Highly Pipelined Asynchronous FPGAs

John Teifel and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, U.S.A.
{teifel,rajit}@csl.cornell.edu

ABSTRACT

We present the design of a high-performance, highly pipelined asynchronous FPGA. We describe a very fine-grain pipelined logic block and routing interconnect architecture, and show how asynchronous logic can efficiently take advantage of this large amount of pipelining. Our FPGA, which does not use a clock to sequence computations, automatically "self-pipelines" its logic without the designer needing to be explicitly aware of all pipelining details. This property makes our FPGA ideal for throughput-intensive applications and we require minimal place and route support to achieve good performance. Benchmark circuits taken from both the asynchronous and clocked design communities yield throughputs in the neighborhood of 300–400 MHz in a TSMC 0.25μm process and 500–700 MHz in a TSMC 0.18μm process.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuit**]: Types and Design Styles—*Gate Arrays, VLSI*; B.6.1 [**Logic Design**]: Design Styles—*Parallel Circuits*

General Terms

Design

Keywords

Asynchronous circuits, concurrency, correctness by construction, pipelining, programmable logic.

1. INTRODUCTION

We investigate the properties of pipelined FPGA architectures from the perspective of an asynchronous circuit implementation. Asynchronous circuits, whose temporal behavior is not synchronized to a global clock, allow logic computations to proceed as concurrently as possible. Concurrent asynchronous logic, in the context of pipelined FPGAs, enables a different architectural design point than what is possible in a more traditional clock-based design.

FPGA'04, February 22-24, 2004, Monterey, California, USA.

While the first FPGA was introduced by Xilinx in 1985 and the first asynchronous microprocessor was designed at Caltech in 1989 [16], very little work in the last two decades has successfully combined asynchronous and programmable circuit technology. Previously proposed asynchronous FPGA architectures [5, 9, 12, 18] based on clocked programmable circuits were not efficient at prototyping pipelined asynchronous logic and did not demonstrate significant advantages over clocked FPGAs. However, recent work by the authors has developed programmable asynchronous circuits that are inherently pipelined, efficiently implement asynchronous logic, and are competitive with clocked circuits [21].

In this paper we design a high-performance FPGA architecture that is optimized for asynchronous logic and features pipelined switch boxes, heterogeneous pipelined logic blocks, and pipelined early-out carry chains. The logic block contains a 4-input LUT and additional asynchronous-specific logic to efficiently implement asynchronous computations. The asynchronous FPGA interconnect is compatible with the `VPR` place and route tool [1], and we show that this FPGA architecture achieves high-throughput performance for automatically placed and routed asynchronous designs.

The main benefits of our pipelined asynchronous FPGA include:

- **Ease of pipelining:** enables high-throughput logic cores that are easily composable and reusable, where asynchronous handshakes between pipeline stages enforce correctness (not circuit delays or pipeline depths as in clocked circuits).
- **Event-driven energy consumption:** automatic shutdown of unused circuits (perfect "clock gating") because the parts of an asynchronous circuit that do not contribute to the computation being performed have no switching activity.
- **Robustness:** automatically adaptive to delay variations resulting from temperature fluctuations, supply voltage changes, and the imperfect physical manufacturing of a chip, which are increasingly difficult to control in deep submicron technologies.
- **Tool compatibility:** able to use existing place and route CAD tools developed for clocked FPGAs.

This paper is organized as follows. Section 2 reviews asynchronous logic and pipelined circuits. In Sections 3, 4, and 5 we describe the architecture, logic block, and interconnect of our pipelined asynchronous FPGA. Section 6 discusses

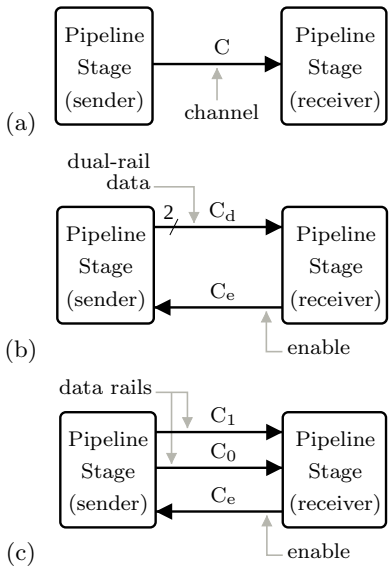

Figure 1: Dual-rail four-phase asynchronous pipelines: (a) channel abstraction, (b) handshake abstraction, and (c) signal level.

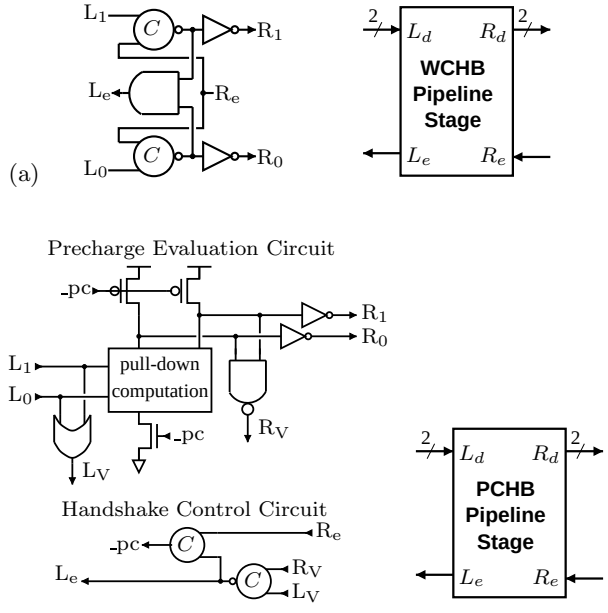

Figure 2: Asynchronous pipeline stages: (a) weak-condition half-buffer (WCHB) and (b) precharge half-buffer (PCHB).

asynchronous logic synthesis, Section 7 presents experimental results, and Section 8 reviews related work. We discuss future directions for asynchronous FPGA design in Section 9 and conclude in Section 10.

2. ASYNCHRONOUS LOGIC

The class of asynchronous circuits we consider in this paper are quasi-delay-insensitive (QDI). QDI circuits are designed to operate correctly under the assumption that gates and wires have arbitrary finite delay, except for a small number of special wires known as isochronic forks [15] that can safely be ignored for the circuits in this paper. Although we size transistors to adjust circuit delays, this only affects the performance of a circuit and not its correctness.

