An Architecture for Asynchronous FPGAs

Catherine G. Wong, Alain J. Martin, and Peter Thomas
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125
{wongca,alain,petert}@async.caltech.edu

Abstract— **We present an architecture for a quasi delay-insensitive asynchronous field-programmable gate array. The logic cell is a complete asynchronous pipeline stage, and the interconnects are entirely delay insensitive, eliminating all timing issues from the place-and-route procedure.**

I. Introduction

Programmable logic is becoming increasingly attractive as the continuing decrease in CMOS feature-size keeps improving the performance of programmable devices. However, increased power consumption, and problems with clock distribution and timing closure are the other side of the coin.

Independently, asynchronous logic has made enormous progress in the last decade: complete asynchronous microprocessors are routinely designed in the lab, and CAD tools are being developed. Since two of the main advantages of asynchronous logic are the absence of a clock and low power-consumption, it seems natural to apply asynchronous technology to the design of FPGAs.

This paper presents a new approach to the architecture of asynchronous FPGAs. Rather than augmenting a standard FPGA cell with the ability to program the gates required for asynchronous design (e.g., a C-element) as has been done before, we propose to lift the architecture of the programmable cell from the level of a gate to the level of a complete pipeline stage.

Our continuing work in the high-level synthesis of asynchronous systems has led us to develop a powerful method for decomposing any sequential HDL program into a (possibly large) collection of communicating modules that all fit into one single template. It is this template that constitutes our basic programmable cell. The cell may have several conditional input and output ports; it may hold state and compute complex functions. Control and datapath are not separated inside the cells.

First, we briefly describe the decomposition method and the basic module, called precharge half-buffer (PCHB). We then present our design for a programmable logic cell, give performance figures from simulations of a layout prototype, and propose solutions for clustering cells into larger components. We discuss different choices for the architectures of programmable cells, clusters, and interconnect, and describe how we are setting up a general model to investigate them. Finally, we implement example applications on the preliminary architecture that we have chosen.

II. Timing Issues and Asynchronous Approaches

Asynchronous techniques are based on local communications among concurrent units. Communication and synchronization among the units are implemented by handshake protocols. There is no concept of global time—no clocks are used—and only minimal assumptions are made about the duration of an action or of a communication.

Timing issues present a difficulty in synchronous FPGA design because the mapping procedure (partitioning, placement, and interconnect routing) may cause violation of some timing constraints. Eliminating the clock is a big improvement in relaxing the timing constraints, but there are still some timing requirements to be met in asynchronous design. The different asynchronous techniques distinguish themselves by the types of timing assumptions they use. The QDI (*quasi delay-insensitive*) style used in our approach to asynchronous FPGAs makes the weakest timing assumptions. The circuits are completely delay-independent except for some forks—called *isochronic forks*—in which the propagation delays on the different branches of the fork are similar.

The automatic placement by a mapping procedure could violate the delay assumption on isochronic forks. An important advantage of the proposed method is that the synthesis procedure guarantees that all isochronic forks are local to the logic cells and therefore are unaffected by the placement procedure. Communication between cells is entirely delay-insensitive: it is implemented as a four-phase handshake, where the data to be transmitted is encoded in 1of N code. Hence, the mapping procedure does not have to meet any timing requirement, which makes this architecture particularly suitable for dynamically configurable systems.

Of course, the mapping algorithms will consider performance metrics such as latency and path-length. The advantage of delay-insensitive design can be seen when one part of system does not meet timing constraints but is off the critical path or is used infrequently. Instead of dedicating computation time to eliminating timing errors from it, it can be left as is, and the whole system will still function correctly and at the desired speed.

A. Quasi Delay-Insensitive Circuits

Consider a cell that receives a parameter a on an input channel, computes $f(a)$, and sends the result on an output channel. (The cell may have several inputs and outputs, some of them conditional. In the implementation we will describe, the cell has three inputs, two of which are con-

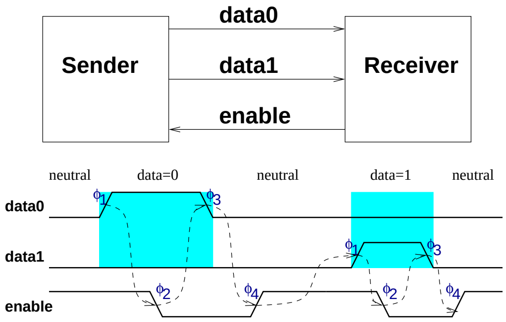

Fig. 1. One-bit communication channel encoded as $e1of2$.

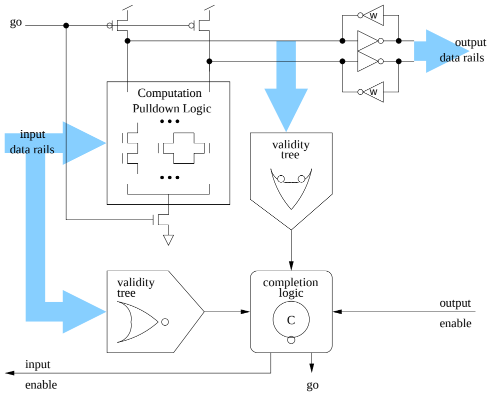

Fig. 2. General template for a precharge half-buffer circuit.

ditional, and one output, which may be replicated.) Our synthesis method demonstrates that we can decompose any program into a network of such cells.

