A Self-Reconfigurable Gate Array Architecture [⋆]

Reetinder Sidhu[1], Sameer Wadhwa[1], Alessandro Mei[2], and Viktor K. Prasanna[1]

[1] Department of EE-Systems, University of Southern California,
Los Angeles CA 90089, USA
sidhu@halcyon.usc.edu, sameer@halcyon.usc.edu,
prasanna@ganges.usc.edu

[2] Department of Mathematics, University of Trento
38050 Trento (TN), Italy
mei@science.unitn.it

Abstract. This paper presents an innovative architecture for a reconfigurable device that allows single cycle context switching and single cycle random access to the unified on-chip configuration/data memory. These two features are necessary for efficient self-reconfiguration and are useful in general as well—no other device offers both features. The enhanced context switching feature permits arbitrary regions of the chip to selectively context switch—its not necessary for the whole device to do so. The memory access feature allows data transfer between logic cells and memory locations, and also directly between memory locations.
The key innovation enabling the above features is the use of a mesh of trees based interconnect with logic cells and memory blocks at the leaf nodes and identical switches at other nodes. The mesh of trees topology allows a logic cell to be associated with a pair of switches. The logic cell and the switches can be placed close to the memory block that stores their configuration bits. The physical proximity enables fast context switching while the mesh of trees topology permits fast memory access. To evaluate the architecture, a point design with 8×8 logic cells was synthesized using a standard cell library for a 0.25 μm process with 5 metal layers. Timing results obtained show that both context switching and memory access can be performed within a 10 ns clock cycle. Finally, this paper also illustrates how self-reconfiguration can be used to do basic routing operations of connecting two logic cells or inserting a logic cell by breaking an existing connection—algorithms (implemented as configured logic) to perform the above operations in a few clock cycles are presented.

1 Introduction

By exploiting the reconfigurability of devices such as FPGAs, significant performance improvements have been obtained over other modes of computation for several applications. Such a device provides configurable logic whose functionality is governed by bits written into its configuration memory, which is typically SRAM. Thus device functionality can be quickly reconfigured to suit application requirements by writing appropriate bits into the configuration memory—this is the key advantage of reconfigurable computing over other modes of computation.

In most cases however, reconfiguration of the device, whether at compile time or at runtime, is performed externally. Much greater performance gains and a high degree of flexibility can be obtained if the device can generate configuration bits at runtime and use them to modify its own configuration—the ability of a device to do so is what we call *self-reconfiguration*.

Self-reconfiguration is a powerful feature that allows configured logic to adapt itself as the computation proceeds, based on input data and intermediate results. It can be used for simple tasks such

[⋆] This work was supported by the National Science Foundation, Grant CCR-9900613. Alessandro Mei was supported by MURST, "Progetto Giovani Ricercatori 1998".

as reconfiguring the constant in a KCM (constant coefficient multiplier)—a self-reconfigurable device can do so on its own, which is faster than reconfiguring the device from an external source. Self-reconfiguration can also be used for non-trivial tasks such as constructing an FSM for string matching [6], or evolving genetic programs [5]. The above applications achieve efficient computation through a fine-grained interleaving of computation and configuration which would not be possible without a self-reconfigurable device.

A self-reconfigurable device needs to be able to store multiple contexts of configuration information and context switch between them. Also, it should allow configured logic to access the configuration memory. The configured logic can then perform self-reconfiguration by modifying configuration memory contents of a particular context and then switching to that context. Hence for efficient self-reconfiguration, it is crucial that the device should enable configured logic to perform

- fast context switching,
- fast random access of the configuration memory,

Even for applications that do not use self-reconfiguration, the above two features can be useful—the former reduces reconfiguration overhead while the latter allows configuration memory to be used for data storage as well.

So far, no device architecture has been designed specifically to support self-reconfiguration. Existing devices offer at most one of above two features—none offers both. Devices such as the Sanders CSRC [4] can switch contexts in a single clock cycle but provide only serial configuration memory access—it can take hundreds of clock cycles to access a particular location [3]. On the other hand, a device like the Berkeley HSRA [9] provides fast random access to the configuration memory (which can thus be used for data storage too) but requires hundreds of clock cycles to switch context—a complete reconfiguration takes about 5 μs [2].

In this paper we present an innovative architecture (Section 2) that supports both single cycle context switching (Section 2.8) as well as single cycle random memory access (Section 3.1), thus providing both features necessary for efficient self-reconfiguration. Further, the context switching feature permits arbitrary regions of the chip to selectively context switch—it is not necessary for the whole device to do so. The memory access feature permits data transfer with single source multiple destinations, and—with restrictions—multiple sources and destinations. In addition, the architecture has a simplicity and regularity that makes it efficient for configured logic to generate configuration bits for self-reconfiguration. This is demonstrated by describing how self-reconfiguration can be used to do basic routing operations such as connecting two logic cells or inserting a logic cell by breaking an existing connection (Section 4). Finally, implementation results are presented (Section 5) followed by conclusion and future directions (Section 6).

2 Architecture

2.1 Overview

The Self-Reconfigurable Gate Array (shown in Figure 1) consists of a rectangular[1] array of *PEs*. Each PE consists of a *logic cell* and a *memory block*. A logic cell contains a 16-bit LUT and a flip-flop. A memory block can store one or more configuration contexts as well as data for the configured logic. PEs are connected to each other through direct connections to 4 nearest neighbors as well as a *mesh of trees network*. As shown in Figure 1, it consists of a complete binary tree along each row and column of the array, with PEs at the leaves of the trees and identical *switches* at the non-leaf nodes.

The mesh of trees is a well studied network in parallel processing [1]. It has useful properties such as a small diameter and a large bisection bandwidth. The mesh of trees is also suitable for

[1] Henceforth we assume, for convenience, an array size of $N \times N$ where N is a power of 2 ($N = 2^n$) although it is not a fundamental limitation.

VLSI implementation since it permits area efficient layout of the PEs in a 2D mesh. The mesh layout also makes possible local, nearest neighbor interconnections. Also, since the area occupied by the mesh of trees network grows only logarithmically with PE array size, the architecture can be efficiently scaled to bigger devices. There is, however, a much more important reason for using this network. It is the use of the mesh of trees with memory blocks and logic cells at its leaves and switches at the non-leaf nodes, that makes it possible for context switch and memory access operations to be performed in a single clock cycle, as explained below.

