H-TREE BASED CONFIGURATION SCHEMES FOR A RECONFIGURABLE DSP ARCHITECTURE

By

ANDY WIDJAJA

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of

ANDY WIDJAJA find it satisfactory and recommended that it be accepted.

_____
Chair

_____

_____

Acknowledgement

First and foremost, I would like to thank my advisor Professor José G. Delgado-Frias for his endless support and guidance in my research work and completion of this thesis. I would also like to thank Professor Valeriu Beiu and Professor Jabulani Nyathi for being the committee members of this research.

Next, I would like to thank the School of Electrical Engineering and Computer Science for awarding me Teaching Assistantships.

And last but not least, I would like to thank my parents for always being there for me.

H-TREE BASED CONFIGURATION SCHEMES FOR A RECONFIGURABLE DSP HARDWARE

Abstract

by Andy Widjaja, M.S.
Washington State University
May 2005

Chair: José G. Delgado-Frias

Reconfigurable computing has attracted considerable attention recently because of the potential to deliver the performance of application-specific hardware along with the flexibility of general-purpose computers. Many reconfigurable architectures have been proposed in the last few years, however, few discussions have been conducted on the specifics of the reconfiguration scheme itself. This thesis describes two efficient configuration schemes for a reconfigurable DSP hardware that utilizes an H-tree interconnection network to link clusters of logic blocks, or cells, to map the desired circuits. The schemes make use of the existing hardware in a two-level reconfigurable cell array for communication of configuration data. The result is a speedy configuration that requires minimal additional control wires and hardware. Circuit simulations indicate that a configuration speed of 1 GHz can be achieved using a modest 0.18-μm CMOS technology.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The introduction of the microprocessor in the late 1970's and early 1980's made it possible for DSP techniques to be used in a much wider range of applications. However, general-purpose microprocessors such as the Intel x86 family are not ideally suited to the numerically-intensive requirements of DSP, and during the 1980's the increasing importance of DSP led several major electronics manufacturers to develop chips with architectures designed specifically for the types of operations required in digital signal processing.

Since then application specific architectures have been used to achieve higher performance than general-purpose processors. However, their circuits cannot be altered after fabrication. They require a redesign and chip refabrication if any part of the circuit needs modification. This inflexibility and the high design cost make them unattractive for a wide-spread application like digital signal processing. Reconfigurable computing has the potential to achieve most of the performance of tailored architectures while maintaining the flexibility of general purpose processors.

The following sections discuss the main requirements of DSP systems and compare reconfigurable hardware to other alternatives. The subsequent chapters begin by describing the system architecture and its interconnect structures. The main body of this thesis covers the configuration and reconfiguration schemes of this reconfigurable system.

1.1 Requirements of DSP Hardware

While DSP covers a very wide range of applications, a number of common metrics for its hardware can be recognized. They are:

- **Performance:** DSP imposes great demands on the processing power of any hardware implementation. For example, a 512-point Fast Fourier Transform (FFT) requires approximately 16,000 multiplications and 9,000 additions [1]. The hardware typical applies the same basic operation to multiple data points. Hardware implementations that exploit the parallelism of DSP algorithms will achieve much higher throughput.
- **Flexibility:** Commercial products will naturally choose their implementation strategy based on total cost. A few commodity devices with widespread usage are preferred over a large number of application-specific devices. To lower development, devices need to be designed with high flexibility so that they can be used in a large number of applications.
- **Power consumption:** Many DSP applications have recently been included in wireless communications and mobile computing. As a result, power consumption is a crucial design factor for many DSP systems.

- **Fault tolerance:** Radiation-induced errors, such as latch-up, burn-out , and single event upsets, are major concerns in environments with high background radiation, such as space. Hardware used for mission-critical applications, such as communication satellites and real-time monitoring equipment, must contain mechanisms to detect and overcome faults. Memory elements are particularly vulnerable to single even upsets, which occur when a charged subatomic particle causes a transient voltage spike that can subsequently change the state of the circuit.

Most applications require a balance between two or more of these metrics. Therefore, the ability of a DSP hardware to meet the particular needs of an application is another key factor influencing the design choice.

1.2 DSP Implementations

Digital systems typically use a variety of components to perform DSP operations. These range from application-specific integrated circuits (ASIC) to general-purpose microprocessors [2]. Table 1.1 shows a comparison of these approaches in terms of the four metrics described in previous section [3].

TABLE 1.1

Comparison of DSP implementations

Device	Performance	Flexibility	Power	Fault tolerance
General-purpose processor	Low	High	Medium	None
Digital signal processor	Medium	Medium	Medium	None
Configurable processor	Medium	Medium	Med-Low	Possible
Reconfigurable hardware	Med-High	High-Med	Med-Low	Inherent
ASIC	High	Low	Low	Possible

General-purpose processors can execute a wide variety of software programs, including DSP algorithms. However, their performance may not meet the requirements of the application [3]. Specialized digital signal processors include some instructions tailored for DSP computations. They generally achieve better performance than their general-purpose counterparts, but their architecture may not be optimized for the different requirements that DSP applications may have, such as speed, power, and word length. In addition, fault-tolerant processors are generally not commercially available.

Configurable processors have a customizable instruction set, datapath, and memory organization. Devices of this type are configured for a particular application prior to fabrication [4]. However, each configuration requires a new complier to generate optimal code. In addition, the use of such a processor may be limited to a specific application, so this approach does not achieve as high flexibility as the other alternatives.

Reconfigurable hardware allows designers to change the configuration of the hardware at any time. This approach provides great alternative for performance, flexibility, power, and fault tolerance [5]. Users also have the option to select between different trade-offs, such as performance over fault tolerance, or power over flexibility, according to the intended application.

Finally, ASICs are optimized for a particular DSP algorithm. These devices can achieve maximum performance and minimum power consumption, but at the expense of high development costs. Due to the cost and limited flexibility of an ASIC, this approach is only best-suited for extremely high-volume applications.

1.3 Reconfigurable Hardware

The goal of reconfigurable hardware is to combine the performance of an ASIC with the flexibility of a microprocessor. This approach has recently become practical for DSP due to the increasing capabilities of VLSI systems. In general, reconfigurable devices contain an array of programmable cells and interconnections. DSP algorithms are divided into small portions and mapped onto the structure. Unused portions of the hardware can be disabled to lower the total power consumption. Since the hardware configuration can be changed at any time, even after deployment, reconfigurable hardware achieves great flexibility [5]. In addition, the design process can be automated using appropriate software tools [6]. Finally reconfigurable hardware possesses a certain degree of fault tolerance, such that DSP algorithms can be remapped around faulty cells if the circuit is damaged.

Traditional reconfigurable devices such as field-programmable gate arrays (FPGA) place little functionality in the cells [7]. These fine-grain devices work well for implementing combinational or sequential logic. However, DSP uses mathematical operations such as multiplication extensively. Unless the architecture contains dedicated hardware for this purpose, mapping a multiplier onto a fine-grain device creates a complex structure that yields poor performance [8].

New reconfigurable devices that incorporate adders, multipliers, lookup tables and other functional units in the cells have surfaced in recent researches [9, 10, 11]. To some degree, these coarse-grain devices are successors to the older reconfigurable systolic array architecture, as in [12]. In general, coarse-grain reconfigurable hardware achieves good performance for mathematical functions, but may not implement all the

control logic necessary for DSP. The fixed number of functional units also limits their flexibility.

This thesis focuses on the configuration and reconfiguration scheme of a novel medium-grain reconfigurable architecture for DSP [13, 14, 15, 16]. The chapters are organized as follows: Chapter 2 describes the system architecture and shows examples of how various operations can be mapped onto the array of cells. Chapter 3 discusses the issues concerning the design of a reconfiguration scheme. Chapter 4 describes a broadcast based configuration scheme that makes use of special broadcast switches to shorten configuration time. Chapter 5 considers a unicast based configuration scheme that achieves slightly lower configuration speed but incur significantly less complex hardware implementation; it also performs better in conjunction with partial reconfiguration. Chapter 6 illustrates the hardware implementation of the unicast scheme and shows its simulation results. Chapter 7 looks at the scheme performances and compares them to other reconfigurable system currently available in the market. Finally, chapter 8 provides some concluding remarks.

Chapter 2

System Description

Most of the contents in this chapter are summaries of Mitchell Myjak's thesis [17]. The reconfigurable architecture is best described in different levels of organization. We begin the system description with the upper level organization. At this level the architecture consists of an array of reconfigurable cells and interconnection structures. The lower level organization focuses on the processing core within each cell and the arrangement of the reconfigurable elements within the core. Description of the hardware organization of the cell follows that of the lower level architecture. And finally, the chapter ends with illustration and detailed description of the interconnection structure.

2.1 Upper-Level Organization

At the upper level of the two-level architecture, the array of cells can be grouped into functional blocks to implement basic operations, such as multiplication and addition. Figure 2.1 illustrates a portion of the reconfigurable cell array. Each cell performs

operations in 4-bit units. The choice of 4-bit cells was selected to give designers enough control over the data word length and maximizes the utilization of the device [18]. Using larger cells would increase the fan-in and fan-out of the gates which in turn may disrupt signal integrity and impede the datapath. A mesh of 4-bit busses connects neighboring cells horizontally and vertically. Additional busses allow data to be routed between non-adjacent cells and will be covered in more details in a later section in this chapter.

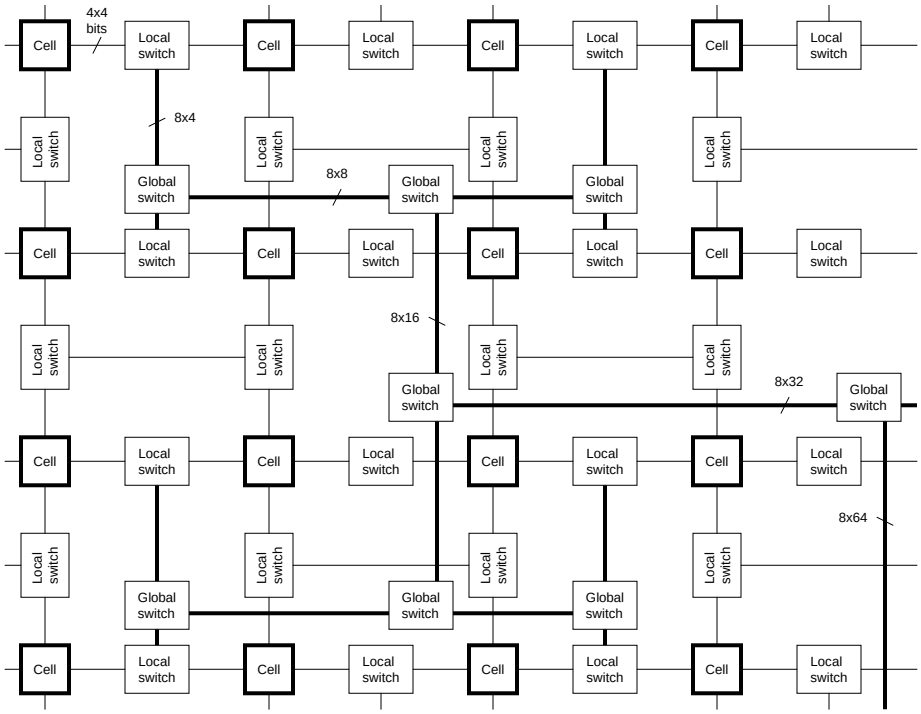


Figure 2.1 Array of cells in reconfigurable architecture

2.1.1. Mathematics Operations

We first illustrate cell arrangements for mathematic operations. Almost all DSP algorithms use multiplication of some form. Depending on the target application, the algorithm may require signed or unsigned multiplication of 16, 20, 24 32-bit, or larger. The use of 4-bit cells enables applications to implement a multiplier of the precise size

required, while benefiting from the inherent parallelism of the operation. Suppose the reconfigurable device is to multiply two unsigned 16-bit numbers *A* and *B* to generate a 32-bit output *Y*.

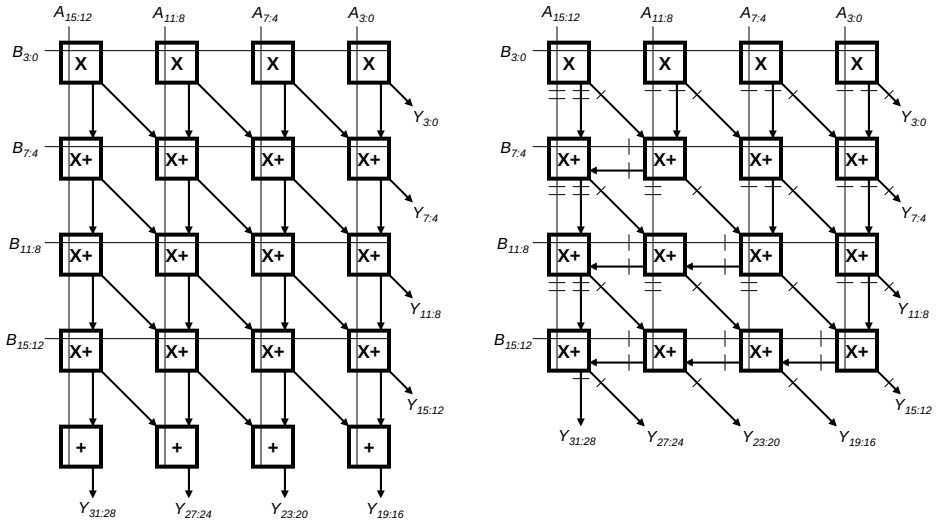


Figure 2.2 (a) 16-bit carry-save and (b) 16-bit modified multipliers

Figure 2.2 illustrates two possible options for mapping the multiplier onto the array of cells. Figure 2.2(a) outlines a straight forward implementation of a carry-save multiplier. This multiplier requires twenty cells. The critical path involves eight cells. By rearranging the interconnection structure, it is possible to reduce the hardware required. Figure 2.2(b) illustrates a more compact multiplier that uses sixteen cells and has a critical path of seven cells. The hash marks in the figure indicate the number of pipeline stages that separate each cell so that intermediate results arrive at the next cell at the proper times. The top row of cells in Figure 2.2(b) performs multiplication but not

addition. If two additional 16-bit terms can be added, the multiplier can be modified to be a multiply-accumulate (MAC) unit which prove to be useful in many algorithms.

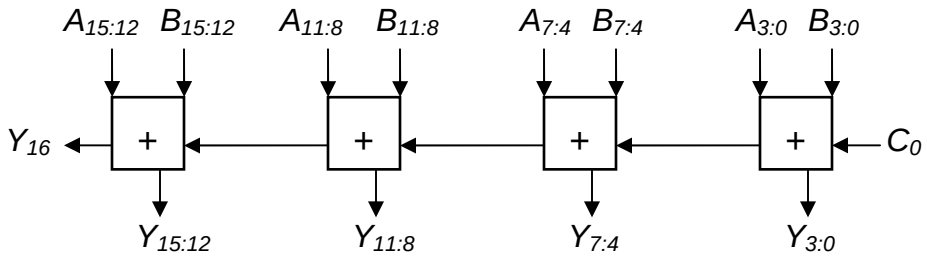


Figure 2.3 16-bit adder

Addition operations are required equally as much as multiplications. In many cases, an addition can be combined with a multiplication and implemented with MAC unit. However, some algorithms still require a dedicated adder. Cell arrangement of a 16-bit adder is shown in Figure 2.3

2.1.2 Memory Operations

Memory is often needed to store intermediate results when mapping DSP algorithms onto the hardware. For example, the Fast Fourier Transform (FFT) requires a working buffer approximately the size of the input data. Other reconfigurable devices typically embed memory blocks within the logic blocks of the array. The two-level cell array is unique in that each cell itself can implement a 64x8-bit memory. Portions of the reconfigurable device can implement the random-access memory while other parts of the array implement the algorithms.