Communications on channels are implemented with four-phase handshake protocols. The type of four-phase protocol we have chosen is the so-called *e1ofN* protocol. A channel between a sender and a receiver is implemented with enable wire (which is an output wire on the receiver side and an input wire on the sender side), and N data wires: one for each possible value of the data being sent. Initially all data wires are low (false), encoding a special *neutral* value. A *valid* value is encoded with exactly one data wire high (one-hot encoding).

For a single bit a, we need two data wires (figure 1). This $e1of2$ encoding is also called *dual-rail*. When the cell is ready to receive a value, the enable is high (true) and the data wires are low (neutral). A receive consists of four actions: (1) wait for one of the data wires to become true (data valid), (2) lower the enable, (3) wait for both data wires to become false (data neutral), (4) raise the enable. A send consists of the four complementary actions: (1) wait for the enable to be high, (2) raise one of the two data wires (set the data valid), (3) wait for the enable to be low, (4) reset the data wires to false (set the data neutral).

B. Precharge Half-Buffer Circuits

Our asynchronous FPGA logic cells are based on the precharge half-buffer (PCHB) circuit family. Among the various quasi delay-insensitive circuit families, PCHBs are both fast (high throughput and low latency) and compact, making them ideal for FPGAs. The fastest asynchronous microprocessors to date use PCHBs for more than 90% of their circuits.

A PCHB can perform computations (in a pulldown network of n-transistors), have multiple input and output channels, have conditional inputs and outputs, and hold state. The circuit is similar to a precharge domino logic circuit in synchronous design except that instead of a clock signal guarding each domino stage, a local signal, *go*, is computed for this purpose. PCHBs also contain circuitry to generate the enable signals for each of the input channels. Any size constraints for PCHBs are either physical (number of transistors in series) or performance-based (desired cycle time). The template for a general PCHB is shown in figure 2.

During each cycle, the PCHB implements the following interleaving of the send and receive four-phase handshakes described in the previous section: (1) wait for high output enable, (2) wait for valid input data, (3) perform computation and set valid output data, (4) lower input enable, (5) wait for low output enable, (6) reset output data to neutral, (7) wait for input data to be neutral, (8) raise input enable.
The cycle time of an asynchronous PCHB is the time required to complete all eight of these steps.

III. Related Work

Several proposals for asynchronous FPGAs have already appeared in the literature [1, 2, 3, 4, 5, 6]. Some of the approaches provide the same level of functionality as our PCHB pulldown logic (3-4 input LUTs), but most do not offer an implementation of the input and output handshakes and sequencing—say, the equivalent of the control part of the PCHB. In these alternative proposals, the control part has to be implemented explicitly, exposing the issues of timing assumptions and isochronic forks, and adding large efficiency penalties in terms of area, cycle time, and energy. In all but [5] and [6], either timing assumptions are needed for separating the control part from the datapath, or their implementation of the sequencing results in unpredictable placement of isochronic forks.

Rettberg and Kleinjohann offer a delay-insensitive FPGA architecture [5] but it has been proved that the class of entirely delay-insensitive circuits is very limited [10].

Recently, Teifel and Manohar have published a new design called PAPA that also uses PCHB circuits for its building blocks [6]. While the basic template for the individual cells is the same as that used in this paper, the choice of cells and clustered architecture is fundamentally different. In searching for a general reconfigurable design, we have created a logic cell that can be programmed to fit many different communication patterns—the number of active channels and the conditions for communication on these channels are configurable. In contrast, the PAPA architecture presents clusters that include different types of cells, each with set communication patterns. The individual cells are therefore simpler but when the system is mapped, the majority of cells in the cluster may be unusable. It may be that the two architectural choices are each better suited for different classes of applications. A clearer picture will emerge when we combine our synthesis tools with the area and performance estimation tools that we are developing based on the architectural model described in section VII.

```
Cell ≡
 *[A?a;
    [¬Sb ⟶   b := 0, c := 0
    ▯ Sb ∧ ¬Si ⟶ [¬Sc ⟶ B?b, c := 0  ▯Sc ⟶ B?b, C?c]
    ▯ Sb ∧  Si ⟶
          [¬Sc ⟶ [¬a ⟶ B?b, c := 0  ▯a ⟶ b := 0, c := 0]
          ▯  Sc ⟶ [¬a ⟶ B?b, c := 0  ▯a ⟶ b := 0, C?c]
          ]
    ];
    [¬Sz ⟶ Z!f(S, a, b, c)
    ▯ Sz ⟶ [a ⟶ Z!f(S, a, b, c)  ▯¬a ⟶ skip]
    ]
 ]
```

Fig. 3. Specification of logic cell behavior in the CHP language. (Basic CHP syntax is summarized at the end of the paper). Sb, Si, Sc, and Sz are programmable SRAM bits that configure the cell's communication patterns, while S is an array of eight programmable SRAM bits that configure the function to be computed.

IV. High-Level Synthesis and Process Decomposition

Our synthesis approach for QDI circuits allows the designer to take a system described in a high-level language and decompose it into readily implementable components by a procedure called process decomposition. Process decomposition exposes concurrency and allows mapping to the basic cells while producing a system with an acceptable throughput but not a surfeit of communications. (In most QDI systems, computation consumes significantly less energy than communication.)