We first describe the *ownership* relation that exists between PEs and switches. Consider any row of the PE array. It consists of N PEs and $N-1$ switches. We associate with each PE, the switch that succeeds it in the in-order traversal of the tree. The above associations are made for each row and column tree. As a result each PE is associated with two[2] switches—a row switch and a column switch. Switches associated with a PE are owned by that PE.

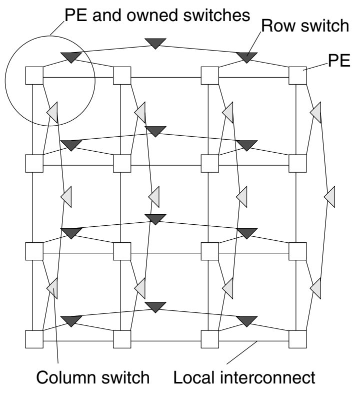


Fig. 1. SRGA architecture is based on a mesh of trees interconnect with PEs (containing a memory block and logic cell each) at the leaves and identical switches at other nodes.

A configuration context contains bits that configure all the logic cells and all the switches in the mesh of trees network. The configuration contexts are stored in memory blocks. Each memory block not only stores configuration bits for the logic cell in that PE, but also for the switches owned by that PE. In the VLSI layout, a PE and its switches can be placed close to each other[3]. This makes it practical to have a dedicated wire for each configuration bit to be transferred from a memory block of the logic cell and switches—the large number of wires required (about 80) is not a problem as they are very short in length. All memory blocks locally transfer data simultaneously over their dedicated wires (the context address is broadcast to all PEs). In this manner a context switch operation can be performed in a single clock cycle (please see Section 2.8 for a detailed explanation).

A memory access operation transfers data between rows or between columns of PEs. The source and destination are the logic cells and/or the memory blocks of the PEs. Each memory block is implemented as a random access memory that can read or write a single bit every clock cycle (the address used by the memory blocks is broadcast to all PEs). Also, as mentioned earlier, memory blocks are located only at the leaf nodes of the mesh of trees network. Thus an N-bit data word can be transferred between rows over column trees or between columns over row trees. In this manner a memory access operation can be performed in a single clock cycle (please see Section 3.1 for a detailed explanation).

2.2 Interconnection Network

The interconnection network of the proposed device consists of 2 parts—the *logic interconnection network* (LIN) and the *memory interconnection network* (MIN). The mesh of trees network mentioned above is composed of a part of the LIN and all of the MIN as described in the following sections. Section 2.3 describes the switch at each non-leaf node of the mesh of trees network.

Logic Interconnection Network The LIN serves the same purpose as the interconnection network in a typical FPGA—that of connecting together the logic cells as specified by the config-

[2] The exceptions are PEs in the right column which do not have an associated row switch and PEs in the bottom row which lack an associated column switch.

[3] The tree of switches is "flattened" with the $N-1$ switches placed in a single row (or column) adjacent to their owner PEs.

uration bits controlling the network switches. All LIN wires are in pairs. Each wire always carries signals in a single direction and wires forming a pair carry signals in opposite directions.

The network consists of 2 types of interconnections. One type are the local connections between each logic cell and its 4 nearest neighbors. These are direct connections—they do not pass through any switches. The other type of connections are in the form of a mesh of trees with PEs at leaf nodes and switches at others.

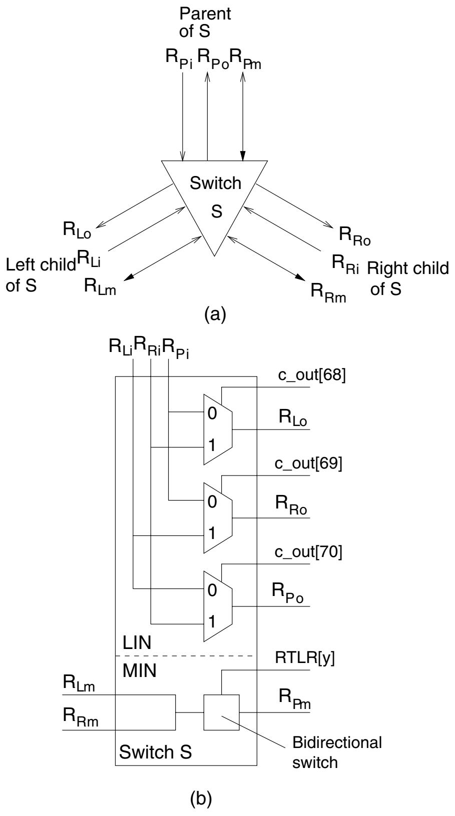


Fig. 2. (a) Switch input and output connections. (b) Internal details. Column switches have identical structure. Please see Figure 4 for connections with owner PE.

Memory Interconnection Network The MIN is used for performing data transfers during the memory access operations. Unlike the LIN, the wires are not in pairs—a single wire is used for each connection, and it may carry signals in either direction. The MIN also forms a mesh of trees network with PEs at the leaf nodes and switches at the remaining ones.

2.3 Switch

For each non-leaf node of the mesh of trees network there is an switch, the structure of which is shown in Figure 2. Each switch is connected to 2 child nodes and a parent node. Both child nodes are either switches or logic cells while the parent node is a switch.

For the LIN part of the switch, each connection is a pair of wires and so it has 3 inputs and 3 outputs. As shown, each output can be connected to either of 2 inputs via the muxes. The switch thus allows any input to be connected to any output without any restriction, except connecting an input to its output pair. Such a connection would only route a signal back where it came from, which is not useful. To configure the LIN part of the switch, 3 bits are required—1 for the control input of each of the 3 muxes.

For the MIN part of the switch, each connection is a single wire. The wires from the child nodes are permanently connected together and are connected to the parent wire through a bidirectional switch. By opening all switches at a particular level, a memory tree can be broken into multiple smaller trees.

2.4 Registers

The SRGA contains a number of registers that are accessed by the configured logic for performing context switch and memory access operations. The registers are shown in Figure 3 and described below.

The SRGA contains 3 global registers—their contents are broadcast to all PEs. They are described below.

Operation Register (OR) It is a 2-bit register that specifies what operation (if any) shall be initiated in the next clock cycle, as shown in Table 1(a).

Memory Operation Register (MOR) It is also a 2-bit register that specifies (if the OR indicates a memory operation in the next clock cycle) source and destination of the data transfer as shown in Table 1(b).