Figure 2.4 illustrates a 512x16-bit memory. The rightmost column of "D" cells decodes the 8-bit address *A*. The main 8x8 block of "M" cells implements the actual memory. The entire module operates in a pipelined fashion. As an access request travels through the pipeline, each decoder cell determines whether *A* falls within the address range of the corresponding row of memory cells. If so, the *re* or *we* signals of the respective cells are asserted. If not, a no-op (no operation) occurs and the memory cells pass the data unchanged to the next row.

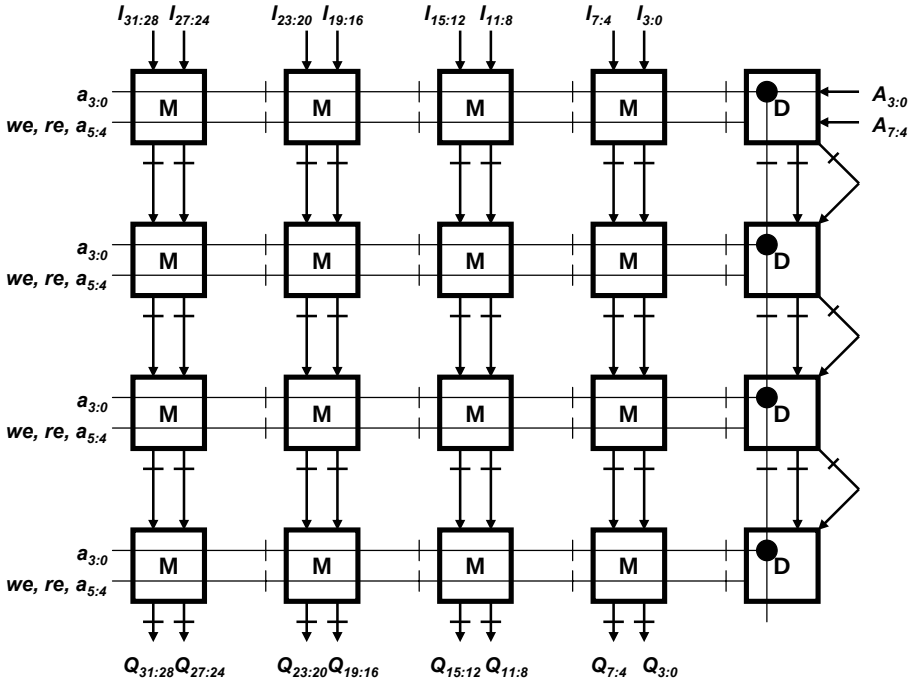


Figure 2.5 A 512x16-bit memory

DSP operations are not composed entirely of mathematical functions, but also require some control logic for proper operation. The control logic may include ANDs, ORs, decoders, multiplexers, and simple state machines. The examples on cell arrangements described in this section and those of other operations can be found in [14, 15, 16].

2.2 Lower-Level Organization

At the lower level, the main component of the cell is a processing core that is made up of a 4x4 matrix of reconfigurable elements. Each element contains a 16x2-bit memory. The processing core can be configured into two structures. The first structure is optimized for memory operations, and the other for mathematical functions. Both structures execute one operation during the evaluation phase of the clock. This section describes the two modes of operation in detail.

2.2.1 Memory Mode

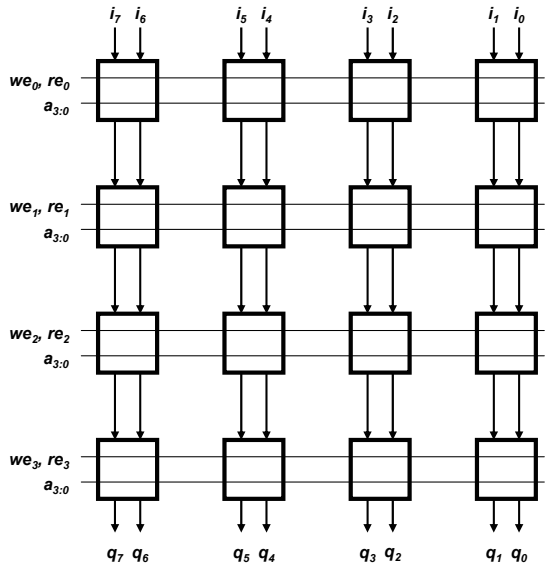


Figure 2.6 Processing core in memory mode

In memory mode, the sixteen elements in the processing core implements a 64x8-bit memory. Figure 2.6 shows the processing core in memory mode. The lower four bits of the address bus *a* connect to every element. The control module uses the upper two bits of *a* to generate *re* (read) and *we* (write) signals for each row of elements. Lines *i*

and *q* are the input data and output data, respectively. Each column of elements handles two bits of data. The memory mode is used for storing of intermediate results, creating a table of constant coefficients, and implementing multivariable logic functions. These functions are necessary to implement the control logic required in DSP algorithms.

2.2.2 Mathematics Mode

In mathematics mode, the processing core reuses the same memory elements to implement mathematical functions. Figure 2.7 shows the processing core in memory mode. The matrix of elements now assumes a structure similar to the MAC unit in section 2.1.1. This structure is optimized for the MAC equation:

$$y_{7:0} = (a_{3:0} \times b_{3:0}) + c_{3:0} + d_{3:0}$$

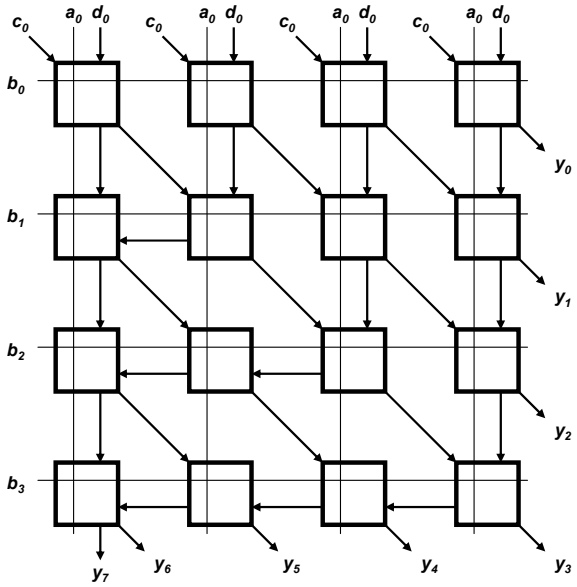


Figure 2.7 Processing core in mathematics mode

Each element acts as a 16x2-bit lookup table, thus the carry-save structure can implement many functions besides multiplication. Details on other types of cells

implemented using the 4x4 matrix of reconfigurable elements can be found in [14, 15]. Besides mathematical functions, the two references also illustrate the processing core alignment for logical functions such as shifting, ANDing and ORing.

2.3 Hardware Organization of Cell

This section completes the description of the lower-level organization by detailing the circuit design of an element. Figure 2.8 depicts the organization of one element in the processing core. Each element contains a 16x2-bit memory. This memory is arranged into a 4x4 array of 2-bit latches through the use of additional glue logic. In memory mode, the element has a 4-bit address *a*, a 2-bit input data *i*, and a 2-bit output data *q*. In mathematics mode, the four address bits are pre-empted by inputs α, β, γ, and δ. The lower two bits control a row decoder, and the upper two bits control a column decoder.

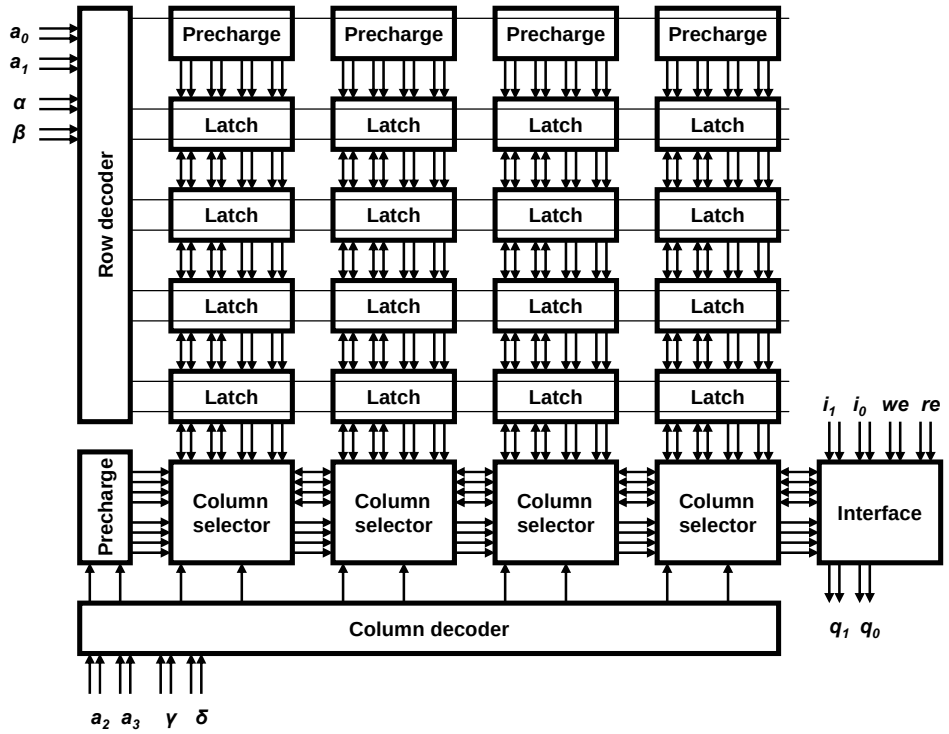


Figure 2.8 Organization of the reconfigurable element

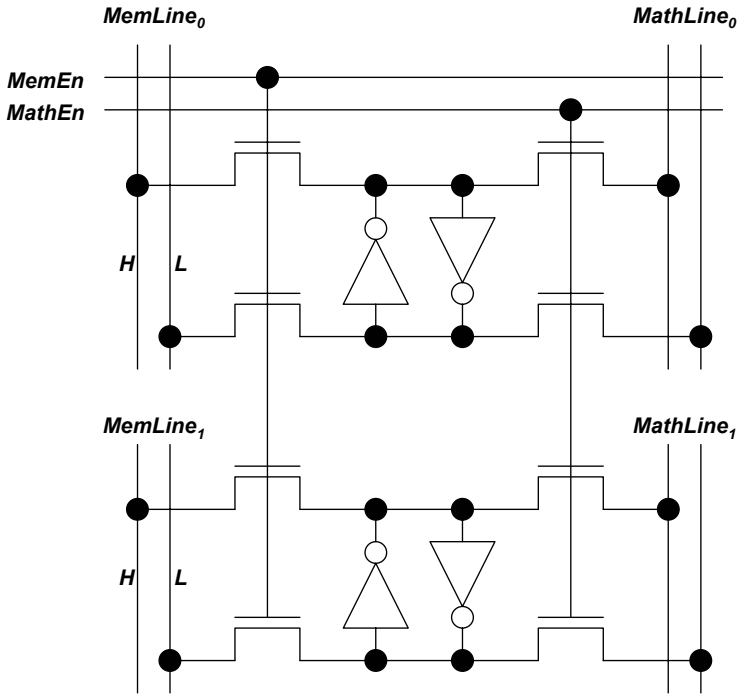


Figure 2.9 A 2-bit latch with separate paths for memory mode and mathematics mode

Each 2-bit latch contains two static random-access memory (SRAM) as shown in Figure 2.9. The circuit provides separate paths for memory mode and mathematics mode. For a read operation in memory mode, the element first precharges $MemLine_i(H)$ and $MemLine_i(L)$ to V_{DD}. The row decoder then asserts the *MemEn* input, allowing the latch to discharge one of these signals to ground. Strong *n*-transistors are used in the latch to expedite this operation. For a write operation in memory mode, the element drives the new data in $MemLine_i(H)$ and $MemLine_i(L)$. When *MemEn* is asserted, the data overwrites the value previously stored in the latch. A read operation in mathematics mode proceeds in a similar fashion as a read in the memory mode.

2.4 Interconnection Structure

To implement a DSP algorithm, reconfigurable hardware is often partitioned into discrete functional units, such as multipliers, adders, memories and control logic. Each unit is then mapped onto a block of cells and signals are routed in-between the blocks. However, the data transfer required within a block differs from that required between the functional units. Adjacent functional units typically exchange data in units of words, whereas the cells within a unit handle data in smaller portions. The reconfigurable cell array uses a novel interconnection structure that implements data transfer both within and between functional units. As shown in Figure 2.10, a mesh of 4-bit busses connects neighboring cells horizontally and vertically. Integrated on top of the mesh is a structure known as the "H-tree". Each level of this global binary tree contains a fixed number of busses. However, the number of bits per bus increases (doubles) at each higher level and eventually taper off after a certain level.

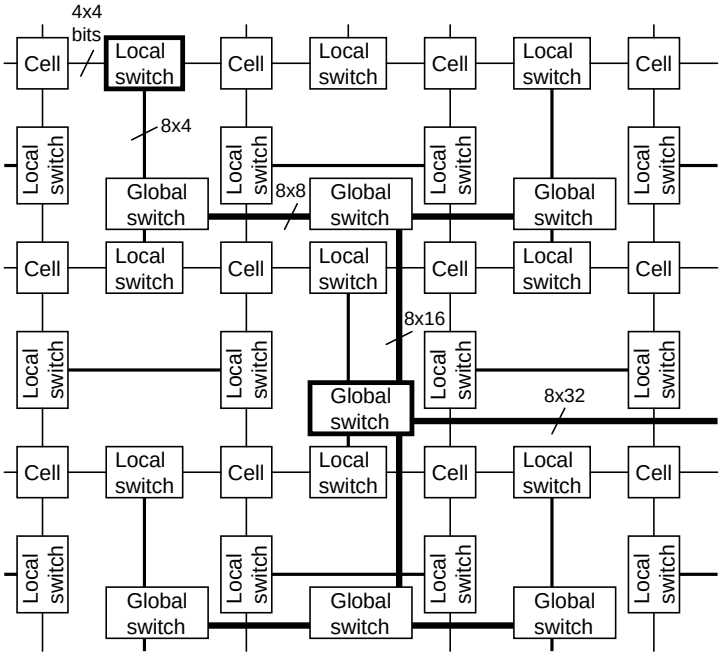


Figure 2.10 Interconnection structure in the reconfigurable architecture

2.4.1 Local mesh

The local interconnect is shown in Figure 2.11. It allows cells to transfer intermediate results within a functional unit (or block). A mesh of 4-bit busses connects cells horizontally and vertically. All busses are unidirectional. Additional "center beams" allow data to be routed in diagonal directions.

