The Lark Project: Design of a Highly Parallel Programmable Logic Array

Scott Apgar
Lloyd Greenwald
Daniel P. Lopresti

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-04
January 1991

The Lark Project: Design of a Highly Parallel Programmable Logic Array

S. Apgar *L. Greenwald* *D. Lopresti*

Department of Computer Science
Brown University
Box 1910
Providence, RI 02912

January 14, 1991

Abstract

In this paper we describe the design and implementation of Lark, a highly parallel programmable logic array. Our goal was to create a chip containing as many simple logic cells as possible, arrayed in a fixed, easy-to-program interconnection scheme. We incorporated as much flexibility and functionality as would fit within these bounds. The 2μ CMOS prototype holds 198 basic cells, each possessing two flip-flops and two independent ALU's capable of performing arbitrary boolean functions of up to three inputs. It is currently queued for fabrication by MOSIS. Our estimates indicate that this chip will have a peak performance of 3.7 billion gate evaluations per second.

1 Introduction

Advances in semiconductor fabrication technology over the past decade have fueled the tremendous growth in the variety of parallel architectures we see today. The availability of this hardware has encouraged computational scientists working in diverse fields to attempt problems larger than once considered possible. This, in turn, creates a demand for still bigger and faster machines. Fortunately, the steady shrinking of device sizes that encouraged the trend will continue for the foreseeable future.

A particularly promising new lineage is the *programmable logic array* (sometimes called *configurable array logic* to avoid confusion with the PLA layout style practiced in VLSI). Such machines are built through the large scale replication of logic cells that, by themselves, are capable of only the simplest operations. Typically, a basic cell consists of a single-bit ALU and a couple bits of storage, and can be implemented using tens or hundreds of transistors. As a result, many cells are needed to perform any worthwhile computation. On the other hand, programmable logic arrays can be tuned to a particular application without sacrificing cycle time, chip area, and ultimately dollars for unused hardware (e.g., providing floating point dividers given integer-only computations). These architectures combine the speed of a custom-built, special-purpose device with the flexibility of a general-purpose computer. Impressive cost/performance ratios have already been demonstrated for a number of important problems.

Two building-blocks that reflect this philosophy are the Configurable Array Logic (CAL) chip [Gray89] and the Xilinx 3090 field-programmable gate array (FPGA) [Xili89]. The former integrates 256 cells arranged in a 16 x 16 grid on a single 2μ CMOS chip. Each cell contains a two-to-one boolean function unit, a flip-flop, and four independent multiplexers that allow for the simultaneous routing of data. Connections between non-contiguous cells must be made through intermediate cells. As of March 1989, several applications had been developed including DES encryption, which was estimated to run several orders of magnitude faster on CAL than on existing workstations.

The Xilinx 3090 differs somewhat in that it is a commercial part not originally intended for building parallel machines (Xilinx markets its FPGA's as a replacement for "jellybean" TTL logic). The basic 3090 cell (called a *configurable logic block*, or CLB) is significantly more powerful than its CAL counterpart, allowing an arbitrary boolean function of up to five inputs, or two functions of up to four inputs each. Each CLB also contains two

flip-flops. Moreover, routing resources are treated separately from computational resources; the CLB's are embedded in a "sea" of switchboxes and wires. There are 320 logic cells per 3090 chip (20 rows by 16 columns).

Researchers at the Supercomputing Research Center used the Xilinx 3090 as the heart of the SPLASH programmable logic array [Gokh90], [SRC90]. The hardware consists of two VME boards that plug into Sun workstation slots. One is an "off-the-shelf" 8-megabyte staging memory. The other is the SPLASH engine, containing a logic array consisting of 32 Xilinx 3090 FPGA chips connected linearly. Interspersed between each pair of adjacent 3090's is a 128K x 8 50ns static RAM chip. This memory is accessible to the FPGA on either side and can be used as a scratchpad, or for preloading initial data (e.g., a look-up table). Successive array stages are joined by 68-bit data paths. The first stage (X0) is connected to an input FIFO through 32-bit data and 4-bit control paths. The last (X31) is connected to an output FIFO similarly. Stages 0 and 31 are joined in wrap-around fashion by a 35-bit data path. All connections (except those to the FIFO's) are bidirectional, so data can flow through the array in either direction. While the parts cost for SPLASH is only about $13,000, its performance surpasses that of current generation supercomputers for certain systolic pattern matching algorithms.

Still, there are shortcomings with the present SPLASH system. It had been hoped that developers coming from a software background could learn to program SPLASH, but this has not been the case. Myriad hardware-level details must be mastered, including issues in routability, clocking, and the crossing of chip boundaries [Lopr91]. Many of these stem from the Xilinx 3090 not being designed with this specific purpose in mind. To address these concerns, a highly simplified spreadsheet model called *POOL* was proposed [Lipt89]. In POOL, each CLB is limited to reading inputs from its neighbors to the north and west, and writing outputs to its neighbors to the south and east. As a result, all connections between CLB's can be pre-routed using fast local lines. POOL has its own disadvantages, however, including limiting the data flow to be unidirectional and not making full use of the underlying SPLASH/Xilinx hardware.

This paper presents another approach to the problem, a new programmable logic array called *Lark*. Lark was originally intended to be a direct implementation of the POOL model, but has evolved considerably to a point that lies between the CAL chip and the Xilinx 3090 in terms of complexity of the basic cell. Its salient features include:

1. Logic cell - two independent arbitrary boolean functions of three inputs, two flip-flops.

2. Interconnectivity - regular fixed alternating grid with four basic orientations, programmable asynchronous pass-through.

3. Chip-level architecture - row-wise programmable reset, bit-serial reconfiguration, 27 inputs and 27 outputs driven directly to pins.

A prototype Lark chip was designed, layed out, and simulated using the Berkeley CAD tools over a six week period as a final project for an upper-level VLSI course at Brown University. The 2μ CMOS, 4.6mm x 6.8mm chip fits 198 logic cells in a 9 x 22 grid and is currently queued for fabrication by the MOS Implementation Service (MOSIS). Crystal timings indicate that its peak performance will be approximately 3.7 billion gate evaluations per second. Assuming a more aggressive process and larger die size, we would be able to place over 1,500 of these same cells on a single chip.

The remainder of this paper is organized as follows: Section 2 describes the high-level Lark architecture, including the various design decisions we made. Section 3 provides more specific details about the prototype implementation. Some detail on the programming of the configurable array is presented in Section 4 and some elementary Lark computation structures and a simple applications are discussed in Section 5. Finally, we present our conclusions and a statement of ongoing research in Section 6.

2 The Lark Architecture

In this section we describe the high-level architecture of Lark. We first provide motivation for the major design decisions and then present the chosen design. In Section 3 we will describe the lowest-level design in more detail.

The Lark chip started out as a hardware implementation of a programming model designed at Princeton University[Lipt89], known as POOL. The original Lark architecture was described as follows: Each logic cell (or processing element (PE), used interchangeably in this paper) in the grid had two inputs, one from the north, one from the west, and outputs going to the south and the east neighbors (see figure 1a) . Each PE was designed to perform 2 arbitrary boolean functions, X and Y, of 4 inputs. Each function had a storage bit associated with its output. The four inputs to each function unit corresponded to the Y input from the north neighbor and the X input

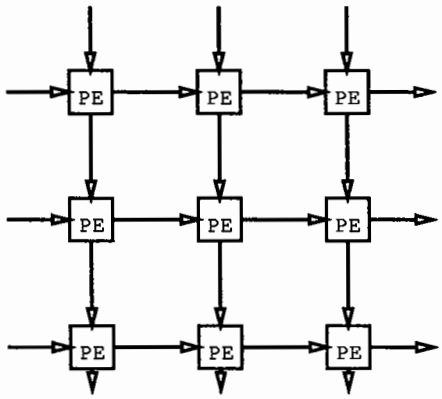
from the west neighbor and the two state bits from the results of the previous cycle's computation within that cell. This modeled the POOL functionality exactly. Unfortunately, this architecture was unidirectional, somewhat difficult to program, and was prone to using cells strictly for routing purposes due to the rigidity of the data flow. Some changes were desired to maintain the functionality of the POOL model, and at the same time aid the application designer's ability to route information between cells.