We use *data-driven decomposition* (DDD), the first automated decomposition method to target the fast PCHB asynchronous circuit family [8]. DDD performs data-dependency analysis on the sequential high-level description and then decomposes the program into a system containing one module for each variable. The modules contain both the computation of the variable (encapsulating all assignments to that variable in the original program) and new intermodule communications that read in values of other variables and send out the newly computed value. The original description and the generated modules are all expressed in the same concurrent programming language, CHP. (Basic CHP syntax is summarized at the end of this paper, and VHDL-to-CHP translators are under development.)

DDD decomposes any deterministic program into a system of PCHB-implementable modules by first rewriting the sequential program so that every variable is assigned a value at most once during the execution of an iteration of the main loop. This transformation to *dynamic single assignment (DSA)* form may involve splitting variables, eliminating inner loops, and adding assignments to the ends of selection statement branches. Other steps include adding variables to encode the conditions of selection statements to reduce the number of communications channels in the final system.

When synthesizing asynchronous FPGAs, there is an extra synthesis step involved in DDD: splitting up wide channels into the small dual-rail ($e1of2$) channels on the FPGA. This step, called vertical decomposition, includes the distribution of control signals to the new parallel modules that each implement one bit of the original channel.

After DSA conversion, data-dependency analysis and decomposition, DDD has created a concurrent system where every module can be implemented by a PCHB circuit. The last stage of DDD is to recompose the DSA modules back into larger clusters. The recomposition algorithm focuses on shared inputs and chains of communication, two features that have parallels in the input-copy and feedback channels of the FPGA's cluster design (see section VI). Thus, DDD can map a sequential program into a collection of FPGA cells, and cluster the cells to fit the architecture described in this paper.

V. Logic Cells

Each of our basic logic cells consists of a single PCHB circuit. The cells are clustered in larger blocks where they can be configured to share inputs and send outputs to each other through local feedback channels. Clusters also contain slack-matching buffers that are inserted to increase the performance of asynchronous QDI circuits. This section presents the design of a programmable $e1of2$ cell. Cells with other channel encodings are discussed in section VII.

A. Cell Design

The asynchronous logic cells in this section are limited to PCHBs with three dual-rail input channels and one dual-rail output channel. Each cell contains twelve programmable SRAM cells: eight to configure the cell's computation and four to configure the cell's communication pattern. The high-level specification of the logic cell, including potential conditional communications, is given in figure 3.

Following the PCHB template given in figure 2, the logic cell circuit consists of a pulldown computation network of n-transistors, input and output channel validity trees (that must also account for data dependencies when conditional communications are programmed), and completion logic that generates the input enable signals and the local signal *go* that guards the domino logic.

We limit the cell to one output channel as multiple output channels would require multiple separate precharge domino stages that could go unused. Instead, by grouping single-output logic cells inside clusters where they can share input channels, there is some redundancy in validity circuitry when multiple outputs are computed from the same inputs, but no unusable circuitry when there is no sharing (as is often the case). Arguably this choice is more suited for some applications (e.g., general asynchronous microprocessors) than others (e.g., arithmetic or DSP) and it merits further study. We limit the cell to three input

