DESIGN OF A RECONFIGURABLE PULSED QUAD-CELL FOR CELLULAR-AUTOMATA-BASED CONFORMAL COMPUTING

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Zhou Tan

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Electrical and Computer Engineering

June 2011

Fargo, North Dakota

North Dakota State University

Graduate School

Title

Design of a Reconfigurable Pulsed Quad-Cell for

Cellular-Automata-based Conformal Computing

By

Zhou Tan

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

ABSTRACT

Zhou Tan, M.S., Department of Electrical and Computer Engineering, College of Engineering and Architecture, North Dakota State University, June 2011. Design of a Reconfigurable Pulsed Quad-Cell for Cellular-Automata-Based Conformal Computing. Major Professor: Dr. Chao You.

This paper presents the design of a reconfigurable asynchronous unit, called the pulsed quad-cell (PQ-cell), for conformal computing. The conformal computing vision is to create computational materials that can conform to the physical and computational needs of an application.

PQ-cells, like cellular automata, are assembled into arrays with nearest neighbor communication and are capable of general computation. They operate asynchronously to minimize power consumption and to allow scaling without the limitations imposed by a global clock. Cell operations are stimulated by pulses which use two wires to encode a data bit. Cells are individually reconfigurable to perform logic, move and store information, and coordinate parallel activity.

The PQ-cell design targets a 0.25 μm CMOS technology. Simulation results show that a PQ-cell, when pulsed at 1.3 GHz, consumes 16.9 pJ per operation. Examples of self-timed multi-cell structures include a 98 MHz ring oscillator and a 385 MHz pipeline.

Keywords: Conformal Computing, Cellular Logic Array, Asynchronous Reconfigurable Computer, Cellular Automata

ACKNOWLEDGMENTS

This thesis represents my endeavor during the past two years in my graduate study life. Though the final result is far away from perfection, it deserves my devotion and hard work.

I would first express my appreciation to Dr. Mark Pavicic and Dr. Chao You, who have been acting as my mentors during this two-year research life, helping me from the very beginning all the way to what I own today.

I am also thankful for Dr. Rajendra Katti, Dr. Sudarshan Srinivasan, and Dr. Yechun Wang for being my committee members and supervising my final examination. You really did bring some brilliant ideas and useful instructions during this time.

My parents, my girlfriend, all of my friends and lab mates, thank you for being supportive and friendly to me every day.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

In recent years there has been widespread interest in making things out of very large numbers of very small parts. These parts could be special molecular structures, micro-fabricated devices, or even living cells. The parts are so small and numerous that new approaches are sought for assembly, programming (defining local interactions to achieve global behavior), dealing with faults, and so on. There are many ideas about what such an ensemble might be useful for. It could be some form of programmable material, "smart matter", swarms of tiny robots, or simply a computer. Related research areas that have computing as a desired outcome include molecular computing [1] - [2], bio-molecular computing [3], bio-inspired computing [4] - [5], and amorphous computing [6].

For computer systems with many small parts, the programming models tend to be quite different from what is used in conventional computers. For example in amorphous systems [7], information essentially diffuses through the system. This is similar to node-to-node "hopping" in wireless sensor networks. In both cases information moves in steps that are much shorter than the dimensions of the system. How to deal with such issues is of interest because it may enable the realization of systems that are superior to today's programmable systems in important ways. In particular, it would be very useful to be able to perform brain-like tasks with systems that are much smaller and more efficient than what can be expected from today's computing architectures.

Our interest is in non-biological cellular arrays in which the parts are densely packed in a regular structure and the need for power and communication is met by electrically conductive wires or planes. In particular, we[1] [2] envision sub-arrays

[1]People who have made contribution to this design are: M. Hoseini, Z. Tan and C. You from Electrical and Computer Engineering Department, M. Pavicic from Center of Nanoscale Science and Engineering.

[2]My task in this PQ-cell design includes some alterations on first version design which is in

fabricated on CMOS chips, and the chips, in turn, are arrayed on large thin flexible substrates, or sheets. Sheets may be cut, joined, bent, and stacked to conform to the physical and computational needs of an application. We refer to this flexible and scalable form of computing as "conformal computing" [8].

Our long-term vision is to help make progress toward systems capable of efficiently performing brain-like tasks. Conformal computing moves in that direction by exploring a computational medium assembled from "cells" that are much simpler than conventional instruction-processing nodes. Although the cells can be used to assemble conventional structures, our desire is to explore alternatives that are more similar to cellular automata [9], crystalline computing [10], cell matrices [11], BLOB computing [12], cellular neural nets [13], and the like. Therefore the cell designs emphasize simplicity (small multiplexers, 2-input logic units) and scalability (clockless synchronization, multi-chip arrays) combined with features of cellular automata (regular structure, local communication) and FPGAs (reconfigurable function and initial state).

Thus, we present a particular cell design, called the pulsed quad-cell (PQ-cell), for constructing a conformal computer. The PQ-cell is a quad cell because it consists of four orientations of an elementary cell called a quarter. The PQ-cell is the latest in a series of cell designs that include a clocked cell [14] and a triggered cell. These designs are distinguished by the source of the stimulus that causes a cell to perform an operation. Clocked cells use pulses generated from a central source and distributed throughout the array. Triggered cells use pulses generated by other cells in response to previous pulses and routed along computational paths. In the clocked and triggered schemes, a transferred data bit is accompanied by a pulse which stimulates the receiving cell to accept and process the bit. Because the data and stimulus are

Chapter 3, the simulation part which is in Chapter 4, and optimizations which are covered in Chapter 5.

conducted on separate wires, it is necessary to design for worst-case delays to ensure the data is set up before the pulses arrive. This necessity is eliminated if the data itself is the stimulus. This is the idea in the PQ-cell design. The PQ-cell design uses dual-rail encoding in which a unit of data is a single pulse that appears on one of two rails (wires): one rail for '0' pulses and the other rail for '1' pulses. The cells route the data pulses along computational paths.

Rather than a theoretical design, or one based on a future technology, the PQ-cell design is targeted for fabrication in a 0.25 μm CMOS technology. Therefore the PQ-cell array can serve as a concrete example of a novel computational host for new and beneficial forms of computation. The remainder of this thesis is as follows: Chapter 2 introduces some background about asynchronous circuit design, meanwhile making comparisons with CA and FPGAs to describe the general features of PQ-cell arrays. Chapter 3 presents the specifics of the PQ-cell design. Chapter 4 shows simulation results for a single cell and a variety of useful multi-cell structures. Chapter 5 brings in some latest update in PQ-cell design. Finally, Chapter 6 contains a summary and conclusions.

CHAPTER 2. BACKGROUND

2.1. Asynchronous circuits

Today's most popular digital circuits are so-called "synchronous", and in the design process, it obeys two fundamental principles: (1) binary signal transmit; and (2) throughout the circuit, a global discrete time is shared by all components in the circuits, which is acknowledged as a "clock". Asynchronous circuits, on the other hand, do not have a global discrete time shared by the whole circuits. So it is also called clockless. To achieve synchronization and communicate within a system, an asynchronous circuit utilizes some particular protocols such as bundled data transmit and handshaking. Nowadays, the need for asynchronous circuits is crucial, mostly because of the dramatic on increase integration degree of Very Large Scale Integrated (VLSI) system, in which the variation across the chip makes the control of clock and other global signals extremely difficult. In addition to the clock distribution issue, some other advantages of the asynchronous circuits have been exploited to overtake synchronous ones. They include:

- Low power consumption
- High operating speed
- Less emission of electro-magnetic noise
- Robustness towards variations in supply voltage, temperature, and fabrication process parameters
- Better composability and modularity
- No clock distribution and clock skew problems

Despite all these positive characteristics over synchronous circuits, development of asynchronous circuits started decades ago but somehow grew slowly before the late 1990s. The following paragraph uncovers a very short history on asynchronous circuit design.

2.1.1. A brief history

Starting as early as the 1950s, the University of Illinois started to contain both synchronous parts and asynchronous parts in circuits design. In the 1960s, the proposition of asynchronous building blocks in "macromodule" was very close to a modern approach. Some other significant contributions were also made by Huffman [15], Muller [16] and Unger [17]. However, when clocked techniques provided an easy way to deal with timing issues, the asynchronous techniques were forgotten for quite a while. it was not until the late 1990s that projects in academia and industry demonstrated that it is possible to design asynchronous circuits which exhibit significant benefits in real-life examples. Among this period of time, Caltech designed and fabricated the first single-chip asynchronous microprocessor in 1988. Shortly after, in 1993, the University of Manchester implemented asynchronous techniques on the famous ARM processor, and made the family of clones called "Amulet". In 1997, a 32-bit MIPS R3000 microprocessor, MiniMIPS, was developed in Caltech. MiniMIPS still holds the record of the fastest complete asynchronous microprocessor chip. Today, the design techniques have developed to satisfy both entire asynchronous and globally asynchronous with locally synchronous requirements. In addition, more computer-aided design tools are developed in designing asynchronous digital systems.

2.1.2. Classication of asynchronous circuits

Depending upon the timing assumptions, asynchronous circuits can be classied as self-timed, speed-independent or delay-insensitive and quasi-delay-insensitive.

Circuits whose correct operation relies on more elaborate or engineering timing assumptions are simply called self-timed. In self-timed asynchronous circuits, each functional block is controlled by some handshake circuit such that each functional block is operated in correct order. Each functional block should also be able to acknowledge the completion of its operation to the handshake control circuit.

A speed-independent circuit is the one of the kind that ignores delay in wire and fork elements, compared with delay in gate components. More specifically, as shown below in Figure 1, if the speed-independent condition is assumed, then d_A, d_B and d_C are some arbitrary finite and also positive values, along with $d_1 = d_2 = d_3 = 0$.

A delay-insensitive circuit, however, assumes arbitrary bounded delay existing in all circuit components. In Figure 1, this means the value of d_A, d_B, d_C, d_1, d_2 and d_3 are greater than zero. Circuit of this kind is certainly more robust than any other ones, because it works properly, regardless of delay of any amount that may occur anywhere in the circuit. Unfortunately, delay-insensitive attribute is very hard to achieve, and up until today, the class of all delay-insensitive circuits are limited.

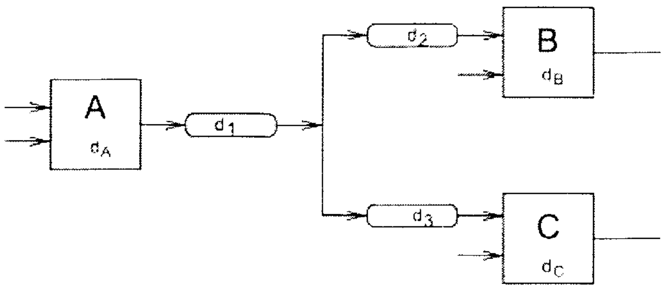

Figure 1. A circuit with wire and gate delays.

Nevertheless, some weak assumption can made to form a more flexible circuit. Still take a look at Figure 1, instead of assuming $d_1 = d_2 = d_3 = 0$, which is the case in delay-insensitive circuits, we assume only $d_2 = d_3 = c$, which is a constant but unknown value. This class of circuits is called quasi-delay-insensitive, and the property applied to a wire fork is called isochronic. Isochronic forks are those if the acknowledging target has seen a transition on their end of the fork then the transition is assumed to have also happened on the other end of the fork too. One advantage is that it allow signals to travel to two destinations and only receive an acknowledge signal from one. Apart from this, the assumption of isochronic is rather weak, since it can be achieved by implementing symmetrical structures in each branch, so that they tend to introduce the same amount of delay.

As a conclusion, by making a weak assumption, a quasi-delay-insensitive circuit is almost retains both robustness and adaptability, which ensures its wide range use throughout asynchronous circuit application.

2.2. Asynchronous communication protocols

In synchronous circuit, a global clock guarantees the safety and success of data transmission between different logic modules. However, when two asynchronous components (or two GALs) are getting communicated, it is essential to have some request and acknowledgement to signal senders and receivers, respectively, to assure the success of communication. We will discuss later in this section in specific communication protocol design with the lack of a global clock.