2.1 Interconnection Design

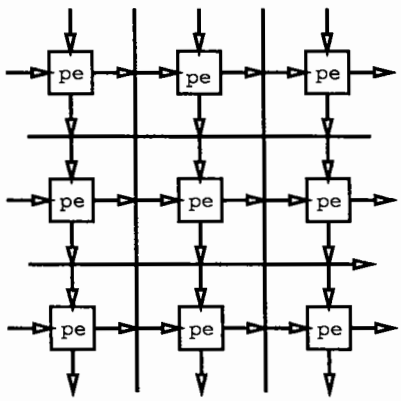
A number of alternatives were discussed before the final architecture was agreed upon. The most extensive discussion concerned changing the inherent interconnection scheme between PE's. Some elegant routing solutions were proposed, including routing switches in between the rows and columns of Lark PE's, crossbars between PE's, and utilizing some number of global bus wires running through the array. The following is a brief discussion of the reasons some solutions were rejected, for the purpose of background. This is intended to motivate the ultimate design decisions for the reader. Refer to Figure 1 for a visual description of each of the routing solutions proposed. After the brief description of the routing alternatives considered, a complete description of the chosen design, including other important decisions will be presented in subsequent sections.

In order to analyze whether or not to include a crossbar switch between each row/column intersection, a 4x4 crossbar switch was designed (see Figure 1b). In evaluating this design, it was determined that it would be almost as large as a PE, thus cutting the number of PE's on a chip in half. It would have increased the efficiency of the existing PE's, but not by a large enough margin. Halving the number of available PE's would detract from our ability to implement the type of algorithms for which Lark was intended (See Section 5).

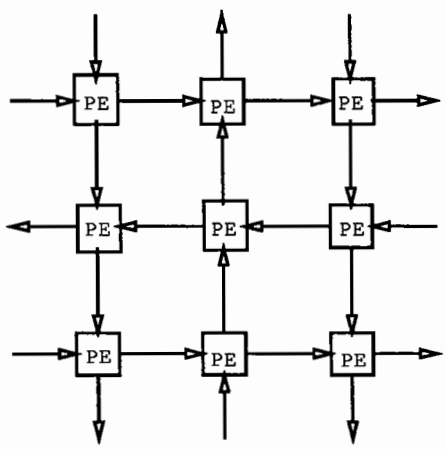
We also studied the possibility of using a 2x2 crossbar switch in the same orientation. This also required significant chip area, thus cutting down on the number of PE's that would fit on a chip. It was considerably smaller than the 4x4 switch, but due to the placement of the switch, (the switches would have had to have been situated on the top and the side of each PE), much of the area of a PE/switch combination would have been wasted space. Since they were somewhat smaller than the PE itself, there was considerable area surrounding each PE which was wasted. Given sufficient time to optimize the layout, this solution may have proven to be viable, however, it did not



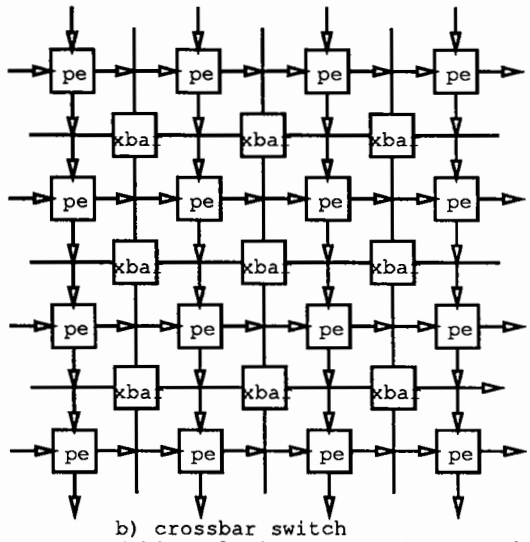
a) Original POOL model



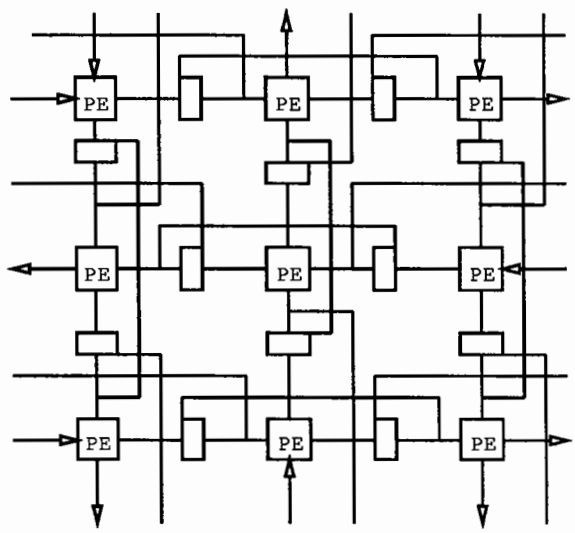
c) simple bus
(this solution requires a protocol and some extra hardware to implement the bus)



e) Chosen topology: Alternating Grid, no switches, cells are used to perform routing.



b) crossbar switch
(this solution required too much real estate too be viable. Even a small xbar switch did not fit into the layout nicely)



d) The small blocks are single bit switches. The grid alternates the direction of data flow. The switches are used to skip rows or columns. This allows the data to change direction.

Figure 1: Routing Topologies Considered

appear so at the time.

We also looked at routing some number of global wires through the PE's. The goal was to provide a simple way for data to leave one logic element and enter another without using the elements in between the two strictly as routing cells. This would have required some control logic to avoid collisions, and would have required some extra logic to implement the protocol this simple bus would have needed (see figure 1c). After some consideration, this solution was dismissed. In its place a much simpler strategy was employed.

One solution discussed (see Figure 1d), and partially adopted, was to run the inherent flow of data for each row/column of PE's in opposite directions (as opposed to the unidirectional nature of the original design), and also allowing the data to jump over a column or row using a bypass bus. This required only very simple programmable switches for each PE. Despite the small area required by these switches, the topology of the PE was such that the switch could not have been easily added. Also, the direction of data flow would be difficult to decipher in this topology.

It was decided to implement a simpler, more regular network, without utilizing switch boxes. The chosen design uses cells to perform routing when necessary, as opposed to a switch or a bus. This is less restrictive then it might seem. As will be discussed later, each PE has two semi-independent function units and data paths. If one is being used for routing, the other can be used for any number of computational uses.

The chosen interconnection design is shown in Figure 1e. In this scheme the data flows in rows and columns in alternating directions. The motivation behind this was to provide a method for data to flow in all directions in the array, without spending the resources to provide global busses. It should be evident that any cell in the array is reachable from any other cell, unlike in the original POOL model. This topology also allows an applications designer to bring together values calculated at cells widely separated on the chip in an efficient manner. Many implications of this design decision are touched upon in later sections.

2.2 Basic Logic Element (PE) Design

Figures 2 and 3 give an overview of the logical components of a single PE, as well as the general layout of the PE. This section gives a high-level description of the functionality of an isolated logic element (PE), including major design components.

As described in Section 1, Lark is configurable hardware. Therefore,

there must be a method to load a program onto the chip. Lark must have two modes: program and compute. This section is concerned with the high-level aspects of the compute mode. The program mode will be treated in detail in Section 4.

2.2.1 Function Computation

Each processing element is capable of performing any two arbitrary 3:1 boolean functions. This is done using 16 single bit programmable latches. Each function has 8 latches dedicated to it. Given any possible permutation of input values from the 3 inputs, a single programmed latch is selected to provide the output of the function (it may help to visualize a 3-input truth table).

The direction of each of the two outputs is fixed for a given cell and depends upon the orientation of the cell in the array. The outputs either go E and S, E and N, W and S, or W and N. These outputs correspond to inputs coming from the complementary directions. While this fixed interconnection scheme may seem rigid, the patterning of the array as an alternating connection grid provides much flexibility. Associated with each function output is a flip-flop (latch), storing the result.