Context and Memory Address Register (CMAR) It specifies (depending on OR contents) the context to switch to or the memory address to be accessed in the next clock cycle. It consists of 2 fields. Bits $0 : \log_2 nc - 1$ form the *context field* of the CMAR—only these bits need to be specified when the CMAR is used for a context switch. The remaining $\lceil \log_2 cs \rceil$ bits form the *offset field*. This field (along with the context field) is utilized when the CMAR is used to specify a memory address. nc is the number of contexts and cs is the configuration word size—that is, the number of bits required to configure a logic cell and its 2 owned switches (each memory block thus stores $nc \times cs$ bits).

The SRGA contains 4 periphery registers—they are located along the boundary of the $N \times N$ PE array. Each register is N-bits long.

Table 1. (a) OR operations, (b) MOR operations.

OR[1]	OR[0]	Operation
0	0	No operation
0	1	Context switch
1	0	Row memory access
1	1	Column memory access

(a)

MOR[1]	MOR[0]	Source and destination
0	0	Memory to memory
0	1	Memory to logic (read)
1	0	Logic to memory (write)
1	1	Logic to logic

(b)

Source Row Register (SRR) It is located along the left side of the PE array. A set bit implies that the corresponding PE row will be the source for the next row memory access.

Destination Row Register (DRR) It is located along the right side of the PE array. A set bit implies that the corresponding PE row will be a destination for the next memory access operation.

Row Mask Register (RMR) It is located along the bottom of the PE array. A set bit indicates that, during a row memory access, no data transfer will take place for the corresponding column. The RMR and the DCR are physically the same register (this is not a problem as both a row and column memory access cannot occur in the same clock cycle).

Source Column Register (SCR) Same as SRR except for columns and located at the top of the array.

Destination Column Register(DCR) Same as DRR except for columns and located at the bottom of the array.

Column Mask Register (CMR) Same as RMR except for columns. The CMR and the DRR are physically the same register.

The SRGA contains 2 memory mapped registers. Each has N^2 bits—1 bit in each of the N^2 memory blocks. These registers can be accessed by the configured logic using memory access operations.

Context Switch Mask Register (CSMR) If the CSMR bit in a PE is set, the PE does not switch contexts even when a context switch operation occurs. Thus the CSMR enables the context switch operation to be controlled for each PE thus providing flexibility in context switching.

Data Restore Mask Register (DRMR) If the DRMR bit in a PE is set, it prevents the flip-flop contents of the logic cell in the PE from being restored when a context switch operation occurs. Thus, the DRMR enables data sharing between logic configured on different contexts.

2.5 PE

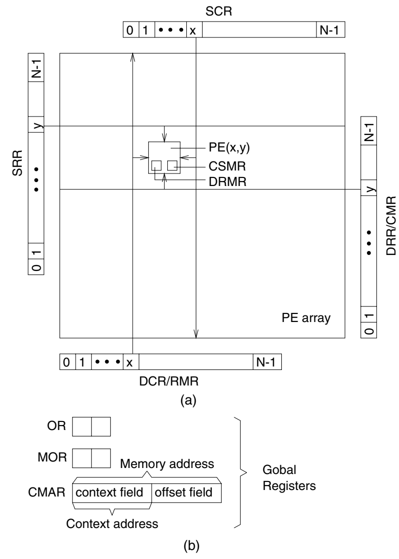


Fig. 3. (a) PE connections to periphery and memory mapped registers. (b) Global registers.

Figure 4 shows the structure of a PE (and also the connections to the 2 switches owned by it). The PE receives various signals from registers described in the preceding section. These are used by the control logic shown on the top of the figure to generate wr_mem, wr_log and switch_context which are used during context switch and memory access operations as described in Section 3.

The LIN nearest neighbor connections (N_i, E_i, S_i, W_i, N_o, E_o, S_o, W_o) and the row tree and column tree connections (R_i, C_i, R_o, C_o) are connected to the logic cell and so are the MIN connections (the bidirectional R_m and C_m are converted to the unidirectional R_{mi}, C_{mi}, R_{mo} and C_{mo}). The memory block supplies configuration bits to the logic block over the wires c_out[0:67] and to the 2 owned switches over c_out[68:73]. These (c_out[0:74]) are the large number of short wires for transferring configuration bits mentioned in Section 2.1.

For memory reads and writes, the memory block is connected to the row and column MIN trees through d_in and d_out. The context_state and DRMR signals are used to restore logic cell flip-flop contents when context switching (as described in Section 2.8).

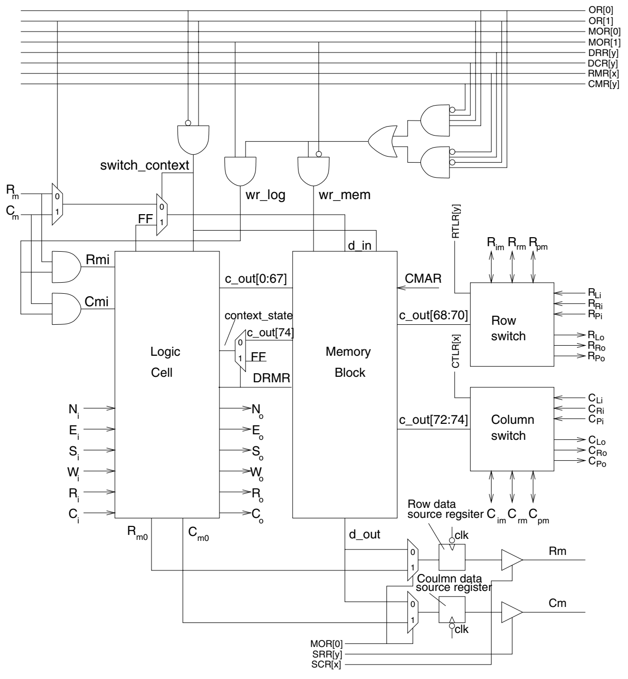


Fig. 4. PE structure.

The 2 muxes at the bottom are used to select either the logic cell or memory block output to drive the MIN when the PE is a source in a memory access operation. The tristate buffers are used since the MIN wires are bidirectional.