2.2.1. CHP, HSE notation and production rules

In this section, we will follow syntax of a high-level language called Communicating Hardware Processes (CHP) [18], which is widely utilized in most of asynchronous circuits behavior description. The HSE notation [19] will also be used as well. It has no distinction from CHP, except that it only accepts Boolean variables.

We will first describe some notations that are going to be used in the rest of the section, starting with communicating process. Figure 2 is showing two processes, p1 and p2, namely two logic modules working concurrently. A sending port from p1, S, sends out the logic value of a local variable x. We denote this sending procedure as R!x. On the other hand, a receiving port from p2, R, receives what has been sent, and stores it into its own local variable y. Again, this is denoted as R?y. Overall, an assignment $y := x$ is achieved.

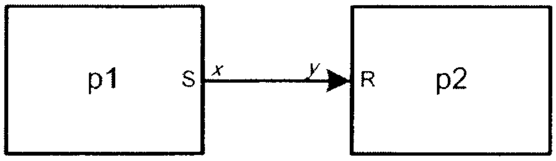

Figure 2. Communication of p1 and p2. Port S sends out the value of local variable x, and port R receives the value from S and assigns it to local variable y.

As stated above, an assignment has the form of *var* := *expr*. For a Boolean variable b, $b :=$ true and $b :=$ false can also be represented as $b\uparrow$ and $b\downarrow$.

There are two composition operators for processes. The sequential operator S1 S2, showing S2 carries out after S1; the parallel operator S1||S2, showing that S1 and S2 compose concurrently. Additionally, notation S1,S2 is also defined as a parallel operator but with noninterfering property: say a variable x is being written by S1, then x is guaranteed to be neither read nor written in process S2.

Another group of important notations are selection, wait and repetition. The selection is represented as [B1 $\rightarrow$ S1|B2 $\rightarrow$ S2], where each of B1 and B2 are called a guard, and each of S1 and S2 is a process. The selection works just like an if statement, as the value of each guard Bi is evaluated in the first place, and with the Bi whose value is true, the corresponding Si will be executed. Particularly, when neither of Bi is evaluated true, the whole process of selection is suspended. In this case, the selection waits for at least one guard to be true. A very straightforward example is [B0]. This selection waits B0 to be true and terminates afterwards, moving on to the following process. If an asteroid is added before a selection, the selection is repeated forever. e.g. $*$[1 $\rightarrow$ S0] will execute S0 forever.

Each circuit consists multiple logic gates, within which there are multiple inputs and one output (most likely). For some Boolean conditions, say Bu, the output z will be set to true. In other conditions, say Bd, z will be set to false. Write them in HSE notation, there are

$$Bu \rightarrow z\uparrow$$

$$Bu \rightarrow z\downarrow$$

Each row from above is considered as a production rule (PR). A production rule has the form of B $\rightarrow t$ where t is a binary transition, and B is a logic expression (also called a guard). All production rules of a single logic gate forms a production rule set (PRs) Obviously, a production set of a combinational gate can be inferred from its truth table. Another example of PRs is for a state-holding element: set-reset latch. Shown in Figure 3, it is constructed by two cross-coupled NOR gates, with two input s and r and two complementary output z and $\overline{z}$. When s is true, it sets output z to be true; when r is true, it sets complementary output $\overline{z}$ to be true. The production rule set of set-reset latch can be generalized as

$$s \rightarrow z\uparrow$$

$$s \rightarrow \overline{z}\downarrow$$

$$r \rightarrow z\downarrow$$

$$r \rightarrow \overline{z}\uparrow$$

Some restrictions should be applied to a PRs. First, complementary PRs must be noninterfering. For instance, in the above PRs, Bu and Bd can neither be true nor false at the same time, otherwise the value of z turns out ambiguous. The best way to resolve this contradiction is to set Bd $= \neg$Bu, thus z can only be true or false at a time.

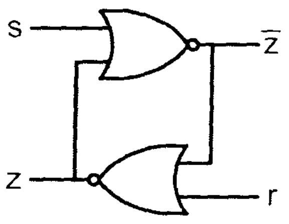

Figure 3. A set-reset latch with complementary outputs.

2.2.2. Bare handshake protocol

Figure 4 below is an easy implementation of a "bare" communication between p1 and p2. It is called "bare", because no data between these two processes are transmitted. Two wires $(s1, r1)$ and $(s2, r2)$ exist for synchronization in this implementation, in which the values of $s1$ and $s2$ are sent out from one process to another, and are respectively received by target process, eventually assign to $r1$ and $r2$. According to conventions, all variables are initialized to be false.

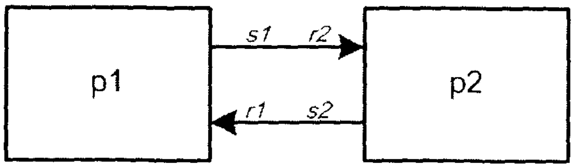

Figure 4. Implementation of bare handshake with two wires.

Based on the methodology of synchronization, bare handshake protocols are classified into two classes: two-phase handshake and four-phase handshake protocol.

1) *Two-phase Handshake* The two-phase handshake is the simplest implementation of asynchronous synchronization. The following sequence defines the behavior of this protocol:

$$Su: \ s1\uparrow;\ [r1]$$

$$Ru: \ [r2];\ s2\uparrow$$

Giving above behavior, there is only one possible transition sequence in two-phase handshake: $s1\uparrow; r2\uparrow; s2\uparrow; r1\uparrow$. Since all states in the system during the communication process should be meaningful somehow, and now only the $0 \rightarrow 1$ transition is defined, we continue to define the following protocol as well:

$$Sd:\ s1\downarrow;\ [\neg r1]$$

$$Rd:\ [\neg r2];\ s2\downarrow$$

From these two protocols, we discover the equivalence of transition $0 \rightarrow 1$ and $0 \rightarrow 1$ on all variables. We can then deduct a more general form of two-phase handshake protocol that is adaptive to both phases:

$$S:\ s1 := \neg s1;\ [s1 = r1]$$

$$R:\ [r2 \neq s2];\ s2 := \neg s2$$

The possible transition sequence of this protocol is: $s1\uparrow; r2\uparrow; s2\uparrow; r1\uparrow; s1\downarrow; r2\downarrow; s2\downarrow; r1\downarrow$ ... In spite of the protocol's simplicity, the implementation in terms of logic components is relatively complicated, since it requires XOR gates and storage element of current status. Hence, in most cases, two-phase handshake protocol is overtaken by four-phase handshake protocol, which we are going to introduce next.

2) *Four-phase Handshake* As stated above, a system should not include any meaningless states. To resolve this problem, it is straightforward to reset all variables to their initialized value before a communicating process ends. One kind of this handshake protocol is called four-phase handshake protocol. It works as follows:

$$S:\ s1\uparrow;\ [r1];\ s2\downarrow;\ [\neg r1]$$

$$R:\ [r2];\ s2\uparrow;\ [\neg r2];\ s2\downarrow$$

The only possible transition, in four-phase handshake, is now: $s1\uparrow; r2\uparrow; s2\uparrow; r1\uparrow$ $. s1\downarrow; r2\downarrow; s2\downarrow; r1\downarrow$. Note that it might seem identical to the transitions in two-phase handshake protocol. Here, the differences are: for two-phase handshake, a complete communication process consists four transitions, since both p1 and p2 acknowledge and react according to the transition of variables in spite of whichever the transition is; for four-phase handshake, different processes react according to the value of variables. As a result, considering the same time cost for each transition, the four-phase handshake takes twice as much time as the two-phase handshake.

2.2.3. Bundled data

When combining data transmission with synchronization, it comes to a hybrid communication protocol called bundled data. Figure 5 shows such an implementation.

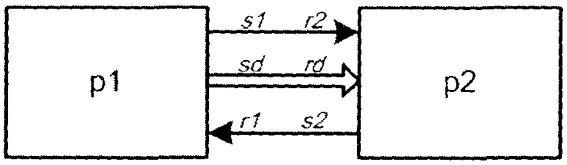

Figure 5. Implementation of bundled data with handshaking protocol and data transmission.

Consider the following circumstance: S sends out value of x ($S!x$), then R receives and assigns it to y ($R?y$). Use HSE notation to describe the communicating process:

$$S!x:\ sd := x;\ s1\uparrow;\ [r1];\ s1\downarrow;\ [\neg r1]$$

$$R?y:\ [r2];\ y := rd;\ s2\uparrow;\ [\neg r2];\ s2\downarrow$$

This communication is guaranteed by the synchronization of p1 and p2. First, p1 starts to send data to p2, meanwhile sending out the request signal by setting $s1$ to high. On the other hand, p2 waits for the request signal until $r2$ turns high. Since data rail is already valid when p2 receives request signal, p2 can then start to receive data. Right after the receiving step, p2 sends back acknowledge signal by setting $s2$ to high, and the following steps are the same as the four-phase handshake protocol. This protocol works under the assumption that the delay through wire $(s1, r2)$ is longer than (sd, rd), so that data signal arrives destination faster than the request signal.

2.2.4. Dual-rail code

Besides bundled data, there are still other options for safe asynchronous data transmission. Dual-rail code is one of them. In dual-rail code, two wires-bit 0 and bit 1-exist in representing one data bit. The table for representation is as follows:

value :	neutral	0	1
bit.0 :	0	1	0
bit.1 :	0	0	1

it can be inferred that for n-bit data transmission, there are necessarily 2n wires to finish encoding. This protocol is also delay-insensitive, since the communicating process can work reliably regardless of arbitrary delay in the wire, as long as it is finite.

2.2.5. 1-of-N code

In 1-of-N code, only one wire will be selected during the data transmission. Compared with dual-rail code, encoding n-bit data requires 2^n wires. A two-bit code-word is encoded using 1-of-N code is shown below:

2.3. PQ-cell, cellular automata and FPGAs

To conform to a wide range of computational needs, a computing system needs to scale from small to very large sizes. This need can be met by an extensible system of small computational elements. Cellular automata (CA) have these properties [20]

value :	neutral	0	1	2	3
d.0 :	0	1	0	0	0
d.1 :	0	0	1	0	0
d.2 :	0	0	0	1	0
d.3 :	0	0	0	0	1

- [22]. A CA cell is simple and the cells are arranged on a lattice that can have a periphery to which cells can easily be added. Similarly, PQ-cell arrays are scalable because, like CA, the cells are simple and are arranged on a two-dimensional lattice.

PQ-cell arrays are also like CA in that a cell has a state, and state transitions follow rules that are based on the states of nearby cells and possibly its own state. The state transitions occur when cells perform an update. In CA, updates are performed throughout the array in a parallel fashion, which may be either synchronous or asynchronous [23]. If synchronous, all the cells update their states once within each of a series of discrete time steps. Each step involves two phases: input and output. During the input phase, the cells input the states of certain nearby cells. During the output phase, the cells update their states. All the cells complete a phase before any of the cells move on to the next phase. If asynchronous, there is no global synchronization and updates depend on other factors. The PQ-cells update asynchronously in response to pulses sent by neighboring cells. By eliminating the need for global synchronization, the PQ-cell architecture is easier to scale to large sizes.

Computing with PQ-cell arrays can use any of the methods in use for CA. The two most common methods are digital circuit emulation and spatio-temporal modeling of a dynamic system. This paper focuses on circuit emulation; however, specially designed cells may be emulated by PQ-cell sub-arrays and used to model dynamic systems. The emulated cells then become the building blocks for larger computation structures such as cellular neural nets.