In the original Lark design there were four inputs with the latter two coming from both previous state bits (output flip-flops). The major benefit realized by using a 4-input function, which would not be available using three inputs would be for an algorithm which was highly dependent on using pairs of functions and their states. Any other function of four inputs could be performed using some combination of 3 input cells. Given that we could fit almost twice as many PE's on the chip if we cut the number of inputs to 3, and that most algorithms that we looked at could be implemented with 3-input functions, we decided to implement it that way. We did not reduce the number of inputs any further, as the basic adder function requires 3 inputs to be performed in a single PE. This is a significant difference between Lark and the CAL chip.

We have described a basic cell as having two inputs from neighboring cells, yet each of the two functions are of three inputs. Where does the third input come from? Obviously, only one state bit can be used for each function of the PE in this design. A low-cost solution is to fix the third input of each function to either one or the other state bit. But, we desired more flexibility. In our design the reduction from 4 to 3 inputs is accompanied by the addition of two programmable multiplexers. Either state bit can be programmed to

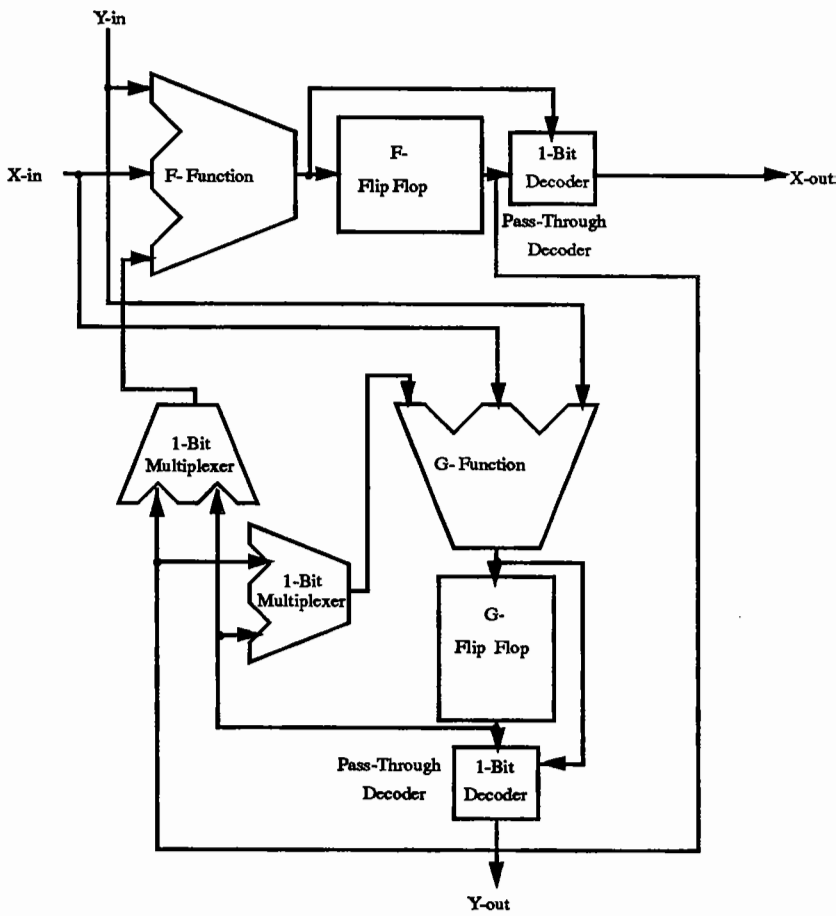


Figure 2: Block Level Description of the PE

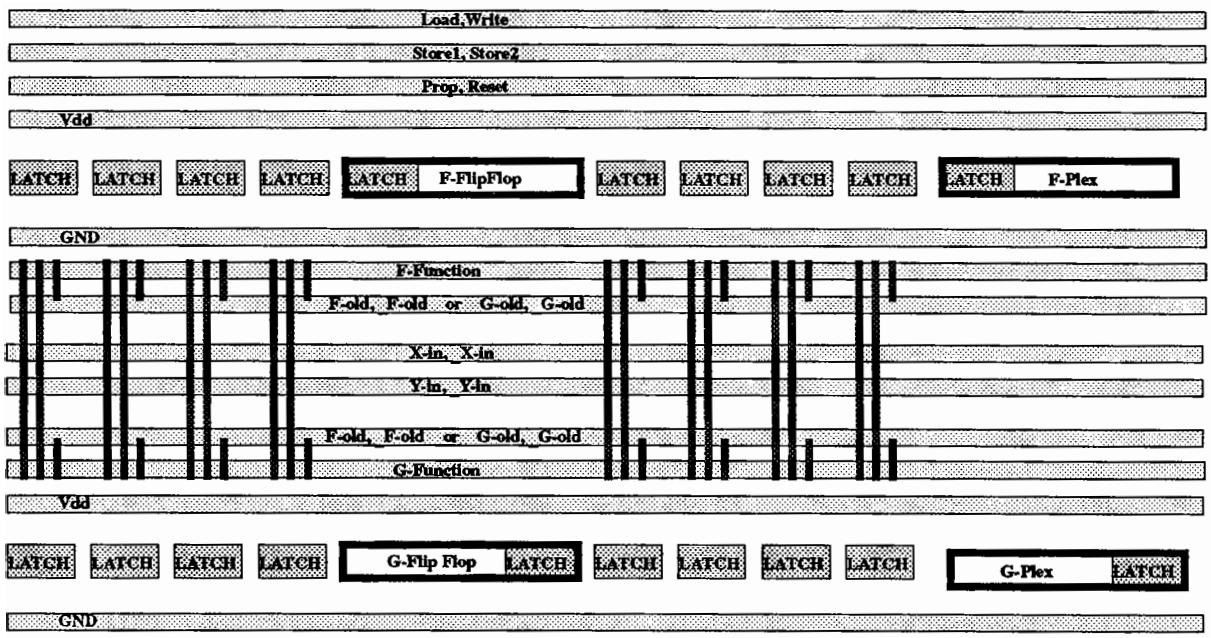


Figure 3: Floor Plan of the PE cell, and the Clock Cell

act as the third input to either of the two functions. This is done using two multiplexers, with the selection performed via a programmable bit for each function. This allows us to reduce the size of the the basic cell by close to half (32 programmed latches vs. 16 latches and two programmed multiplexers).

2.2.2 Asynchronous Programmable Pass-through

A final significant design feature of the basic cell is the *asynchronous programmable pass-through.* The PE can be programmed to act in zero-delay mode. In this mode the calculated output of the function is passed directly through to the output of the PE. In normal operation the output is taken off of the storage bit, thus taking a clock cycle to get through to the next cell. In zero delay mode, the output can be passed through up to 2 consecutive cells in one clock cycle. This mode of operation can also save cells from being used simply as routers. The other function unit of the cell can continue to function in normal operation mode. If it is desired to pass through more than 2 consecutive cells, it is necessary to slow down the clock speed. The limit of 2 cells is due to the time required to get through the PE into the neighboring cell. Our current estimates allow a 10 Mhz clock in the prototype Lark chip.

2.3 Chip Level Design

This section discusses the chip-level design of Lark. This includes global signals and special *row-wise reset* feature of the chip-level design.

The basic PE cell is replicated in four different orientations to provide the alternating network discussed in Section 2.1. The PE's have been designed to abut on the E/W sides, and have the clocks running on the N/S sides. The outputs of the PE's flow in the direction of the arrows seen in Figure 1e.