2.6 Memory Block

Figure 5 shows the structure of a memory block. The memory cell array is internally arranged as nc columns of cs(=75) bits each. Thus each column can store a configuration word. This arrangement enables all cs bits of a configuration word to be read out in a single clock cycle (and registered in c_out). Also, in case of a memory read or write operation, a single bit can be read from or written to the memory cell array. As should be clear from the figure, the CSMR and DRMR can also be accessed through memory operations. The CCR stores the address of the current context and is used during context switching as described in Section 2.8.

2.7 Logic Cell

Figure 6 shows the structure of a logic cell. It consists of a 16-bit LUT and a flip-flop. The LUT can implement 2 boolean functions of 3 inputs (with outputs $L1_o$ and $L2_o$) or a single boolean function of 4 inputs (output $L0_o$).

As can be seen, the mux $M0_i$ enables any of the inputs received by the logic cell to be used as input $L0_i$ of the LUT—the inputs $L1_i$, $L2_i$ and $L3_i$ are driven by muxes $M1_i$, $M2_i$ and $M3_i$ respectively which are identical to $M0_i$. Similarly, the output N_o of the logic cell can be connected to any of the inputs or any of the outputs of the LUT or flip-flop. Identical muxes $M1_o$–$M7_o$ drive the other outputs of the logic cell.

The complete flexibility in configuring connections allows the LUT and flip-flop to be used while other signals are routed though the logic cell. Also, since each mux has similar inputs and requires 4 control bits, the configuration word format (shown in Figure 7) is simple and regular, which considerably eases generation of configuration bits required for self-reconfiguration.

2.8 Context Switch Operation

Performing a context switch operation from current context *a* to another context *b* involves saving the state of context *a*, restoring the state of context *b* and replacing the configuration bits of context *a* with those of *b* in registers that determine the functionality of the configurable logic. A context switch operation completes in a

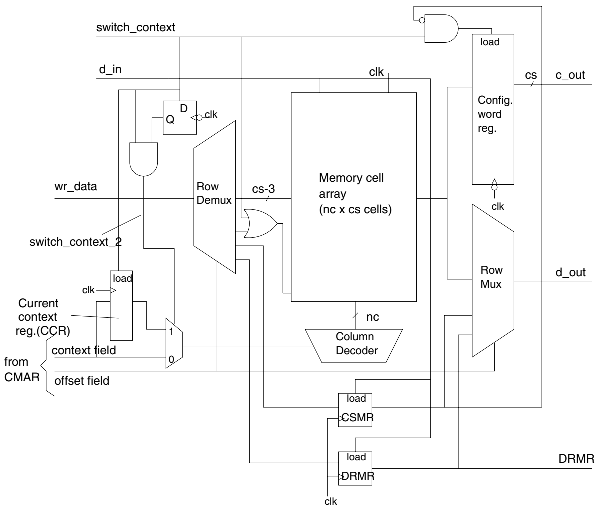


Fig. 5. Memory block structure.

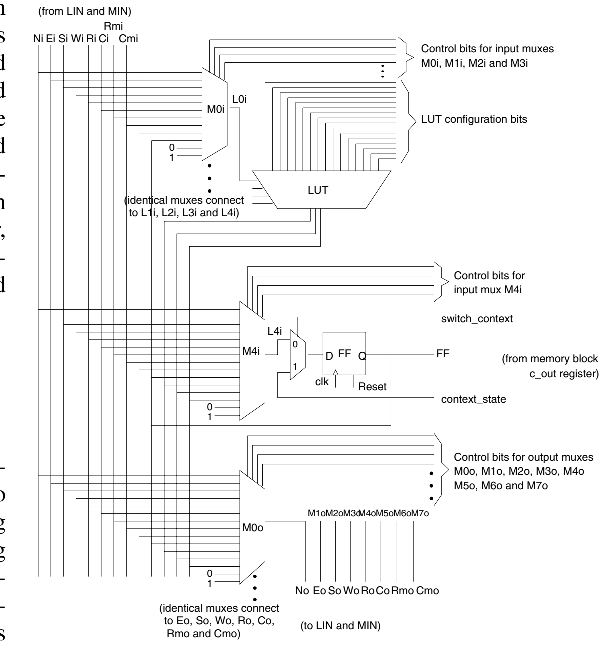

(from LIN and MIN)
Ni Ei Si Wi Ri Ci Rmi Cmi
Control bits for input muxes M0i, M1i, M2i and M3i
M0i
L0i
LUT configuration bits
0
1
LUT
(identical muxes connect to L1i, L2i, L3i and L4i)
Control bits for input mux M4i
switch_context
M4i
L4i
0
1
D FF Q
clk
Reset
FF
(from memory block c_out register)
context_state
0
1
Control bits for output muxes M0o, M1o, M2o, M3o, M4o M5o, M6o and M7o
M0o
M1oM2oM3oM4oM5oM6oM7o
0
1
No Eo So Wo Ro Co Rmo Cmo
(identical muxes connect to Eo, So, Wo, Ro, Co, Rmo and Cmo)
(to LIN and MIN)

Fig. 6. Logic cell structure.

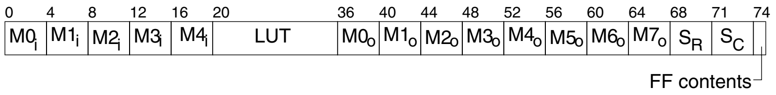

0 4 8 12 16 20 36 40 44 48 52 56 60 64 68 71 74
M0i M1i M2i M3i M4i LUT M0o M1o M2o M3o M4o M5o M6o M7o SR SC
FF contents

Fig. 7. Configuration word format.

single clock cycle. The context state consists of the $N \times N$ bits stored in the logic cell flip-flops (FF in Figure 6). The registers that determine the functionality of the configurable logic are the configuration word registers in each memory block (shown in Figure 5). The state of context *b* is restored only in those PEs which have their DRMR (Data Restore Mask Register) bit reset—in other PEs, the context *a* state is retained. In this manner data can be shared between contexts. Also, only those PEs switch to context *b* which have their CSMR (Context Switch Mask Register) bit reset—other PEs retain context *a*. In this manner, arbitrary regions of the $N \times N$ PE array may switch contexts while remaining regions retain the current context. In order to have static logic (logic that does not change on context switches) using above approach, it needs to be configured only on one context. This is more efficient than the static logic mode in [8] which required the same logic to be configured in all contexts. Also, the proposed approach permits multiple contexts to be active in different regions of the PE array at the same time.