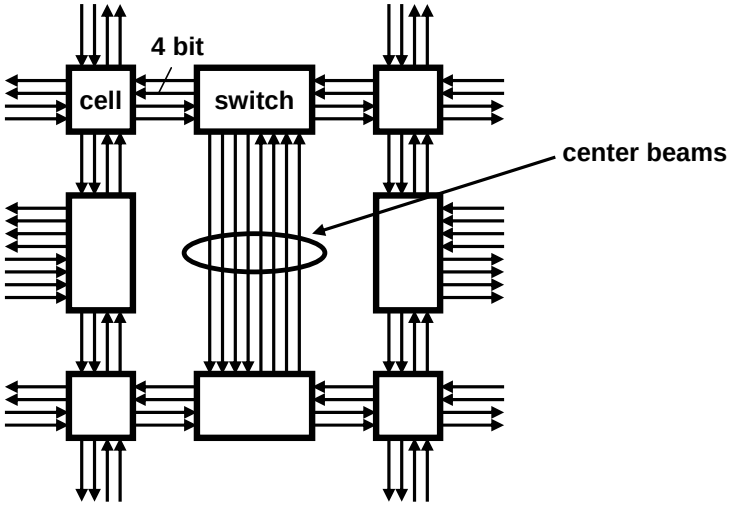


Figure 2.11 Local mesh of 4-bit busses with additional "center beams"

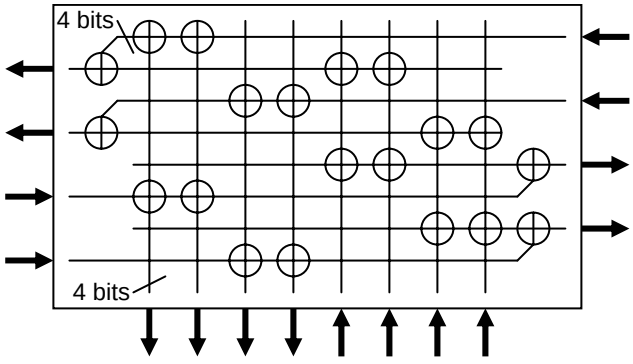


Figure 2.12 A local mesh switch

Figure 2.12 shows one of the switches in the local mesh. The switch manipulates each 4-bit bus separately. Incoming data from a cell can either be routed to the cell opposite the switch, or through the center beam to two more diagonal cells.

2.4.2 Global H-tree

The global H-tree, illustrated in Figure 2.13, is used to route inputs and outputs of functional units across the reconfigurable cell array. The two lowest levels of the tree are shared with the local interconnect, therefore cells that only interact with other cells within the same functional unit do not waste the capacity of the global interconnect. The root of the tree connects to an internal memory or to the external pins of the device.

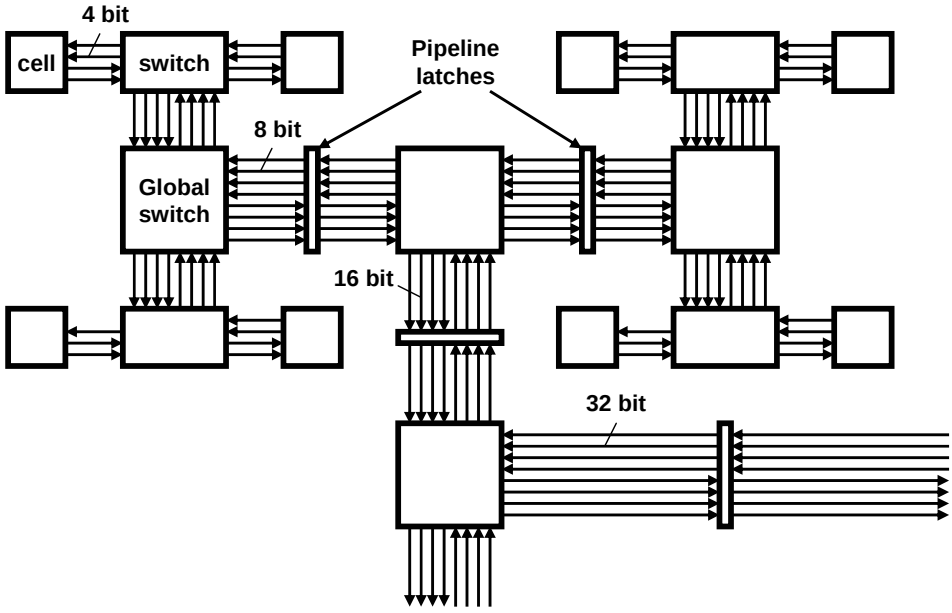


Figure 2.13 Global H-tree

Each level of the H-tree contains four input busses and four output busses. Data originating from a cell travels up the output path until it reaches the highest level required. It then cuts through to the input path and descends to its destination cell. Both directions of travel on the global interconnect require routing through multiple levels of switches. Therefore, the H-tree incorporates pipeline latches to enable higher clock frequencies. To allow simultaneous data transfer to and from each cell, the number of lines in each bus doubles at each level, up to a maximum of 64 bits.

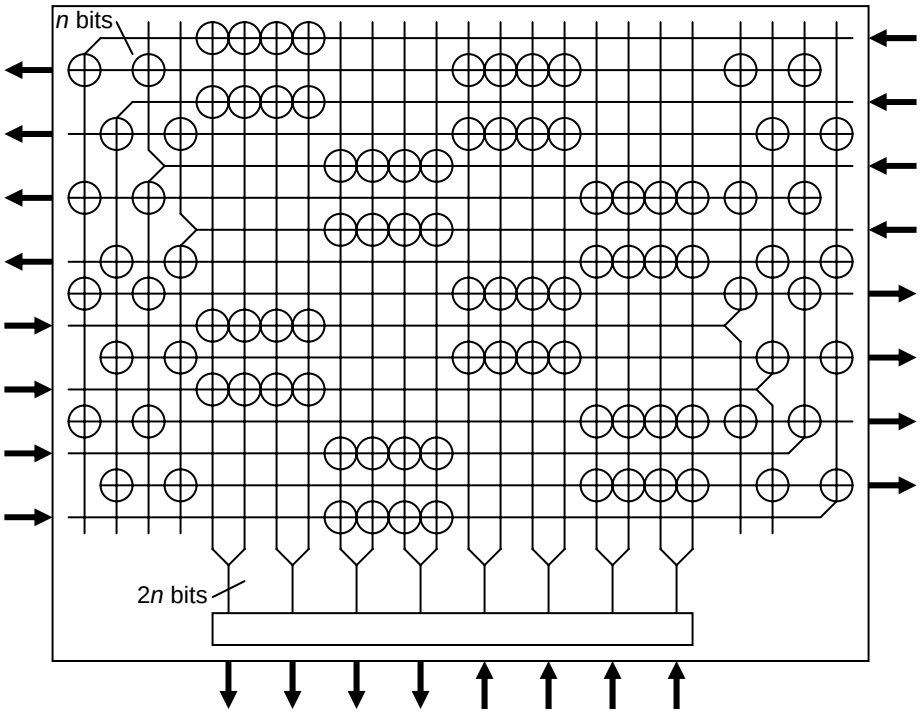


Figure 2.14 A typical switch in the global H-tree

The above diagram (Figure 2.14) depicts a typical switch in the global H-tree. Similar to the switches in the local mesh, busses are divided into two groups. However, the switches route data in units of 8, 16, 32, or 64 bits. The structure of each switch is similar from level to level, only the number of bits per bus changes on each level. On the input path, the 2*n*-bit busses from the upper level can be routed onto the *n*-bit busses of the same group in the two lower levels. The least significant and most significant *n* bits of the input are handled separately. On the output path, each *n*-bit bus from the lower level can be copied onto an outgoing 2*n*-bit bus of the same group. Alternatively, the switch can transfer data from the output path to the input path on the same level. In this case, the group designations are not observed. This approach allows designers to create libraries of functional units that can be connected easily without conflicts.

Chapter 3

Reconfiguration Issues

Application specific architectures have long been used to achieve higher performance than general-purpose processors. However, their circuits cannot be altered after fabrication. They require a redesign and chip refabrication if any part of the circuit needs modification. This inflexibility and the high design cost make them unattractive for a wide-spread application like digital signal processing. Reconfigurable computing has the potential to achieve most of the performance of tailored architectures while maintaining the flexibility of general purpose processors. This is done through configurable logic blocks that are connected using a set of routing resources that are also programmable.

3.1 Configuration Speed

The operation of a reconfigurable system occurs in two distinct phases, namely configuration and execution. Depending on the system design, configurations can be

loaded exclusively at start-up of a program, or periodically during runtime. The targeted system is aimed to support run-time reconfiguration. The concern that arises from run-time reconfiguration is that it involves reconfiguration during program execution. Therefore the reconfiguration process must be done as efficiently and as quickly as possible. Without a fast reconfiguration, the overhead of configuring the hardware diminishes any acceleration gained by the system.

For example, to illustrate the seriousness of this issue, the DISC II system [19, 20] spends 25%-71% of its execution time on reconfiguration. Other systems like the ATR work by UCLA [21] consume as much as 98.5% of its execution time on reconfiguration. If the delays caused by reconfiguration are reduced, performance can be greatly increased. Therefore, fast configuration is an important area of research for run-time reconfigurable systems.

3.2 Data Volatility

While fast configuration can reduce the run-time reconfiguration overhead, another method to accelerate this process is partial reconfiguration. In some cases, configurations do not occupy to the full reconfigurable array, or only a portion of the mapped circuit requires modification. In such situations, a partial reconfiguration is more suitable than a full reconfiguration.

In partial reconfiguration, the consideration to be made is on the retention of configuration data of the reusable circuit, and thus the concern about data volatility. Due to the hierarchical nature of our interconnect structure, it is necessary to configure the components at the lowest level (i.e. the logic cells and local mesh connections) before

configuring the layers above it (i.e. the global interconnects). As such, configuring a cell layer will impose routing configuration changes on the global interconnects leading to it as we will see more clearly in Chapter 5.

One method of retaining a reusable configuration from the previous execution is to store the configuration data in a nearby memory, and restoring it after the reconfiguration process. However, this also means additional memory space and clock cycles to perform the storing and restoring. Other methods involve logic gates to determine which parts of the array to be modified and which parts to be left unchanged. The consideration is thus on the amount of additional hardware to support data retention and the extra time required to manage the reusable circuits.

3.3 Configuration Caching

A great amount of delay caused by configuration is due to the distance between the host processor and the reconfigurable hardware. The delay is further increased if the reconfigurable hardware is housed in a casing with limited bandwidth leading in/out of the chip. A configuration cache can significantly reduce the costs of reconfiguration [22, 23].

By storing the configurations in memory near to the reconfigurable array, the data transfer during reconfiguration can be greatly accelerated, and the overall time required reduced. Additionally, a configuration cache can allow for a direct output bus to the reconfigurable hardware [24]. This bus can further reduce configuration times by taking advantage of the close proximity of the cache by providing high-bandwidth communications that facilitate a wide parallel loading of the configuration data.

3.4 Hardware Considerations

The design of the logic blocks within the reconfigurable hardware varies from system to system. Some of which are described in [2]. For the purpose of this thesis, the targeted system is a two-level reconfigurable architecture optimized for DSP described in [14, 17]. The routing between the logic blocks is also of great importance. To configure the routing, typically a passgate structure is used (as in Fig. 1). The programming bit will turn on a routing connection when configured with a true value, allowing signal to flow from one wire to another. It will disconnect the wires when the bit is set to false.

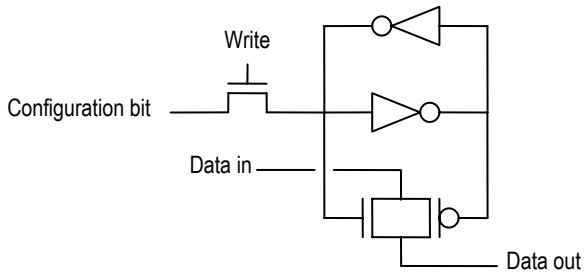


Figure 3.1 An SRAM-based programmable routing bit

Local and global routing resources usually come in two flavors, namely segmented and hierarchical routing. Systems such as RaPiD [25] and LEGO [26] use segmented routing. In segmented routing, short wires provide local communications. These short wires can be connected together using switch crossbars to emulate a longer wire. Bypass wires are often employed to allow signals over long distances without passing through many switches. Hierarchical routing, on the other hand, tends to group logic blocks into clusters. Local meshes provide routing within a cluster of logic blocks. At higher levels, longer wires connect the different clusters together. This is repeated for

a number of levels. The idea is that most communication should be local and only a limited amount of signals will travel long distances.

In a large array of logic blocks, the routing and its programmable bits usually contribute to a significant area of the reconfigurable hardware. Furthermore, the amount of routing required does not grow linearly with the amount of logic present. Larger devices require even more routing resources per logic block than smaller ones [27]. DeHon showed in [28] that the most area efficient designs will be those that optimize their use of the routing resources rather than the logic resources. The contribution of this thesis is a configuration scheme that maximizes speed and the use of existing routing resources in a reconfigurable architecture.

Chapter 4

Broadcast Based Configuration Scheme

Hierarchically, the global interconnect switches form a balanced binary tree with the logic cells as leafs. The depth of the tree is determined by the size of the cell array. A 16x16 cell array, for example, requires a 9-level binary tree of global interconnecting switches. The global interconnect forms an integral part of the DSP operation as data often require to be routed to distant cells within the array [29, 30]. Since the global interconnect reaches out to every cell in the array, it is beneficial to make use of this existing communication scheme to transfer configuration data. As such, minimal or no additional wiring is required for configuring the cells. In this chapter, a novel configuration scheme is described; this scheme utilizes the hierarchical tree to rapidly broadcast configuration data to the cells and programmable switches.

To implement a DSP operation, the reconfigurable cell array is partitioned into blocks of different sizes. Each block is configured to implement an adder, multiplier, memory module, or other functional unit [31]. In order to configure a block of functional

unit, four layers of components require configuration. They are (in the order of first to last to be configured) local interconnects, cell's processing core, cell's internal switches, and global interconnects.

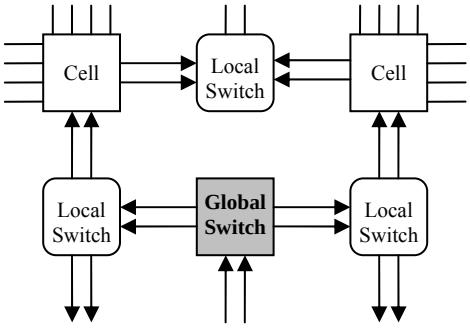


Figure 4.1 Close-up of cells and nearby switches

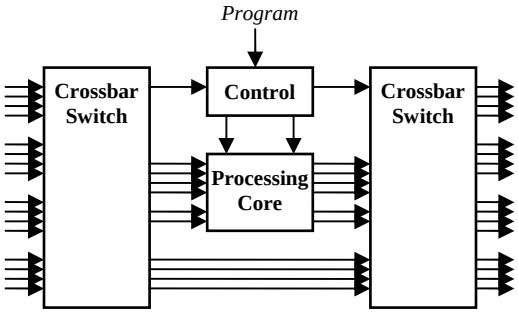


Figure 4.2 Simplified components of a cell

The local interconnect switches are first to be configured since they form the deepest component to be reached by the H-tree structure. In fact, these switches are not directly connected to the global interconnects. Therefore, their configuration bits are transmitted through the reconfigurable cells, before the cells themselves are configured (as illustrated in Figure 4.1). Within the cell, there are two components to be configured, the processing core and the internal switches. Figure 4.2 shows the organization of a cell.