2.3.1 Global Signals

Clock routing is layed-out between PE's. The clocks are generated off the chip and brought in through input pins. They are immediately buffered and driven onto the metal lines running between each row of PE's. There is a 3-phase non-overlapping clock for the chip when in compute mode, and another 3 phase clock when in program mode. The compute mode clock uses the following signals: WRITE, PROP and STORE2 (see Figure 4). For the

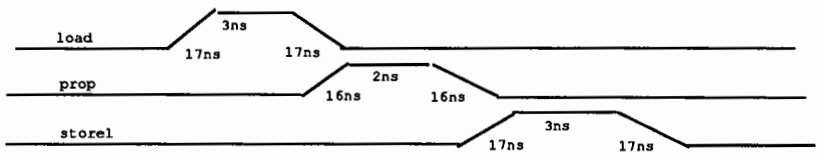


Figure 4: Programming Cycle

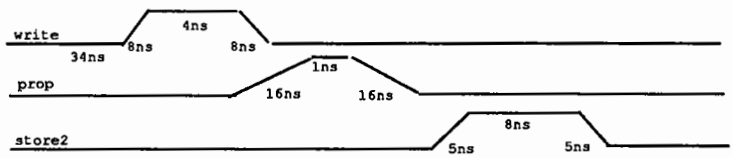


Figure 5: Computation Cycle

programming mode the clocks are as follows: LOAD, PROP and STORE1 (see Figure 5).

When LOAD is being clocked the data is taken off of the programming shift register (as described in Section 4) and stored in the latches. LOAD also controls the two input and output pins which perform double duty between the shift register data and the PE input data. We plan to drive the clocks using a 10Mhz signal.

2.3.2 Row-Wise Reset

This /em row-wise reset design provides the capability of selectively clearing any or all rows of function output flip-flops at any point in the computation. Each row of PE's has a program bit assigned to it. This serves as the reset mask for that row. If the bit is asserted, when the reset clock signal is asserted the entire row of PE function output flip-flops (state bits) are reinitialized to 0. The reset logic was incorporated into the latch using simple steering logic at at a very low cost to the total design. The program bits of the shift chain are not affected by this reset feature.

Experience with a previous design done by this group led us to conclude that the ability to reset rows of PE's individually would be extremely useful. It allows a designer to begin an algorithm in a known state, without having

to reprogram the entire chip. This also makes simulation and debugging easier. It takes roughly 4K cycles to program our prototype chip. This is a significant amount of time in a simulator. The RESET capability has the added benefit that it allows the user to run multiple sets of data through the chip without having to reprogram the same algorithm over and over again to clear out the stale data.

2.4 Prototype Design Issues

While Lark is interesting as a "paper" design, we are pleased to report that it is also to become a fabricated chip. The chip has been completely layed out and is currently queued for fabrication in 2μ CMOS by MOSIS. A chip-level Flea layout of the prototype Lark is shown in Figure 6. This section describes some additional design considerations that arose from producing a prototype chip.

2.4.1 Array Size

The chip we are sending out for fabrication uses the 64P46X68 standard pad frame available from MOSIS. We are using a 2μ double metal CMOS process. Using this technology we were able to fit a grid of 198 processing elements ($\sim$ 400 function units) on our prototype chip. In addition, we have calculated that using the largest chip size MOSIS has available, an 84 pin 7900x9200μ frame, in concert with a 1.2μ process would allow us to fit roughly 1584 PE's in the frame. In this technology we would be capable of driving the clock speed up to 16 Mhz. That would give us a total of over 3100 1-bit functions computed per cycle. We estimate that using this fabrication technology a Lark chip could be built to perform 50 billion gate evaluations per second.

2.4.2 Pin Assignment

Due to the limitation of the pad frame size we had to work with, we only have 64 pins available in our prototype chip (even with larger pad frames we would still be pin-bound). We have a total of 8 global signals which are driven directly from input pins, leaving us only 56 pins for the input/output streams. The global signals are: RESET, LOAD, PROP, STORE1, STORE2, WRITE, Vdd, and GND.

In the prototype Lark chip, we have a grid of 9x22 cells, giving us 62 border cells which needed input or output pins, 6 pins short of the ideal

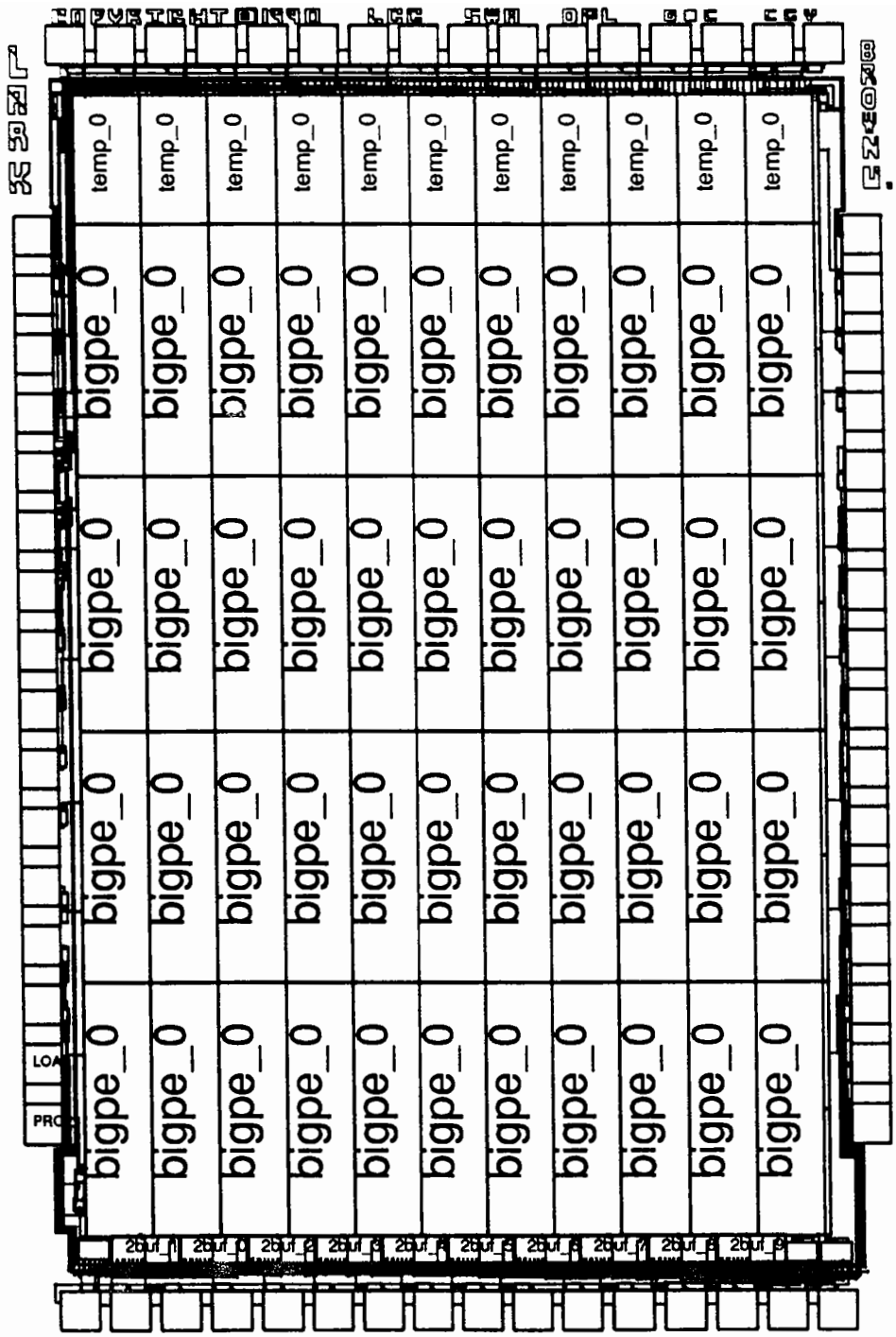


Figure 6: The Prototype Lark Chip

number. As is common in VLSI applications, no matter what technology is used, there are always going to be fewer I/O pins than ideally desired. Our solution to this problem is as follows: we chose to ignore the input and output pin requirements of the 4 PE's in the corners of the grid. In the place of an input pin we provided a program latch (programmable in concert with the rest of the programmable latches an the chip, see Section 4). Thus the application designer has the capability of setting that input for the duration of the algorithm being computed. The outputs are ignored. In the pad frame that our prototype chip is designed in, this technique leaves us 2 spare pins. Thus we were able to assign pins to the input and the output of the last column of PE's in the grid.