CA are typically uniform, which means all the cells follow the same rules for state transitions. Therefore, for digital circuit emulation, the usual approach is to create patterns of cells to perform the functions of wires, logic gates, and registers [24]. These patterns involve multiple cells and may take many cycles to advance a signal. PQ-cell arrays, however, are like non-uniform CA. The cells may have different rules. This allows a more effective method in which a single PQ-cell can perform the function of a wire, a logic gate, a storage element, or simple combinations thereof. By directly implementing these circuit elements, computation is faster and more compact. Furthermore, PQ-cells can be cascaded without intermediate storage elements. This optimizes the performance of multi-level combinatorial logic. Customization of a PQ-cell is achieved by loading a set of configuration bits. A similar initialization step is needed for CA, but only the initial state is specified. The configuration of a PQ-cell specifies its initial state, its transition rules (inputs and functions), and how it will synchronize parallel activity.

The ability to configure and re-configure the cells is a feature PQ-cell arrays share with FPGAs. The origins of the FPGA include the cellular arrays surveyed by Minnick in 1967 [25]. An early and enduring motivation for cellar arrays was to be able to use low cost batch processing methods. An accompanying idea is some form of configurability, which is needed to customize the array to a particular application. It was also recognized early on that it would be desirable to do this configuring "in the field", as opposed to in the factory, and ultimately to be able to configure repeatedly without removing or even having physical access to the device. These desires are now met by FPGAs and similar devices.

The PQ-cell explores a variation on the FPGA theme in which emphasis is on support for computational paradigms that deal directly with the spatio-temporal realities of a physical computing system. For some problems, including brain-like tasks like pattern recognition, this approach may lead to significantly improved performance and scalability. For this reason the cell designs developed so far have not adopted some of the features that optimize the mapping of arbitrary circuits onto an array. In particular, the PQ-cells route pulses through cells rather than through an interconnection network. Also, since it may be used for routing, and it is not yet clear what functions are needed, each quarter of a PQ-cell uses a simple 2-input logic unit rather than a 4- to 6-input look-up table.

Another difference is explicit support for asynchronous operation. Each PQ-cell contains a unit for synchronizing pulses. This unit also enables each cell to be configured as a stage in a pipeline for processing and transporting information. Pipelines can cross chip boundaries. This supports extreme scalability and allows portions of the array to operate at different speeds (for purposes such as local heat management). So a PQ-cell array is like an extensible FPGA whose reconfigurable elements are simple cells that communicate asynchronously with nearby cells to update their states.

CHAPTER 3. THE PQ-CELL DESIGN

This section presents the PQ-cell design, beginning with basic features and then focusing on facilities for processing and storing bits, routing pulses, coordinating parallel activity, maintaining pulse integrity, satisfying timing requirements, and configuring PQ-cell arrays.

3.1. Basic features: pulsed operation, quarters

At a high level, a PQ-cell is a unit that receives and sends pulses. It can receive a pulse at any one of four inputs and respond by sending a pulse on any number of four outputs. The received pulse may be interpreted as a bit of data or as a control signal. The sent pulses are always in response to a received pulse. So, without stimulation by a pulse, a PQ-cell does nothing. This is one reason PQ-cell arrays are expected to be efficient consumers of power.

A PQ-cell is called a quad-cell because it consists of four elementary cells called quarters. A quarter is the basic operational unit of a cell.

Fig. 6a shows a quarter (shaded box) and its connections to four neighboring quarters (open boxes). A quarter receives pulses from two quarters, one internal to the cell and one external. Likewise it sends pulses to two quarters, one internal and one external. Each pulse appears on one of a pair of wires, which is what allows the pulse to be interpreted as a bit. Figures 6b and 6c show how four quarters are combined to form a quad-cell and how the connections between quarters form the connections between cells.

3.2. Processing and storing bits: logic units and data latches

In response to a pulse from the internal quarter, a quarter records which wire the pulse arrived on. This record is stored in a data latch (RS latch) and becomes the

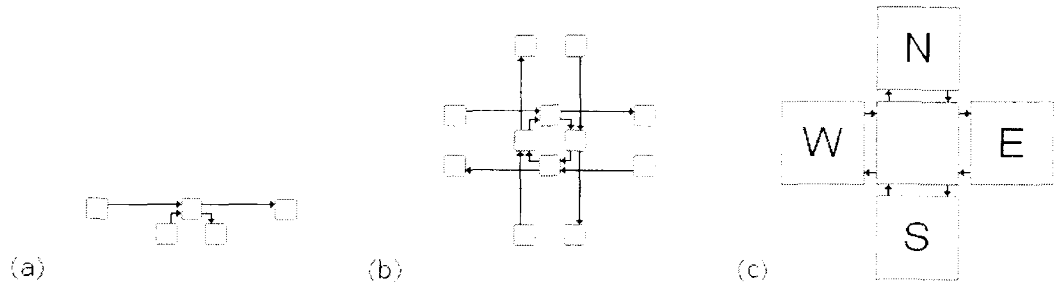

Figure 6. Where pulses are received from and sent to by (a) a quarter, (b) the four quarters of a cell, and (c) a cell. Each arrow represents a pair of wires.

B-input to a logic unit (LU) within the quarter. The LU also has an A-input, which receives pulses from an external quarter. Each pulse at the A-input causes the LU to form a result based on the A and B inputs. This result is represented by a pulse that is sent to an internal quarter and an external quarter. Fig. 7 gives an internal view of the quarters that shows the connections between the LUs and the data latches.

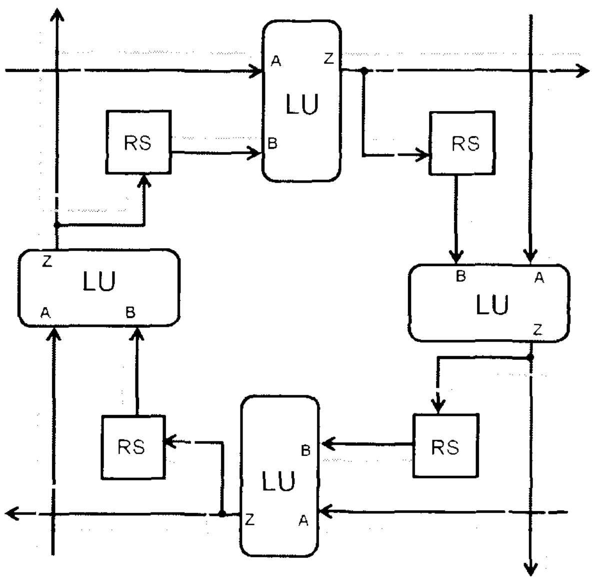

Figure 7. Logic units (LUs) and data latches (RS latches). To the A input and from the Z output, the black wires carry '1' pulses and the gray wires carry '0' pulses. To the B input, the black and gray wires are the Q and $\overline{Q}$ outputs, respectively, of the RS latch.

The dashed lines outline the quarters (shown as shaded boxes in Fig. 1b). Each data latch is implemented as an RS latch. The result formed by an LU is a logical combination (Boolean function) of the A and B inputs and appears as a pulse at the Z-output. The LU functions are listed in Table 1.

Table 1. The PQ-cell LU functions

D	E	F	G	Z	$\overline{Z}$
0	0	0	0	0	1
0	0	0	1	$A \cdot B$	$\overline{A} + \overline{B}$
0	0	1	0	$A \cdot \overline{B}$	$\overline{A} + B$
0	0	1	1	A	$\overline{A}$
0	1	0	0	$\overline{A} \cdot B$	$A + \overline{B}$
0	1	0	1	B	$\overline{B}$
0	1	1	0	$A \oplus B$	$\overline{A} \cdot \overline{B} + A \cdot B$
0	1	1	1	$A + B$	$\overline{A} \cdot \overline{B}$
1	0	0	0	$\overline{A} \cdot \overline{B}$	$A + B$
1	0	0	1	$\overline{A} \cdot \overline{B} + A \cdot B$	$A \oplus B$
1	0	1	0	$\overline{B}$	B
1	0	1	1	$A + \overline{B}$	$\overline{A} \cdot B$
1	1	0	0	$\overline{A}$	A
1	1	0	1	$\overline{A} + B$	$A \cdot \overline{B}$
1	1	1	0	$\overline{A} + \overline{B}$	$A \cdot B$
1	1	1	1	1	0

Columns D, E, F, and G correspond to configuration bits that select which one of the 16 functions is performed by the LU. Z and $\overline{Z}$ are the '1' and '0' output wires, respectively, of the LU. The expressions in the Z and $\overline{Z}$ columns specify which wire will carry the output pulse. For example, if DEFG = 0011, then Z = A means a pulse that enters the LU on the A wire will exit the LU on the Z wire. Likewise, $\overline{Z}$ = $\overline{A}$ means a pulse that enters the LU on the $\overline{A}$ wire will exit the LU on the $\overline{Z}$ wire. Another example: if DEFG = 0101, then Z = B and $\overline{Z}$ = $\overline{B}$. In this case there is no dependence on where the pulse enters the LU. If B = 1, the pulse exits on the Z wire and, if B = 0, the pulse exits on the $\overline{Z}$ wire.

Fig. 8 shows an implementation of half of the LU using combinatorial logic. It is essentially a selector that chooses some combination (either, neither, or both) of $\overline{A}$ and A to exit at Z. The choice is based on four configuration bits and the state of the data latch. The circuit for $\overline{Z}$ is equivalent.

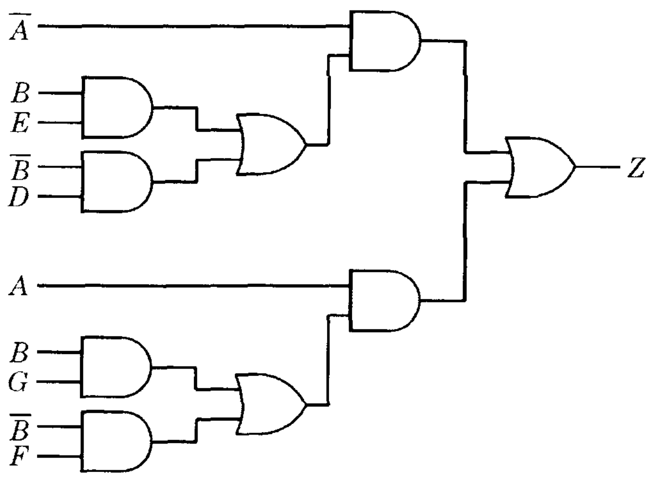

Figure 8. Implementation of half of the LU.

3.3. Routing pulses: paths, turns, crossovers, and forks

In its response to a pulse, a PQ-cell may send a pulse to one or more of its neighbors. A neighbor may, in turn, send a pulse to one or more of its neighbors, and so on. This sequence of operations forms a path through the array. It is necessary that a means be provided for steering pulses along these paths.

Pulse steering in a PQ-cell is achieved by using a selector to insert a right turn. Fig. 9 shows the cell with the selectors. Each selector chooses one of two sources for the A input of the LU. Since each quarter has a selector, up to three successive right turns can be made in a cell. Examples are shown in Fig. 10.

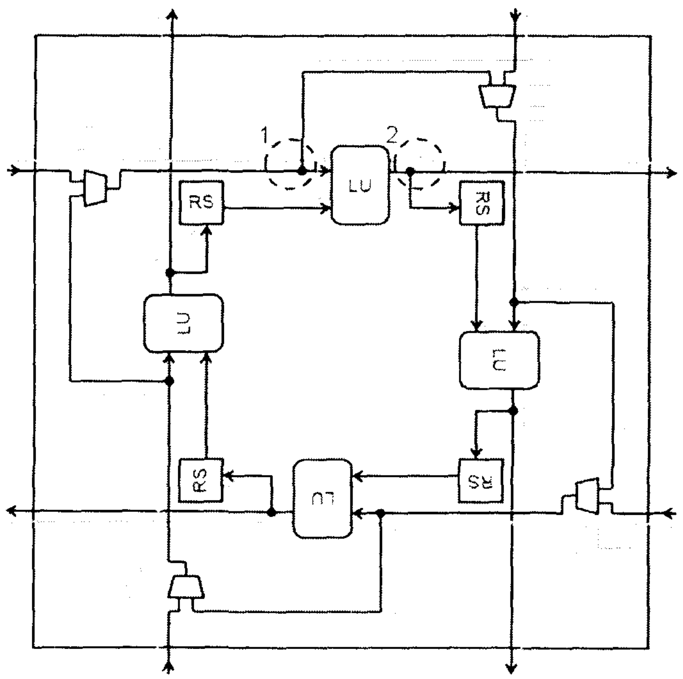

Figure 9. Adding selectors for making right turns. The numbered circles locate forks.