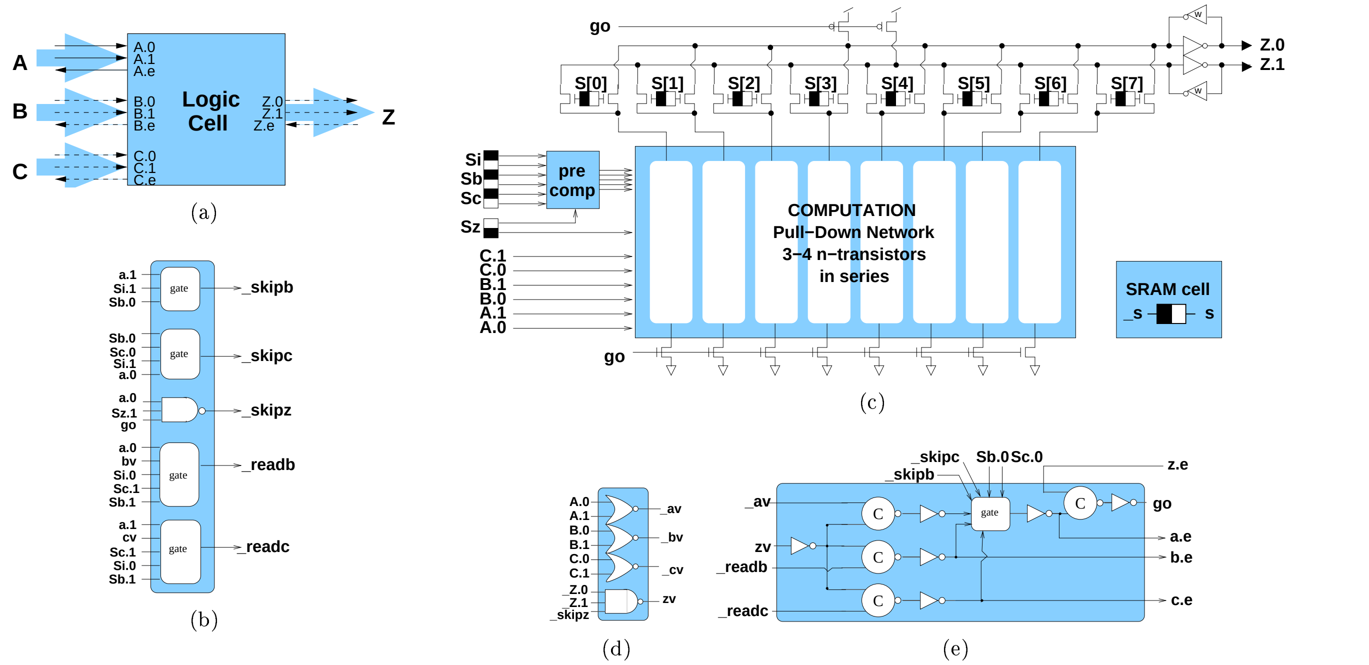

Fig. 4. (a) Logic cell for the asynchronous FPGA. The dashed lines indicate channels on which communications may be conditional. Different parts of the cell: (b) circuitry for conditional communications; (c) computation circuitry; (d) input and output validity generation; (e) completion circuitry.

channels because with four, the pulldown networks of the domino logic grow too long for high performance.

The basic circuit for the cell is illustrated in figure 4. The "pre-comp" unit in figure 4(c) consists of combinational gates that have been pulled out of the pulldown network to reduce the number of n-transistors in series. Since the unit operates only on SRAM-cell outputs, it does not add to the forward latency or the cycle time of the PCHB in general. Figure 4 is not an exact representation of the circuit: details such as the reset circuitry and keepers for each of the C-elements have been omitted.

B. Communication Patterns

Using the programmable bits Sb, Si, Sc and Sz, the cell can be configured to use either one, two, or three inputs. The cell's input channel $A?$ is always unconditional (a communication must occur on every cycle) but communications on all of the other channels—input channels $B?$ and $C?$, and output channel $Z!$—can be programmed to depend upon the value input upon $A?$. These conditional communications allow each cell to be configured to fit ten different communication patterns, some of which include data-dependency. From these patterns, a cell can not only implement basic computation blocks but also, either singly or in combination with another cell, two basic control circuits in asynchronous QDI design: the controlled merge and the controlled split. The importance of these cells is demonstrated in the $FBLOCK$ example of section VIII.

C. Simulation Results

We created layout and performed SPICE simulations for the basic cell in TSMC 0.18-μm technology. The simulations show that a cell operates with a cycle time of 190−235 MHz (depending upon its configuration), and consumes between 2.1 and 3.1 pJ/cycle. The results are summarized in figure 5.

Sb	Sc	Si	Sz	communication	cycle time MHz	energy pJ/cycle
0	X	X	0	$A?,Z!$	206	2.4
0	X	X	1	$A?,(Z!)$	235	2.1
1	0	0	0	$A?,B?,Z!$	195	2.8
1	0	0	1	$A?,B?,(Z!)$	222	2.4
1	0	1	0	$A?,(B?),Z!$	200	2.6
1	0	1	1	$A?,(B?),(Z!)$	217	2.4
1	1	0	0	$A?,B?,C?,Z!$	190	3.1
1	1	0	1	$A?,B?,C?,(Z!)$	220	2.7
1	1	1	0	$A?,(B?,C?),Z!$	199	2.8
1	1	1	1	$A?,(B?,C?,Z!)$	214	2.6

Fig. 5. Results of SPICE simulations for different configurations of the basic cell in 0.18-μm technology. Parentheses indicate that communications depend upon the data input on channel $A?$.

The dimensions of the prototype FPGA cell, including twelve bits of SRAM, are $334\lambda\times323\lambda$. While the design of the asynchronous logic cell may seem bulky compared to basic logic cells in synchronous FPGAs, we note that QDI designs face a similar area penalty in custom VLSI design yet outperform their synchronous counterparts in both speed and energy consumption [11]. A more detailed consideration of cell area is given in section VII.

VI. Cluster Design

The logic cells serve as versatile building blocks for an asynchronous FPGA. However, additional functionality and interfacing can help implement programs and embed the cells into an interconnect mesh more efficiently. To incorporate features such as channel replication, slack-matching buffers for performance optimization, and initial tokens generated at reset, we group cells together into structures called *clusters*. While a more general model is presented in section VII, a cluster in this section consists of four basic logic cells, $L[0]-L[3]$, that can share inputs and send outputs back to inputs as feedback channels. This design is illustrated in figure 6, and its different features are described below.

A. Copy Channels

QDI channels cannot be split or shared between PCHBs without additional completion circuitry to acknowledge every transition on the channel. If the data rails of a channel are split and copied to two different input ports, then the two enable rails from those ports must be collected in a C-element with that gate's output serving as the new enable rail for the copied channel. Clusters include extra enable rails and C-elements so that any input channel can be copied either to $L[0]$ and $L[1]$, to $L[2]$ and $L[3]$, or to all four logic cells. To configure the input copy channels, simply connect either the original separate enable rails (e.g., both A[0].e and A[1].e) or the generated copy enable (e.g., cpA_01_e) to the interconnect mesh.