We design asynchronous systems as a collection of concurrent hardware processes that communicate with each other through message-passing channels. These messages consist of atomic data items called *tokens*. Each process can send and receive tokens to and from its environment through communication ports. Asynchronous pipelines are constructed by connecting these ports to each other using channels, where each channel is allowed only one sender and one receiver.

Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens on channels. Most of the channels in our FPGA use three wires, two data wires and one enable wire, to implement a four-phase handshake protocol (Figure 1). 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". A dual-rail encoding is a specific example of a 1ofN asynchronous signaling code that uses N wires to encode N values, such that setting the nth wire encodes data value n. The four-phase protocol operates as follows: the sender sets one of the data wires, the receiver latches the data and lowers the enable wire, the sender lowers all data wires, and finally the receiver raises the enable wire when it is ready to accept new data. The cycle time of a pipeline stage is the time required to complete one four-phase handshake. The throughput, or the inverse of the cycle time, is the rate at which tokens travel through the pipeline.

2.1 Pipelined Asynchronous Circuits

High-throughput, fine-grain pipelined circuits are critical to efficiently implementing logic in an asynchronous FPGA architecture. Fine-grain pipelines contain only a small amount of logic (e.g., a 1-bit adder) and combine computation with data latching, removing the overhead of explicit output registers. This pipeline style has been used in several high-performance asynchronous designs, including a microprocessor [17].

Figure 2 shows the two types of asynchronous pipelines that are used in our asynchronous FPGA.[1] A weak-condition half-buffer (WCHB) pipeline stage is the smaller of the two circuits and is useful for token buffering and token copying. The half-buffer notation indicates that a handshake on the receiving channel, L, cannot begin again until the handshake on the transmitting channel, R, is finished [11]. A precharge half-buffer (PCHB) pipeline stage has a precharge pull-down stack that is optimized for performing fast token computations. Since WCHB and PCHB pipeline stages have the same dual-rail channel interfaces, they can be composed together and used in the same pipeline.

Asynchronous pipelines can be used in programmable logic applications by adding a switched interconnect between its pipeline stages, which configures how their channels connect together. Figure 3 shows one such programmable interconnect, where connection boxes and switch boxes are used to connect channels between logic blocks. However, the throughput of an asynchronous pipeline is severely de-

[1]The *C-element* is an asynchronous state-holding circuit that goes high when all its inputs are high and goes low when all its inputs are low.

graded if its channels are routed through a large number of non-restoring switches. For example, a pipeline that contains four switches between pipeline stages is 59% slower than its full-custom implementation [21]. Using SPICE we measured only a 4% improvement on an asynchronous interconnect when these non-restoring switches were interspersed with restoring switches. As a result, we instead designed a pipelined interconnect architecture that uses pipelined switch boxes and guarantees at most two non-restoring switches between pipeline stages. This ensures a high throughput asynchronous FPGA and minimizes the performance lost due to using a switched interconnect.

2.2 Retiming and Slack Elasticity

A slack-elastic system [14] has the property that increasing the pipeline depth, or *slack*, on any channel will not change the logical correctness of the original system. This property allows a designer to locally add pipelining anywhere in the slack-elastic system without having to adjust or resynthesize the global pipeline structure of the system (although this can be done for performance reasons, it is not necessary for correctness). While many asynchronous systems are slack elastic, including an entire high-performance microprocessor [17], any non-trivial clocked design will *not* be slack elastic because changing local pipeline depths in a clocked circuit often requires global retiming of the entire system.[2]

To simplify logic synthesis and channel routing, we designed the pipelines in our asynchronous FPGA to be slack elastic. This allows logic blocks and routing interconnects to be implemented with a variable number of pipeline stages, whose pipeline depth is chosen for performance and not because of correctness. More importantly, in a pipelined interconnect, the channel routes can go through an arbitrary number of interconnect pipeline stages without affecting the correctness of the logic. The logic mapped to a slack-elastic FPGA need not be aware of the depth of the logic block pipelining or the length of its channel routes, since it will operate correctly regardless of how many pipeline stages exist. We call this property *self-pipelining* because the designer only specifies the functionality and connectivity of the logic, but does not explicitly specify the pipeline details.

A slack-elastic FPGA has an increased amount of flexility over a clocked FPGA, where pipeline depths must be deterministic and specified exactly for the logic to function correctly. For example, we will show that our pipelined asynchronous FPGA yields good performance without requiring banks of retiming registers, which are necessary for logical correctness in highly pipelined clocked FPGA architectures [23]. In addition, we use simple place and route tools that do not need knowledge of logic block pipeline depths, pipelined interconnects, or even asynchronous circuits!

3. FPGA ARCHITECTURE

Our asynchronous FPGA has an "island-style" architecture with pipelined logic blocks, pipelined switch boxes, and unpipelined connection boxes (Figure 3). A logic block has four inputs and four outputs, equally distributed on its north, east, south, and west edges. The routing tracks are

[2]This limitation in a clocked system can be lifted by adding valid bits to all data, which emulates an asynchronous handshake at the granularity of a clock cycle.

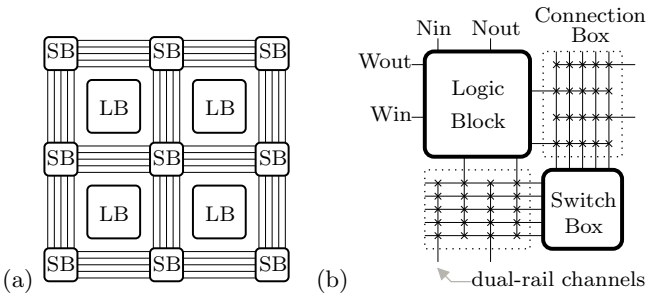

Figure 3: Asynchronous FPGA: (a) island-style architecture and (b) fully-populated connection boxes.

dual-rail asynchronous channels and span only single logic blocks. The connection boxes are fully-populated and the switch boxes have Xilinx-style routing capabilities.

Configuration of our asynchronous FPGA is done using clocked SRAM-based circuitry. This allows us utilize the same configuration schemes used in clocked FPGAs. In this design we constructed the simplest configuration method using shift-register type configuration cells that are connected serially throughout the chip. During programming the asynchronous portion of the logic is held in a passive reset state while the configuration bits are loaded. The configuration clocks are disabled after programming is complete and the asynchronous logic is enabled.