The cell configuration determines which source is connected to the A input. Since there are two wires coming from each source, there are two 1-of-2 selectors. For added flexibility, these selectors are configured independently. This is useful when routing control signals. To prevent a pulse from initiating further activity, a cell may configure its input selectors so that it does not accept pulses from that source.

A path may need to cross itself or another path. In a PQ-cell, this need is met by the connections between the quarters, which include four crossovers (Fig. 6b).

If a cell, in response to a single pulse, sends pulses to multiple neighbors, the cell is initiating parallel activity. This is called a fork. In a PQ-cell, a fork results when a quarter uses the turn selector to accept an internal input (Fig. 10b). This can happen at most three times in succession because there are four quarters and the input pulse must be accepted by one of them. So one input pulse could result in up to four output pulses, each of which heads in a different direction.

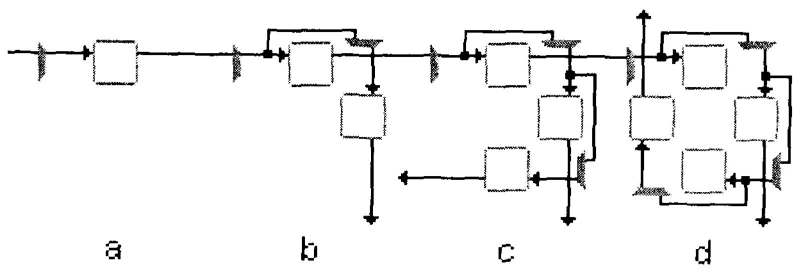

Figure 10. Using selectors to make turns of: (a) 0 degrees, (b) 90 degrees, (c) 180 degrees, and (d) 270 degrees.

3.4. Coordinating parallel activity

To coordinate parallel activity, the PQ-cell includes a synchronizing operation called a join. (It could also be called a rendezvous.) A join involves two or more quarters within a cell. A quarter participates in a join if it is configured to do so. A participating quarter is either ready or not-ready to send an output pulse. The join condition is satisfied when every participating quarter is ready. If a participating quarter is ready, it was either initialized to be ready or it became ready by performing an operation in response to an input pulse. Once the join condition is satisfied, each of the participating quarters outputs a pulse and becomes not-ready.

The PQ-cell implementation of the join operation is shown in Fig. 11. The additional circuitry is collectively referred to as the synchronizer. It has four configuration bits, J1, J2, J3, and J4, which indicate which cells are participating in the join. These bits also control the output selectors, choosing either the path from the LU (if a quarter is not participating in the join) or the switched path from the synchronizer (if a quarter is participating in the join). Each quarter has an RS latch that is set when a pulse exits the LU. This is called the event latch because it records an event of interest to the synchronizer. Also associated with each quarter is an OR gate whose output feeds into a 4-input AND gate.

The join condition is satisfied when the output of the AND gate, labeled R (for

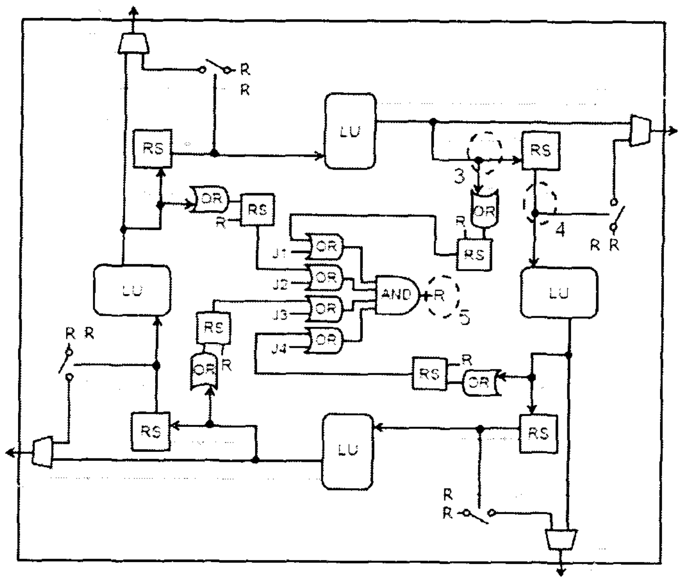

Figure 11. PQ-cell circuitry involved in performing the join operation.

reset), is high. Therefore the outputs of all the OR gates must be high. For each quarter, the OR gate output is high if the corresponding configuration bit is high (the quarter is not participating in the join) or the event latch output is high (the quarter is ready to output a pulse). When the join is satisfied, R is used to reset the event latches, which also causes R to return low. This produces a reset pulse that passes through the switch closed by an output of the data latch. So the effect of a join is to delay the LU outputs of participating quarters until the join condition is satisfied.

3.5. Maintaining pulse integrity

As a pulse travels along a path, its amplitude is restored each time it is re-driven. However, its width may get shorter or longer, depending on the relative speeds with which leading and trailing edges are generated by the circuitry. If a pulse becomes too short, it may fail to stimulate further logic and vanish. If a pulse becomes too long, it may interfere with other pulses. Therefore some means is required for maintaining pulse width. This is the purpose of the pulse regenerator (PR).

The PR outputs a pulse of width W in response to an input pulse that may be shorter or longer than W. Fig. 12 shows one form for the PR. It has two delays, D1 and D2. This circuit outputs a pulse of width W = D2 - δ (where δ is the delay through the first NOR gate) in response to an input pulse whose width is at least D1, where $W/2 < D1 < W$. Choosing D1 near W/2 allows for narrower input pulses.

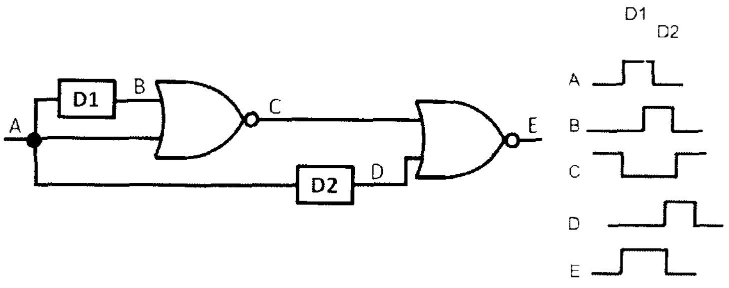

Figure 12. A design for the pulse regenerator (PR).

A PR is at every output from a cell. Fig. 13 is a composite diagram of the PQ-cell that shows where the PRs are located.

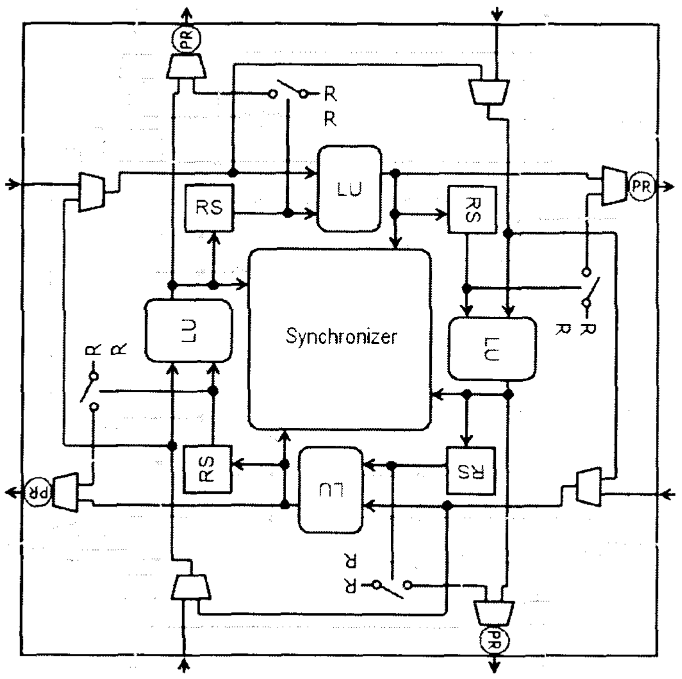

Figure 13. Composite diagram of the PQ-cell including a pulse regenerator (PR) at each output.

3.6. Satisfying timing requirements

Correct operation of computations in PQ-cell arrays requires certain timing requirements to be observed. Pulse width requirements are managed by the pulse regenerator. Pulse separation is managed by handshaking such as in pipelines. A remaining requirement is to ensure that the B-input to an LU is set before a pulse arrives at the A-input. The output of the LU is produced in response to a pulse at the A-input. Since the B-input prepares the LU to produce this response, it must arrive a short time before the pulse at the A-input. Fig. 14 shows three ways to achieve this.

The first solution depends on other cells to delay the A-input relative to the B-input (Fig. 14a). The other solutions use the join operation and are independent of delays external to the cell. In these solutions, the path leading to the A-input passes through the cell before re-entering the cell and delivering a pulse to the A-input. This path includes an LU that participates in a join with the LU that delivers the B-input (Fig. 14c). If one LU is the source of both inputs, then only that LU participates in the join (Fig. 14b).

The solutions using the join are dependent on the design of fork 3. Fork 3 is one of five forks within the PQ-cell whose locations are circled and numbered in figures 9 and 11. Each fork is a point at which a signal simultaneously enters two or more paths. If these paths interact within the cell, and the result of this interaction depends on the order in which the signals arrive, then the cell needs to be designed to ensure a consistent outcome.

Fork 3 creates two paths, one that goes to the B-input of the LU and one that exits the cell under the control of the synchronizer. By involving the synchronizer, the delay in the shortest path from fork 3 to the A-input (which involves exiting then re-entering the cell) is guaranteed by design to exceed the time required for the

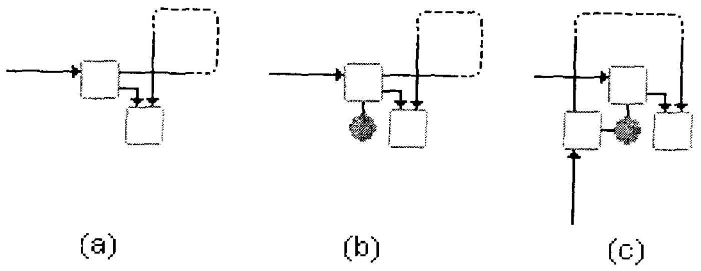

Figure 14. Three ways to properly order the two inputs to an LU: (a) introduce sufficient external delay; (b) supply both inputs from an LU within the same cell that participates in a join; (c) supply the A-input from an LU that participates in a join with the LU that supplies the B-input.

B-input to set up the LU. Therefore, even if there are no external delays, the input order is still correct.

Another design consideration for fork 3 results from an internal interaction between its two paths. One path sets the data latch and the other path could cause the synchronizer to generate a reset pulse. Since the data latch sets up the route by which the reset pulse exits the cell, the path through the data latch must be shorter than the path through the synchronizer.

Fork 1 is also of interest. This fork is located at the input of the LU in the N quarter and creates two paths that may come together at the LU in the E quarter. One path leads to the B-input and the other path (if chosen by the input selector) leads to the A-input. The path to the A-input is shorter, so the LU should be configured to perform a function that uses only the A-input; i.e. Z = 0, 1, A, or $\overline{A}$. Forks 2, 4, and 5 create independent paths and therefore present no special timing issues.

3.7. Configuring PQ-cell arrays

The behavior and initial state of a cell are determined by a set of configuration bits that are loaded into the cell before it is used. A simple way to load these bits is to shift them serially into a long shift register that contains all the bits of all the cells in the array. However, this would be a slow process, especially for large arrays.

A faster and more flexible scheme is envisioned for PQ-cell arrays.