3 Basic Operations

For a context switch to occur, some logic on the currently active context (context *a*) needs to write into the CMAR (explained in Section 2.4) the address of the context to switch to (context *b*) and into the OR the bits 01—writing these bits into the OR initiates a context switch operation in the next clock cycle. At the positive edge which marks the beginning of the next clock cycle, the CMAR and OR contents are registered and broadcast to all the memory blocks.

In each memory block (shown in Figure 5), in the first half of the clock cycle, the configuration word for context *b* is loaded into the configuration word register as follows. The switch_context signal is 1 while the switch_context_2 signal is 0. As a result the context field of the CMAR gets applied to the column decoder selecting the column corresponding to context *b* for loading into the configuration word register. Also its load enable input (EN) is 1 (assuming that the CSMR bit is 0). Therefore at the negative clock edge at the end of the first half of the clock cycle, the configuration word register gets loaded with the configuration word for context *b*.

During the second half of the clock cycle, context *a* state is saved and context *b* state is restored as follows. The signal switch_context_2 becomes 1 applying contents of the current context register (which contains the value *a*) to the column decoder. The signal row_select[cs-1] also becomes 1. These signals together select for writing the memory cell that stores FF contents for context *a*. Also the value of d_in, the data input to the memory array is the output of FF. Thus at the end of the second half of the clock cycle, the contents of FF get stored in bit 73 of the configuration word (shown in Figure 7) for context *a*. At the same clock edge, the switch_context signal ensures that FF gets loaded with the context_state signal. The value of context_state is either the FF contents saved in the configuration word of context *b* (if DRMR is 0) or the current contents of FF (if DRMR is 1). Also at the same clock edge, the current context register is loaded with the value in the context field of the CMAR (*b*) which would then be used to save the state of context *b* at the next context switch. In this manner, the context switch operation is performed in a single clock cycle.

3.1 Memory Access Operations

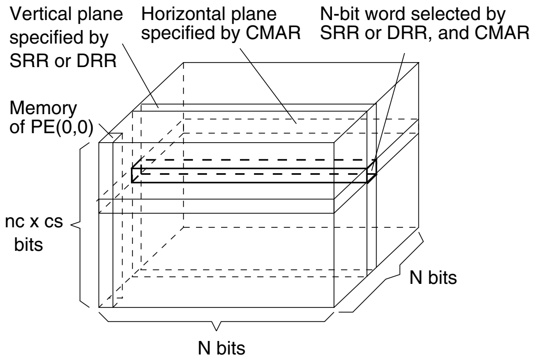

Vertical plane specified by SRR or DRR
Horizontal plane specified by CMAR
N-bit word selected by SRR or DRR, and CMAR
Memory of PE(0,0)
nc x cs bits
N bits
N bits

Fig. 8. Selection of an N-bit data word along a row.

A memory access operation transfers data between rows or between columns of PEs. The source and destination of data are the logic cells and/or the memory blocks in the PEs. Data transfers can occur from memory blocks to logic cells (memory read), from logic cells to memory blocks (memory write) or directly from memory block to memory block[4]. Each data transfer is of an N-bit word, with each PE in the source row (or column) contributing 1 bit. Transfer of any arbitrary subset of the N-bits can be done using the mask registers RMR and CMR. All memory access operations complete in a single clock cycle. In the first half of the clock cycle, data is read out of the memory blocks or the logic cells of the source PEs. In the second half, the data bits are broadcast over the column (or row) memory trees and written into the logic cells or memory blocks of the destination PEs.

The operation is a read, write or memory transfer operation depending on the contents of the MOR. For all operations, OR contains 10 indicating a row memory operation. Also a single 1 bit in the SRR indicates the source row while the 1 bits in the DRR specify the destination rows. The CMAR contains the memory address and the RMR is used to mask any bits, if required. Figure 8 shows the selection of an N-bit word of data along a row. The SRR or DRR selects a vertical plane while the CMAR specifies a horizontal bit-plane—the selected bits are at their intersection. Writing 10 in the OR register initiates the operation in the following clock cycle.

Memory Read Operation The MOR contains 01 which indicates that the source is memory blocks (address of the N-bit memory word specified by the SRR and the CMAR as shown in Figure 8) while the destination is logic cells (specified by the DRR).

In the first half of the clock cycle, in each memory block, the CMAR contents are applied to the row demux, the column decoder and the row mux, causing the required data bit to be output on d_out (please see Figure 5). At the negative clock edge, d_out is registered in the column data source flip-flop (shown in Figure 4).

Since the SRR bit for the source row is 1, the tristate buffers in the source row PEs are enabled, driving the flip-flop contents onto the corresponding memory column trees (described in Section 2.2). In this manner, in the second half of the clock cycle, the N-bit word is broadcast over the N column memory trees.

Finally, in each PE in the destination rows (rows for which DRR is 1), the wr_log signal (shown in Figure 4) is asserted causing the bit broadcast over its corresponding column memory tree to be available as the C_{mi} input to the logic cell. As can be seen from Figure 6, the C_{mi} input can be used by the logic in various ways—as an input to the LUT, the flip-flop FF, the muxes $M0_o$ to $M7_o$ connected to the logic cell outputs, or any combination thereof. The outputs of any of the above that use C_{mi} as input, stabilize by the end of the second half of the clock cycle, thus completing the memory read operation.

[4] Data transfer between logic cells is also supported by the MIN but is not discussed since the LIN is more suitable for connecting logic cells.

Note that for PEs in non-destination rows, $\mathsf{C_{mi}}$ is 0 because wr_log is not asserted. The same is true for those columns of destination row PEs for which the corresponding bit of the mask register RMR is 1.

Memory Write Operation The MOR contains 10 which indicates that the source is logic cells (specified by the SRR), while the destination is memory blocks (address specified by the DRR and CMAR). In the first half of the clock cycle in each PE, the $\mathsf{C_{mo}}$ output of the logic cell is applied to the input of the column source data flip-flop which registers it at the negative clock edge. As shown in Figure 6, any of several wires inside a logic cell may be connected to $\mathsf{C_{mo}}$ by appropriately configuring mux $\mathsf{M7_o}$.