4. PIPELINED LOGIC BLOCK

The pipelined logic block architecture for our asynchronous FPGA is shown in Figure 4. The main parts of the logic block, which will be discussed in detail in the next sections, are the input pipelining and routing, the pipelined computation block, and the output pipelining and routing.

4.1 Input Buffers

The input buffers are single WCHB pipeline stages that buffer input tokens from the connection box switches and the mux switches used to route the logic block inputs to the function unit, conditional unit, or output copy parts of the logic block. Upon system reset, the input buffers can optionally initialize the internal logic block channels (N,E,S,W) with tokens, which is often necessary in an asynchronous system to setup token-ring pipelines and other state-holding pipelines. Three constant token sources can also be routed to any of the function or conditional unit inputs.

4.2 Function Unit

The function unit shown in Figure 5 can implement any function of four variables and provides support for efficient carry generation in addition and multiplication applications. While this unit is logically based on the configurable logic block used in the Xilinx Virtex™ FPGA [24], our design is significantly different because it is pipelined and fully asynchronous. The main pipeline of the function unit consists of an address decoder, a lookup table (LUT), and an XOR output stage. A secondary pipeline, which is optionally configured when carry computations are necessary, examines the function unit inputs and generates the carry-out. The output of the function unit can be copied to the output copy, to the state unit, and/or to the conditional unit.

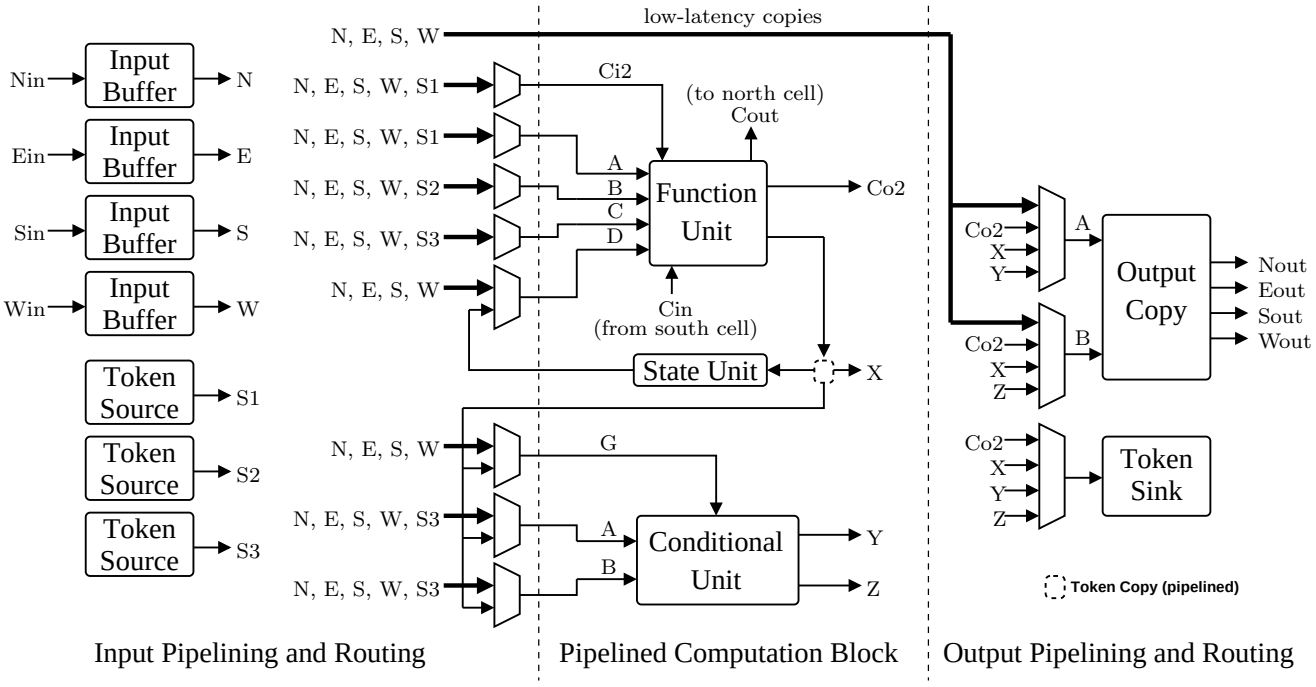

Figure 4: Pipelined asynchronous FPGA logic block.

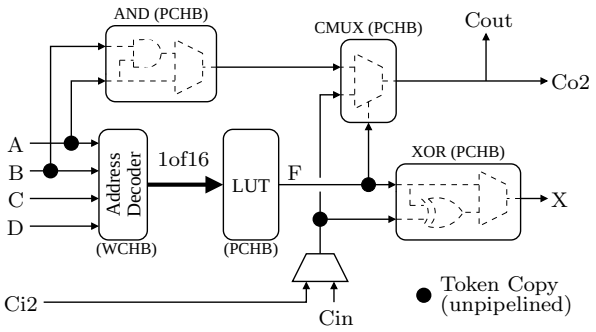

Figure 5: Function unit with carry support.

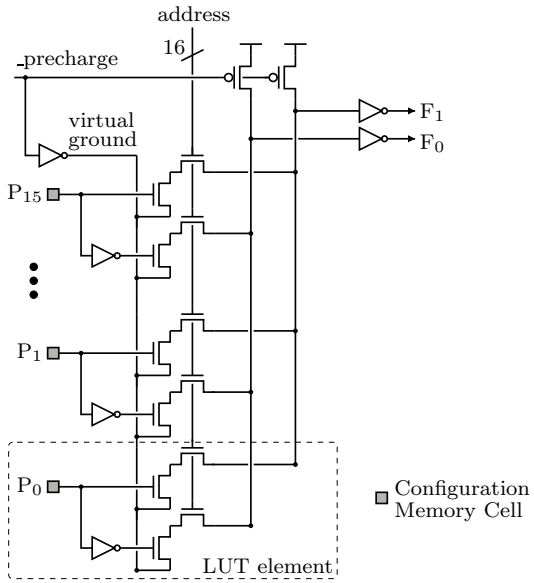

Figure 6: Asynchronous LUT circuit.