A PQ-cell array has many short shift registers. Each register holds the configuration bits for a subset of the array, which may be a single PQ-cell. Each quarter has 9 configuration bits: 4 for the LU, 2 for the input selector, 1 for the data latch, 1 for the event latch, and 1 to specify whether or not the quarter participates in the join.

Each register is at a node of a 2D mesh network. Serial streams of configuration bits pass through the network to reach selected registers, bypassing registers that are not configured by that stream. Multiple streams may be in the network at the same time. Furthermore, cells that are not being configured may continue to operate.

Fig. 15 shows a single node in the configuration network. In this case the node is associated with a single PQ-cell. The controller routes incoming data bits to one of three shift registers or to another node. The first bit to arrive determines whether or not this node will receive configuration bits. The second bit identifies the end of the stream. The next two bits select the next node to be visited by the stream. Then, depending on the first bit, configuration bits for this node will follow. Finally, depending on the second bit, any additional bits are passed on to the selected node.

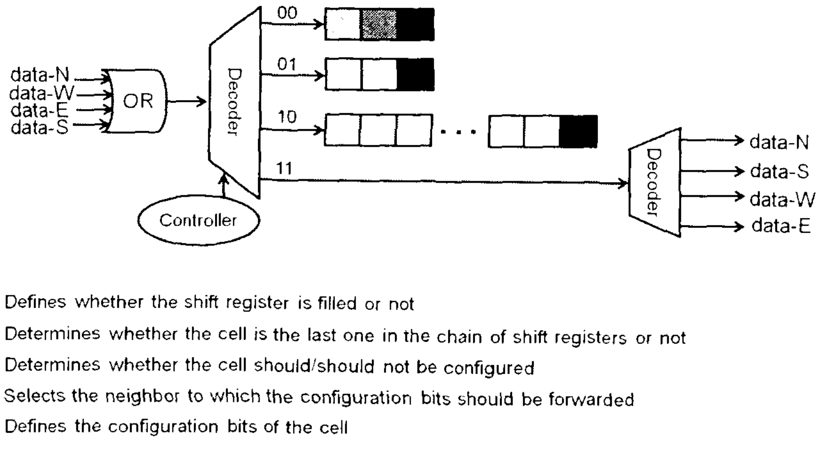
data-N
data-W
data-E
data-S
OR
Decoder
Controller
00
01
10
11
Decoder
data-N
data-S
data-W
data-E
■ Defines whether the shift register is filled or not
▨ Determines whether the cell is the last one in the chain of shift registers or not
□ Determines whether the cell should/should not be configured
□ Selects the neighbor to which the configuration bits should be forwarded
□ Defines the configuration bits of the cell

Figure 15. The behavioral model of the configuration circuitry.

CHAPTER 4. SIMULATIONS OF THE PQ-CELL

An IC design of the PQ-cell was made for fabrication in TSMC's 0.25 μm CMOS technology. This design was simulated to obtain the expected propagation delay and energy consumption of a single cell. Several multi-cell structures were also designed and simulated to exercise the cells wehn they function together in an array. This section describes four of these simulations: a single cell, a full adder, a ring oscillator, and a pipeline. A pipeline is an important structure and a good example of how the join works. Finally, at the end of this section, PQ-cell performance is compared with that of asynchronous FPGA logic cell designs of other researchers.

4.1. Single cell

A single cell was simulated to find the minimum pulse width, the propagation delay through a cell, and the energy consumed in a single operation, and the energy consumed in a single operation, and to explore the effect of supply voltage on these measures. The minimum pulse width was determined to be about 550 ps for supply voltages between 1.8 V and 2.5 V. The temperature was 25°C. This result was independent of the cell function.

For the other measures, the cell was configured to perform the XOR function. The B-input was set to zero and pulses were supplied to the A-input. The input pattern was a series of pulses alternating between the '0' wire and the '1' wire. The pulses were 700 ps wide and were separated on each wire by 800 ps. This is an input pulse rate of 1.3 GHz. The results are shown in figures 16 and 17.

Fig. 16 shows the input and output waveforms when operating at 2.5 V. The propagation delay through the cell is 1.1 ns. At 1.8 V the propagation delay is about 1.5 ns. Fig. 17 shows the current profiles when operating at 2.5 V and 1.8 V. At 2.5 V the average current is approximately 2.5 mA over a period of 2.5 ns, so the energy

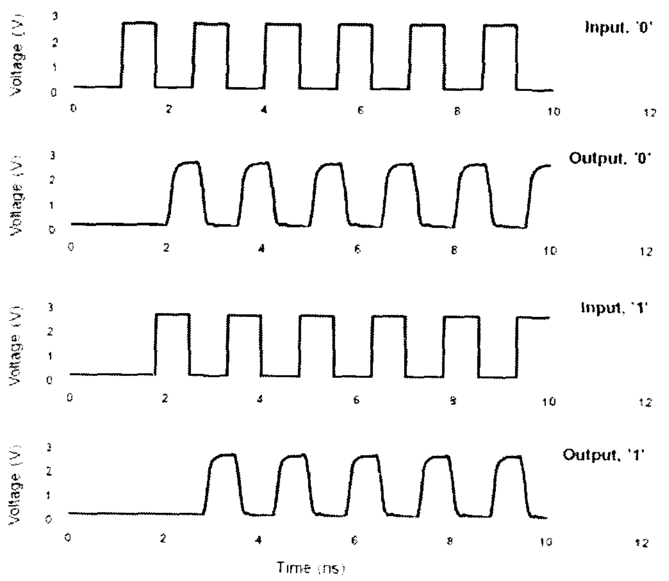

Figure 16. Simulation results of a PQ-cell performing an XOR function.

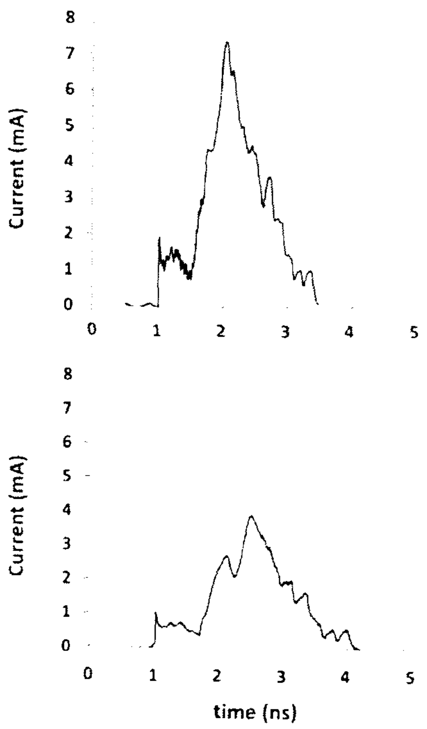

Figure 17. The PQ-cell current profile at (a) 2.5 V and (b) 1.8 V. The profile at 1.8 V is lower (average is 1 mA) and longer (by about 500 ps).

consumption per pulse is 15.6 pJ. At 1.8 V the average current was approximately 1.5 mA over a period of 3.2 ns, so the energy consumption per pulse is 8.6 pJ. This is significantly less than the consumption at 2.5 V. The tradeoff is a modest increase in propagation delay. Note that energy consumption is essentially zero when there are no pulses. This is one of the benefits of asynchronous circuits.

4.2. Full adder

The full adder is an example of using multiple PQ-cells to perform combinatorial logic. Fig. 18 shows how the adder was constructed. The inputs to the adder are A, B, and C. C is carry in. The Sum is formed as $A \oplus B \oplus C = A \oplus (B \oplus C)$. The Carry (carry out) is formed as $AB + AC + BC = A(B + C) + BC$. The supply voltage was set to 2.5 V. Input pulses representing a '1' were sent to the C, B, and A inputs, in that order, at 2 ns intervals. The output pulse for Sum appeared after a 1.1 ns delay and the output pulse for Carry appeared after a 3.6 ns delay. These results are consistent with what was expected considering the propagation delay through a single cell.

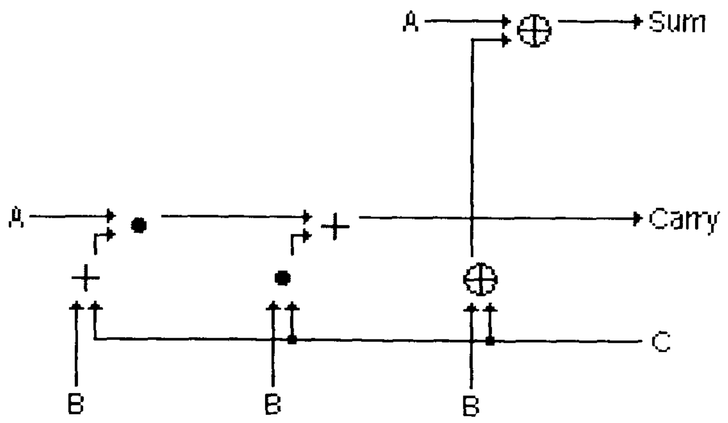

Figure 18. Full adder constructed from four cells in a PQ-cell array.

4.3. Ring oscillator

The ring oscillator is a loop. Four cells were used in its construction (Fig. 19).

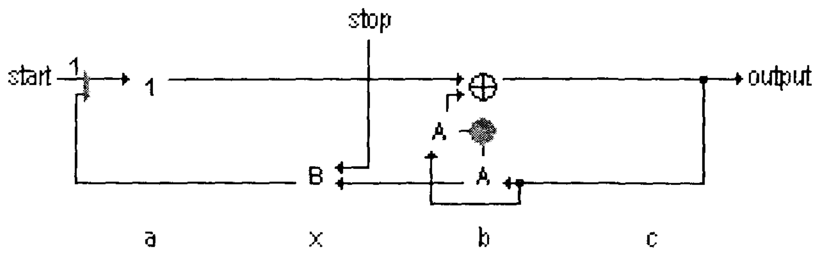

Figure 19. Ring oscillator constructed from four PQ-cells.

The oscillator has a start input, a stop input, and an output. The oscillator is started by supplying a '1' pulse at the start input. The input selector is configured to accept '1' pulses from an external source and '0' pulses from an internal source. The input pulse causes the N quarter of cell a to output a '1' pulse, which passes through the N quarter of cell x and triggers the N quarter of cell b. The N quarter of cell b outputs the complement of the latest value it received from the W quarter; initially, this is a 0. Cell c receives the output from cell b, duplicates it, and sends one copy to the output and the other copy back to cell b. Cell b duplicates this input and stores one copy as the B-input of the N quarter. The other copy goes to the S quarter of cell x which outputs a pulse with the value at its B-input. If this value is a 0, the oscillator continues; otherwise it stops. Also note that the S and W quarters of cell b participate in a join. The effect of this is to delay the cell output from the S quarter until the W quarter has readied its output. This ensures the proper arrival order of the inputs to the N quarter.

When simulating the oscillator, the data latches are initially zero. Nothing happens until a '1' pulse enters the start input. This causes the first output, which is the complement of the data latch at the B-input of the LU in the N quarter of cell b. The simulation results are shown in Fig. 20. The output pulse rate is approximately

98 MHz at 25°C. The oscillator was also simulated at −25°C and 65°C, and the output pulse rates were 119 MHz and 84 MHz, respectively. This inverse relationship is due to decreases in transistor current as temperature increases.

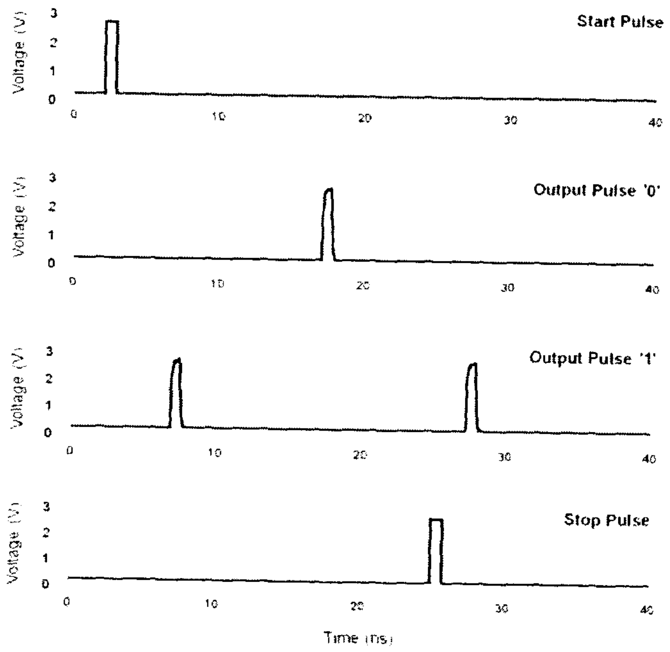

Figure 20. Simulation results of the ring oscillator.