One possibility solution to this problem that could be explored is including a Linear Feedback Shift Register (LFSR) for each unconnected output bit to collect the data as it is shifted off. This technique would be used to encode the bit stream into a unique syndrome (barring synonyms, we would have to make the LFSR as large as possible to ensure that synonyms did not occur) using some CRC polynomial. The LFSR could then be included as part of the scan chain and be shifted off for decoding if the data was critical.

3 Lark Design Details

This section is intended to provide a more detailed description of the internal workings of the Lark chip's lowest level subcells. We performed switch-level and timing simulations at all levels of hierarchy. The results of these simulations, along with a description of the workings of each subcell are available upon request from the authors.

3.1 Program Latch

The most often used layout cell in our design is the *program latch*. This latch is replicated 22 times per PE, and is also used outside the PE in the reset logic, as well as being used to store the value of the inputs on the corners of the grid.

This latch was minimized, in terms of area, very early in the design stage. It is controlled by the following clock signals: LOAD, PROP, and STORE1.

The LOAD signal is used while in programming mode. It controls the gate of a pass transistor which allows the shift register to act as an input to the latch. Thus, all the latches are capable of being programmed. If we are not loading, then that pass transistor is off and the latch operates in

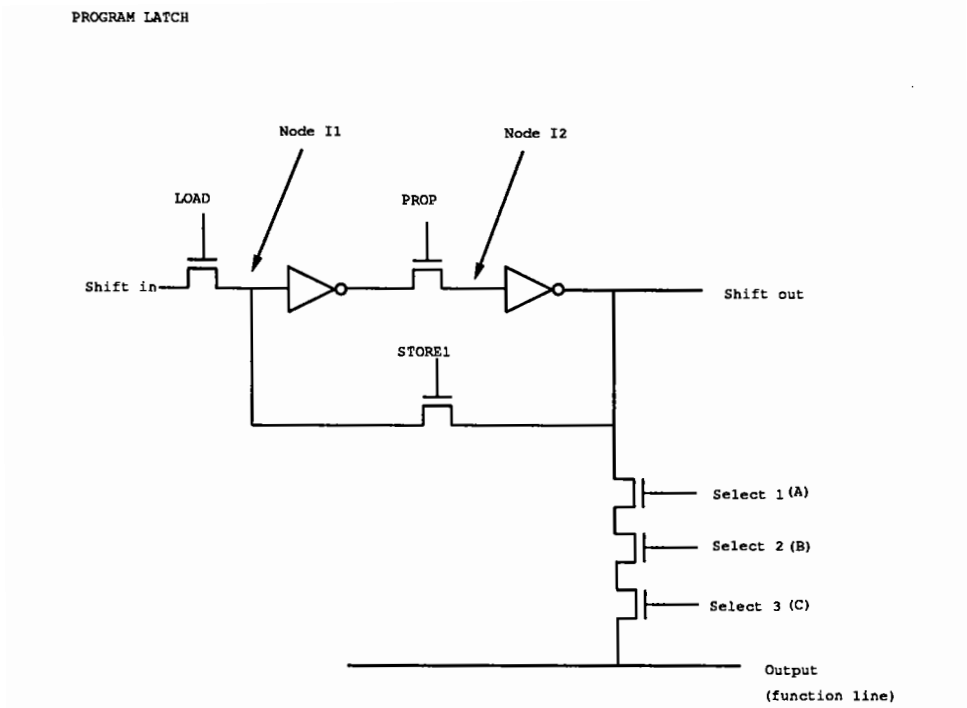


Figure 7: Program Latch Diagram

normal mode. This latch has a feedback path from the output back through to the input. This feedback path is controlled by the signal STORE1. The latch consists of 2 inverters with a pass transistor in between them. PROP controls the flow of data from the first inverter through to the second and out to the next latch or function line. In the function flip-flop, the output of this latch is gated onto a function line by three pass transistors. This is used to implement our function unit. Figure 7 shows the design of the latch. The layout of this latch was done in 2160 square lambda (see Figure 8). Obviously, any possible minimization to this latch would give us a significant savings in area on the chip. We spent an appropriate amount of time making it as small as possible.

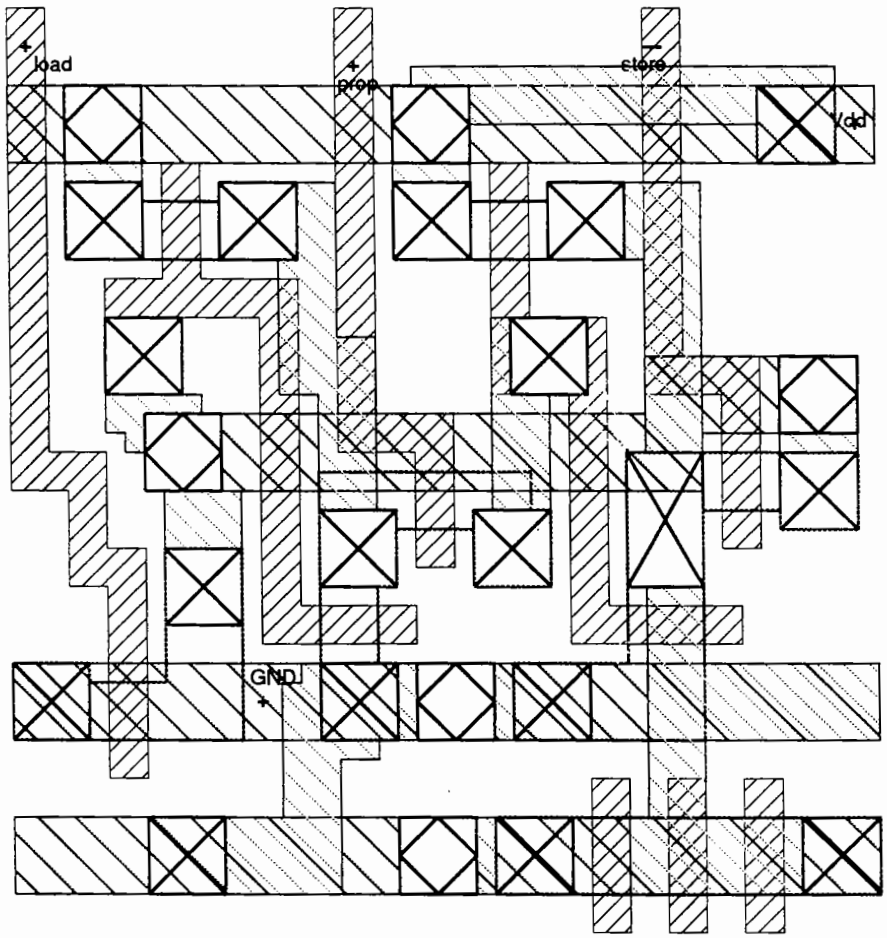


Figure 8: Program Latch Layout

3.2 Function Unit

Our 3:1 function unit is comprised of eight of the program latches. The outputs of these latches are gated onto our function line. If the three pass transistors connecting the output of the program latch to the function line are turned high, then the output of the latch is the only one driving output onto the function line. This is due to the truth table design of the function unit. The output of the function unit drives the input of the function latch, and also the input to the multiplexer which drives the pass-through.

3.3 F & G Flip-Flop with Reset and Input Selector

The flip-flop which is designed to store the output of the function line is based on our basic program latch also. We added the capability of resetting the value of this latch to be a zero. The input of this flip-flop is connected to the function line. This is controlled by the clock signal WRITE. WRITE is high only when we are in compute mode. It allows the value of the function line to be gated into the latch. If LOAD is high, we are in program mode, and the input of the latch is connected to the shift register chain. Due to the addition of the write input, we needed to add an additional clock signal to the feedback path, STORE2, without this signal the latch would not have operated correctly. There was the possibility of writing a new value and overwriting it with the old value via the feedback path before the new value had a chance to be propagated through to the output.

This latch also has a multiplexer contained inside it. The mux is used to select which function result it will gate onto the third input of the associated 3 input function unit. It can select either its own output, or the output of the second function in the PE. This selection is controlled via another programmable bit, i.e., another latch.