The address decode stage reads the four function unit inputs and generates a 1of16 encoded address. The 1of16 encoding on the address channel simplifies indexing into the LUT and is the only asynchronous channel in our design that is not dual-rail encoded. The asynchronous four-input LUT circuit shown in Figure 6 uses a modified PCHB-style pipeline circuit (the handshake control circuit is not shown). Since there are sixteen memory elements in a four-input LUT and the 1of16-encoded address guarantees no sneak paths will occur in the pull-down stack, the asynchronous LUT circuit can use a virtual ground generated from the "_precharge" signal instead of the foot transistors used in a normal PCHB pipeline stage. This reduces the area required for the LUT and makes it faster than our previous asynchronous LUT circuit [21] because it has fewer series transistors and does not need internal-node precharging circuitry to compensate for charge sharing issues. The address decode and function lookup table circuits are the throughput-limiting part of our pipelined asynchronous FPGA, and we sized the transistors in these circuits to operate at 400 MHz in TSMC 0.25μm.

When the function unit is configured to perform carry computations, the XOR stage outputs the logical "xor" of the LUT output and the carry-in to produce the function unit's output. A carry computation would require two LUTs if the function unit did not have this extra built-in XOR stage. However, adding an additional pipeline stage to the function unit increases the input-to-output latency on its critical pipeline path. To evaluate the performance impact

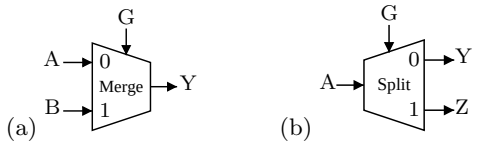

Figure 7: Conditional unit configurations: (a) 2-way merge process and (b) 2-way split processes.

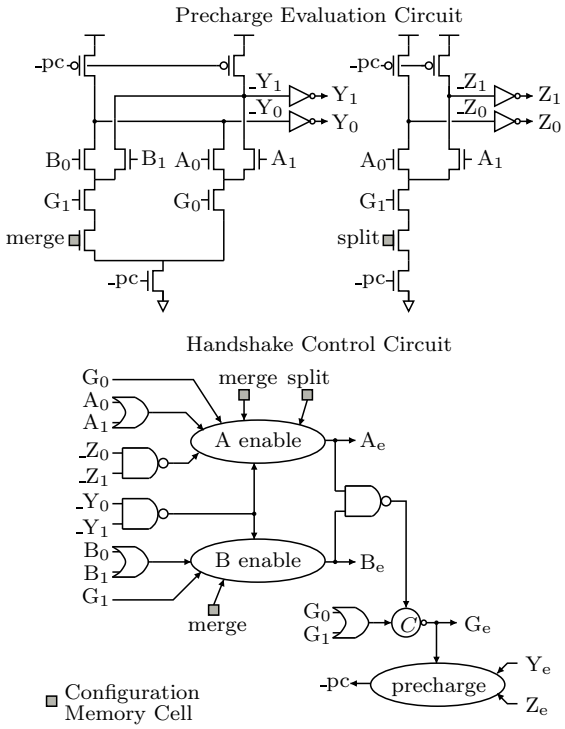

Figure 8: Conditional unit circuit.

of this pipeline stage we designed an unpipelined version of the XOR stage that has minimal latency when it is configured not to use the carry-in. We measured the performance of the unpipelined XOR on typical pipelined designs mapped to our asynchronous FPGA architecture and compared them against the performance of the proposed pipelined XOR stage. For linear pipeline designs the pipelined XOR stage was 20% faster in throughput, but with token-ring pipelines the unpipelined XOR stage was 8% faster. By using the pipelined XOR stage we chose to trade slightly decreased token-ring performance for greatly increased linear pipeline performance.

The carry pipeline in the function unit is used to create carry chains for arithmetic computations. The AND stage either propagates the A input to the CMUX stage for addition carry chains or outputs the logical "and" of the A and B inputs to the CMUX stage for multiplication carry chains. Depending on the value of the LUT output, the CMUX stage generates the carry-out by selecting between the carry-in and the output of the AND stage. When the carry-out does not depend on the carry-in and the CMUX has received a token from the AND stage, it does not need to wait for the carry-in token to generate the carry-out token. This early-out CMUX allows the construction of asynchronous ripple-carry adders that exhibit average-case behavior in their carry chains. In contrast, clocked ripple-carry adders must tolerate the worst-case behavior of their carry chains. Carry chains can be routed using the normal interconnect or using low latency carry channels that run vertically south-to-north between adjacent vertical logic blocks.

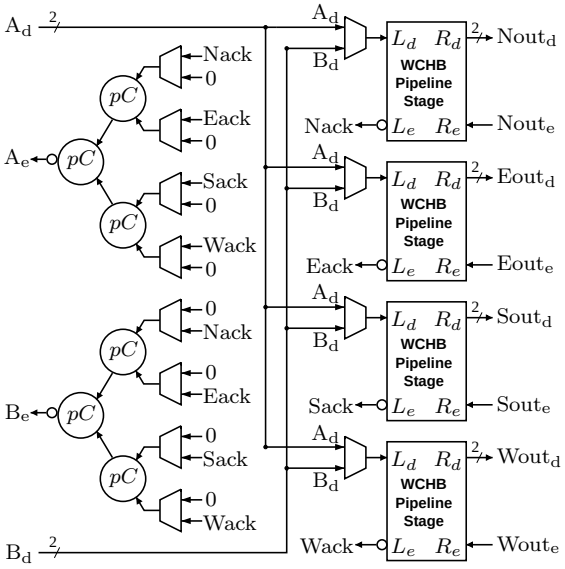

Figure 9: Output copy circuit.

4.3 Conditional Unit

The conditional unit allows logic processes to conditionally send tokens on output channels *or* to conditionally receive tokens on input channels. The conditional unit is heavily used in control dominated asynchronous circuits and less so in other computation circuits.

Figure 7 shows the two possible configurations for the conditional unit: a two-way controlled merge process or a two-way controlled split process. The merge process is a conditional input process that operates by reading a "control" token on the G channel, conditionally reading a "data" token from A (if the "control" token equals zero) or from B (if the "control" token equals one), and finally sending the "data" token on the Y output channel. Likewise, the split process is a conditional output process that operates by reading a "control" token on the G channel, reading a "data" token on the A channel, and then conditionally sending the "data" token on Y (if the "control" token equals zero) or on Z (if the "control" token equals one). The asynchronous merge and split processes are similar to clocked multiplexers and demultiplexers, which operate on signals instead of tokens.

The conditional unit is only slightly larger than a standard two-way asynchronous merge process. Figure 8 shows the circuit used to implement the conditional unit. We note that the handshake completion circuitry in the conditional unit is much more complex than for a normal PCHB-style pipeline stage and is the reason why we cannot simply make all channels conditional in the logic block. By using the same precharge transistor stacks and handshake control circuits for both the merge and split processes, the area of the conditional unit is approximately 40% smaller than our previous conditional unit design [21] that used separate split and merge circuits.