In cases where a channel needs to be copied to two cells in different clusters, an extra enable rail is also provided for the output channel. This rail is sent through a C-element with the original output enable rail and a multiplexer programmed by the SRAM bit *Scpz* is used to choose the enable signal from either the original single output enable signal or the combined copy enables.

B. Feedback Channels

There are many cases when the output channel of a logic cell in a cluster is required as an input to another cell (or, perhaps to several other logic cells) in that same cluster. One example is a full adder. To implement this scenario as simply as possible, feedback channels are provided within the cluster. The logic cells can be programmed to connect to these channels in the same way that they can connect to external interconnect—the new channels are completely local.

C. Buffering and Initial Tokens

Asynchronous pipeline dynamics show that the throughput of a system is optimized when the number of PCHBs in a cycle (N) is related to the PCHB's forward latency (l) and the cycle time for a handshake between PCHBs (t), through the equation $N*l = t$ [13]. As well, any datapaths that split into different branches of PCHBs and then reconverge should be matched in length for optimal throughput. The task of adding simple QDI buffers to a system to meet these constraints is called *slack-matching*. (QDI "buffers" are circuits that implement four-phase handshakes on both their input and output channels.) Each output channel contains a QDI "slack buffer" that the output signals may be programmed (using the SRAM bit $Sbuf$) to either skip or pass through (see figure 6).

Slack buffers are also used at reset, when a system may be initialized with tokens of data on certain channels. Because such channels require slightly different completion circuitry within the attached PCHBs, the slack buffers can be set up with different reset token configurations (no token, initial token=0, initial token=1).

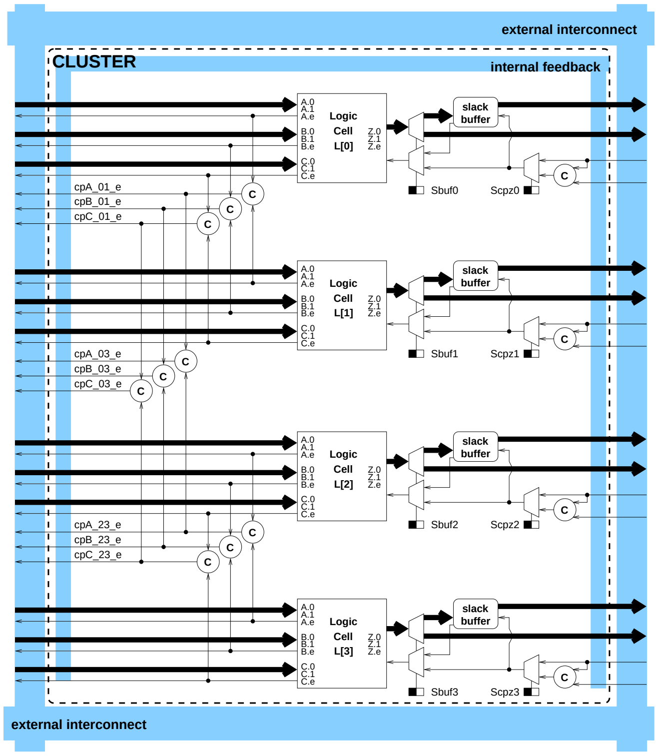

Fig. 6. Cluster block diagram.

D. Summary

Aside from interconnect, the cluster described in this section contains 60 programmable SRAM cells: twelve within each logic cell, four to select whether or not each logic cell output will pass through a slack buffer, four to choose whether or not each output channel will be copied, and four to program initial tokens on the output channels.

VII. Architectural Models and Interconnect

We have presented the design of a basic $e1of2$ logic cell, and of a cluster that contains four such cells as well as four slack buffers. However, there are many other design points to consider when constructing a fast, energy- and area-efficient asynchronous FPGA. We have begun studying different designs with varying channel types (performance considerations limit us to considering $e1of2$ and $e1of4$ encodings), cluster configurations, interconnect switches and routing architectures. This section describes our approach to creating a parameterizable model (including both computation blocks and interconnect) for an asynchronous FPGA, and notes some initial results and trends concerning area that have surfaced when comparing synchronous FPGAs with asynchronous FPGAs that use $e1of2$ and $e1of4$ channel encodings.

In our analysis, we use the minimum-width transistor area estimation technique presented by Betz, Rose and

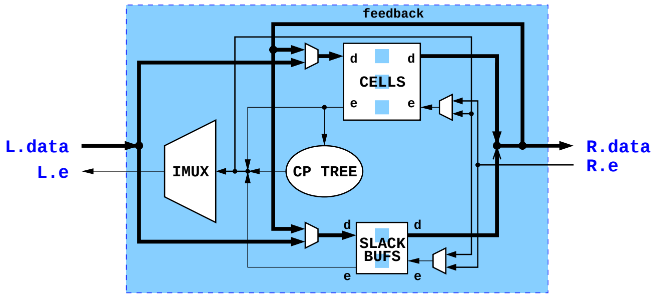

Fig. 7. General cluster design.

Marquardt [7]. This method estimates total circuit area by using an experimentally-derived function to compute the area required by a transistor of a given size in units of the area required by a minimally-sized transistor in the same technology.

