, where d specifies the logic rail (e.g., d=0 computes F_0). As noted in [2], a function computation block of this style will suffer from charge sharing problems, which we solved using aggressive transistor folding and internal-node precharging techniques.

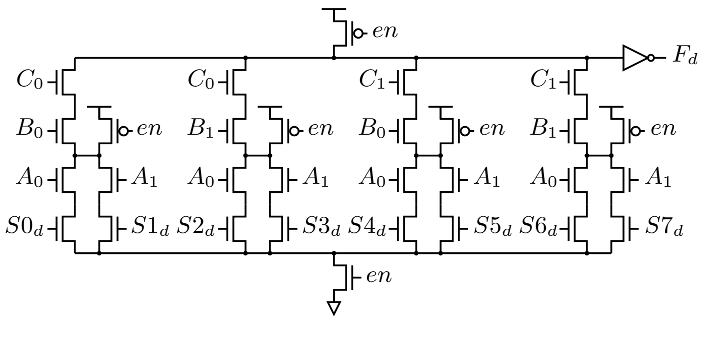


Fig. 3. One rail of a dual-rail precharge computation block for a 3-input function unit.

Physical Design. A prototype PAPA device has been designed and preliminarily layed out in TSMC's $0.25\mu m$ CMOS process (FO4 delay$\approx$120ps) available via MOSIS. An arrayable PAPA cell that includes one logic cell and two channel routers is 144 x 204 μm^2 (1200 x 1700 λ^2) in area, which is 50-100% larger than a conventional clocked FPGA cell but 100-200% smaller than the pipelined clock FPGA in [17]. To minimize cell area and simplify programming, configuration bits are programmed using JTAG clocked circuitry. The area breakdown for the architecture components is: function unit (14.4%), merge unit (2.5%), split unit (2.9%), token unit (2.6%), output copies (12.5%), configuration bits (37.7%), channel/input routers (18.2%), and miscellaneous (9.1%).

We have simulated our layout in SPICE (except for inter-cell wiring parasitics) and found the maximum inter-cell operating frequency for PAPA logic to be 395MHz. Internally the logical units can operate much faster, but are slowed by the channel routers. To observe this we configured the logical units to internally source "logic-1" tokens on their inputs and configured the *Output-Copy* stages to sink all result tokens (bypassing all routers). The results are: *Function* unit (498MHz, 26pJ/cycle), *Merge* unit (543MHz, 11pJ/cycle), *Split* unit (484MHz, 12pJ/cycle), and *Token* unit (887MHz, 7pJ/cycle). These measurements compare favorably to the pipelined clock FPGA in [17] that operates at 250MHz and consumes 15pJ/cycle of energy per logic cell. Our current work focuses on intelligently pipelining the channel routers to match the internal cycle times of the logical units and using improved circuit techniques to reduce the energy consumption of the PAPA logic cells.

5 Performance Analysis

The pipeline dynamics of asynchronous pipelines, due to their interdependent handshaking channels, are quite different from the dynamics of clocked pipelines. To operate at full throughput, a token in an asynchronous pipeline must be physically spaced across multiple pipeline stages, whereas in a clocked pipeline the optimum results when there is one token per stage [18]. The optimal number of pipeline stages, n_0, per token in an asynchronous pipeline is attained when $n_0 = \tau_0/l_0$, where τ_0 is the cycle-time of a pipeline stage and l_0 is its forward latency. For circuits used in the PAPA design, n_0 ranges from 5 to 8 pipeline stages per token (for pipelines without switches). If a pipeline has fewer stages per token than n_0, it will operate at a slower than maximal frequency but consume less energy [15]. On the other hand, if the pipeline has more stages per token than n_0, it will both operate slower and consume more energy than the optimal case.

To observe the pipeline dynamics when there are programmable switches between pipeline stages, we modeled a PAPA pipeline with a linear pipeline of n weak-condition pipeline stages that contain a variable number of routing switches between each pipeline stage. This model uses layout from the *Token* unit, has n_0=5, and measures all results from full SPICE simulations (including inter-cell wiring parasitics). This model gives an upper bound on the performance of

PAPA pipelines and shows the behavioral trends of inserting switches between fine-grain asynchronous pipeline stages.