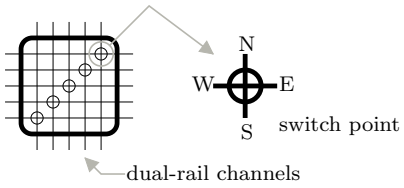

Figure 10: Switch box.

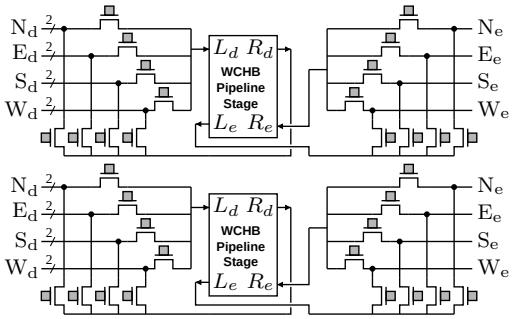

Figure 11: Pipelined asynchronous switch point.

4.4 Output Copy

The output copy pipeline stage performs both token copying and token routing. This stage is used to copy result tokens from the logic block, which arrive on channels A and B, and statically route them to the four output ports (Nout,Eout,Sout,Wout) of the logic block. The output copy stage can support at most two concurrent copy processes (with input channels A and B respectively) that copy tokens to 1, 2, 3, or 4 of the output ports, with the restriction that the output ports cannot be shared between the two copy processes. Sharing output ports is not allowed because the same channel would have two token senders, which is equivalent to having two competing drivers in a clocked circuit.

As shown in Figure 4, the function unit and conditional unit can both source tokens to the output copy pipeline stage. The input buffers can also source tokens to the output copy, which allows the logic block to support low-latency copy processes that bypass the function and conditional units. However, since the output copy has only two input channels, the output copy can handle at most two token streams. Result tokens that are not needed by other logic blocks should be routed to the token sink instead of the output copy.

The circuit that implements the output copy, shown in Figure 9, is approximately 35% smaller than our previous output copy design [21] because it uses half the number of WCHB pipeline stages. It is important to note that after the muxes in the output copy circuit have been configured by the configuration memory, the output copy circuit operates as two completely independent and concurrent copy processes.

4.5 State Unit

The state unit is a small pipeline built from two WCHB stages that feeds the function unit output back as an input to the function unit, forming a fast token-ring pipeline. Upon system reset, a token is generated by the state unit to initialize this token-ring. This state feedback mechanism is superior to our previous design [21], where all feedback token rings needed to be routed through the global interconnect.

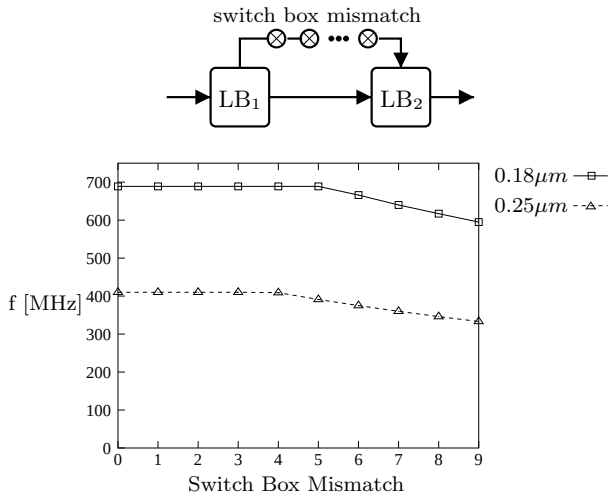

Figure 12: Performance of linear pipelines.

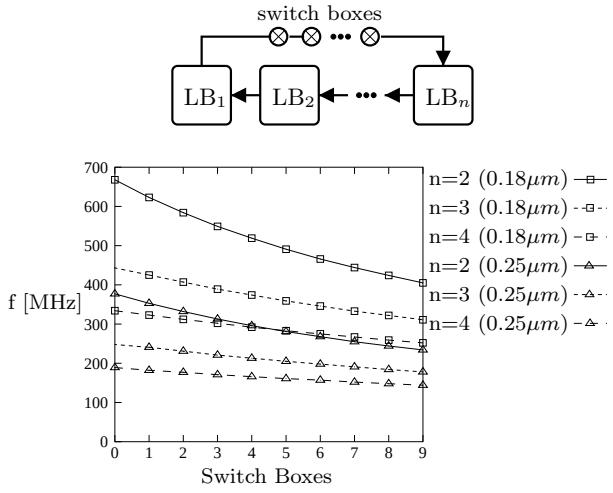

Figure 13: Performance of token-ring pipelines.

5. PIPELINED INTERCONNECT

We built pipelined switch boxes into our asynchronous FPGA interconnect to ensure high token throughput between communicating logic blocks. Figure 11 shows a pipelined switch point in our asynchronous "Xilinx-4000 style" switch box, where wire segments only connect to each other when they are in the same track. This is similar to the switch points used in high-speed clocked FPGAs [23].

In our asynchronous interconnect, a channel connecting two logic blocks can be routed through an arbitrary number of pipelined switch boxes without changing the correctness of the resulting logic system since our asynchronous FPGA is slack elastic. However, the system performance can still decrease if a channel is routed through a large number of switch boxes. To determine the sensitivity of channel route lengths on pipelined logic performance, we varied the number of switch boxes along a route for typical asynchronous pipelines.

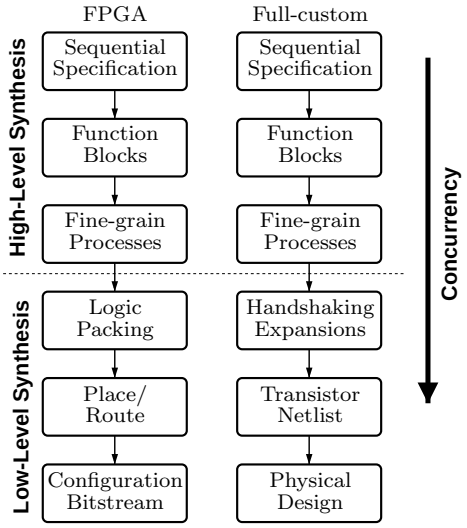

Figure 14: Asynchronous synthesis flow.