The design of this entire flip-flop can be seen in Figure 10. Note that there are actually 2 latches in this subcell: one to hold the program bit of the mux, and the second storing the function result.

3.4 Asynchronous Programmable Pass-Through

This design provides the capability of passing the output of the function unit directly through to the next cell without latching it. This will allow us to pass through 2 complete cells in one clock cycle. The user has the option of using a longer clock cycle to increase the number of cells that can be bypassed in one clock cycle. The intent of this feature was to reduce

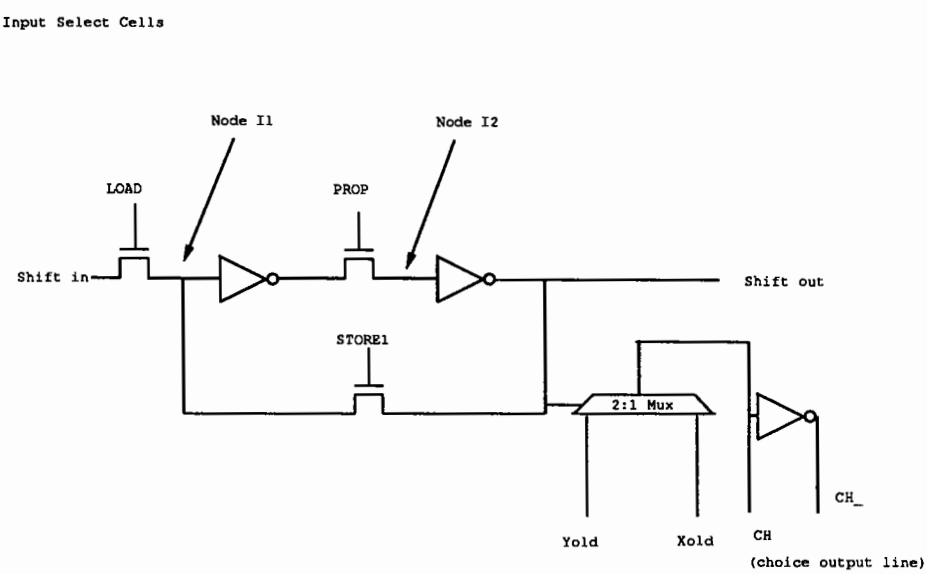


Figure 9: Input Selection Cells

the number of cycles required for an algorithm that moves data from cell to cell before actually getting the required elements in the same processing element. The result of the function unit is still stored in the latch, thus can be reused on subsequent cycles.

This pass through is controlled via a program latch also. If the bit is asserted, the pass through is utilized, otherwise the output is not available until after the Result latch stores it.

3.5 Processing Element

The PE is composed of the subcells mentioned in the previous subsections. It consists of 22 program latches, 2 output flip-flops (with reset and the input selector), the asynchronous pass-through logic and the clock line routing. The floorplan of a PE is shown in figure 3. Also shown in this figure are the orientation of the clock lines. The Flea plot of this layout is shown in figure 11. This basic PE cell was joined together to form a larger cell of 4 PE's. This larger cell was then used as the building block for the chip. This was done strictly to make the task of completing the chip layout more

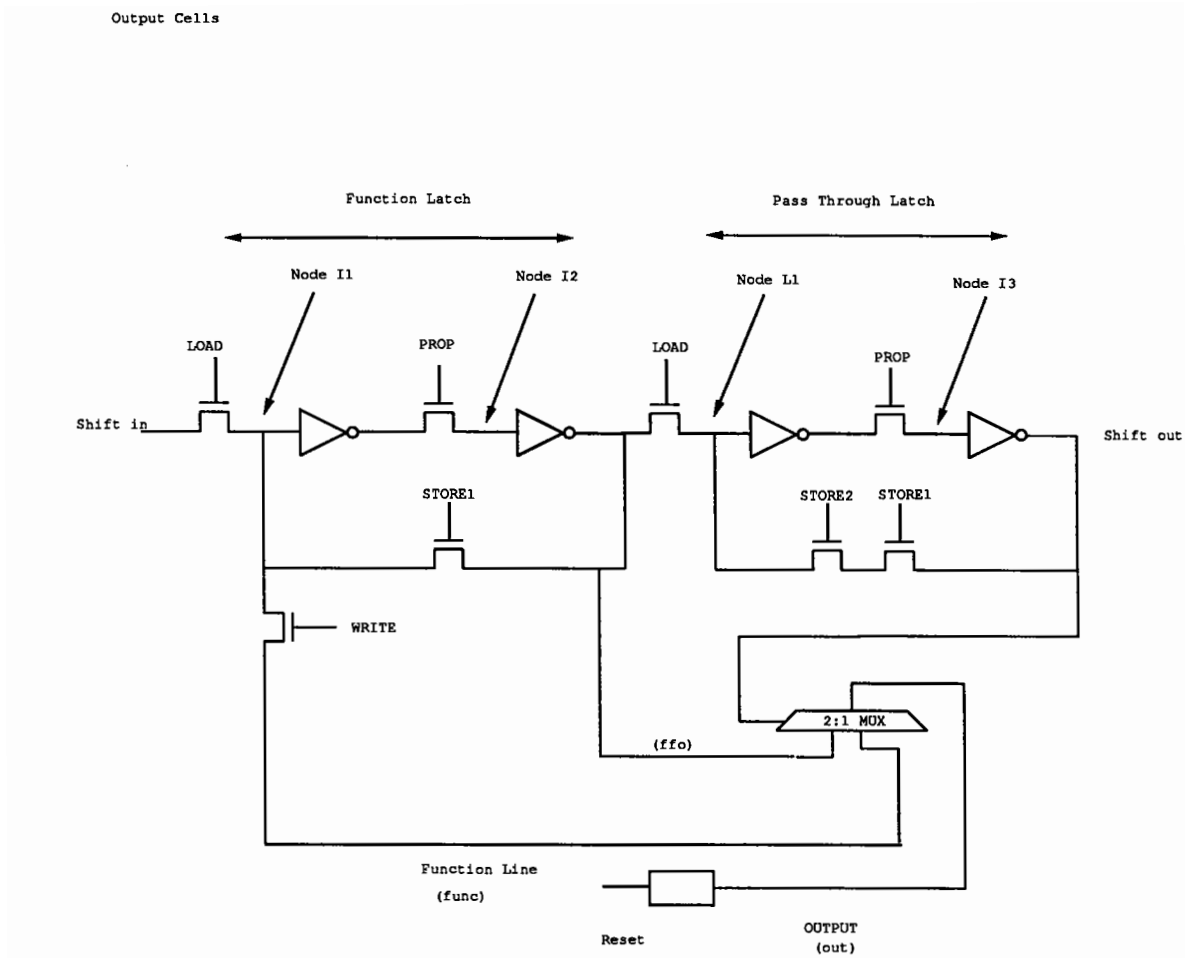


Figure 10: Output Cell

hierarchical (we could let the layout tool, Magic, perform more of the work!). In the chip-level layout of figure 6, the inner boxes are groups of four PE's each.

4 Hardware Reconfiguration Details

As mentioned previously, a programmable array is programmed by loading the program into a set of local memory positions on the chip. This is usually done at power-up by a host processor or dedicated PROM. Also, the program is generally bit-serial. By this, we mean that the program path is designed to be a large shift register encompassing all static memory elements on the chip. A stream of bits is serially loaded into the chip from a single location and pushed one position further per clock cycle. It therefore takes n clock cycles to program n bit positions.

The Lark configuration procedure is a slight variation of this general technique. The Lark chip employs a two-path bit-serial method for programming. Both paths are loaded in parallel and each path visits approximately half of all programmable memory.