At the negative clock edge, $\mathsf{C_{mo}}$ is registered in the column data source flip-flop (shown in Figure 4). Since the SRR bit for the source row is 1, the tristate buffers in the source row PEs are enabled, driving the flip-flop contents onto the corresponding memory column trees (described in Section 2.2). In this manner, in the second half of the clock cycle, the N-bit word is broadcast over the N column memory trees.

Finally, in each PE in the destination rows, the wr_mem signal (shown in Figure 4 is asserted causing the bit broadcast over its corresponding column tree to be available as the d_in input to the memory block. Also, the CMAR contents are applied to the row demux and the column decoder of the memory array (shown in Figure 5), selecting the memory cell into which d_in will be written. At the positive clock edge, d_in gets written into the memory array, thus completing the memory write operation.

Note that for PEs in non-destination rows, wr_mem is not asserted, preventing any memory write from taking place. The same is true for those columns of destination row PEs for which the corresponding bit of the mask register RMR is 1.

Memory to Memory Data Transfer Operation The MOR contains 00 which indicates that the source is memory blocks (address specified by the SRR and CMAR) and the destination is also memory blocks (address specified by the DRR and CMAR). Note that the CMAR is used for both source and destination addresses. Thus this operation is useful only if source and destination are on the same horizontal memory slice (shown in Figure 8).

In the first half of the clock cycle, data bits are read from the source row memory blocks into the corresponding column data source flip-flops as explained in Section 4. In the second half of the clock cycle, the data bits get written into the memory blocks of the destination PEs as described in Section 4. In this manner, the memory to memory data transfer operation is performed in a single clock cycle. As usual, the RMR can be used to prevent the transfer of any of the N bits.

4 Basic Routing Operations Using Self-Reconfiguration

Modification of configured logic using self-reconfiguration typically occurs as follows. Active context (a) decides that some logic on it needs to be modified. Context a then writes certain parameters (in a predetermined location—flip-flops or memory) that specify the reconfiguration required. It then switches to context b. Logic configured on context b reads the supplied parameters and uses them to generate the required configuration bits. Next, it writes the bits to appropriate locations in the memory (these locations store configuration bits for context a). Finally, context b switches back to context a, which now continues processing using logic that has been modified through self-reconfiguration.

In this section, we first look at the problem of connecting 2 logic cells in the same row. Since the SRGA architecture is symmetric w.r.t rows and columns, connecting 2 logic cells in the same column can be done in a similar manner. Next, we extend the operation to perform insertion of a logic cell between 2 logic cells previously connected (Section 4.2), and connecting two logic cells which are not in the same row or column (Section 4.3).

4.1 Connecting 2 logic cells in the same row

The problem is to connect the output of a logic cell to the input of another in the same row, using only row tree wires[5]. The LIN row tree to be used for the routing is a complete binary tree containing $N - 1$ switches (since each row has N PEs). Thus creating the required connection means appropriately configuring a subset of the $N - 1$ switches. As can be seen from Figure 9, connections need to be created up the tree starting from the source logic cell, and then down the tree till the destination logic cell is reached. The highest node through which the connection passes is the *least common ancestor* of the source and destination logic cells.

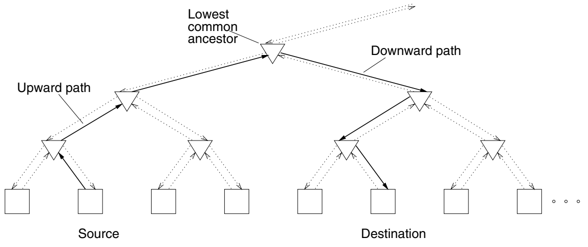
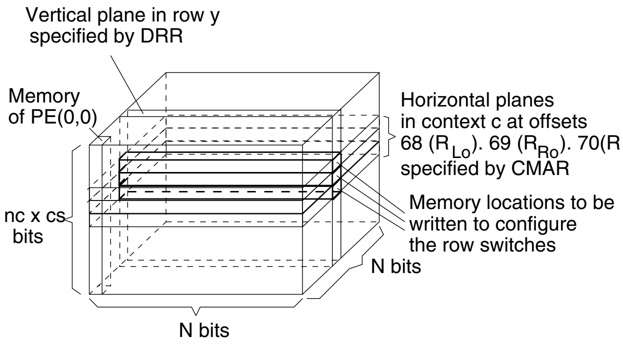


Fig. 9. Connection between 2 logic cells using row switches.

To create the connection, the context which requires the routing (a) needs to specify to the context that will perform the routing, the following information:

- The context address c ($0 \leq c < nc$) on which the routing is to be performed (typically it would be a itself).
- The row number y ($0 \leq y < N$) in which the logic cells to be connected are located.
- The column numbers x_s and x_d ($0 \leq x_s, x_d < N$) of the source and destination logic cells respectively.

The first 2 parameters are used to determine the memory locations in which the configuration bits will be written. Each switch is configured using 3 bits (see Figure 2) which are stored in the memory block of the PE that owns the switch. The required memory locations are thus $N - 1$ columns of 3 bits each—each column is associated with one of the $n - 1$ switches. Since all the switches (and hence their memory blocks) are in the same row, the memory locations that need to be written are 3 rows of $N - 1$ bits each. Figure 10 shows these locations and how they are accessed. The DRR uses the supplied row *y* to specify the vertical bit plane. The CMAR uses supplied context *c* as the contents of its context field while the offset field contains 68, 69 or 70 to access one of the 3 horizontal planes (corresponding to muxes driving the R_{Lo}, R_{Ro} and R_{Po} outputs respectively, as shown in Figure 2). The memory locations that control the switches of the muxes in row *y* in context *c* are at the intersections of the planes.



Vertical plane in row y specified by DRR
Memory of PE(0,0)
Horizontal planes in context c at offsets 68 (R_{Lo}). 69 (R_{Ro}). 70(R_{Po}) specified by CMAR
Memory locations to be written to configure the row switches
nc x cs bits
N bits
N bits

Fig. 10. Memory locations that configure switches of row y in context c. See Figure 7 for offset values.

[5] Connecting using only local, nearest neighbor wires is a much simpler problem. Also the routing delay would be linear compared to logarithmic in case of tree switches.

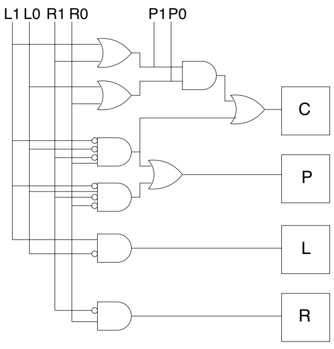


Fig. 11. Logic module structure.