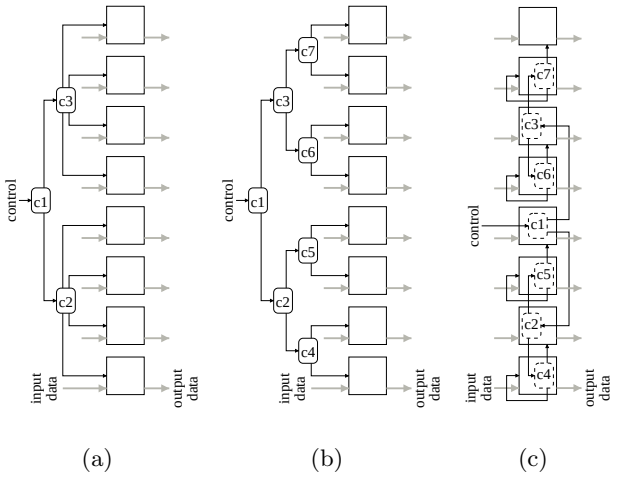

Figure 15: Copying control tokens to wide datapaths: (a) minimum latency, (b) cluster-friendly, and (c) integrated copy trees.

Figure 12 shows the performance of a branching linear pipeline using pipelined switch boxes and the FPGA logic blocks configured as function units. Tokens are copied in logic block LB_1, travel through both branches of the pipeline, and join in logic block LB_2. Since the speed of the function unit is the throughput-limiting circuit in our asynchronous FPGA design, this pipeline topology gives an accurate measure for linear pipeline performance. The frequency curve shows that an asynchronous pipelined interconnect can tolerate a relatively large pipeline mismatch (4–5 switch boxes) before the performance begins to gradually degrade. This indicates that as long as branch pathways have reasonably matched pipelines we do not need to exactly balance the length of channel routes with a bank of retiming registers. In contrast, in clocked FPGAs it is necessary for correctness to exactly retime synchronous signals routed on pipelined interconnects using banks of retiming registers [23].

Figure 13 shows the performance trends for token-ring pipelines using the FPGA logic blocks configured as function units, such that one token is traveling around the ring. For pipelined interconnects, adding switch box stages to a token-ring will decrease its performance and indicates that the routes of channels involved in token-rings should be made as short as possible. The frequency curves in Figure 13 are worst-case because all the logic blocks were configured with their function units enabled, requiring a token to travel through five pipeline stages per logic block. If the logic blocks were instead configured to use the conditional unit or the low-latency copies, then the token-ring performance would approach the performance of a linear pipeline because a token would travel through fewer pipeline stages. In addition, token rings used to hold state variables can often be implemented using the state unit, which localizes the token ring inside the logic block and has the same throughput as a linear pipeline.

6. LOGIC SYNTHESIS

Logic synthesis for an asynchronous FPGA follows similar formal synthesis methods to those used in the design of full-custom asynchronous circuits [13], whose steps are shown in Figure 14. We begin with a high-level sequential specification of the logic and apply semantics-preserving program transformations to partition the original specification into high-level concurrent function blocks. The function blocks are further decomposed into sets of fine-grain, highly concurrent processes that are guaranteed to be functionally equivalent to the original sequential specification. To maintain tight control over performance, this decomposition step is usually done manually in full-custom designs. However, for FPGA logic synthesis we are developing a *concurrent dataflow decomposition* [20] method that automatically produces fine-grain processes by detecting and removing all unnecessary synchronization actions in the high-level logic.

The resulting fine-grain processes are small enough (i.e., bit-level) to be directly implemented by the logic blocks of our asynchronous FPGA. Currently, the logic packing step, which clusters multiple fine-grain processes into a single physical logic block, is done manually and the place/route and configuration steps are done automatically. Observe that the asynchronous FPGA synthesis flow avoids the low-level, labor-intensive, and asynchronous-specific steps of the full-custom synthesis flow (e.g., handshaking expansions, transistor netlist generation, and physical design).

Logic computations in asynchronous designs behave like fine-grain static dataflow systems [3], where a token traveling through an asynchronous pipeline explicitly indicates the flow of data. Channel handshakes ensure that pipeline stages consume and produce tokens in sequential order so that new data items cannot overwrite old data items. In this dataflow model, data items have one producer and one consumer. Data items needed by more than one consumer are duplicated by copy processes that produce a new token for every concurrent consumer. In contrast, clocked logic uses a global clock to separate data items in a pipeline, which allows data items to fan out to multiple receivers because they are all synchronized to the clock. Furthermore, the default behavior for a clocked pipeline is to overwrite data items on the next clock cycle, regardless of whether they were actually used for a computation in the previous cycle.

To synthesize logic for our asynchronous FPGA, a designer only needs to understand how to program for a token-based dataflow computation model and is not required to know the underlying asynchronous pipelining details. This type of asynchronous design, unlike clocked design, separates logical pipelining from physical pipelining. An FPGA application that is verified to functionally operate at the dataflow level is guaranteed to run on any pipelined implementation of our asynchronous FPGA. For example, an application that operates correctly with the FPGA described in this paper will also work with an FPGA that contains twice as much pipelining, without requiring the retiming conversions necessary for clocked applications.

The main difference in logic density between asynchronous and clocked logic is the overhead of copying tokens to multiple receivers. For logic with small copies to four or fewer receivers, the output copy in the logic block provides zero overhead copy support. However, logic requiring wide copies suffers from the overhead of having to construct a copy tree with additional logic blocks. A typical example of a copy tree, shown in Figure 15, is when a control token needs to be copied to each bit of a wide datapath. Often the latency of a control token is not critical and the copy tree can be constructed using two-way copies as shown in Figure 15(b). This potentially allows the copy tree to be integrated with the datapath logic and have zero overhead in terms of logic block counts. The copy overhead for the benchmarks used in this paper ranged from 20%–33% compared with equivalent synchronous implementations, although we did not attempt to aggressively integrate copy trees.

7. EXPERIMENTAL RESULTS

We laid out our asynchronous FPGA design using conservative SCMOS design rules that obey rules for both TSMC $0.18\mu m$ and TSMC $0.25\mu m$, which utilizes five layers of metal. The area of an arrayable tile is approximately $2.6M\lambda^2$ ($37440\mu m^2$ in $0.25\mu m$ and $21060\mu m^2$ in $0.18\mu m$) with routing tracks of width four. The computation units consume 40% of the total area, followed by the switch box (25%), input pipelining (16%), output pipelining (10%), and connection boxes (8%). Using SPICE we obtained delay values from the extracted layout and used these to back-annotate an asynchronous switch-level simulator of our FPGA that allowed us to accurately evaluate the performance of benchmark circuits.