The processing core implements the operations required for DSP. It contains a 4x4 matrix of elements which implement a look-up table that serves the desired function. Each element is made up of 16x2 (or 32) bits SRAM [13]. The two crossbar switches connect the inputs and outputs of the processing core to the interconnection network.

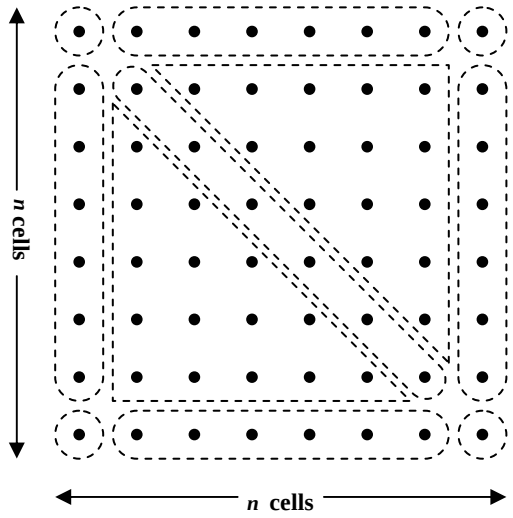


Figure 4.3 Generic grouping of a functional block

Although the configuration data are different for each layer of components, the way in which they are transmitted is the same. Additionally, Figure 4.3 shows a generic grouping of cells that form all the functional blocks of DSP operations. All the cells within each looping have the same configuration. Thus, there can be at most 11 different configurations within a block. Moreover, the internal switches and local switches within each looping have also the same configuration. This in turn provides a significant advantage by minimizing the configuration bits. Figure 4.4 shows the cell configurations and data flow of a 32-bit multiplier-adder block. A, B, C, D, E, F and H represent the different cell configurations. The arrows depict the data flow that is determined by the

internal and local switches of each cell. Notice that the switch directions are similar among the same cell configurations.

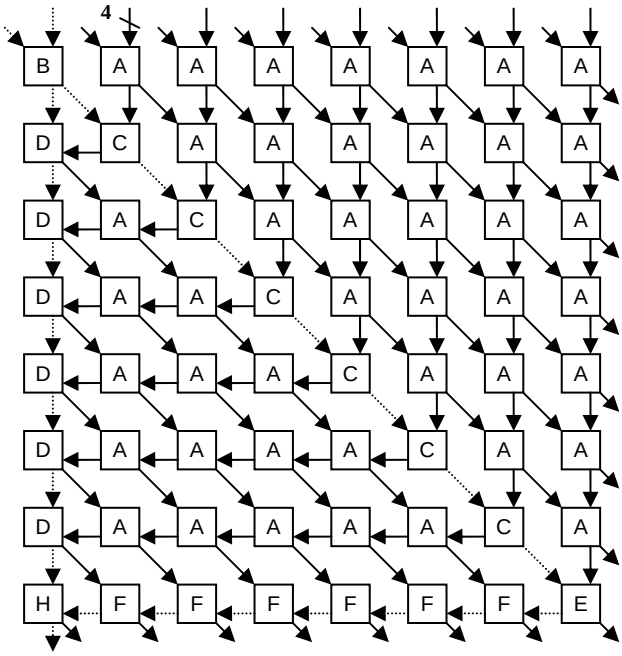


Figure 4.4 A 32-bit multiplier-adder

The global interconnects (not shown in Fig. 4.4) are the last to be configured. Naturally, during configuration of the global interconnects, the lowest level switches are configured first followed by those on the next upper level, up till the topmost switch. The discussion in the next section uses this cell core configuration as illustration. However, the same scheme is applicable to the four layer types of components to be configured.

4.1 Scheme Description

The existing architecture allows for two modes of operation, namely memory mode and mathematics mode. During normal operations, the logic cells are set in mathematics mode, though some cells may be set in memory mode for storage of intermediate processing data [14]. During configuration and reconfiguration, the cells are switched to memory mode to receive configuration bits through the global interconnects.

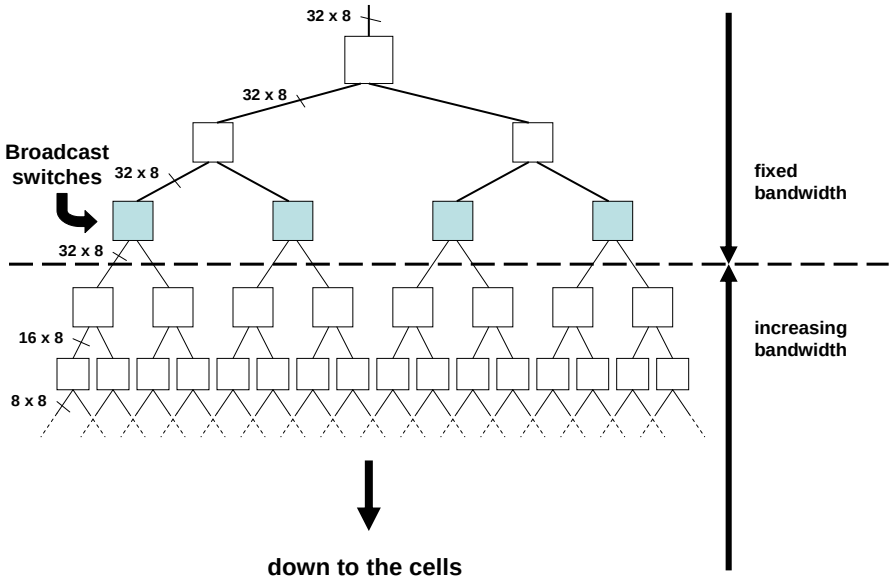


Figure 4.5 Global Interconnect bandwidths

Configuration data is channeled from the highest node of the H-tree, through the network of interconnect switches, down to the receiving reconfigurable cells. The H-tree structure supports an increasing bandwidth with each higher switch level, usually up to the maximum determined by the hardware/technology used. In our DSP hardware, the global interconnect bandwidth is maxed at 32x8 bits of bus going each way. Since the

configuration bits are only transferred top-down in the hierarchy, 32x8 bits is the maximum bandwidth available.

Figure 4.5 illustrates the H-tree in a binary tree structure for a 16x16 cell array. Notice that the switch bandwidths increases up to the maximum of 32x8 bits, beyond which they remain constant up to the top of the tree. This property splits the H-tree into two portions, one with increasing bandwidths and one with fixed bandwidths. The portion with fixed bandwidths covers only a small number of switches at the uppermost levels. This is advantageous as any additional hardware needed to decode a configuration compression need only be placed in this small number of switches. As for the rest of the switches, they need only be setup in one broadcast connection (as shown in Figure 4.6) to perform data transfers to the reconfigurable cells. Notice that there is only an 8-bit bus going into each cell. Since there are 512 bits of data per cell configuration, 64 clock cycles are required to configure each cell.

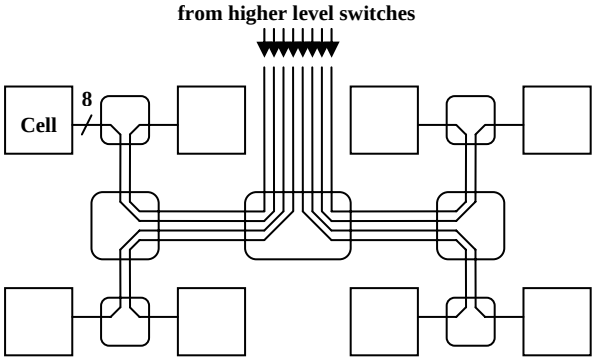


Figure 4.6 Increasing bandwidths of the global switches

The increasing bandwidth of the H-tree provides a direct connection from the **broadcast switches** (see Figure 4.5) to virtually every cell in the array. With all of the

switches below the broadcast switch setup as in Figure 4.6, the data transfer makes full use of the entire available bandwidth. Transmission of configuration bits from this point onwards is instantaneous. It will take 64 clock cycles to configure 32 cells as each cell will only take the delivery of 8 bits at a time. It will take 512 cycles to configure the entire 256 cells in a 16x16 cell array. With some modification to the broadcast switches, the configuration cycles can be reduced by half. The task at hand lies on properly configuring the broadcast switches before sending the configuration bits to the cell array.

4.2 Broadcast Switches

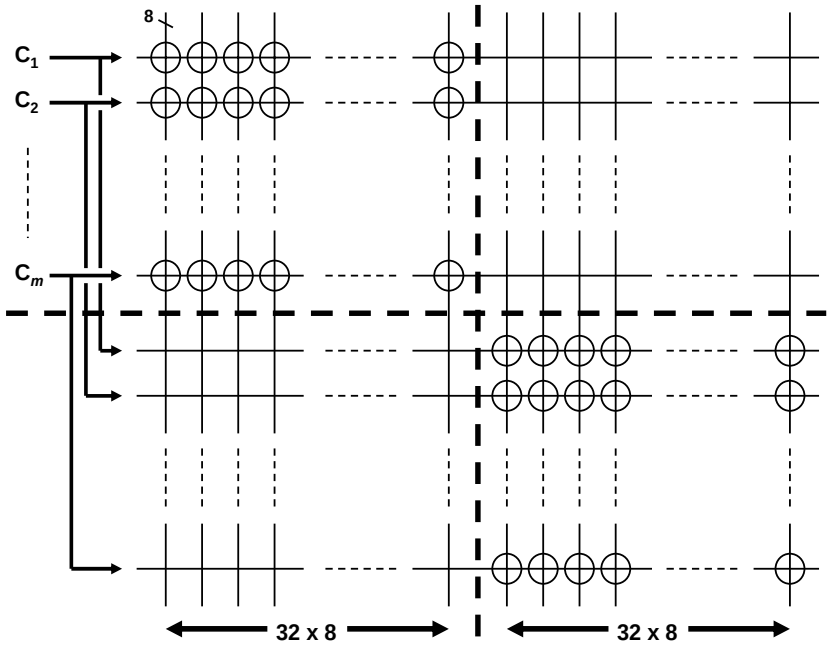


Figure 4.7 Broadcast Crossbar Switch

The broadcast switches are located at the frontier between two separate sets of switches. The switches below them have increasing bandwidths at each higher level. The

broadcast switches themselves and any switches above them have fixed bandwidths. During configuration, the broadcast switch crossbar is setup as shown Figure 4.7. Each switch is responsible for sending broadcast data to two sub-trees. Each sub-tree receives up to 32x8 bits of data each clock cycle. C_1, C_2, … C_m are the different types of cell configurations. Considering *m* number of configuration types, this requires 64 x *m* crosspoints (32 x *m* crosspoints each side). However, as discussed in the system description, we know there are at most 11 types of configurations per functional block of cells. This reduces the number of crosspoints to be configured in the broadcast switch considerably.

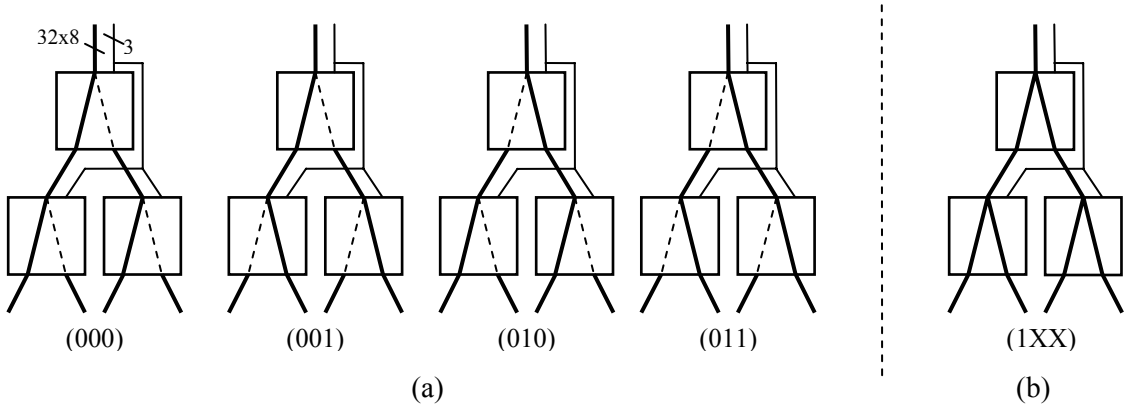


Figure 4.8 (a) during configuration of broadcast switches, (b) when sending cell configuration data

It is worth noting that the discussion thus far excludes the routing of switches above the broadcast switch. We have assumed them to convey the required information to the broadcast switches at the proper time. It is crucial that these switches do not incur additional time that will impair the configuration efficiency. With the help of a small decoder, these switches can be synchronized with the incoming data so that the correct data reaches the intended broadcast switch. Figure 4.8(a) illustrates the different routes

set on these switches during the steps taken to configure the four broadcast switches. Figure 4.8(b) shows the subsequent broadcast of the cell configuration data to all the broadcast switches after they have been properly setup.

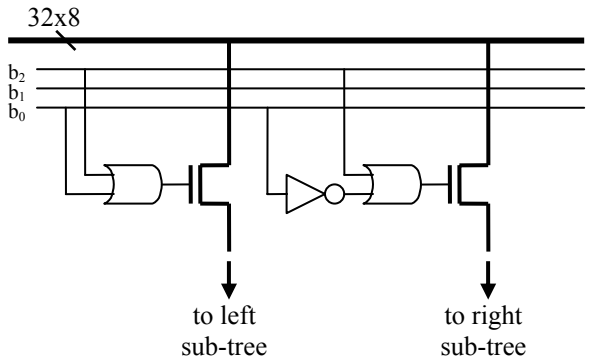


Figure 4.9 Schematic of simple decoder

To implement the decoder, 3 additional bits (b_2, b_1, and b_0) are supplemented to the top three global switches' bandwidth. These 3 bits will be issued from the host processor together with the respective 32x8 bits of broadcast switch configuration. Referring to Figure 4.8(a), the lower two switches are made to listen to the least significant bit, i.e. b_0. The topmost switch is made to listen to the second bit, b_1. And the third bit, b_2, is reserved for setting the broadcast branching that is shown in Figure 4.8(b). Figure 4.9 illustrates a simplified schematic for the implementation of this decoder.

The additional decoder is justified by the fact that it is only required at the uppermost three switches of the tree structure. The increasing bandwidth in the lower portions of the H-tree made it possible to do a data broadcast from high up the switch hierarchy which in turn benefited us in having only to focus our configuration scheme design on the top level switches.

4.3 Performance Estimation

In this section we take a closer look at configuring a system that allows up to 16 cell configuration types. Therefore, in this system, there are 1024 crosspoints to be configured at each broadcast switch. Fortunately, there are only four broadcast switches in a 16x16 cell array. With a fixed 32x8 bits bandwidth from the top of the tree to the broadcast switches, it takes 4 cycles to configure each switch (assuming proper pipelining), or 16 cycles to configure all the four broadcast switches. However, 1 cycle is needed to send a *control word* preceding each switch configuration (more on this *control word* in chapter 5). So a total of 20 cycles is needed each time the broadcast switches are to be configured.