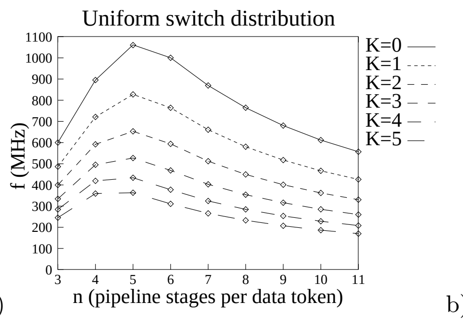
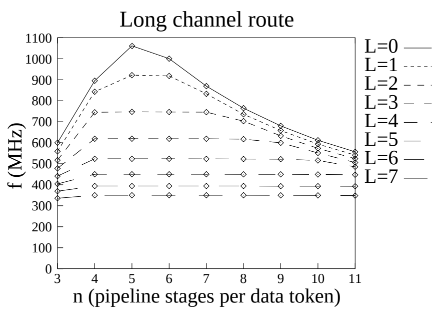
a)  b) 

Fig. 4. Maximum operating frequency curves for one token in a linear pipeline of n weak-condition pipeline stages, when (a) there are K routing switches between every pipeline stage and (b) one pipeline stage has a long route through L switches.

Figure 4a shows the maximum operating frequency curves for our model pipeline when there are K routing switches between every pipeline stage (K=0 is the "custom" case when there are no switches between stages). We observe that as K increases, n_0 decreases from 5 stages to 4 stages and the frequency curves shift downward because the switches uniformly increase the cycle time of every pipeline stage. Figure 4b shows the effect of one pipeline stage having a long route through L switches (when the other pipeline stages have no switches). In this case, the frequency curves flatten as L increases because the cycle time of the pipeline is mainly determined by the cycle time of the stage containing the long route (i.e., the long route behaves as a pipeline bottleneck).

In addition to decreasing their operating frequency, the energy consumption of asynchronous pipelined circuits also increases when routing switches are added between pipeline stages. To observe the energy effect of adding switches to asynchronous pipelines we use the $E\tau^2$ energy-time metric [13, 15]. E is the energy consumed in the pipeline per cycle and τ is the cycle time ($1/f$). Since E is proportional to V^2 and τ is proportional to $1/V$, to first order this metric is independent of voltage and provides an energy-efficiency measure to compare both low-power designs (low voltage) and high-performance designs (normal voltage). Figure 5 shows energy-efficiency curves for our model pipeline under the two switch scenarios examined earlier (lower values imply more energy efficiency).

The maximum operating frequency and energy-efficiency curves for a PAPA pipeline will look like a mixture of the two switch scenarios we investigated, since some pipeline stages will have no switches between them (channels inside of the logic cell) and some will have two or more (channels going through the input and channel routers). We have found that in synthesized PAPA logic there is at most six switches between logic cells, and on average two to four (including input routers). While the plots in this section show that (as expected) adding routing

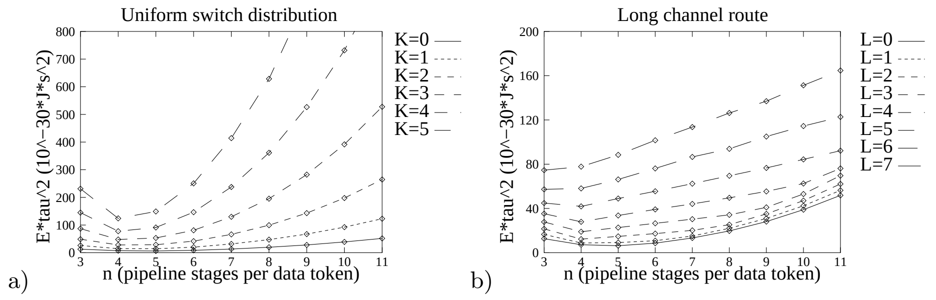


Fig. 5. Energy-efficiency curves for one token in a linear pipeline of n weak-condition pipeline stages, when (a) there are K routing switches between every pipeline stage and (b) one pipeline stage has a long route through L switches.

switches to full-custom, high-throughput pipelined circuits decreases both their speed and energy efficiency, they also show that there is still much performance remaining ($\approx 50\%$) to make them attractive for high-speed programmable asynchronous logic.