The peak operating frequency of our asynchronous FPGA is 400 MHz in $0.25\mu m$ and 700 MHz in $0.18\mu m$. Our design is about twice as fast as commercial FPGAs, but approximately two times larger. However, our FPGA is half the size of a highly pipelined clock FPGA [23] and of comparable performance (250 MHz in $0.4\mu m$). Although our performance is 36% slower than hand-placed benchmarks on a "wave-steered" clock FPGA [19], it is almost twice as fast for automatically-placed benchmarks.

The peak energy consumption is 18 pJ/cycle in $0.25\mu m$ and 7 pJ/cycle in $0.18\mu m$ for a logic block configured as a four-input LUT.[3] In addition, the interconnect energy consumption per switch point is 3 pJ/cycle in $0.25\mu m$ and 1 pJ/cycle in $0.18\mu m$. Due to the large amount of pipelining in our current design, the energy consumption of our asynchronous FPGA is higher than the 26pJ/cycle of our previous design [21] in $0.25\mu m$. However, most of the transistors in our layout are over-sized and we have yet to optimally size them for the peak operating frequency, which will make the energy consumption more manageable. Since QDI circuits do not glitch and consume power only when they contain tokens, an asynchronous FPGA automatically has perfect clock gating without the overhead associated with dynamic clock gating in synchronous designs. As a result of this event-driven energy consumption, the power consumption of an asynchronous FPGA is proportional to the number of tokens traveling through the system.

To evaluate the performance of our asynchronous FPGA, we synthesized a variety of benchmark circuits that were used in previous clocked and asynchronous designs. The benchmarks in Table 1 are classified into three categories: arithmetic, signal processing, and random circuits. Approximately half of these benchmarks were optimized for FPGA implementations (e.g., scaling accumulator, FIR filter core,[4] and cross-correlator) and the other half were not developed specifically for FPGAs (e.g., booth multiplier, systolic convolution, and write-back unit).

We placed and routed these benchmarks using `VPR` [1] and used its default settings, except to disable timing-driven optimizations because they assume a clocked architecture. Since `VPR` does not support macro placement, we hand placed several benchmarks so that they could use the fast south-to-north carry chains. While we made no effort to equalize branch mismatches or to minimize routes on token-ring pipelines and other latency-critical channels, most of the benchmarks performed within 75% of the FPGA's maximum throughput. In contrast, pipelined clock FPGAs require substantial CAD support beyond the capabilities of generic place and route tools to achieve such performance [19, 23].

Our asynchronous FPGA inherently supports bit-pipelined datapaths that allow datapath bits to be computed concurrently, whereas clocked FPGAs implement aligned datapaths that compute all datapath bits together. However, due to bit-level data dependencies and aligned datapath environments (e.g., memories or off-chip I/O) a bit-pipelined datapath in an asynchronous FPGA will behave in-between that of a fully concurrent bit-pipelined datapath and a fully aligned datapath. To evaluate such a datapath fairly, we measured the performance of a 16-bit adder with a fully bit-pipelined environment and a fully aligned environment. Since the fully aligned adder datapath will exhibit data-dependent carry chain behavior, we reported the best-case and worst-case throughputs in Table 1. In the worst case, a fully aligned adder is 7% slower than a fully bit-pipelined adder using fast carry chains and 47% slower using slow carry chains.

8. RELATED WORK

Existing asynchronous FPGA architectures [5, 9, 12, 18] have been based largely on programmable clocked circuits. These FPGAs are limited to low-throughput logic applications because their asynchronous pipeline stages are either built up from gate-level programmable cells (e.g., [5]) or

[3]The absolute energies reported by our asynchronous SPICE simulator have not been validated and are suspected to be higher than the actual values, however, we deem that the relative energies are reasonable to compare against each other (in the same manufacturing process).

[4]Contains only the serial adders, LUT-based multipliers, and scaling accumulator of the FIR filter [4].

Table 1: Benchmark statistics for automatically placed and routed asynchronous designs.

Design	LBs	Proc-esses	Proc./ LBs	Function	Condi-tional	Copy	**Throughput** (MHz) $0.25\mu m$	$0.18\mu m$
Arithmetic circuits								
16-bit adder (bit-pipelined datapath)	16	16	1.0	100%	0%	0%	395	674
16-bit adder (aligned datapath)	16	16	1.0	100%	0%	0%	211/323	359/535
*16-bit adder (aligned datapath)	16	16	1.0	100%	0%	0%	369/377	641/651
4x4 array multiplier	36	36	1.0	78%	0%	22%	321	550
*4x4 array multiplier	21	21	1.0	62%	0%	38%	371	637
12x12 booth multiplier [2]	432	648	1.5	44%	0%	56%	395	674
8-bit scaling accumulator [4]	22	34	1.5	44%	21%	35%	352	607
*8-bit scaling accumulator [4]	15	24	1.6	50%	29%	21%	384	665
Signal processing circuits								
8-tap symmetric 8-bit FIR filter core [4]	49	62	1.3	48%	11%	40%	354	599
*8-tap symmetric 8-bit FIR filter core [4]	42	56	1.3	55%	13%	32%	382	661
2-bit, 7-lag auto cross-correlator [6]	239	254	1.1	83%	0%	17%	363	626
1-D systolic convolution (4-bit, 8-stage) [10]	216	232	1.1	72%	0%	28%	346	578
Random logic circuits								
MIPS R3000 write-back unit [17]	54	63	1.2	10%	60%	30%	346	658

*hand placed using fast south-to-north carry chains

use *bundled-data* pipelines (e.g., [18]). A fabricated asynchronous FPGA chip using bundled-data pipelines operated at a maximum of 20 MHz in a 0.35 μm CMOS process [9]. Another drawback of previously proposed asynchronous FPGAs is that they could not use generic synchronous FPGA place and route tools. For example, the CAD tools for the Montage FPGA architecture [5] needed to enforce the isochronic fork delay assumption required for safe prototyping of QDI circuits.

The first asynchronous FPGA architectures (e.g., [5]) included programmable arbiters, asynchronous circuits that non-deterministically select between two competing non-synchronized signals. However, arbiters occur very rarely in slack-elastic systems because they can be used only when they do not break the properties of slack elasticity [14]. For instance, an asynchronous MIPS R3000 microprocessor used only two arbiters in its entire design [17]. Future work for our asynchronous FPGA will consider replacing a small number of conditional units with arbiter units.