The programmable memory consists of the following:

- Truth Table Program - 16 bits per cell: 8 bits per 3 input, 1 output function, 2 functions per cell;
- Flip-Flop Initialization - 2 bits per cell: F (Xout) flip-flop and G (Yout) flip-flop;
- Pass-Through Bit - 2 bits per cell: one per function, low indicates latch, high indicates pass-through;
- Flip-Flop 3rd Input Multiplex - 2 bits per cell: one per function, low indicates G fli-flop output, high indicates F;
- Row-Wise Mask Resets - 1 bit per row of cells: high indicates reset enabled, low disabled (RESET signal ignored for row);
- Virtual Pin Program Latch - 1 bit per chip input not driven by a pin (in the 9x22 Lark this is 3 total).

This gives a total of 4381 programmable bits.

Lark employs two paths to program the chip. Each path divides the Truth Table Programs, Flip-Flop Initializations, Pass-Through Bits and

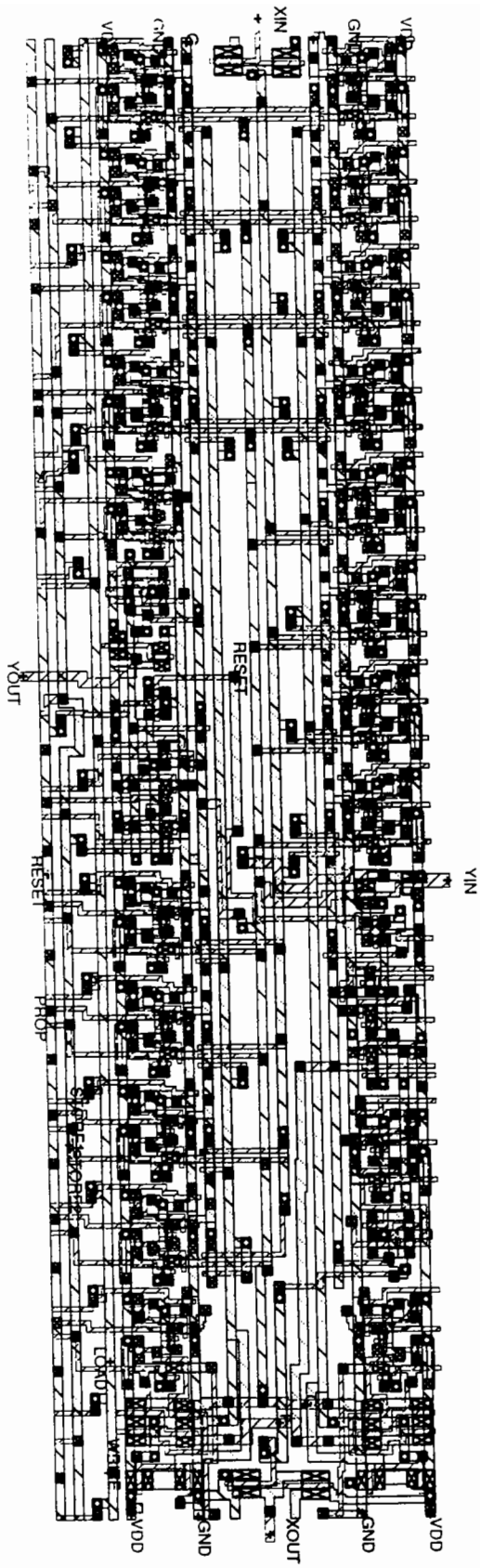


Figure 11: Processing Element Layout

Multiplex Bits evenly. However, path two must also program the Row-Wise Resets and the Virtual Pin Program Latches. Thus path one has a 2178 bit stream and path two has a 2203 bit stream. A program must therefore buffer path one with 25 initial dummy bits.

Additionally, for debugging purposes, Lark also employs a two path shift out at the end of the programming path. This allows the shift register to be used as a scan chain. The outputs of the register are multiplexed to two of the output pins normally used to output the PE data. When in program mode the pins get their output from the shift register, in normal operation, they get their output from the outputs of the cells. This allows the contents of the chip to be dumped out to the host processor in the same number of cycles as it takes to load a program. This will be a very useful feature for the testing and debugging of applications as was discovered with SPLASH.

The application user should never have to program the bit-streams directly. This can easily be abstracted and automated. In fact, there is currently an ongoing effort at Brown to design and implement a high-level graphical design tool for this purpose [Kraf91].

5 Designing Applications for Lark

The Lark chip provides both flexibility and challenge for the application designer. The alternating fixed connection grid alters the programming task by removing the need to program routing. However, this particular interconnection scheme does not naturally apply to all problems. For those that would be more suited to a different connection scheme, it takes a little experience to map them onto Lark. Fortunately, the programmable asynchronous pass-through feature allows a designer to create many interconnection patterns without wasting clock cycles. In this section we first describe simple basic units which can be easily implemented on Lark. We then describe some pattern matching applications. Finally, we mention other classes of problems which would be well-suited to program onto Lark. In our example applications we try to show the use of all of Lark's innovative features.

5.1 Building Blocks

A range of example PE configurations is shown in Figure 12, where each box is the abstract cell for a different basic unit. Inputs and paths which are not needed are not shown to simplify the cells. Each example shows a different use for a single two-function, two-flip-flop basic PE cell ranging from a single

directional shift register unit to a multi-input state feedback counter unit. The truth tables for the multi-input basic cells are also included in figure 12. These units are the building blocks for the multiple cell applications to follow. Some example basic units are:

1. Shift Register (S). Also double delay shift registers and arbitrarily large patterns of shift register cells.

2. Router (R). Using the asynchronous pass-through we can route a signal completely through one cell in a single-cycle (current timing estimates), or through many cells (with a modified clock). The router cells can be used to help signals turn corners by using the inherent bidirectional input and output of the basic cell.

3. Comparator (C); And (A): The comparator and And cells are functions of 2 inputs and one output and do not use feedback from the output latch.

4. Comparator/Routing Cell (C/R): Any two single output units can be combined into a single unit (cell). When a single input cell is used in one linear direction, a completely independent function can be implemented to flow in the other direction and even use its own latch feedback (or comparator latch output) if desired).

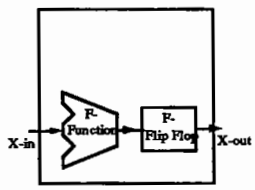
5. Combinational Logic: A three-input function can be computed during a pass-through. This is where we get the phrase "pass-through". The signal passes through the function unit, not just around it.

6. Counter: An arbitrarily sized counter can be implemented by using a linear array of basic function (counter) cells.

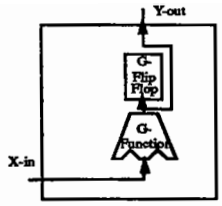
These basic units serve to demonstrate a small sample of the range of building blocks available when designing an application using Lark. Many other variations are possible.

5.2 Composite Units

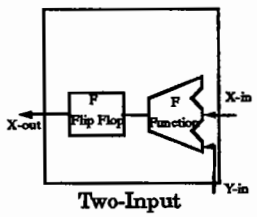
We can then take any variety of basic units and create applications. The straightforward applications shown here make use of the above basic units and are chosen to demonstrate many of Lark's innovative features.



Single-Input
Latched Function
(Shift Register-S)



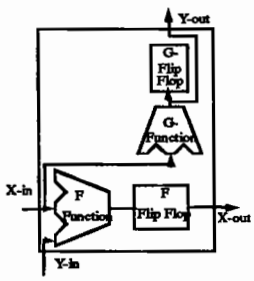
Single-Input
Unlatched Function
(Corner Routing Cell-R)



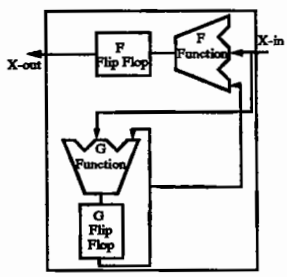
Two-Input
Latched Function
(Comparator-C)

Xin	Yin	Fnew (Xout)
0	0	1
0	1	0
1	0	0
1	1	1

Truth-Table Implementation
of Comparator



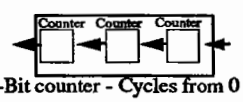
Two-Input Latched Function
And
Single-Input Unlatched Function
(Comparator/Routing Cell- C/R)



Two, Two-Input
Latched Functions
Using State Feedback
(Counter)

Xin	Gold	Fnew (Xout)	Gnew (Yout/ Feedback)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth-Table Implementation
of Counter Using State Feedback



8-Bit counter - Cycles from 0 to 7

Figure 12: Example PE Configurations

5.2.1 Character Adjacency Matching

In this application the basic cells described above are tiled onto Lark to perform a pair-wise 8-bit character match. Consider a string of characters. This application compares each character to its two nearest neighbors to find matches. This type of application may be suitable as a component in a data compression application, for example. This application is shown in figure 13.