The remaining parameters (x_s and x_d) are used to compute the configuration bits to be written to the locations determined using the first 2 parameters. We now look at the problem of computing these bits. Each of the $N-1$ switches is configured by 3 bits. However, looking at Figure 9, it can be seen that each switch to be configured receives a single input and supplies a single output. Thus only a single mux needs to be configured for each switch. Therefore, we compute 4 bits for each switch—bits L, P, R specify respectively whether the mux driving the left child (R_{Lo}), right child (R_{Ro}) or parent (R_{Po}) outputs is to be configured, and bit C specifies with what value.

The logic used to compute the bits required consists of $N-1$ identical logic modules, one module corresponding to each row switch. Each module generates the 4 bits (L, R, P and C) for its corresponding switch. Figure 11 shows the structure of the logic module. Each module requires 5 logic cells. Just as the switches to be configured are arranged as complete binary tree, so also we configure the $N-1$ logic modules as a complete binary tree—each module and the switch it represents are in the same position in their respective trees. The edges of the logic module tree consist of 2 unidirectional links from each child node to its parent. The lowest level modules are connected to flip-flops—a pair of flip-flops represents each logic cell.

Computation starts by setting the flip-flops corresponding to the source and destination logic cells to contain 01 and 10 respectively. Each logic module receives 2 bits from each child node. If it receives 01 from one child and 00 from the other, it is on the upward path (see Figure 9). Thus it needs to configure the parent mux and hence writes a 1 into it. Based on whether the 01 was received from the left or right child, 0 or 1 is written to the C flip-flop (see Figure 2). The logic module passes the received 01 to its parent. If a node receives a 10 input from one child and 00 from the other, then it is on the downward path. The left or right mux needs to be configured and a 1 is written to the L or R flip-flop depending upon which child node the 10 was received from. In both cases, input from parent needs to be selected and hence 0 is written to the C flip-flop. The module passes 10 to its parent. Finally, if a module receives a 01 from one child and 10 from the other, it represents the switch which is the least common ancestor of the source and destination logic cells. A 1 is written to the L or R flip-flop depending upon whether the 10 was received from the left or right child. Also, a 1 is written to the C flip-flop since the left input needs to be connected to the right mux or vice versa. The logic module passes neither 01 or 10 to its parent.

The module logic shown in Figure 11 performs the above functions. Since only combinational logic is required to compute configuration bits, the signals travel up the tree and bits in all logic module are computed in a *single clock cycle*[6]. The subsequent task of writing configuration bits into the memory becomes very simple if the computed bits for a switch are located in the same column in which they are to be written. Therefore we map logic modules to $(N-1) \times 5$ logic cells, each module located in the column in which are to be written the 3 bits that configure the switch it represents.

Routing of the modules thus placed to connect them in a complete binary tree can be efficiently performed. Figure 12 shows how an $N-1$ ($N=8$) node tree, with 2 logic cells per node can be configured with a single upward link from each child node to its parent. Since the required logic modules have 5 logic cells (and hence the tree requires 5 rows), they can be connected as a tree with 2 upward links from each child node to its parent.

[6] The clock period increases logarithmically with tree size.

Finally, the computed bits (L, R, P and C bits in all modules) are used to configure the switches. As discussed above, there are 3 $(N-1)$ bit memory locations, 1 each for the control bits of the muxes driving the $\mathsf{R_{Lo}}$, $\mathsf{R_{Ro}}$ and $\mathsf{R_{Po}}$ outputs. Each clock cycle, the one of L, R or P bits in all the $N-1$ logic modules are inverted and written to the RMR and the C bits of all $N-1$ logic modules are written to the location addressed by DRR and CMAR as discussed previously.

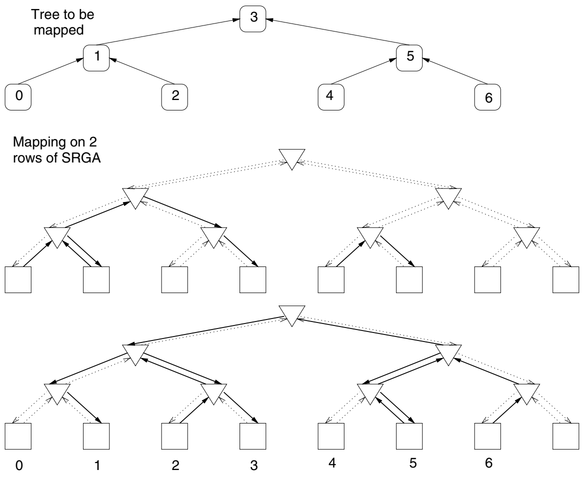


Fig. 12. Mapping of a complete $N-1$ node binary tree (with a unidirectional link from each child node to its parent) onto 2 rows of N logic cells. Each node consists of 2 logic cells.

In this manner, in only 3 clock cycles, the configuration bits to perform the routing operations are written. Thus, a connection between 2 logic cells in the same row (or column) can be created in a constant number of clock cycles—it does not depend upon the size of the row. The length of the clock cycle would depend upon the row length but it would grow only logarithmically with row length (since signals only need to propagate up the tree of logic modules). A related observation is that several such connections can be created in a row (or column) in parallel time if they occur in separate subtrees.

4.2 Inserting a logic cell between 2 connected logic cells in the same row

The output of logic cell l_s is connected to the input of logic cell l_d in the same row, using only row tree switches. The problem is to insert another logic cell l_i, also in the same row into the connection between l_s and l_d. Doing so requires breaking the above connection and creating 2 new connections—from the output of l_s to the input of l_i, and from the output of l_i to the input of l_d.

The input parameters are x_i (column of l_i) and all the parameters required for the row routing operation described in Section 4.1. The required operation can be very simply implemented using 2 invocations of the above mentioned row routing operation. It is invoked once with row and destination column parameters x_s and x_i, and with x_i and x_d the second time.

It should be noted that the above operations overwrite row switch configurations which had created a connection between x_s and x_d—thus the original connection need not be explicitly broken. In this manner, the logic cell insertion operation can be efficiently performed in a constant number of clock cycles.