A. Logic Cell Comparisons

At the logic cell level, the three-input $e1of2$ cell described in section V requires twelve SRAM bits and roughly 2.5 times the area of an equivalent three-input synchronous cell (which includes a 3-LUT, a multiplexer, and nine SRAM bits). Recall that the asynchronous $e1of2$ cell includes circuitry that allows conditional communications by only generating enable signals under certain data-dependent conditions. This circuitry may not be necessary in a synchronous design, but contributes to the energy-efficiency of our circuits since the majority of energy in asynchronous chips is consumed in communications and not computation. Also, conditional communications on channels allow entire cells to be used (i.e., be "active") only conditionally. (Consider the scenario where cell $C3$ receives all of its inputs from cells $C1$ and $C2$. If $C1$ and $C2$ send their data only when the condition B is true, then $C3$ is inactive whenever B is false.) Since there is no global clock, asynchronous QDI cells do not consume any dynamic energy when they are inactive.

Compared to the $e1of2$ cell, an equivalent $e1of4$ cell (with the same control logic but twice the datapath) requires 132 SRAM bits and more than five times the area. The $e1of4$ cell can programmed to compute 4^{4^3} different functions while the $e1of2$ version can compute only 2^{2^3}, but whether all of this added functionality is helpful remains to be seen. Also, details at the analog circuit level may cause the $e1of4$ cell to run at a slower pace than the smaller $e1of2$ cell. We are investigating these issues further.

B. Parameterizable Clusters

For programmable clusters, we are modeling architectures using parameters such as channel size, number of inputs per logic cell, number of logic cells per cluster, number of slack buffers per cluster, number of interconnect channels that enter the cluster, etc. A model for this general cluster is shown in figure 7.

Currently, the feedback network is assumed to be fully connected and the copy tree design is more general than the two/four-way copy trees described in section VI. This flexibility will place a practical limit on the size of the clusters, and we are working to determine a desirable balance between clusters that are either large and flexible or small and less generally programmable. This involves developing tools that (given the parameters listed above) estimate area and delay, taking into account the extra electrical buffering (as opposed to slack buffering) required when the choice of parameters creates nodes with large fanout and placing the buffers so as to avoid isochronic forks.

Channel Type	#Gates per channel switch	#SRAM cells per channel switch	#Channels per $2n$-bits of datapath
sync/1-bit	1	1	$2n$
sync/2-bit	2	1	n
async/e1of2	3	1	$2n$
async/e1of4	5	1	n

Fig. 8. Interconnect switches for different channel encodings.

C. Interconnect

Asynchronous circuits require more wires than synchronous circuits to communicate information. However, the general interconnect of asynchronous FPGAs can be conceptually modeled after that of synchronous FPGAs, with an asynchronous channel being the equivalent of a synchronous wire. The area required by asynchronous interconnect does not simply scale with the number of wires, though: while each switch for an $e1ofN$ channel involves $N+1$ wires and programmable gates (either pass-gates or tri-state buffers), only one SRAM cell is required for configuration. Figure 8 characterizes the interconnect switches for some basic synchronous and asynchronous channel encodings. We include synchronous interconnect where each wire can be routed individually, synchronous interconnect where the wires are grouped in pairs to carry 2-bit quantities and are always routed together, and the asynchronous $e1of2$ and $e1of4$ channel encodings.

Consider the situation in which interconnect area is dominated by switches and not wires. (Conventional experience indicates this is the case, and while the growing number of metal layers in new technologies help us increase the density of wiring networks, switches currently require space on the substrate and cannot be easily stacked [9].) The specific computations performed in this section assume a network that employs 50% pass-gates and 50% tri-state buffers. Since tri-state buffers are usually more than twice the size of pass-gates, the numerical results will vary depending on the fraction of pass-gates used in interconnect. However, for each switching network architecture considered here, the relative ordering of the switching areas required by the various channel encodings remains the same no matter what combination of pass-gates and tri-state buffers are used.

We begin by noting that both synchronous 1-bit and $e1of2$ interconnect require the same number of channels to route a given datapath. Given a switching network architecture then, both types of interconnect will use the same number of switches; only the area per channel switch will differ. Following the circuit area estimation technique described earlier, $e1of2$ switches require, on average, about 2.2 times the area of their synchronous 1-bit counterparts. Hence, no matter what type of switching network is chosen (i.e., no matter whether the number of switches grows linearly or super-linearly with the number of inputs and outputs to the network), $e1of2$ interconnect will use up roughly 2.2 times as much area as synchronous 1-bit interconnect. Similarly, $e1of4$ interconnect will use up about 2.1 times the area as synchronous 2-bit interconnect, regardless of the type of switching network chosen. (The comparisons in this section ignore the wires required to distribute clock signals throughout a synchronous FPGA.)

In contrast, when comparing the two asynchronous encodings, $e1of4$ interconnect requires fewer channels but larger individual channel switches than $e1of2$ interconnect. The ratio of total interconnect areas for the two channel-encoding schemes therefore depends on the switching network architecture chosen. For example, consider a network that uses $O(N^2)$ switches, where N is the number of network sources and sinks. Given this switching architecture and equally wide datapaths, $e1of4$ interconnect uses up only 39% of the area of $e1of2$ interconnect. When the switching network uses $O(N)$ switches, $e1of4$ interconnect still requires less area than $e1of2$ interconnect, but at the higher fraction of 77%.