TABLE 4.1

Number of configuration clock cycles
for a 16x16 cell array

Component	Broadcast switch setup cycles	Configuration cycles	Total cycles
Local switches	20	2 x (3 + 1)	28
Cell cores	-	64 + 1	65
Internal switches	-	2 x (8 + 1)	18
Global switches	5 x 20	17	117
		Total	228

Once the broadcast switches are properly configured, the cell configuration data, i.e. C_1, C_2, … C_{16} can now be broadcasted to all the cells. As shown in Table 4.1, the local switches are the ones to be configured first, therefore a 20-cycle setup time is

required to configure the broadcast switches. The major advantage of this scheme is that once the broadcast switches are configured, their configurations can be reused to send subsequent configuration data to the cell core and cell internal I/O switches. As such, no setup time is required prior to sending configuration data for these components after the initial setup for the local switches. Unfortunately, the same cannot be done for the global switches. The routing connections of the global switches are usually highly asymmetric. Therefore they usually do not follow the same pattern as the local components (i.e. local mesh, cell core and internal switches). To configure the global switches, the broadcast switches will usually require a different setup for every layer along the H-tree.

In a 32x32 cell array, a full configuration process will have to be split into four 16x16 configuration sequences (assuming the maximum bandwidth of the bus is still capped at 256-bit per direction). Therefore, a full configuration of a 32x32 cell array requires four times the total cycles shown, or approximately 912 clock cycles.

Chapter 5

Unicast Based Configuration Scheme

In the previous chapter, a configuration scheme was described. This scheme provides a fast configuration; however, it requires a large and complex *broadcast switch* design. In order to minimize transmission of the same configuration data to several cells located throughout the array, the broadcast switches are configured to identify these target cells. Only after that, the configuration data for the cells themselves can be transmitted. This method reduces the number of clock cycles for configuring the cells significantly. However, the number of configuration cycles required for the H-tree global communication switches remained high due to the highly irregular data these switches require. As the reconfiguration of the global switches is expected to occur more frequently than that of the cells, the speed advantage in configuring the cells alone may become less significant in the long run. In addition to that, the high number of cross-points in the broadcast switch that need to be managed greatly increased its design complexity and space requirement.

In this chapter, we propose a simpler configuration scheme that requires almost no additional hardware other than those that are already present in the existing architecture. The scheme continues to make use of the H-tree global communication network as the main channel of transmitting configuration data. However, instead of having a differentiated and complex broadcast switch, all the global switches have the same design. Figure 5.1 illustrates a typical setup in a global H-tree switch. Each level of the H-tree contains eight busses, four in each direction. The number of lines per bus doubles at each level from 4 bits to 8, 16, 32 and eventually 64, at which point the bandwidth levels off. As such the number of cross-points in the global switches remains the same throughout the levels of the H-tree.

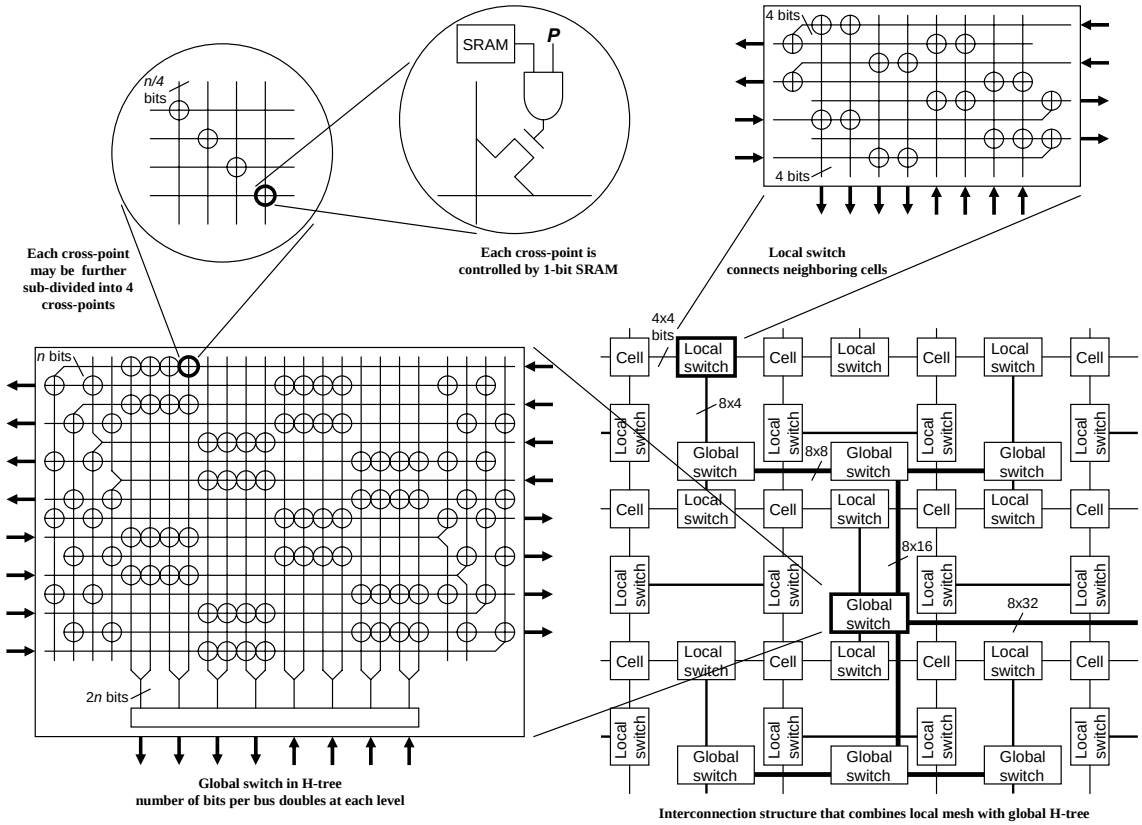


Figure 5.1 Global and local interconnect switches in cell array

5.1 Scheme Description

Figure 5.2 shows the hierarchical bandwidth nature of the global H-tree network. The bandwidths shown include only the downstream busses. As configuration data will only be communicated in a top-down manner, only the downstream bandwidths (i.e. half the total bandwidths) are utilized for configuration purposes. Configuration data will be sent in words of 4x64 (or 256) bits wide each clock cycle. As such, each word is sufficient to provide data for a sub-tree of 32 cells depicted by the shaded triangular area. Data will be channeled to this sub-tree until all the cells and interconnect switches within are configured before moving on to the next sub-tree. Naturally, all the global switches above the cells will have to be connected in the manner shown in Figure 5.3. We call this the *default* (switch) *connection*. The *default connection* ensures that switches above the cells pass the configuration data all the way through to the targeted components.

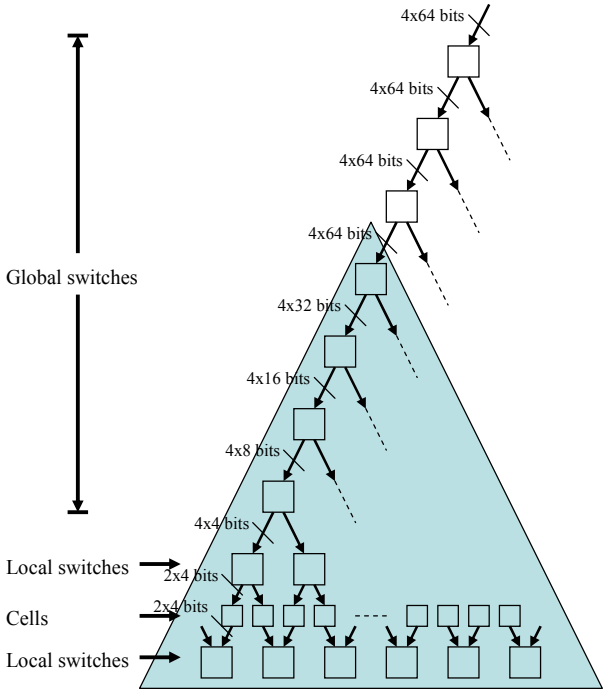


Figure 5.2 Hierarchical view of the H-tree

5.1.1 Configuring by layers

The reconfigurable architecture is made of several components that require configuration, namely:

- reconfigurable elements or cells
- local interconnect switches
- global interconnect switches

In addition, each cell can be further divided into two reconfigurable components, which are the cell processing core, and the input/output internal switches. From Figure 5.1 and 5.2, it can be observed that half of the local interconnect switches form the lowest layer in the hierarchy. As configuration data is communicated in a top-down manner, components at the lowest layer will naturally have to be configured first. Configuration is executed in layers, moving up along the hierarchy. Therefore, in a fresh configuration or a complete reconfiguration, the sequence will be:

1. local switches (those that are not directly connected to the global H-tree)
2. cell processing core
3. cell internal switches
4. local switches
5. global switches

After its configuration, each layer will be *closed off* to further configuration signals while the system configures components of the next higher layer.

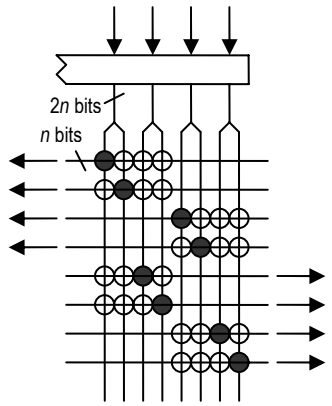


Figure 5.3 Default global switch connections during configuration

5.1.2 Global control signals

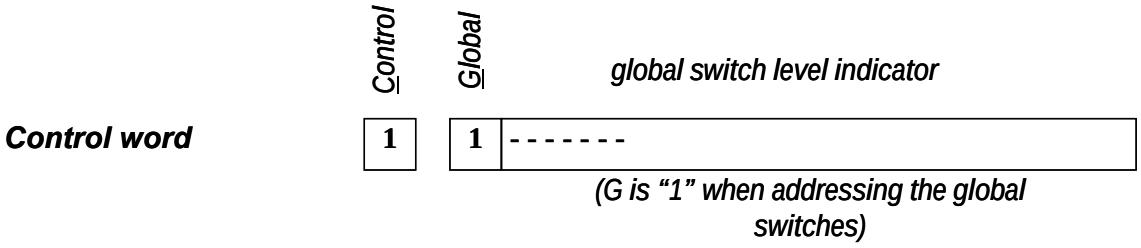
To perform configuration of the hardware components in hierarchical layers, a mechanism is needed to tell the targeted components to receive/store the incoming data, and the other components above them to pass the data through. This also means that each component recognizes if the next data word that is arriving is meant to be stored in its SRAM's or to be rerouted to the next layer of components. To facilitate this, control signals can be sent concurrently or prior to the configuration data. To send the control signals concurrent to the data, additional bus lines will be required. However, this means additional hardware to an already crowded architecture. Alternatively, some of the data bit lines can be used to serve the control signals. However, this leads to longer configuration cycles as less data will be conveyed each cycle. Yet another alternative is to use the data bus for a control word one cycle before sending the configuration data. The purpose of this control word is to signal the targeted layer of components to be ready to store the subsequent data words. We chose this option because its implementation requires minimal additional hardware and configuration speed is not compromised.

To minimize the number of wires in the architecture, we aim to achieve the necessary data control with minimal number of global signals. There are, however, two global signals that this scheme requires. The first, and more obvious, is a 1-bit signal known as *Control* (from here onwards designated as *C*). The signal *C* indicates whether the busses are carrying *control word* or *data word*. A control word is carried in the data bus when *C* is exerted with a true value, otherwise a data word (or configuration data, to be more specific) is carried on the bus. The second global signal is known as *Programming mode* signal (designated as *P*). *P* is responsible for the *default* global switch connections mentioned earlier in this section. When *P* is exerted, the switches will revert to their *default connection* settings. As the communication network is highly pipelined, the global signals can also be channeled downstream in a pipelined manner along with the corresponding data bus. More on the use of these signals will be discussed in the next few sections.

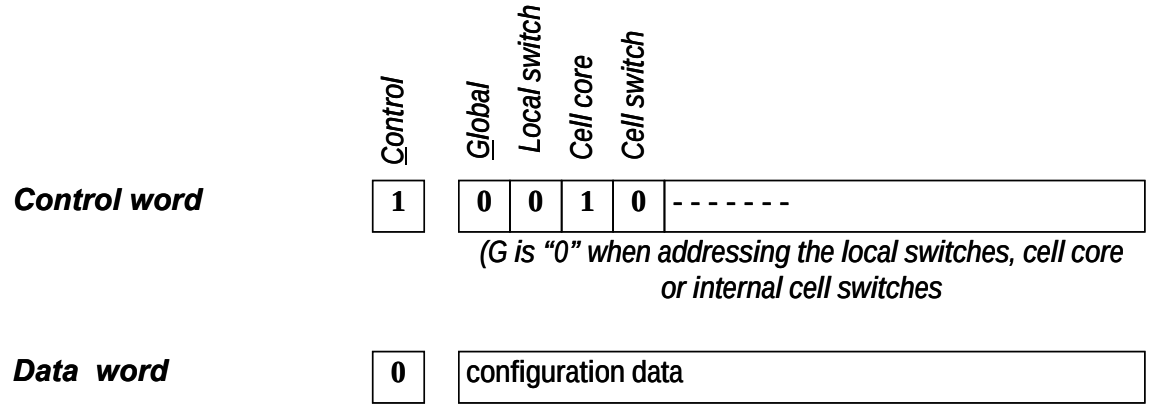
5.1.3 Configuration words

In order to utilize the available interconnects without adding extra wires to the architecture, the global H-tree busses are used to transmit both control words and configuration data. This is achieved with the help of the two global signals *P* and *C*. *P* is exerted to set the global switches in a *default connection* to pass configuration data all the way to the lowest layer of components. *C* is exerted whenever a control word is to be communicated to a particular layer. All configuration data will always be preceded by a control word. The *control word* comes in two flavors. The first is used to communicate

with the global interconnect switches. This is done by exerting the *Global* bit (*G*) to a true value:

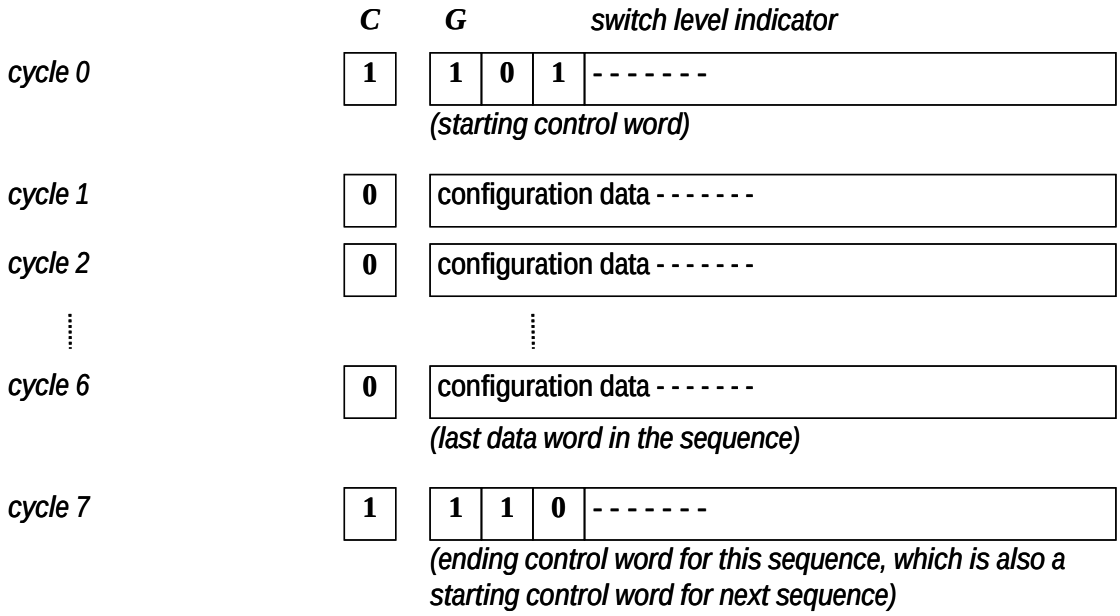


The second is used to communicate with the local switches and cell components. This is done by exerting the *Global* bit to a false value:



Configuration data is sent in *data words* following the *control word*. If the configuration bits required for a particular component exceed the bus bandwidth, multiple data words may be sent before the next control word. The configuration sequence for a particular component (or a layer of components) is terminated by a control word that closes the data gates leading to that component (details on this in section III). A terminating control word can also carry information for the next component (or layer of

components) to be *opened* for configuration. A typical sequence of configuration words will thus look like the following:



5.1.4 Decoding the control word

Figure 5.4 illustrates an example of the logic gates used to decode the control word for a cell core. The signal *C* is exerted to allow writing to a 1-bit latch that controls a series of transmission gates which in turn determines the flow of data into the component (i.e. the cell core in this example). One bit in the control word is assigned to the cell cores. When this bit is exerted to a true value, together with a true value to *C* and a false value to *G* (since the cell core is not part of the *Global* switches), a true value is written to the 1-bit latch. This latch opens the gates for the data bus, making the cell core ready to receive configuration data in the next clock cycle. In the following clock cycle, the signal *C* will be exerted to a false value to indicate that the data bus is carrying a data word instead of a control word. It will disable writing to the 1-bit latch. The data bus,

now carrying a data word, will flow freely into the cell core for every subsequent clock cycle until the arrival of the next control word.

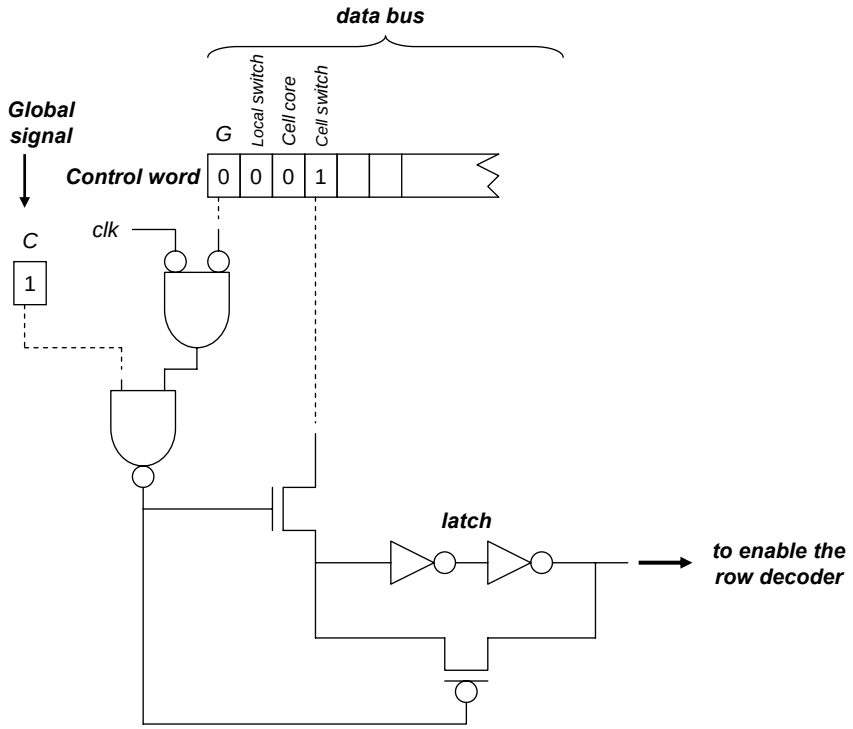
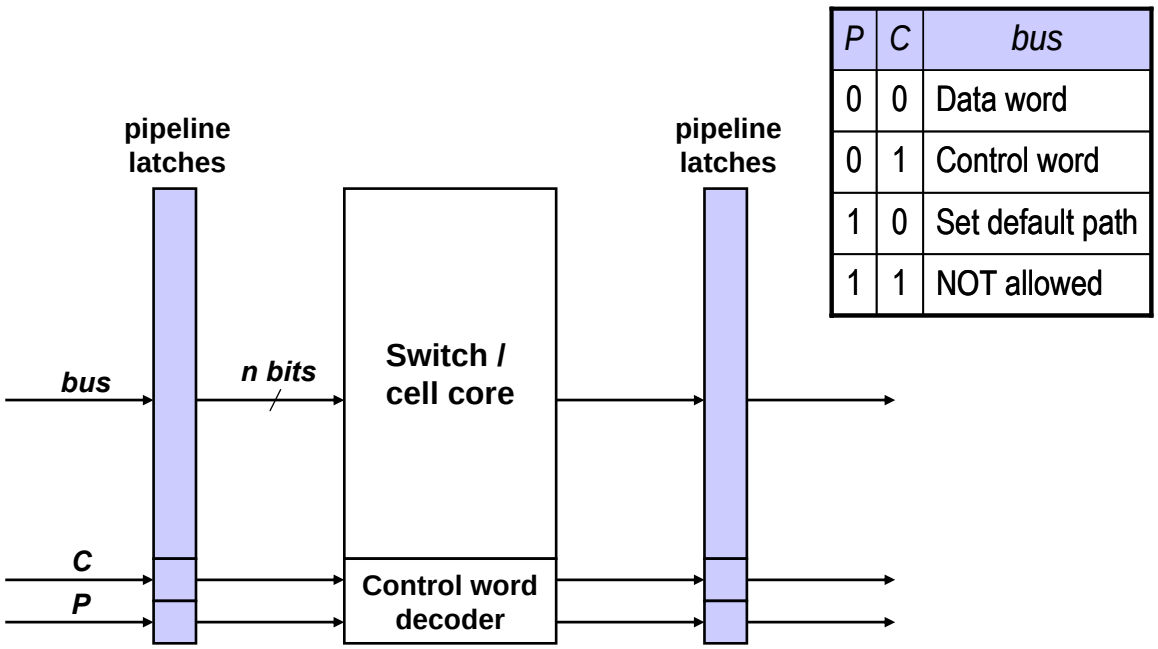


Figure 5.4 Control word decoder for the cell cores

At the end of the configuration cycles for the cell cores, an ending control word is sent to write a false value to the 1-bit latch. This control word closes the data gates into the cell core and can be used to open the data gates of the components in the next upper layer above the cell cores (which are the cell's internal I/O switches). Similar control word decoding is employed throughout all the components in the hierarchy. Figure 5.5 illustrates a pipeline stage and how signals *P* and *C* determine the type of word carried in the data bus.



P	C	bus
0	0	Data word
0	1	Control word
1	0	Set default path
1	1	NOT allowed

Figure 5.5 Pipeline stage and the functions of the global signals

5.2 Programmable Architecture

The routing between the logic blocks is typically controlled by programmable bits. Each of these programmable bits controls a passgate (or a group of passgates in the case of a bus) that determines whether a signal flow from one wire to another.

5.2.1 The programmable bit

We begin by looking for an appropriate SRAM design to store the programmable routing bits. We chose the SRAM as our main storage device mainly for its low power consumption. Figure 5.6 shows the two designs that we narrowed down to. Both designs use a total of 6 transistors. Figure 5.6(a) depicts a more typical SRAM with a

transmission gate feedback to assist in getting both strong '1' and strong '0' at the input to the double inverters. Figure 5.6(b) illustrates a modification to the SRAM where the second inverter's rail voltages are cut off during a write session. This is done so as to minimize the load on the incoming data signal. Additionally, with the *ground* and *vdd* supplies removed during a write, there is minimal power drainage during a transition between a '1' and a '0' in the stored bit. As a result, we found (b) to have 10% lower power consumption than (a) during our simulations.

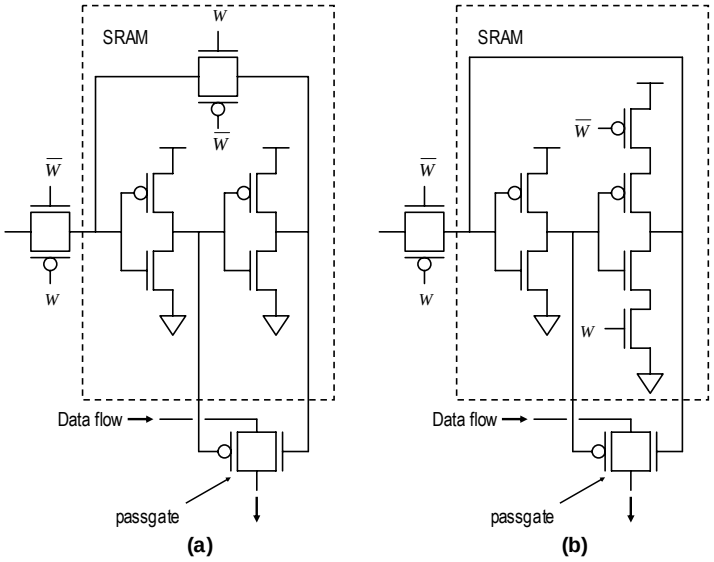


Figure 5.6 SRAM designs: (a) with transmission gate on the feedback, (b) without gate on the feedback

5.2.2 Programming the switch

One of the methods to reduce configuration time is through compression of the configuration data. Lower number of configuration bits means less configuration cycles. The number of programmable bits per switch depends on the switch function itself and its size. Each global interconnect switch, for example, has 96 cross-points, which translate

to 96 programmable bits. The local switches have 20 bits each, and the cell internal I/O switches have 64 bits each. The arrangements of the cross-points for the global and local switches are depicted in Figure 5.1. The cell internal I/O switches are made up of the full 8x8 crossbars.

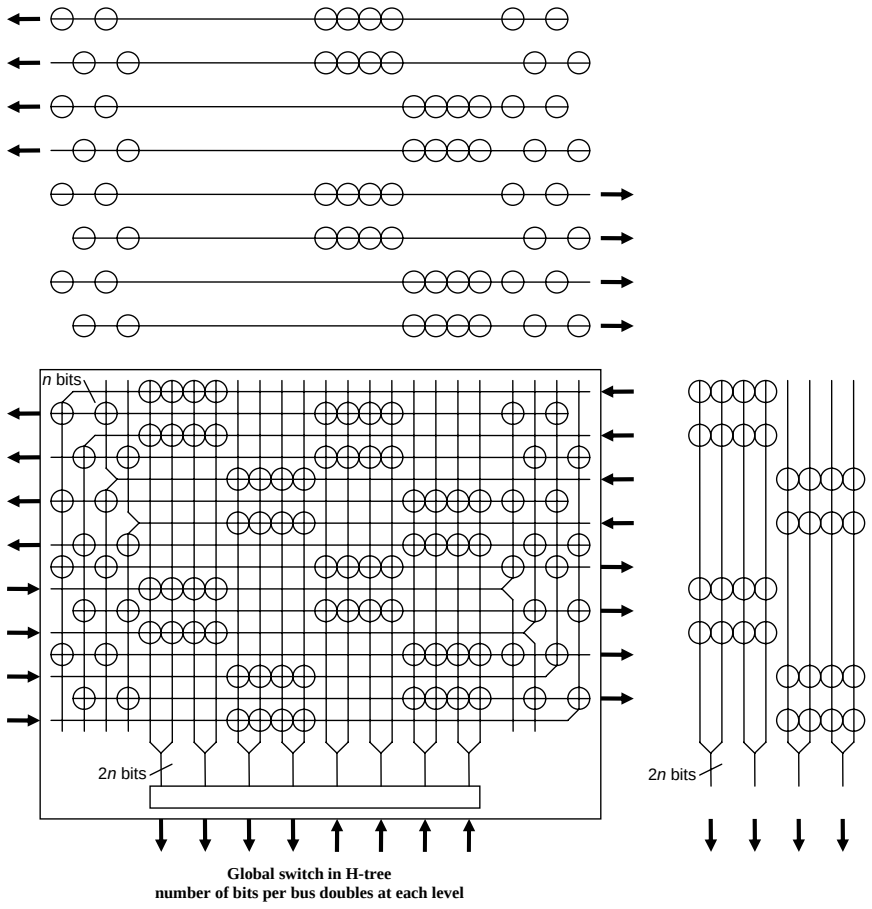


Figure 5.7 The output rows and columns of a global switch

As in any switch crossbars, one input line can be connected to multiple outputs, but each output row (or column) can only have one connected cross-point. Figure 5.7 shows an example of our global interconnect switch. Its output rows and columns are laid out on the sides to illustrate the number of cross-points per output row or column

more clearly. As only one cross-point per output line can be active at any one time, a column/row decoder can be used to compress the configuration data.

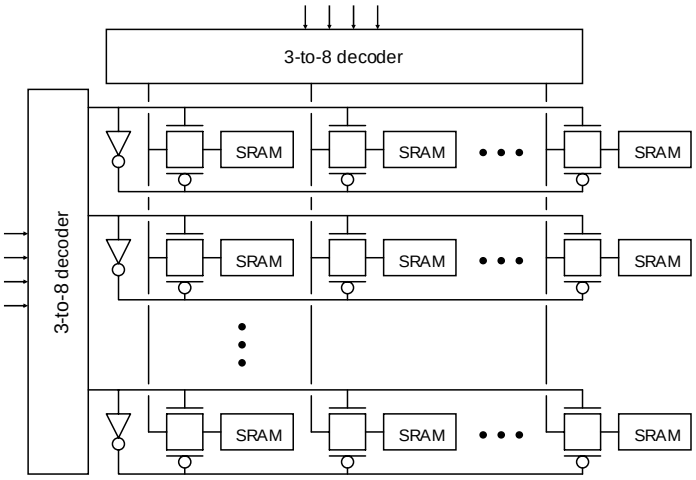


Figure 5.8 Decoding the configuration data

Figure 5.8 illustrates the use of the decoders on an 8x8 programmable bits of the crossbar for a cell internal I/O switch. The row decoder determines the specific output row to be written. The column decoder determines the cross-point to connect an input wire to an output wire. Only three bits per decoder are required for eight rows or columns. So in each clock cycle, a 6-bit data word can be used to configure an output row. However, at the cell layer of the H-tree interconnection network, each cell receives 8-bits of input data bus. Therefore, an extra bit is used for the row decoder to select all rows at once for writing. An extra bit is also used for the column decoder to write a '0' (false value) to all the SRAM's in a row. This is useful when we need to deactivate all the connections in the switch, which now can be done in merely one clock cycle with the extra bits. Turning off all connections in a switch is useful for setting up the *default* connections.