6 Logic Synthesis Results

High-level logic synthesis for PAPA designs borrows heavily from the formal synthesis methods we use to design full-custom asynchronous circuits [10]. We begin with a sequential description of the logic that is written in the `CHP` (Communicating Hardware Processes) hardware description language and apply (already existing) semantics-preserving program transformations to get a set of fine-grain concurrent `CHP` processes. Each of the resulting processes can be implemented in a single PAPA logic cell. The processes are then physically mapped onto PAPA logic cells. While this procedure is currently only semi-automated, it is not as tedious a task as for gate-level FPGAs. Finally, channels connecting logic cells are automatically routed and a configuration file generated.

We report SPICE simulations for several synthesized logic examples:

– N-bit ripple-carry adder (N logic cells) – Throughput of 292MHz, with a data input-to-output latency of 1.91ns, and a carry input-to-output propagation latency of 1.04ns per bit (the router was directed to minimize carry latency).
– Pipelined Booth encoded multiplier 1-bit cell (12 logic cells) – Throughput of 222MHz (original full-custom version ran at 190MHz in $0.8\mu m$ [1]).
– Register bit (5 logic cells) – Throughput of 272MHz, can read and/or write on same cycle.

7 Summary

We introduced a new high-performance asynchronous FPGA architecture. The architecture uses fine-grain asynchronous pipelines to implement a coarse-grain

FPGA and is suitable for prototyping pipelined asynchronous logic. Our preliminary circuit simulations demonstrate that PAPA logic systems are a promising alternative to full-custom asynchronous designs.

Acknowledgments

This research was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by an NSF CAREER award under contract CCR 9984299. John Teifel was supported in part by an NSF Graduate Research Fellowship.

References

1. Cummings, U.V., Lines, A.M., Martin, A.J.: An Asynchronous Pipeline Lattice-structure Filter. *Proc. Int'l Symp. on Asynchronous Circuits and Systems* (1994)
2. Hauck, S., Burns, S., Borriello, G., Ebeling, C.: An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers* **11**(3) (1994) 60-69
3. Ho, Q.T., et al.: Implementing asynchronous circuits on LUT based FPGAs. *Proc. 12th Int'l Conf. on Field Programmable Logic and Applications* (2002)
4. How, D.L.: A Self Clocked FPGA for General Purpose Logic Emulation. *Proc. of the IEEE 1996 Custom Integrated Circuits Conf.* (1996)
5. Keller, E.: Building Asynchronous Circuits with JBits. *Proc. 11th Int'l Conf. on Field Programmable Logic and Applications* (2001)
6. Konishi, R., et al.: PCA-1: A fully asynchronous self-reconfigurable LSI. *Proc. 7th Int'l Symp. on Asynchronous Circuits and Systems* (2001)
7. Lines, A.M.: *Pipelined Asynchronous Circuits*. M.S. Thesis, California Institute of Technology (1996)
8. Maheswaran, K.: *Implementing Self-Timed Circuits in Field Programmable Gate Arrays*. M.S. Thesis, U.C. Davis (1995)
9. Manohar, R., Martin, A.J.: Slack Elasticity in Concurrent Computing. *Proc. of the 4th Int'l Conf. on the Mathematics of Program Construction* (1998)
10. Manohar, R.: A Case for Asynchronous Computer Architecture. *Proc. of the ISCA Workshop on Complexity-Effective Design* (2000).
11. Martin, A.J.: Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4) (1986)
12. Martin, A.J.: The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conf. on Advanced Research in VLSI* (1990)
13. Martin, A.J., Lines, A., Manohar, R., et al.: The Design of an Asynchronous MIPS R3000. *Proc. of the 17th Conf. on Advanced Research in VLSI* (1997)
14. Payne, R.: Asynchronous FPGA architectures. *IEE Computers and Digital Techniques* **143**(5) (1996) 282-286
15. Teifel, J., Fang, D., Biermann, D., Kelly, C., Manohar, R.: Energy-Efficient Pipelines. *Proc. 8th Int'l Symp. on Asynchronous Circuits and Systems* (2002)
16. Traver, C., Reese, R.B., Thornton, M.A.: Cell Designs for Self-timed FPGAs. *Proc. of the 2001 ASIC/SOC Conf.* (2001)
17. Tsu, W., et al.: HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. *Proc. 7th Int'l Symp. on Field-Programmable Gate Arrays* (1999)
18. Williams, T.E.: *Self-Timed Rings and their Application to Division*. Ph.D. thesis, Stanford University (1991)