Typically, routing resources take up 90% and logic cells only 10% of the area of current synchronous FPGAs. If this ratio holds true for asynchronous FPGAs then, ignoring clusters and focusing on this section's area comparisons for logic cells and switching networks, an asynchronous $e1of2$ FPGA requires roughly 2.2 times the area of its synchronous 1-bit equivalent. (The fact that $e1of2$ cells and switching networks are similar factors larger than their 1-bit synchronous counterparts points to the ratio remaining roughly the same.)

The computation is more complicated when comparing $e1of2$ and $e1of4$ asynchronous FPGAs. One reason is the effect of different switching network architectures on the area required by $e1of4$ interconnect. Another is that while $e1of4$ cells can handle twice the datapath of $e1of2$ cells, the percentage of cells that make use of this doubled datapath depends on the nature of the system being implemented. In a datapath element such as the *FBLOCK* described in section VIII, close to 95% of the cells would take advantage of the larger cells. In a less regular control element, that percentage might drop. We are working on combining our synthesis and area estimation tools to study this issue further.

Most of our research so far has focused on designing the computation blocks for an asynchronous FPGA but, as this section indicates, there are many interesting problems to explore in the realm of programmable asynchronous interconnect. One point of interest that we have not yet mentioned involves the slack-matching optimization. If the slack buffers in the clusters are not sufficient to increase the slack of the system and optimize its cycle time (because the addition of too many buffers to the cluster could cause the internal feedback interconnect area to grow beyond practical limits), more optional slack-matching buffers can be added at switch-points in the global interconnect.

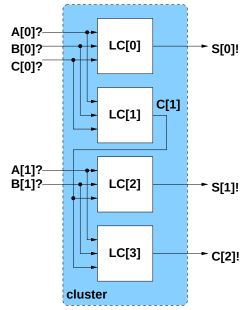

Fig. 9. One cluster implementing two bits of a full adder with input channels A and B, sum output channel S, and carry channels C.

VIII. Examples

In this section, we demonstrate different possible configurations of the clustered architecture described in sections V and VI by decomposing programs of different sizes and mapping them to cells and clusters. We start by considering a simple full adder and then a four-bit ALU. The final example is a 32-bit datapath unit taken from an asynchronous microprocessor.

A. Full Adder and ALU

One bit of a ripple-carry full adder can be implemented using two dual-rail logic cells. When multiple bits are required, copy and feedback channels allow each cluster to implement two bits of the adder. A possible cluster setup is shown in figure 9.

Building upon this full adder, consider the implementation of a four-bit ALU that can perform addition, subtraction, logical AND, and logical OR. Such a system requires two clusters to implement the full adder cells, one cluster to negate the subtrahend in case of subtraction, and one cluster to implement the boolean functions. As well, we need three more clusters to conditionally send the inputs and opcodes to either the full adder or the boolean computation blocks, and one cluster to merge the outputs of these two sets of blocks. The sum total of cells required to implement this four-bit ALU is 32. The cells can be mapped to eight clusters. (On a standard 4-LUT synchronous FPGA architecture, the same circuit can be implemented in 16 cells. The difference in results can be attributed to the extra input in the basic synchronous cell and to the fact that no acknowledgment signals need to be withheld for conditional communications, simplifying the control circuitry.)

B. Microprocessor Execution Unit

We have decomposed the *FBLOCK* execution unit from the asynchronous MiniMIPS [12] for implementation on our clustered architecture. The *FBLOCK* is a classic datapath unit (32-bits wide) with simple control. Its initial high-level CHP specification is as follows:

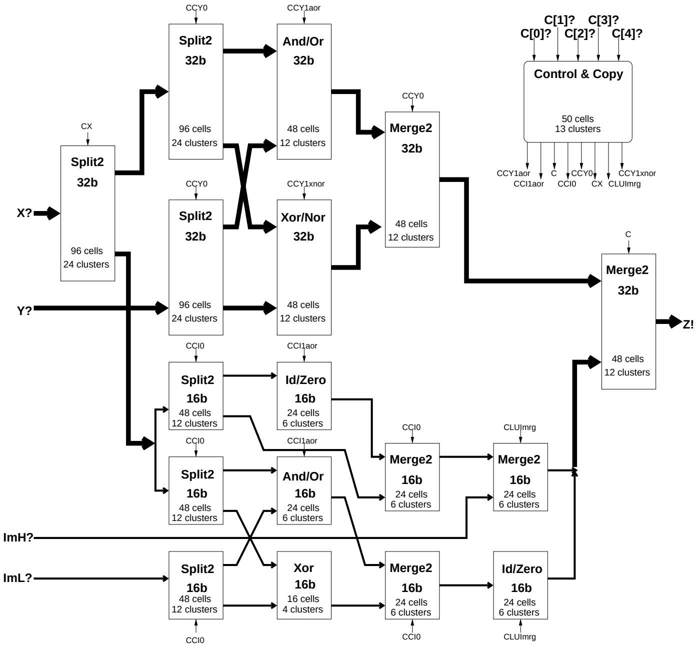

Fig. 10. Decomposition of the *FBLOCK* execution unit from the Asynchronous MiniMIPS.