While it is possible to perform limited prototyping of asynchronous logic on commercial clocked FPGAs, the performance and logic density costs are large. For example, an unpipelined 1-bit QDI full-adder requires ten 4-input LUT cells [7] and six additional 4-input LUT cells to pipeline its outputs. Since this adder is inefficiently pipelined and cannot take advantage of built-in carry chains, it will operate slower than both a clocked adder and an adder implemented in our asynchronous pipelined FPGA. In addition, the circuit area of prototyping an asynchronous adder in a clocked FPGA is 16 times worse than a clocked adder and 8 times worse than an adder in our asynchronous FPGA.

An alternative method for prototyping clocked logic is to map a clocked netlist onto asynchronous blocks, such that the resulting asynchronous system implements the same logical behavior as if it were a clocked system. While we believe this method is less than ideal because clocked logic does not behave like asynchronous logic and need not efficiently map to asynchronous circuits, previous work has designed pipelined asynchronous FPGA cells that support this feature [8, 22]. However, these FPGA designs used unpipelined interconnects and did not demonstrate significant performance advantages over clocked FPGAs.

9. FUTURE WORK

QDI asynchronous circuits are very conservative circuits in terms of delay assumptions and in that regard the results presented in this paper are the "worst" performance we can achieve with asynchronous FPGA circuits. If we use more aggressive circuit techniques that rely on delay assumptions, then it is feasible to design faster and smaller asynchronous FPGAs, at the cost of decreased circuit robustness.

In this paper we explored only a small realm of possible asynchronous FPGA architectures, that of an island-style FPGA with a non-segmented pipelined interconnect. A segmented interconnect, where routing channels span more than one logic block, or a hierarchical interconnect (e.g., a tree structure) could be used to help reduce pipeline latency on long channel routes. In addition, advanced hardware structures found in commercial FPGAs (e.g., cascade chains, LUT-based RAM, etc.) could be added to our asynchronous FPGA to improve performance and logic density.

While we concentrated on designing a bit-level FPGA using dual-rail channels, asynchronous logic may be more area efficient and energy efficient for multi-bit programmable datapaths. These datapaths consist of small N-bit ALUs, which are interconnected by N-bit wide channels that use more efficient 1ofN data encodings. For example, a 1of4 channel will use one less wire than two dual-rail channels and consume half as much interconnect switching energy.

10. SUMMARY

We described a finely pipelined asynchronous FPGA architecture that we believe is the highest performing asynchronous FPGA by an order-of-magnitude. This asynchronous FPGA architecture distinguishes itself from that of a clocked one by its ease of pipelining, its event-driven energy consumption, and its automatic adaptation to delay variations. We showed that asynchronous logic can transparently

use a pipelined interconnect without needing retiming registers and demonstrated that benchmark circuits achieve good performance using synchronous place and route CAD tools.

11. ACKNOWLEDGMENTS

The research described in this paper 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.

APPENDIX

A. PROGRAMMABLE CIRCUITS

Figure 16 shows the circuit details for a programmable C-element (pC), which consists of a standard C-element augmented by a configurable pull-down stack that allows a and/or b to be ignored when pC is part of a completion tree. The environment is responsible for driving unused inputs to ground (e.g., see completion trees in Figure 9).

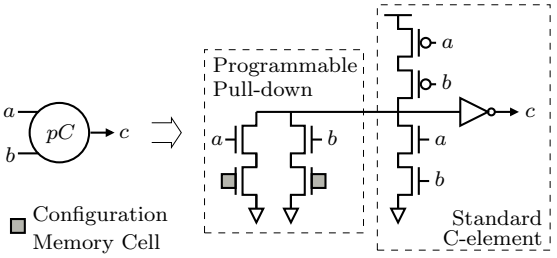

Figure 16: Programmable C-element.

Figure 17 shows the circuit details for an unpipelined programmable copy stage that can be configured to copy input tokens to one or both of its output channels.

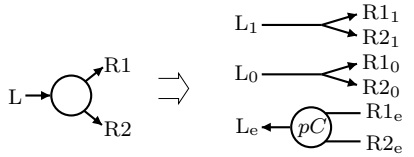

Figure 17: Unpipelined token copy.

B. 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] U. V. Cummings, A. M. Lines, and A. J. Martin. An asynchronous pipeline lattice-structure filter. In *Proc. International Symposium on Asynchronous Circuits and Systems*, 1994.

[3] J. 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.

[4] G. Goslin. A guide to using FPGAs for application-specific digital signal processing performance. Xilinx Application Notes, 1995.

[5] 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.

[6] B. V. Herzen. Signal processing at 250 mhz using high-performance FPGAs. In *Proc. International Symposium on Field Programmable Gate Arrays*, 1997.

[7] 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.

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

[9] 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.

[10] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, 1988.

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

[12] K. Maheswaran. Implementing self-timed circuits in field programmable gate arrays. Master's thesis, U.C. Davis, 1995.

[13] R. Manohar. A case for asynchronous computer architecture. In *Proc. of the ISCA Workshop on Complexity-Effective Design*, 2000.

[14] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *International Conference on the Mathematics of Program Construction*, 1998.

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

[16] A. J. Martin, S. M. Burns, T.-K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Proc. Conference on Advanced Research in VLSI*, pages 351–373, 1991.

[17] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proc. Conference on Advanced Research in VLSI*, pages 164–181, Sept. 1997.

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

[19] A. Singh, A. Mukherjee, and M. Marek-Sadowska. Interconnect pipelining in a throughput intensive FPGA architecture. In *Proc. International Symposium on Field Programmable Gate Arrays*, 2001.

[20] J. Teifel and R. Manohar. Concurrent dataflow decomposition. Cornell Computer Systems Laboratory Technical Report CSL-TR-2003-1036, Cornell University, Sept. 2003.

[21] J. Teifel and R. Manohar. Programmable asynchronous pipeline arrays. In *Proc. International Conference on Field Programmable Logic and Applications*, 2003.

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

[23] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. HSRA: High-speed, hierarchical synchronous reconfigurable array. In *Proc. International Symposium on Field Programmable Gate Arrays*, 1999.

[24] Xilinx. Virtex™ 2.5V field programmable gate arrays. Xilinx Data Sheet, 2002.