Given input: A, B, C, D, E, F, G, H..... The comparisons performed will be: (A.B) (B.C) (C.D) (D.E) (E.F) ....

This application will perform the pair-wise matching of all characters in a string simultaneously in constant time (11 clock cycles) as long as the string length does not exceed the number of rows in the Lark chip. This is a minimum of 22 for a less-dense process and can far exceed these numbers on denser implementations of Lark. For strings of more characters than there are available rows, the time needed to execute all pair-wise matches is equal to a constant times the number of characters divided by the number of rows in the chip. After a 3 cycle start-up time, 22 comparisons are computed in parallel taking 8 clock-cycles for each group of 22 characters. If, after 8 cycles, the high-order counter cell outputs a logical 1 we know that all 8 bits matched. A 0 indicates fewer than 8 matches. Strings cannot be immediately pipelined in this application, so the second group of 22 comparisons (22 strings) is available after a one cycle pause to reset the counter cells to 0 plus a new three-cycle start-up time. This extra three-cycles is necessary because the row-wise reset not only resets the 8-bit counter, but also resets three bits in the shift register. Adding functionality to the chip to allow a per-cell reset mask would be prohibitively high compared to the small price to pay for an extra three-cycle delay. In applications which isolate an entire row for a single function (such as a 256-bit string comparison application which uses 8 cells for its counter) the row-reset capability is used to its full efficiency.

This example demonstrates several key features of Lark: 2 function/2 flip-flop basic unit cells; feedback using the multiplexed flip-flop states; multidirectional data flow (3 directions); pass-through routing; row-wise reset and full utilization of the array.

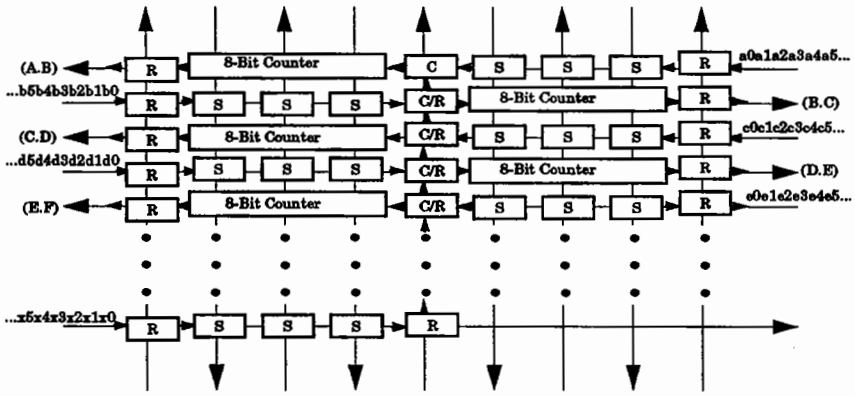


Figure 13: Character Adjacency Matching

5.2.2 Substring Matching

In substring matching an 8-bit key is input and then an arbitrary length sequence of bits is streamed past it. Every time the key is found within the stream a one is output otherwise a zero is output. This example takes advantage of the fact that we can get a two-cell pass-through plus third cell functioning in a single clock cycle with an adjusted clock. If the fastest possible clock speed is desired the application would need to be altered to replace double routers with a combination router/shift register, which would increase the number of cycles necessary. The application requires an 8 cycle start-up delay before processing can begin and an additional 5 cycle lag before the results of the match on the first 8-bits is obtained. Thereafter, a result can be obtained on every clock cycle. This example uses a modified comparator (for directional differences only), an "And" cell, and modified routing and shift registers (for additional directions). C, A, S have single delay latched output; R has zero delay pass-through output. Note, we could also add a counter to the tail of the matcher to count how many times the key has matched within the substring. The counter would be implemented exactly as in our previous example. See Figure 14.

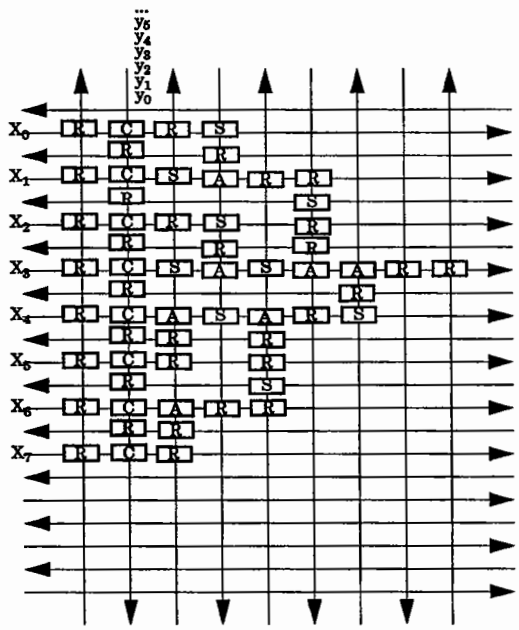


Figure 14: Substring Matching

5.2.3 Other examples

The previous examples demonstrate the use of many of the features of the Lark chip. As we mentioned above, we can extend our character adjacency matcher into a data (file) compression application. With similar variations we expect to be able to implement encryption applications as does CAL. We also mentioned the ease with which we can implement large combinational logic circuits composed of 3:1 gates.

Given the nature of the Lark chip, we expect to be able to implement many other pattern matching and naturally systolic algorithms, either with linear, mesh or feedback structure. These include, but are not limited to: General Pattern Matching; Neural Networks; Cellular Automata and Image Processing Algorithms.

6 Conclusions and Ongoing Research

In this paper we have described the design and implementation of a prototype Lark chip. This development cycle took 6 weeks from inception to submission for fabrication. The prototype chip is currently being fabricated and we expect the chip to perform 3.7 billion gate evaluations per second. Additionally, we estimate that using the most advanced fabrication technology available to us a Lark chip could be built to perform 50 billion gate evaluations per second. We feel that this design is an improvement over previous designs and achieves a high functionality/cell-size ratio.

In designing Lark many alternatives designs were explored and additional lessons learned about the chosen design. In future research, we expect to make use of this experience to improve upon Lark's performance and functionality. Additionally, work toward high-level programming tools for Lark are essential for its ultimate acceptance.

7 Acknowledgements

The authors would like to thank Dave Carey, Curtis Yarvin and Richard Hughey for their contributions to this work.

References

[Gokh90] M. Gokhale, et. al., "SPLASH: A Reconfigurable Linear Logic Array," *Proceedings of the 1990 International Conference on Parallel Processing,* August 1990, pp. 526-532.

[Gray89] J. P. Gray and T. A. Kean, "Configurable Hardware: A New Paradigm for Computation," *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference,* C. Seitz, ed., Cambridge, MA: The MIT Press, 1989, pp. 279- 295.

[Kraf91] J. Kraft and B. Knep, "PEARL: Programming Environment Array Configurable Logic," *Technical Report (in preparation),* Department of Computer Science, Brown University, 1991.

[Lipt89] R. J. Lipton, private communication.

[Lopr90] Daniel Lopresti, CS 165 Lecture Notes, Brown University, 1990.

[Lopr91] D. P. Lopresti, "Rapid Implementation of a Genetic Sequence Comparator Using Field- Programmable Logic Arrays," to be presented at the *1991 Conference on Advanced Research in VLSI,* Santa Cruz, CA, March 1991.

[SRC90] Supercomputing Research Center, SPLASH User's Manual, ver. 2, May 1990.

[Xili89] Xilinx Inc., *The Programmable Gate Array Data Book,* 1989.