4.4. Pipeline

Pipelines are important structures for transporting and processing data. In particular, asynchronous pipelines, because of their ability to store variable amounts of data, can form elastic connections between computations at different locations within a PQ-cell array. Event-driven elastic pipelines, with or without internal processing, was the subject of Sutherland's 1988 ACM Turing Award lecture.

PQ-cells are readily configured as symmetric pipelines that can operate in either direction. Fig. 21 shows a series of PQ-cells forming three stages of a pipeline. Refer to the quarters by their compass locations within a cell: N, S, E, and W. Then this pipeline involves the N and S quarters in each cell. These quarters are configured to participate in a join. To cause the pipeline to operate in the West to East direction, the N quarters are initialized to not-ready and the S quarters are initialized to ready.

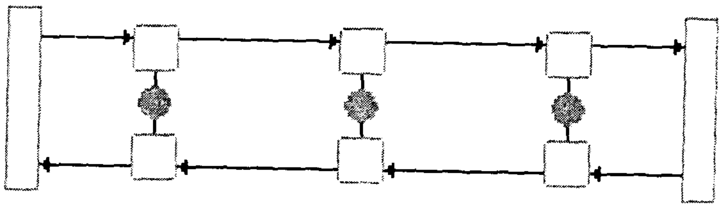

Figure 21. Three stages of a pipeline. Each stage uses two quarters and the synchronizer (filled circle) of a PQ-cell.

For any cell in the initial state, if a pulse arrives at N, the join is satisfied and two pulses are sent, one from N and one from S. If the input pulse is interpreted as data, then the pulse from N is interpreted as data being passed to the next stage and the pulse from S is interpreted as a signal being passed to the previous stage. The new state of the cell is that both quarters are not-ready. The next state depends on which event occurs first; either the cell receives a data pulse from the previous cell or the cell receives a signal pulse from the next cell. But one event will not satisfy the join. Only when both events have occurred is the join satisfied and the cell outputs another pair of pulses: a data pulse to the East and a signal pulse to the West.

A 3-stage pipeline was simulated to see how fast it would run. It was configured as a loop, with a wire connecting the output to the input at each end. By inserting an initial pulse, the pipeline circulated the data that was initially in the data latches. The resulting pipeline speed was 190 MHz. This can be improved by integrating pulse width control (section III.E) with other circuitry, so this figure should be viewed as a lower bound.

4.5. Comparisons

To obtain a view of a PQ-cell's performance compared to other work, the asynchronous FPGA logic cells presented by Wong et al. [26], Teifel and Manohar [27], and Mahram et al. [28] were studied. These were chosen because they are examples of asynchronous cell designs and because the authors included estimates of speed and energy consumption. These estimates are summarized in Table 2, along with those of the PQ-cell. The frequency given for the PQ-cell is the simulated pipeline speed. The PQ-cell energy consumption is from the single-cell simulation when operating at 1.3 GHz.

Table 2. Speed and energy consumption of various asynchronous cell designs

Design	Technology	Voltage	Speed(MHz)	Energy(pJ/cycle)
Wong	TSMC 180	1.8	190-235	2.1-3.1
Teifel	TSMC 250	2.5	400	18
Mahram	TSMC 180	1.8	280	2.2
PQ-cell	TSMC 250	2.5	385	16.9

There are significant architectural differences between the designs, so it is difficult to make meaningful conclusions. Even so, it is interesting to see that the results are relatively close. The biggest difference is the low energy consumption of the Wong and Mahram FPGAs, but at least a factor of 2 can be attributed to the technology and supply voltage. For example, an FPGA cell described in [29] consumed 18 pJ/cycle at 250 nm and was expected to improve to 7 pJ/cycle at 180 nm. A similar improvement can be expected for the PQ-cell. Future work will need to make a more careful comparison to see what can be learned by studying the architectural variations and their ramifications.

CHAPTER 5. OPTIMIZING PQ-CELL

5.1. Logic unit

Figure 22 is the latest schematic of LU. D_0 and D_1 are delays that are part of the pulse width control circuitry. The LU routes a pulse arriving at one of its two inputs, A_1 and A_0, to one of its two outputs, Z_1 and Z_0. The route is determined by D, E, F, G, B, and $\overline{B}$. Pulses can be prevented from passing through the LU by setting the enable signal low. The major improvement of this LU is that it has a built-in circuitry to adjust the pulse width, by trimming the incoming pulse before stretching again. Consequently, neither would the pulse width be too long, nor will it be shorten to disappear after passing multiple cells. In addition, it maintains pulse integrity by introducing only two gate delays, which are two units less than the original design built with pulse generator.

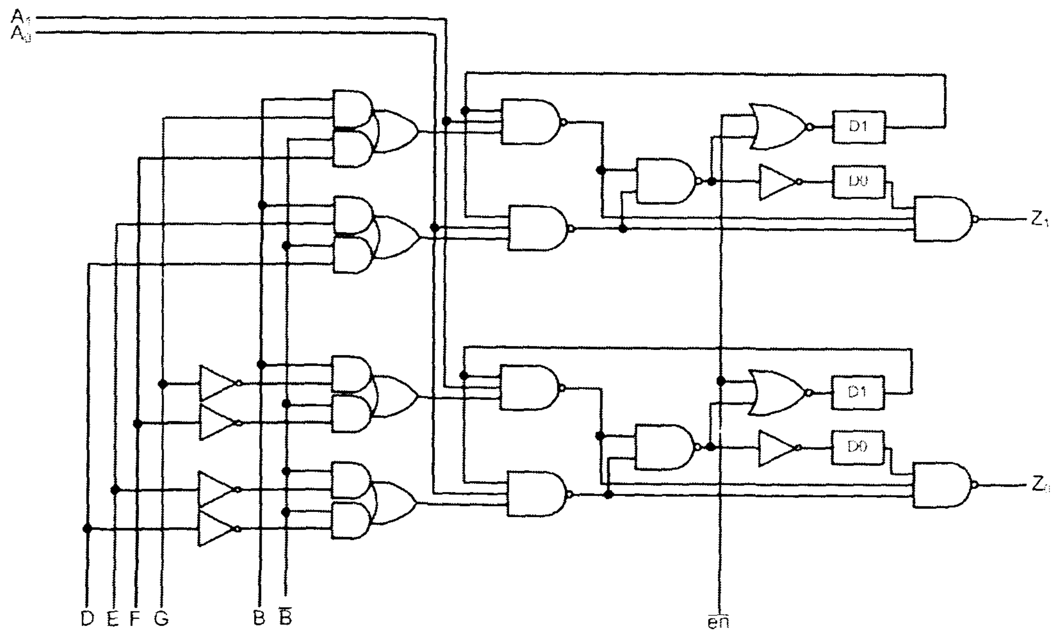

Figure 22. The modified version of LU.

The built-in pulse width control circuitry consists of two parts: pulse trimmer and pulse stretcher. We simplified a subset of the LU for timing analysis, shown in Figure 23. The first 3-input NAND gate and loop AA'a forms the trimmer. When a pulse with exceeding width comes, the trimmer will generate a new inverted pulse with a relatively narrow and fixed width, by carrying out NAND function between the original pulse and its delayed copy. The stretcher is achieved by carrying out NAND function between the trimmed pulse and its delayed copy. Since all the delay elements are fixed in this circuit, we can successfully generate a new fixed-length pulse.

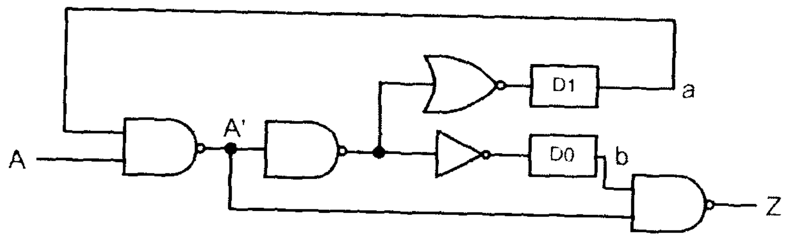

Figure 23. Simplified LU for timing analysis.

However, the width of input pulse not staying in a certain range could lead to a malfunction. For instance, a very narrow incoming pulse will not be stretched but ending up with two separate pulses. Likewise, large pulse width will result in an oscillation due to the existence of the inverting loop. For simplicity, assume $D_0 = 0$ and $D_1 = 0$, and let G = one gate delay. Let D_a = delay from A to a $= \delta_{NAND} + \delta_{NOT} + D_0 = 2G + D_0$. D_b = loop delay $= \delta_{NAND} + \delta_{NAND} + \delta_{NOR} + D_1 = 3G + D_1$. W_I = width of the input pulse and W_O = width of the output pulse. If $W_I < D_a$, the input pulse is too short to stretch. Consequently, there will be two output pulses of width W_I with leading edges separated by D_a. If $D_a < W_I < D_b$, incoming pulses are stretched appropriately, but too short to be trimmed, thus $W_O = W_I + D_a$.If $D_b < W_I < 2D_b$, both the trimmer and stretcher successfully adjust the pulse width. In this case, $W_O = D_b + D_a$. If $W_I > 2D_b$, the loop forms a 3-stage ring oscillator.

in which multiple pulses are generated out of a single long pulse. To meet timing requirement, $D_a < W_I < 2D_b$ should be satisfied. These two statuses also suffice to be considered "steady", since for every W_O there is $D_b < W_O < 2D_b$, which also helps keep the consistency of outgoing pulse width, too. The acceptable range of input pulse width can be increased by adding up more delay to D_b, with the tradeoff of increasing the minimum input pulse width as well.

5.2. Configuration circuitry

Due to the area-consuming property of the originally proposed, which takes twice as much area as the functional circuit, a new configuration circuitry with high configuration efficiency is proposed in this section. There are 11 bits for each of four quarters and 1 bit shard by a whole cell. The configuration bits for a quarter are: 1 for the initial condition of the data latch; 4 to specify the LU function; 1 for the initial condition of the event latch; 1 to specify whether or not the quarter participates in the join, and 4 for the input multiplexer (1 each for the '0' and '1' inputs to select between straight and a turn, and 2 to select one of four turns). The configuration bit shared by the cell is: 1 bit for split join. Each cell has two configurations: a default configuration and a programmable configuration. The bits for the programmable configuration are stored in a shift register. The bits for the default configuration are wired in. A "default" input chooses between the two configurations. If "default" is true, the default configuration is selected; otherwise the programmable configuration is chosen.

5.2.1. Default configuration

Default configuration is accomplished by a selection circuitry, which is a collection of the following circuits. In each case, if "default" is false (low), the output is determined by the programmable configuration bit and, if "default" is true (high), the output is determined by the circuitry.

Figure 24(a) and (b) are used to supply levels. Figure 24(a) is used when default is zero. Likewise, if default is one, Figure 24(b) is used. Figure 24(c) and (d) are used to supply a pulse to one of two outputs. Figure 24(c) is used if the default is to send a pulse to zero_out. Figure 24(d) is used if the default is to send a pulse to one_out.