As mentioned in earlier sections, each switch above a targeted layer of components will revert to the *default* connection to allow configuration data to flow through to the next lower level in the H-tree. During the *default* connection, most of the cross-points in a switch will be deactivated except for the few that pass data directly in a top-down manner. For example, in the cell internal I/O switch, only two cross-points are to be activated to transmit 8-bit data (on two 4-bit buses) to the cell core and below. One way to do this is to deactivate all the connections in the switch, followed by two more cycles to configure the two required cross-points to allow necessary data flow. However, by trading off a minimal amount of space, we find that we can achieve the *default* connections in just one clock cycle with the help of the global *P* signal. Figure 5.9 depicts an 8x8 crossbar with two additional cross-points for the *default* connection.

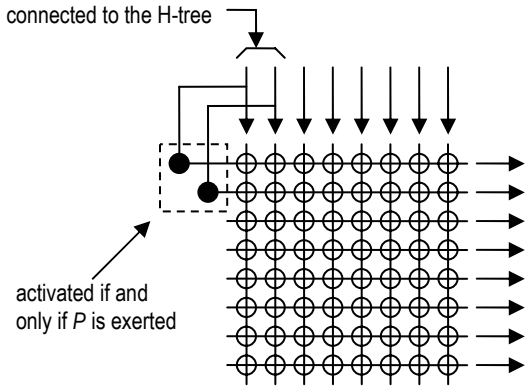


Figure 5.9 An 8x8 switch crossbar with additional cross-points for *default connection*

When *P* is exerted, a signal will be sent to the row and column decoders to deactivate all the connections in the switch. At the same time, the signal *P* is used to activate the extra cross-points to generate the default connections.

5.2.3 Partial configuration

In some cases, configurations do not occupy the entire reconfigurable array. Yet in other instances, only a part of the configuration requires modifications. In these situations, a partial reconfiguration of the array will suffice, and will cost less downtime as opposed to a full reconfiguration. The hierarchical nature of our system allow for configurations in clusters of cells or switches. When necessary, the tree structure also allows for configuring interconnection switches that are at the higher levels without disturbing the lower level switches and logic blocks. We found this useful in large applications like the Fast Fourier Transform (FFT) which is often executed in a DSP operation.

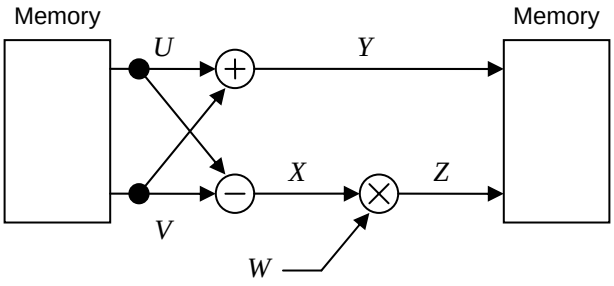


Figure 5.10 Kernel of decimation-in-frequency FFT

For an example, we take an illustration from [16]. Figure 5.10 shows the kernel of the classic decimation-in-frequency FFT. This operation includes an adder, a subtracter, and a multiplier. Initially, the input data is loaded into the memory on the left. Each pair of data is then processed by the butterfly stage, and the result is stored in the memory on the right. The two memories are then reversed and the process is repeated. Such an operation requires the updating of only a portion of a mapped circuit, while the

rest should remain intact. Specifically, only the highest level connections from the logic blocks leading to and from the memories require rerouting. The output memory that stores the results from the previous calculation is switched to be the input memory for the current process. The initial input memory can be reused for storage of results from the current calculation. The cells and lower interconnection network that forms the functional logic blocks (i.e. adder, subtracter, and multiplier) remain intact.

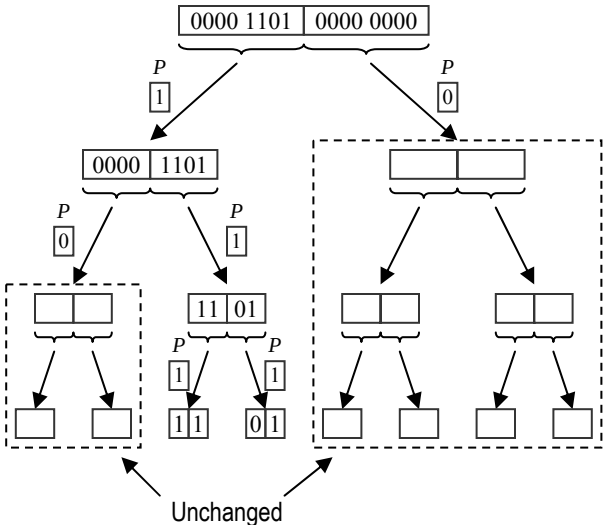


Figure 5.11 Partial *default connections* using signal *P*

To achieve partial reconfiguration, we capitalize on the global signal *P* (programming mode) and the highly pipelined structure of the system. As mentioned previously, the signal *P* is exerted to impose the *default connection* on the interconnection switches to allow the flow of configuration data in a top-down manner. When *P* is exerted, the routing configuration of a switch is wiped off to accommodate the *default connection*. The affected switch will require a reconfiguration even when it did

not need a new configuration. Therefore, instead of a fully global *default connection* the signal *P* can be accompanied by its own data bits, and be trickled down the H-tree just like the signal *C* (which differentiates a *control word* from a *data word* being carried on the data bus). This way, only portions of the array that require modifications will receive the signal *P*, while the rest of the array remains unchanged.

Figure 5.11 illustrates a simplified flow of the signal *P* with its accompanying data bits. Each pipeline stage represents a level in the global interconnect hierarchy. By ORing the corresponding half of the data bits at each level, a decision is made as to whether the signal *P* is to be passed on to the next level. As a result, sub-trees that do not require modification do not receive the signal *P*, and the components within retain their configurations.

Chapter 6

Implementation & Simulations

This chapter describes the circuit level design of an interconnect switch. The interconnect switches are made up of the global H-tree switches, the local mesh switches, and the cell internal I/O switches. Together they control the routing of input, output and intermediate data within the cell array. Although the three different types of switches have different sizes and serve different levels of data communication within the array, their basic control structure and configuration storage mechanism remain similar. In the following sections, we thus use only the cell internal I/O switch for the purpose of illustration and functional verifications.

6.1 Switch Circuit Design

In the previous chapters, we pointed out that each data bus line output on a routing switch can only be connected to a single input at any one time. In order to compress the configuration data, the use of column and row decoders match this purpose.

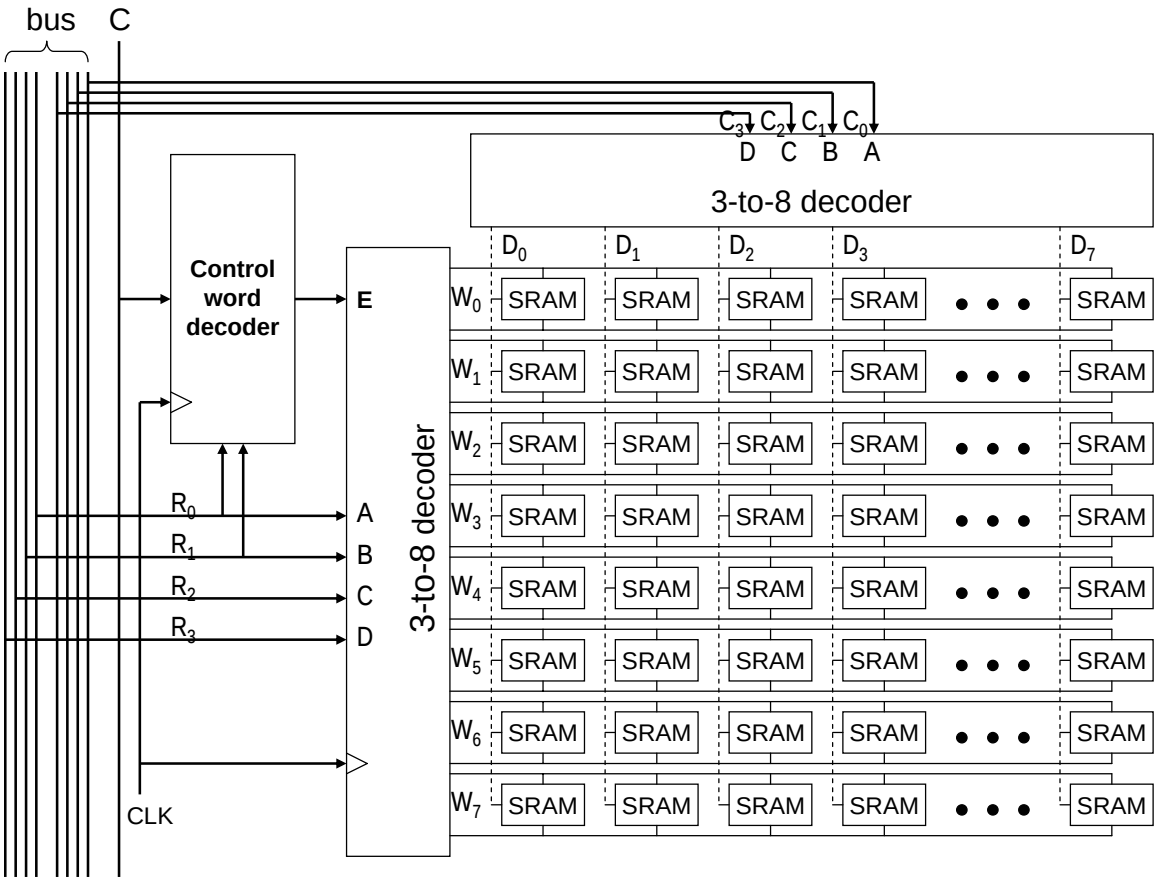


Figure 6.1 Cell internal I/O switch

Figure 6.1 shows the circuit of an 8x8 cell internal I/O switch. In this case, both the column and row decoders are 3-to-8 decoders. Although only three input bits are needed for eight outputs, an additional bit is added to each decoder to serve a special purpose in either turning all the outputs high or all outputs low. This feature allows writing of 0's (turning off the connections) to all the SRAMs in a single clock cycle. It has the advantage of clearing all the connections in the switch prior to setting up the *default connection* that was described in the previous chapter.

6.1.1 Control word decoder

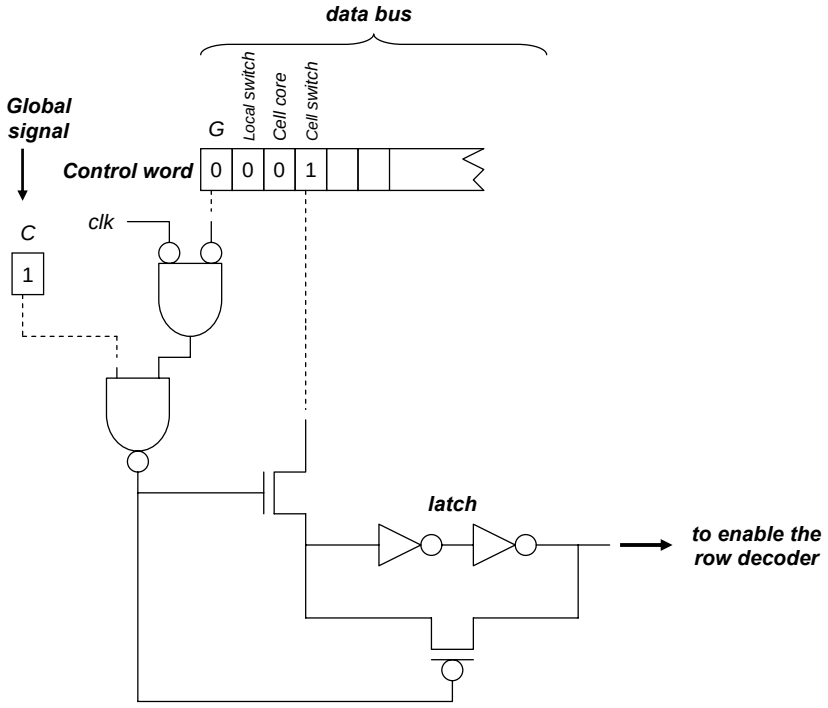


Figure 6.2 Control word decoder

The control word decoder (Figure 6.2) contains a single latch that holds a true value when the switch is selected for writing configuration data. This latch is written when the right combination of global signal **C** and the respective bits in the bus that correspond to this particular switch are exerted. In this case, the signal **G** corresponds to bit $\mathbf{R_0}$ and the cell internal switch is designated by bit $\mathbf{R_1}$ in Figure 6.1. When written with a true value, this latch keeps the row-decoder enabled throughout the configuration cycles. At the end of the configuration sequence for this particular switch, another control word will be expected in order to deactivate the row-decoder by writing a false value to the control latch.

6.1.2 Column decoder

D	C	B	A	Out
0	0	0	0	D0
0	0	0	1	D1
0	0	1	0	D2
0	0	1	1	D3
0	1	0	0	D4
0	1	0	1	D5
0	1	1	0	D6
0	1	1	1	D7
1	X	X	X	Off

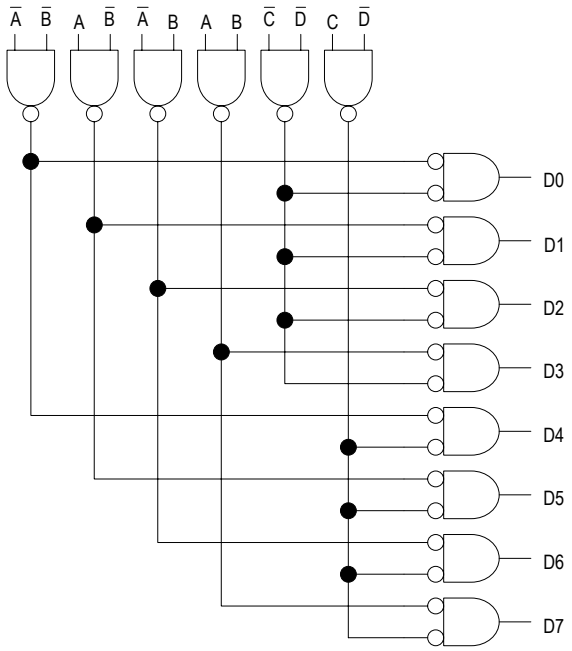


Figure 6.3 Column decoder

Figure 6.3 shows the schematic of the column decoder next to its truth table. The signal **D** is used to generate 0s on all the decoder outputs. When **D** is not exerted, the decoder functions as a normal 3-to-8 decoder.