$$
\begin{array}{lll}
FBLOCK(& C? & : \; e1of2[5]; \\
 & X?, Y?, Z! & : \; e1of2[32]; \\
 & ImL?, ImH? & : \; e1of2[16] \;\;) \;\; \equiv \\
\end{array}
$$

$$
\begin{array}{l}
*[\;\; C?c; \\
\quad [c \in \{and, or, xor, nor\} \longrightarrow \;\; X?x, Y?y, \;\; Z!op(c)(x, y) \\
\quad [\!]\, c \in \{andi, ori, xori\} \longrightarrow \;\; X?x, ImL?il; \;\; Z!op(c)(x, il) \\
\quad [\!]\, c = lui \longrightarrow \;\; ImH?ih; \;\; Z!(ih{*}2^{16}) \\
\;\;]\;\;]
\end{array}
$$

The final decomposition of this unit consists of 834 logic cells, grouped in 209 clusters. All but two of the clusters are fully populated. 661 of the cells take advantage of conditional communications. On a related note, only 93 of the cells are active on every cycle. Hence, in the absence of a clock, only 93 of the 834 cells are consuming dynamic energy every cycle. A high-level view of the decomposed system is shown in figure 10.

IX. Conclusion

We have presented a clustered architecture for an asynchronous FPGA. The system is quasi delay-insensitive, and places no inherent timing constraints on any placement and routing tools that target this FPGA. We have briefly described our synthesis approach, and demonstrated how to implement several different circuits on this architecture, including a large datapath unit from an asynchronous microprocessor.

As the synthesis tools for our asynchronous FPGA mature, we can explore different architectures thoroughly. We have discussed design decisions such as channel encoding and cell homogeneity, and mentioned others such as the number of slack buffers per cluster or the number of i/o channels per cell. Another possibility to investigate is the introduction of arbiter and synchronizer cells to the architecture, although they are used infrequently in asynchronous microprocessors.

While there are many issues open to further research, we believe that the cell and cluster designs presented in this paper offer a good general basis for future asynchronous FPGAs.

Acknowledgments

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency and monitored by the Air Force under contract F29601-00-K-0184.

References

[1] R.E. Payne. Asynchronous FPGA Architectures. *IEE Proceedings on Computers and Digital Techniques*, Special Issue on Asynchronous Processors, September 1996.

[2] S. Hauck, S. Burns, G. Borriello and C. Ebeling. A FPGA for Implementing Asynchronous Circuits. *IEEE Design and Test of Computers*, 11(3), 1994.

[3] K. Maheswaran. Implementing Self-Timed Circuits in Field Programmable Gate Arrays. Master's thesis, U.C.Davis, 1995. September 1996.

[4] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N. Imlig, T. Schiozawa, M. Inamori and K. Nagami. PCA-1: A Fully Asynchronous, Self-Reconfigurable LSI, *Proc. Seventh Intl Symposium on Asynchronous Circuits and Systems*, March 2001.

[5] A. Rettberg and B. Kleinjohann. A Fast Asynchronous Reconfigurable Architecture for Multimedia Applications, *Proc. 14th Symposium on Integrated Circuits and Systems Design*, September 2001.

[6] J. Teifel and R. Manohar. Programmable Asynchronous Pipeline Arrays. *Proc. 13th Intl Conference on Field Programmable Logic and Applications*, September 2003.

[7] V. Betz, J. Rose and A. Marquardt. Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, 1999.

[8] C.G. Wong and A.J. Martin. High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition. *Proc. 40th Design Automation Conference*, June 2003.

[9] A. DeHon. Rent's Rule Based Switching Requirements, *Proc. System-Level Interconnect Prediction (SLIP'01)*, 2001.

[10] A.J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conference on Advanced Research in VLSI, ed. W.J. Dally*, MIT Press, 1990.

[11] A.J. Martin, M. Nyström and C.G. Wong. Three Generations of Asynchronous Microprocessors. *IEEE Design & Test of Computers, Special Issue on Clockless VLSI*, November-December 2003.

[12] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings and T.K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *Proc. 17th Conference on Advanced Research in VLSI*, 1997.

[13] A.J. Martin, M. Nyström, K. Papadantonakis, P.I. Penzes, P. Prakash, C.G. Wong, J. Chang, K.S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J.T. Tong, and A Tura. The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller, *Proc. 9th IEEE Intl Symposium on Advanced Research in Asynchronous Circuits and Systems*, May 2003.

CHP Notation

Below is a brief summary of the meanings of CHP features that are used in this paper.

- **Basic Statements:**
  - $x := a$: Assign the value of a to x.
  - $A?a$, $B!b$: Assign the value on input channel A to variable a. Output the value of b onto channel B.
  - **skip**: Do nothing.
- **Composition of Statements:**
  - $A; B$: Execute A and then B in series.
  - A, B: Execute A and B in parallel. Stronger compositional operator than ";".
- **Control Structures:**
  - $[g \rightarrow A [\!] h \rightarrow B]$: Deterministic selection statement. If g is true, execute A. Or else, if h is true, execute B. The program waits for one of g and h to be true; g and h are mutually exclusive.
  - $*[A]$: Repeat A indefinitely.