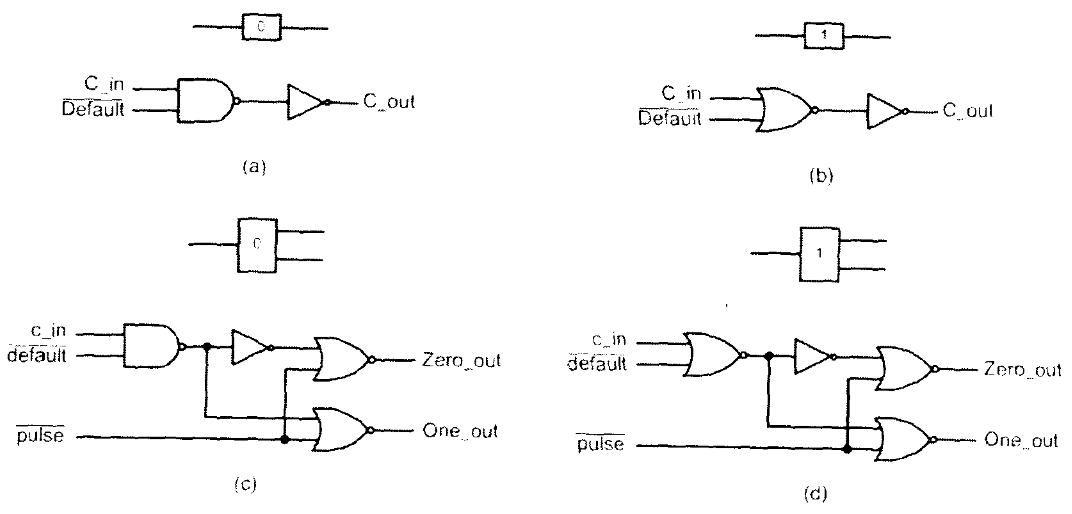

Figure 24. Default selection for (a) set default to zero; (b) set default to one; (c) set default pulse exit zero_out; (d) set default pulse exit one_out.

5.2.2. Programmable configuration

The programmable configuration of each cell is stored in a shift register. Each stage in the shift register is a D-latch (Figure 25, in which a "high" clock signal triggers input passing through to the output (known as the "read" state), while a "low" clock signal disconnects input from the output node, (known as the "store" state). To store a copy of its input, the latch goes through a store-read-store cycle. The input must be steady while the latch is about to reach the "read" state. Moreover, a long shift register can be constructed by cascading the shift register elements and causing each stage store a copy of the output of its predecessor in a ripple-like fashion, starting with the end opposite from the data input. Figure 26 is an example of a 3-stage shift register. Data enters from one end of the chain, while shift pulse enters from the other end. With sufficient delay between stages, a negative at $\overline{G}$ shift pulse will cause stage N+1 to store a copy of the output of stage N.

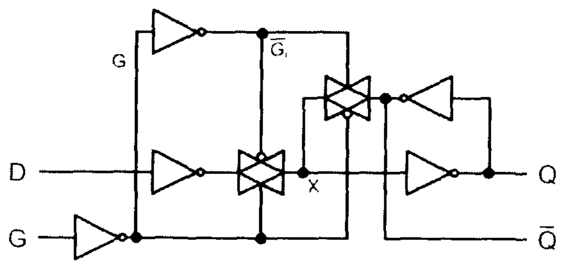

Figure 25. Schematic of a D-latch with complementary outputs constructed by transmission gates.

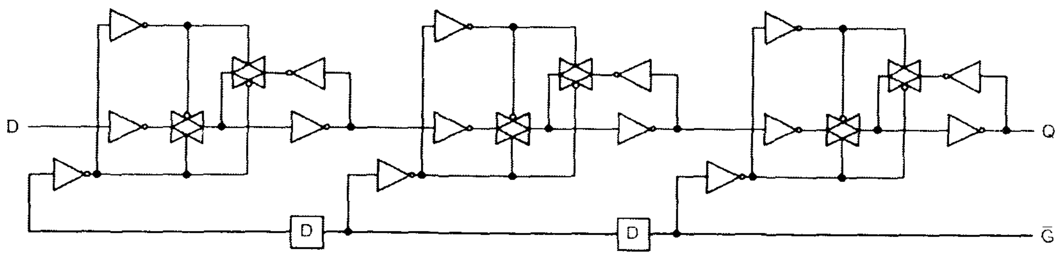

Figure 26. A 3-stage shift register chain.

As described before, the inter-stage delay should be "sufficient". To determine this value, an expression is needed for this inter-stage delay. Let d be the delay through the delay element, let g be the delay through an inverter, let p be the time to pass through a switch, and let s be the time to open or close a switch (assume they are equal). Consider a negative pulse of width W at $\overline{G}$ of stage N+1. The leading edge of this pulse arrives at each successive stage after a delay of d. If d is sufficiently long, stage N+1 will enter the store state before stage N is able to affect node X of stage N+1. First looking at stage N, the leading edge of the pulse will cause stage N to begin the transition to the read state when it reaches the input switch. This transition will cause the D input to stage N to pass through the input switch.

through the Q output inverter, and appear at the Q output. This new output will appear at node X after it passes through the input inverter and input switch of stage N+1. So the shortest path (from $\overline{G}$ of stage N+1, through stage N, and to node X of stage N+1) goes through a delay element, through the $\overline{G}$ input inverter of stage N, closes the input switch (control input to data output), and goes through the Q output inverter, the D input inverter of stage N+1, and the input switch (data input to data output). The length of this path is $d + g + s + g + g + p = 3g + d + s + p$. Within this time, the trailing edge of the pulse must open the input switch of stage N+1. This, along with closing of the loop switch, causes stage N+1 to enter the store state. The longest path from $\overline{G}$ to node X passes through the /G input inverter, through a second inverter (to produce $\overline{G_i}$), and opens the input switch (control input to data output). The length of this path is $g + g + s = 2g + s$. So the needed relationship is $W + 2g + s < 3g + d + s + p$. This reduces to $W < d + g + p$. If $p = 0$, then $W < d + g$.

Below in Figure 27 is the latest design for storing the configuration bits of a quarter. The delay element is a driver with a delay, d, of $2g$. The initial pulse is generated from a low-to-high transition of the input using a "one-shot", which implemented as below in Figure 28. The generated pulse width $W = (1+2K)g$ where K is an integer. Right here we have $K = 1$, so that $W = 3g$. Several issues arise with this design. First, as shown below, in order for data bits to propagate through the shift registers, $W < d + g$ should be satisfied. In other words, there exists $d > 2g$. This could be achieved by slight decreasing both the PMOS and NMOS width within each driver. Second, throughout a entire cell, there are four quarters of configuration circuitry cascading together. For a shift pulse propagate through "smoothly", the same amount of delay between each D-latch unit should be guaranteed. The only exception occurs only when a shift pulse exiting one quarter then entering another. To prevent this exception happening, $d_i + d_o = d$ should be strictly satisfied, in which

d_i and d_o are delay of last inverter and "one-shot" circuit, respectively. Third, since "one-shot" can only shorten a pulse by controlling the width, but cannot lengthen a pulse that is already short. This feature requires delay elements carefully sized to lengthen pulses so that a shift pulse does not die out in the middle.

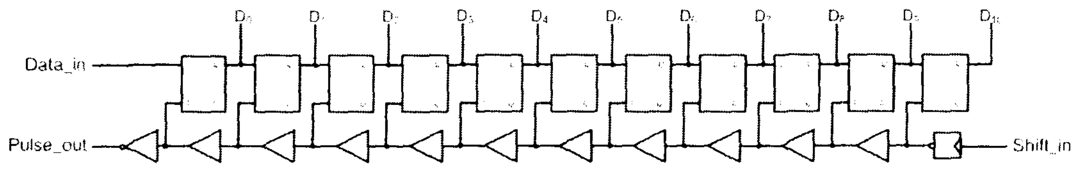

Figure 27. Configuration circuitry for a quarter.

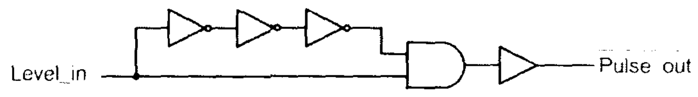

Figure 28. An "one-shot" circuity, generating an inverting fixed-length pulse from a level input.

Below in Figure 29 is a configuration circuitry of an entire cell. Note that there is one configuration bit remained for split-join. An extra one-shot circuit and an inverter are added in order to keep the delay consistency between each individual register, when considering an even longer shift register chain along multiple cells.

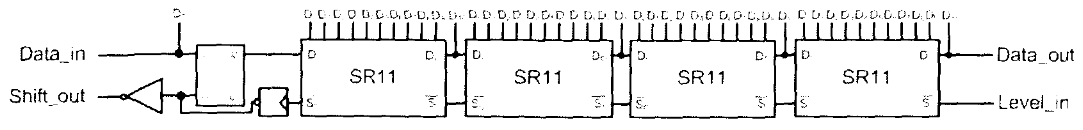

Figure 29. Configuration circuitry for a cell.

CHAPTER 6. SUMMARY AND CONCLUSIONS

The PQ-cell is the latest member of a set of exploratory designs for a simple reconfigurable computing element for cellular-automata-based conformal computing. A single cell includes facilities for performing logic, moving and storing information, and coordinating parallel activity. The PQ-cell is a dual-rail pulse-driven asynchronous primitive that combines stimulus and data in a single pulse that appears on one of two rails (wires). This dual-rail design eliminates any need to maintain a timing relationship between separate stimulus and data signals.

A novel feature of the PQ-cell is its quad-cell design which consists of four elementary quarters, each with a different compass orientation, that share synchronization and configuration circuitry. Each quarter includes a 1-bit storage unit and a logic unit capable of performing any one of the 16 possible functions of two bits. The P-cell, a single cell with 4-fold rotational symmetry, is also being considered and future work will include a careful comparison between these two designs.

The cells are designed to be elements of extensible cellular arrays in which communication takes place directly between neighboring cells. Accordingly, arrays of PQ-cells can be configured to form a wide variety of computational structures including cellular automata and FPGA-like circuits. Because there are no global signals or long wires, and pulse integrity is maintained by the cells, the arrays can be extended to very large sizes. Also, to make the cell configuration process extensible, it was designed to be a selective and highly parallel activity.

In addition to a functional design, this paper presented simulation results for an IC design of the PQ-cell. The design, intended for fabrication in TSMC's 0.25 μm CMOS technology, was used in simulations of basic single and multiple cell structures, including an XOR gate, a full adder, a ring oscillator, and a pipeline. The simulations were important for testing the correctness of the design and for obtaining estimates of speed and power. Comparisons with other work indicate that the PQ-cell is competitive in performance and energy consumption.

The PQ-cell design is a good step toward a reconfigurable asynchronous primitive that can serve as the elemental unit in extensible arrays for conformal computing. The next steps are to refine the design and move on to layout and fabrication of a prototype array chip. Many additional steps are needed to adequately address issues related to the performance, cost, programming, and use of systems employing these arrays. The hope is that this path will lead to an extensible material that is a superior host for computation in applications ranging from small low-power sensors to large high-throughput pattern processors.

REFERENCES

[1] J. Lyke, G. Donohoe, and S. Karna, "Reconfigurable Cellular Array Architectures for Molecular Electronics," Air Force Research Laboratory, Technical Report, AFRL-VS-TR-2001-1039, 2001.

[2] S. Das, G. Rose, M. Ziegler, C. Picconatto, and J. Ellenbogen, "Architectures and Simulations for Nanoprocessor Systems Integrated on the Molecular Scale," in *Introducing Molecular Electronics*, pp. 479-512, Springer Berlin/ Heidelberg, 2005.

[3] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," in *Science*, vol. 266, pp. 1021-1024, 1994.

[4] M. Sipper, "The Emergence of Cellular Computing," in *Computer*, vol. 32, issue 7, pp. 18-26, Jul. 1999.

[5] P.-A. Mudry, F. Vannel, G. Tempesti, and D. Mange, "CONFETTI : A Reconfigurable Hardware Platform for Prototyping Cellular Architectures," in *IEEE Intl Parallel and Distributed Processing Symp. (IPDPS 2007)*, pp. 1-8.