4.3 Connecting 2 logic cells not in the same row or column

The problem is to connect the output of logic cell l_s to the input of logic cell l_d, when l_s and l_d are neither in the same row or same column. Let the location of l_s and l_d be (x_s, y_s) and (x_d, y_d). The required connection can be created by first connecting the output of l_s to the input of the logic cells at (x_d, y_s) (or (x_s, y_d)) and then connecting the output of the intermediate logic cell to the input of l_d. The former operation is the row routing operation described in Section 4.1 while the latter is its column counterpart which can be performed in a similar manner. In addition, the logic cell at (x_d, y_s) needs to be configured to connect the connections along the row and column trees. This can be easily done by configuring the 4 bits that control $\mathsf{M5_o}$ to connect the input $\mathsf{R_i}$ to its output $\mathsf{C_o}$. The logic cell can still be used for other purposes. In this manner, logic cells can be efficiently connected even if they are not in the same row or same column.

5 Implementation

Table 2. Area estimates for the 8×8 SRGA design.

Component	Area (μm^2)
Switch	311
Logic cell	7741
Memory block	81797
PE	90881
2×2 array	363018
4×4 array	1480095
8×8 array	5925859

The complete SRGA architecture presented in Section 2 was described in several thousand lines of Verilog code. The description was at the RTL level with several components explicitly instantiated. It was then synthesized using a library of standard cells for a 0.25 μm process with 5 metal layers. The synthesized design can store 8 configuration contexts and has an array size[7] of 8×8. The timing estimates are expected to increase slightly[8] after place and route. However, delays due to loading and fanout are accounted for in the results shown.

As can be seen from Table 2, most of the area in a PE is taken by the memory block. Its area of 81797 μm^2 for a memory size of only ($nc \times cs = 8 \times 77 =$) 616 bits is quite poor even for SRAM. The reason is that the current implementation uses 2 standard library cells to implement a single memory cell[9]. By designing a custom memory cell, we expect to reduce the area taken by a PE (and hence the array) by about half.

Table 3. Timing estimates for the 8×8 SRGA design.

Operation	Time required (ns)		Total
performed	(first half)	(second half)	time (ns)
Context switch	4.76	4.26	9.02
Memory read	5.09	3.83	8.92
Memory write	5.78	3.15	8.93
Memory to memory	5.09	3.15	8.24
Min. clock cycle	5.78	4.26	**10.04**

Table 3 shows the times required (in both halves of the clock cycle) to perform the context switching operation—please see Section 2.8 for a description of what happens in each clock cycle half. The results obtained through implementation demonstrate that the SRGA is capable of context switching in a single clock cycle.

Table 3 also shows the times required (in both halves of the clock cycle) to perform the memory read, memory write, and memory to memory data transfer operations—please see Section 3.1 for what happens in each half of the clock cycle for the above operations. Again, the times obtained show that the SRGA can perform memory access operations in a single clock cycle. The bottom row of the table shows the minimum time required for each half of the clock cycle (obtained by selecting the maximum times in their corresponding columns) and also the total clock cycle time of 10.04 ns. Thus the SRGA design can be expected to operate in the range of 80-100 MHz without optimization. Since the SRGA design has been shown to perform single cycle context switch and single cycle memory access, while operating at a reasonable clock speed, the chief claims made for the proposed architecture have been validated by the implementation.

6 Conclusion and Future Directions

This paper presented the detailed description of an innovative reconfigurable device architecture that performs single cycle context switching as well as single cycle memory access to the unified on-chip configuration/data memory. These 2 features were realized through the novel use of a mesh of trees interconnect with logic cells and memory blocks at the leaves and identical switches at the other nodes. Timing estimates obtained from an SRGA design synthesized using a standard cell library demonstrated that the architecture could perform both above features while operating at a reasonable clock speed.

[7] Synthesis of larger array sizes failed due to large database sizes.

[8] Unless design is optimized for speed. Results shown are for unoptimized design.

[9] The standard memories created by memory generators were not found suitable as the required memory block needs extra logic to handle the context switch operation.

The SRGA architecture is suitable for a large class of reconfigurable computing applications since it reduces the reconfiguration overhead and provides fast on-chip memory for data storage. But more important is the ability of the SRGA to perform efficient self-reconfiguration—it is made possible by the fast context switching and memory access capabilities. Self-reconfiguration is a powerful feature since it enables the reconfigurable device to modify its own configuration logic at runtime without any external intervention. This power is demonstrated by showing how the SRGA can perform basic routing operations very efficiently using self-reconfiguration—part of the efficiency is due to the simplicity and regularity of the interconnection structure. Further, significant speedups using self-reconfiguration have been obtained for string matching [6][7] and genetic programming [5] applications. The above applications require the self-reconfigurable device to provide fast context switching and memory access, which are precisely the characteristics of the SRGA.

Following are the future directions we plan to explore:

Interconnect As mentioned in Section 2.2, for efficient mapping of various types of logic, the interconnection resources of the SRGA may need to be increased. This can be done by adding more wires to each row and column tree, by connecting same level nodes in a tree, or by connecting row and column trees through non-leaf nodes. Note that all the above can be done while preserving the basic mesh of trees structure with identical switches.

Clocking Logic configured on different contexts would typically operate at different clock frequencies. Support needs to be added to the SRGA to enable configuration contexts to specify the required frequency and accordingly alter operating frequency after a context switch.

Switches Another feature being considered is the addition of configurable logic and/or a flip-flop to each switch. This would enable efficient mapping of muxes and decoders and would also help in retiming. Routing using self-reconfiguration would also become more efficient.

References

1. F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.
2. S. Perissakis, Y. Joo, J. Ahn, A. DeHon, and J. Wawrzynek. Embedded dram for a reconfigurable array. In *Proceedings of the 1999 Design Automation Conference*, Jun. 1999.
3. S. M. Scalera. Personal communication, 1998.
4. S. M. Scalera and J. R. Vazquez. The design and implementation of a context-switching fpga. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–85, Napa, CA, April 1998.
5. R. P. S. Sidhu, A. Mei, and V. K. Prasanna. Genetic programming using self-reconfigurable FPGAs. In *Field Programmable Logic and Applications - 9th International Workshop, FPL'99*, volume 1673 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.
6. R. P. S. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *FPGA '99. Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 217–226, Feb. 1999.
7. R. P. S. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna. A self-reconfigurable gate array architecture. In *Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems*.
8. Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, April 1997.
9. W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. High-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 69–78, Feb. 1999.