[6] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R.Weiss, "Amorphous Computing," in *Communications of the ACM*, vol. 43, pp. 74-82, 2000.

[7] H. Abelson, J. Beal, and G. Sussman, "Amorphous Computing," in *Technical Report*, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, MIT-CSAIL-TR-2007-030, Jun. 2007.

[8] M. Pavicic, "Wallpaper Computers: Thin, Flexible, Extensible, and R2R Ready," in *Flexible Electronics and Displays Conference*, Phoenix, AZ, Feb. 2009.

[9] N. Margolus, "CAM-8: A Computer Architecture Based on Cellular Automata," in *Pattern Formation and Lattice Gas Automata*, pp. 167-187, 1996.

[10] T. Toffoli, "A Pedestrians Introduction to Spacetime Crystallography," in *IBM J. Res. and Dev.*, vol. 48, no. 1, pp. 13-29, Jan. 2004.

[11] N. Macias and P. Athanas, "Application of Self-Configurability for Autonomous, Highly-Localized Self-Regulation," in NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2007), pp. 397-404, Edinburgh, Scotland, Jul. 2007.

[12] F. Gruau, Y. Lhuillier, P. Reitz, and O. Temam, "BLOB Computing," in *2004 International Conference on Computing Frontiers, CF04*, Ischia, Italy, Apr.

[13] L. Chua and T. Roska,*Cellular Neural Networks and Visual Computing: Foundations and Applications*, New York, NY, USA: Cambridge University Press, 2002.

[14] M. Hoseini, M. Pavicic, and C. You, "A Cellular Automata ASIC for Conformal Computing," in *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 305-306, Las Vegas, Nevada, USA, 14-17 Jul. 2008.

[15] D. A. Huffman, "The synthesis of sequential switching circuits, in *Sequential Machines: Selected Papers*, E. F. Moore, Ed. Reading, MA: Addison-Wesey, 1964.

[16] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," *Proc. Int. Symp. Theory of Switching*, pp. 204-243, 1959

[17] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley, 1969.

[18] C. A. R. Hoare, "Communicating sequential processes", *Commun. ACM*, vol. 21, pp. 666-677, 1978.

[19] Martin, Alain J. and Nystrom, Mika and Martin, (2006) Asynchronous techniques for system-on-chip design. *Proceedings of the IEEE, 94 (6)*. pp. 1089-1120. ISSN 0018-9219

[20] S. Wolfram, *A New Kind of Science*, Wolfram Media, January 2002.

[21] P. Sarkar, "A Brief History of Cellular Automata," vol. 32, pp. 80-107, Mar. 2000.

[22] N. Ganguly, B. Sikdar, A. Deutech, G. Canright, and P. Chaudhuri, "A Survey on Cellular Automata," Feb. 2006. Available online at http://www.cs.unibo.it/bison/publications.

[23] S. Adachi, F. Peper, and J. Lee, "Computation by Asynchronously Updating Cellular Automata," in *Journal of Statistical Physics*, vol. 114, no. 1-2, pp. 261-289, Jan. 2004.

[24] F. Peper, J. Lee, S. Adachi, and S. Mashiko, "Laying Out Circuits on Asynchronous Cellular Arrays: A Step Towards Feasible Nanocomputers," in *Nanotechnology* 14, pp. 469-485, Mar. 2003.

[25] R. Minnick, "A Survey of Microcellular Research," in *Journal of the ACM*, vol. 14, no. 2, pp. 203-241, April 1967.

[26] C. Wong, A. Martin, and P. Thomas, "An Architecture for Asynchronous FPGAs," in *Proc. 2003 IEEE Intl Conf. on Field-Programmable Technology*, pp. 170-177, Dec. 2003.

[27] J. Teifel and R. Manohar, "An Asynchronous Dataflow FPGA Architecture," in *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1376-1392, 2004.

[28] A. Mahram, M. Najibi, and H. Pedram, "An Asynchronous FPGA Logic Cell Implementation," in *Proc. of the 17th ACM Great Lakes Symposium on VLSI*, pp. 176-179, 2007.

[29] J. Teifel and R. Manohar, "Highly Pipelined Asynchronous FPGAs," in *Proc. of the 2004 ACM/SIGDA 12th International Symposium on Field-Programmable Gate Arrays*, pp. 133-142, Monterey, CA, Feb. 2004.

APPENDIX A. SOURCE CODE FOR LOGIC UNIT DESIGN

```
library IEEE; use IEEE.STD_LOGIC_1164.all;


entity LU is
  port (A0_IN, A1_IN, B0_IN, B1_IN, D_GATE,
           E_GATE, F_GATE, G_GATE, EN: in STD_LOGIC;
           Z0, Z1 : out STD_LOGIC);
 end entity;


architecture struct of LU is
    component delay is
    port(A0, A1, C0, C1, EN: in STD_LOGIC;
         F : out STD_LOGIC);
    end component;
    component AOI4V1 is
    port (A, B, C, D : in STD_LOGIC;
         F : out STD_LOGIC);
    end component;
    component OAI4V1 is
    port (A, B, C, D : in STD_LOGIC;
         F : out STD_LOGIC);
    end component;

    signal S1, S2, S3, S4, S5, S6 : STD_LOGIC;

 begin
   AOI1 : AOI4V1 port map (B1_IN, E_GATE, B0_IN, D_GATE, S1);
   AOI2 : AOI4V1 port map (B0_IN, F_GATE, B1_IN, G_GATE, S2);
   AOI3 : OAI4V1 port map (B0_IN, E_GATE, B1_IN, D_GATE, S3);
   AOI4 : OAI4V1 port map (B1_IN, F_GATE, B0_IN, G_GATE, S4);
```

```
    delay1 : delay port map (A0_IN, A1_IN, (not S1),
                                             (not S2), EN, Z1);
    delay2 : delay port map (A0_IN, A1_IN, S3, S4, EN, Z0);
  end;

 entity delay is
    port(A0, A1, C0, C1, EN: in STD_LOGIC;
         F : out STD_LOGIC);
    end delay;

architecture struct of delay is
    component NAND3V1 is
     port (A, B, C : in STD_LOGIC;
           F       : out STD_LOGIC);
    end component;
    component NAND2V1 is
     port (A, B : in STD_LOGIC;
           F    : out STD_LOGIC);
    end component;
    component INVX1 is
     port (A : in STD_LOGIC;
           Z : out STD_LOGIC);
    end component;

    signal S1, S2, S3, S4, S5, S6 : STD_LOGIC;

begin
   NAND1 : NAND3V1 port map (S6, A1, C1, S1);
   NAND2 : NAND3V1 port map (S6, A0, C0, S2);
   NAND3 : NAND2V1 port map (S1, S2, S3);
   INV1  : INVX1 port map (S3, S4);
   NAND4 : NAND2V1 port map (S4, EN, S5);
```

```
    INV2   : INVX1 port map (S5, S6);
    NAND5 : NAND3V1 port map (S4, S2, S1, F);
end;
```

APPENDIX B. SOURCE CODE FOR CONFIGURATION CIRCUITRY DESIGN

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity config is
  port (CLK_IN : IN STD_LOGIC;
        DATA_IN : IN STD_LOGIC;
        default , pulse_not : IN STD_LOGIC;
        D, E, F, G, DTA_LTCH_0, DTA_LTCH_1, EVT_LTCH_0,
        EVT_LTCH_1, JOIN, TURN_T0, TURN_T1, TURN_U0,
        TURN_U1 : OUT STD_LOGIC;
        CLK_OUT : OUT STD_LOGIC);
end config;

architecture behavior of config is
  component store_unit is
  port (data_in , clk_in : IN STD_LOGIC;
        clk_out : OUT STD_LOGIC;
        data_out: OUT STD_LOGIC);
  end component;
  component oneshot is
    port (SI : in STD_LOGIC;
          PO : out STD_LOGIC);
  end component;
  component INV_REG is
    port (A : IN STD_LOGIC;
          Z : OUT STD_LOGIC);
  end component;

  signal store_bit , clk : STD_LOGIC_VECTOR (10 downto 0);
  signal clk_ctl : STD_LOGIC;
```

```
begin
  pulse_gen : oneshot port map (clk_in , clk_ctl);
  LATCH0:   store_unit port map
             (data_in , clk (9), clk (10), store_bit (0));
  LATCH1:   store_unit port map
             (store_bit (0), clk (8), clk (9), store_bit (1));
  LATCH2:   store_unit port map
             (store_bit (1), clk (7), clk (8), store_bit (2));
  LATCH3:   store_unit port map
             (store_bit (2), clk (6), clk (7), store_bit (3));
  LATCH4:   store_unit port map
             (store_bit (3), clk (5), clk (6), store_bit (4));
  LATCH5:   store_unit port map
             (store_bit (4), clk (4), clk (5),store_bit (5));
  LATCH6:   store_unit port map
             (store_bit (5), clk (3), clk (4), store_bit (6));
  LATCH7:   store_unit port map
             (store_bit (6), clk (2), clk (3), store_bit (7));
  LATCH8:   store_unit port map
             (store_bit (7), clk (1), clk (2), store_bit (8));
  LATCH9: store_unit port map
             (store_bit (8), clk (0), clk (1), store_bit (9));
  LATCH10:  store_unit port map
             (store_bit (9), clk_ctl , clk (0), store_bit (10));

  process (default , store_bit (0), store_bit (1),
           store_bit (2), store_bit (3), store_bit (6),
           store_bit (7), store_bit (8), store_bit (9),
           store_bit (10))
    begin
    if default = '1' then
```

```
        D <= '0'; E <= '0'; F <= '1'; G <= '1'; JOIN <= '1';
        TURN_T0 <= '0'; TURN_T1 <= '0'; TURN_U0 <= '0';
        TURN_U1 <= '0';
      else
        D <= store_bit (0); E <= store_bit (1); F <= store_bit (2);
        G <= store_bit (3);JOIN <= store_bit (6);
        TURN_T0 <= store_bit (7); TURN_T1 <= store_bit (8);
        TURN_U0 <= store_bit (9); TURN_U1 <= store_bit (10);
      end if;
   end process;

 DATA_LATCH: process (default , pulse_not , store_bit (4)) begin
        if default = '1' then
          DTA_LTCH_1 <= not (pulse_not);
          DTA_LTCH_0 <= '0';
        elsif store_bit (4) = '0' then
          DTA_LTCH_0 <= not (pulse_not);
          DTA_LTCH_1 <= '0';
        else DTA_LTCH_0 <= '0';
          DTA_LTCH_1 <= not (pulse_not);
        end if;
   end process;

EVENT_LATCH: process (default , pulse_not , store_bit (5)) begin
        if default = '1' then
          EVT_LTCH_0 <= not (pulse_not);
          EVT_LTCH_1 <= '0';
        elsif store_bit (5) = '0' then
          EVT_LTCH_0 <= not (pulse_not);
          EVT_LTCH_1 <= '0';
        else EVT_LTCH_0 <= '0';
          EVT_LTCH_1 <= not (pulse_not);
        end if;
    end process;

  INV: INV_REG port map (clk (10), clk_out);

  end;


  entity store_unit is
    port (data_in, clk_in : IN STD_LOGIC;
          clk_out : OUT STD_LOGIC;
          data_out: OUT STD_LOGIC);
  end store_unit;


  architecture structure of store_unit is
    component dlatch is
      port (D, G : IN STD_LOGIC;
            Q : OUT STD_LOGIC);
    end component;
    component buffv1 is
      port (A : IN STD_LOGIC;
            F : OUT STD_LOGIC);
    end component;
    component buff_reg is
      port (A : IN STD_LOGIC;
            F : OUT STD_LOGIC);
    end component;


    signal sig_buff1 : STD_LOGIC;
    signal sig_clk_out : STD_LOGIC;


  begin
    latch : dlatch port map (data_in, sig_clk_out, data_out);
  buff1 : buffv1 port map (clk_in , sig_buff1 );
  buff2 : buff_reg port map (sig_buff1 , sig_clk_out );
  clk_out <= sig_clk_out ;
end;
```