6.1.3 Row decoder

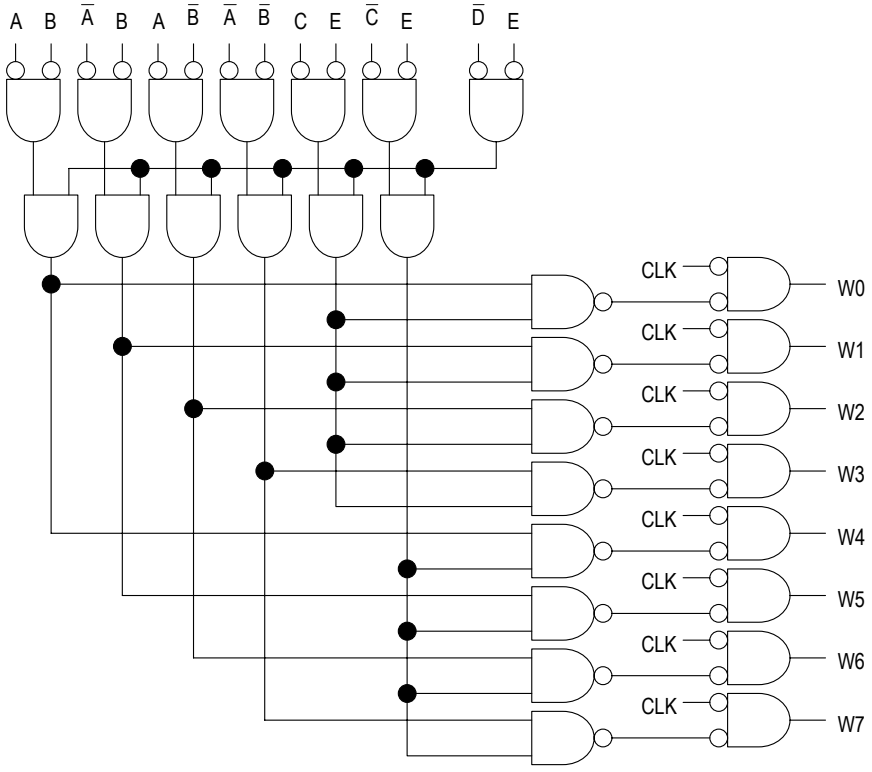


Figure 6.4 Row decoder

The row-decoder contains more features, which in turn make it significantly bulkier and slower, than the column decoder. Instead of turning off all the outputs, the signal **D** exerts 1s on all the outputs. This means that all rows are open for writing when **D** is exerted. Combined with the all 0s exerted by the column-decoder, this feature allows clearing all the SRAMs in one clock cycle. The row-decoder also comes with an enable signal **E** to enable/disable configuration of the switch. Clock is also needed in the row-decoder to synchronize the writing cycles to the valid periods of the data to be written to the SRAMs.

6.1.4 SRAM

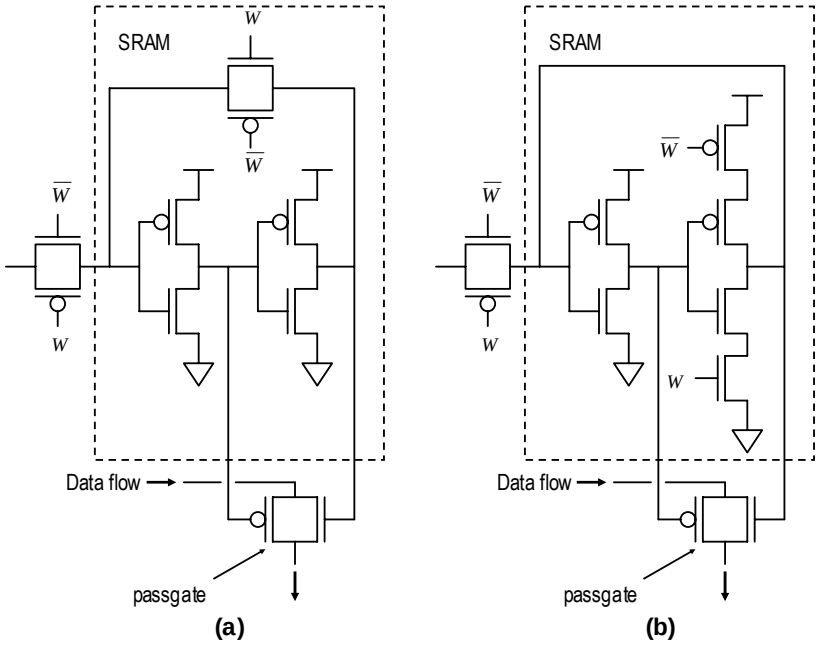


Figure 6.5 SRAM designs: (a) with transmission gate on the feedback, (b) without gate on the feedback

We chose the SRAM as our main storage device mainly for its low power consumption. Fig. 6.5 shows the two designs that we narrowed down to. Both designs use a total of 6 transistors per SRAM. Fig. 6.5(a) depicts a more typical SRAM with a transmission gate feedback to assist in getting both strong '1' and strong '0' at the input to the double inverters. Fig. 6.5(b) illustrates a modification to the SRAM where the second inverter's rail voltages are cut off during a write session. This is done so as to minimize the load on the incoming data signal. Additionally, with the *ground* and *vdd* supplies removed during a write, there is minimal power drainage during a transition between a '1' and a '0' in the stored bit. As a result, we found 6.5(b) to have 10% lower power consumption than 6.5(a) during our simulations.

6.2 Simulations

Ideally, we want the respective decoder signals to be generated according to the timing diagram shown in Figure 6.6. Data is assumed to transmit to the switch on the rising-edge of the clock after an estimated delay of 100 ps. We realize, from simulations that the column-decoder imposes a delay of 240 ps from the clock edge. The row-decoder, on the other hand, generates write signals at the falling edge of the clock. The row-decoder (having more features) imposes a delay of 315 ps, which is considerably longer than the delay through the column-decoder. The write signal, however, has to be active only during the valid part of the data signal (generated by the column-decoder) to be written to the SRAMs. Therefore, modifications were needed on the row-decoder to make it more sensitive to the rising edge of the clock to mark the end of its write signal.

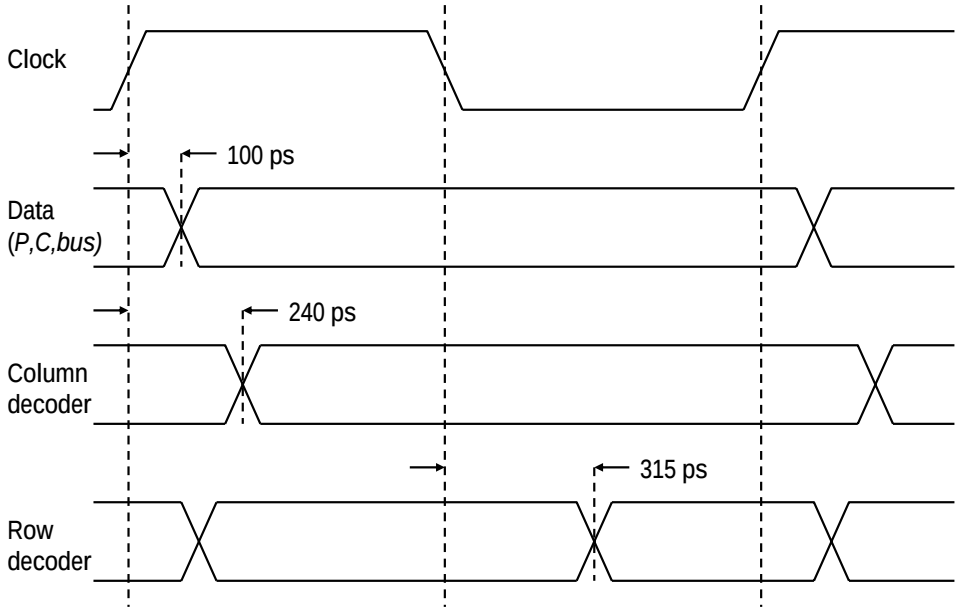


Figure 6.6 The timing diagram

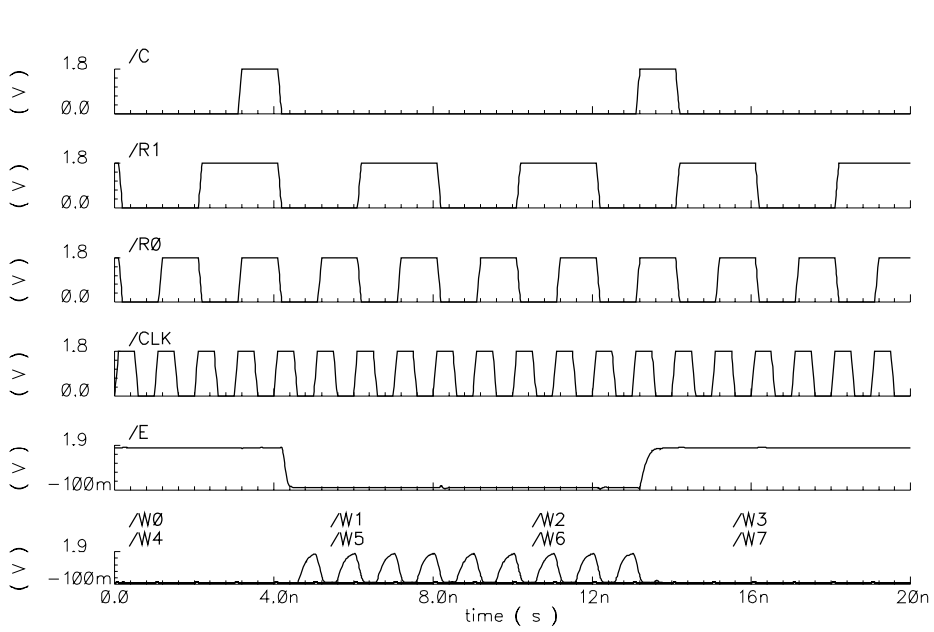


Figure 6.7 Functional verification of a switch being selected for configuration

To start off the simulations section, we show in Figure 6.7 the functional verification of a switch that is selected for configuration. With reference to Figure 6.1, the switch is selected for writing incoming configuration data when signals **C, R_0** and **R_1** are exerted with a true value (in this case high voltage). This in turn generates an enable signal **E** (active low) that enables the row-decoder. When enabled, the row-decoder generates the write signals **W_0** to **W_7** one at a time each clock cycle. Each **W** is responsible for writing the corresponding data signal **D** (from the column decoder, not shown in the figure) into its designated row of SRAMs. The switch is 'open' for writing until the end of its configuration cycles, i.e. when **R_1** = 0 while **C** = 1 and **R_0** = 1.

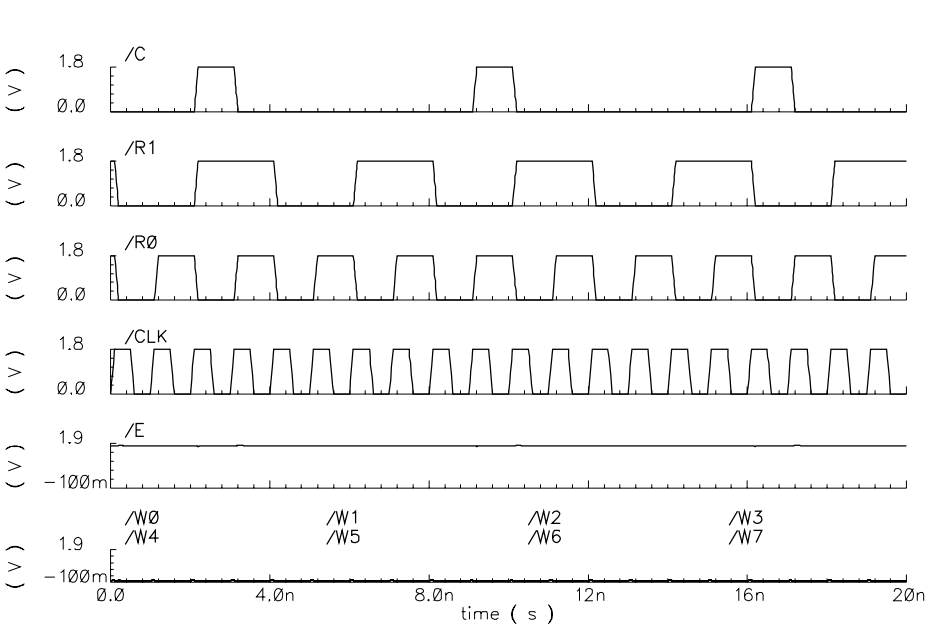


Figure 6.8 Functional verification of a switch that is NOT selected

For verification, we also show in Figure 6.8 a simulation where the switch is not being selected for configuration. The other three combinations of $\mathbf{R_0}$ and $\mathbf{R_1}$ when **C** is exerted true are shown here. The enable signal E is not enabled on all three conditions, and therefore no write signals are generated by the row-decoder. Thus no new data is written and the existing data in the SRAMs remain unchanged.

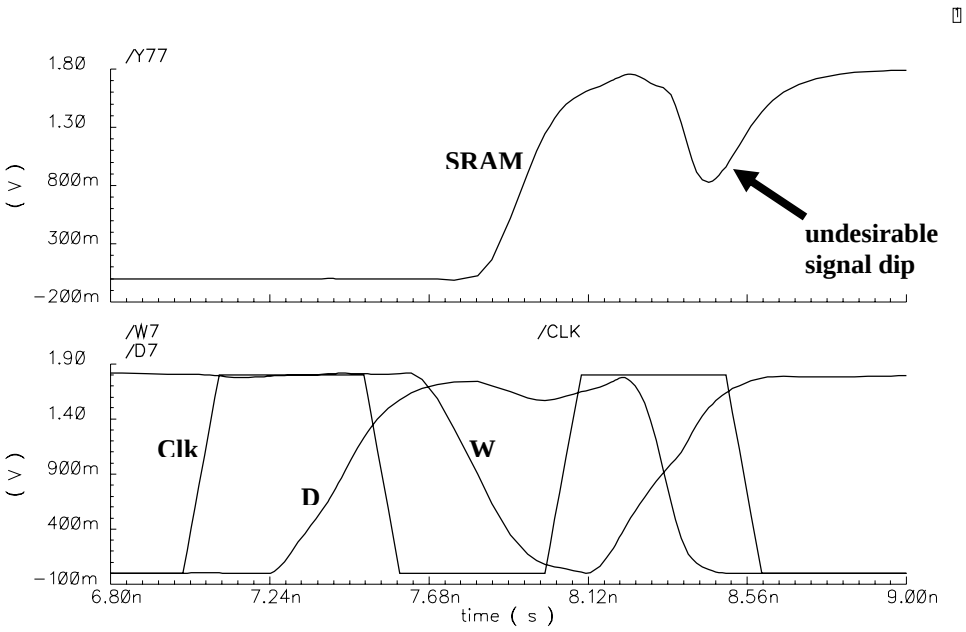


Figure 6.9 Initial Simulation

Using 0.18µ CMOS technology, we managed to achieve a stable configuration speed of up to 1.25 GHz. However, we maintained a speed of 1 GHz for the rest of the simulations for more assured functional verifications. Our initial simulation of the switch (Figure 6.9) indicated that the write signal, **W** (generated by the row decoder), was indeed too slow compared to the data signal, **D** (by the column decoder). In this simulation we set data arriving from outside the switch with a delay of 0.1 ns. Ideally, we want **W** to be deactivated before **D** changes for the subsequent clock cycle. However, due to the longer delay in the row-decoder, **W** deactivates very near to the end of **D**'s valid period, resulting in the undesirable signal dip at the output to the **SRAM**.

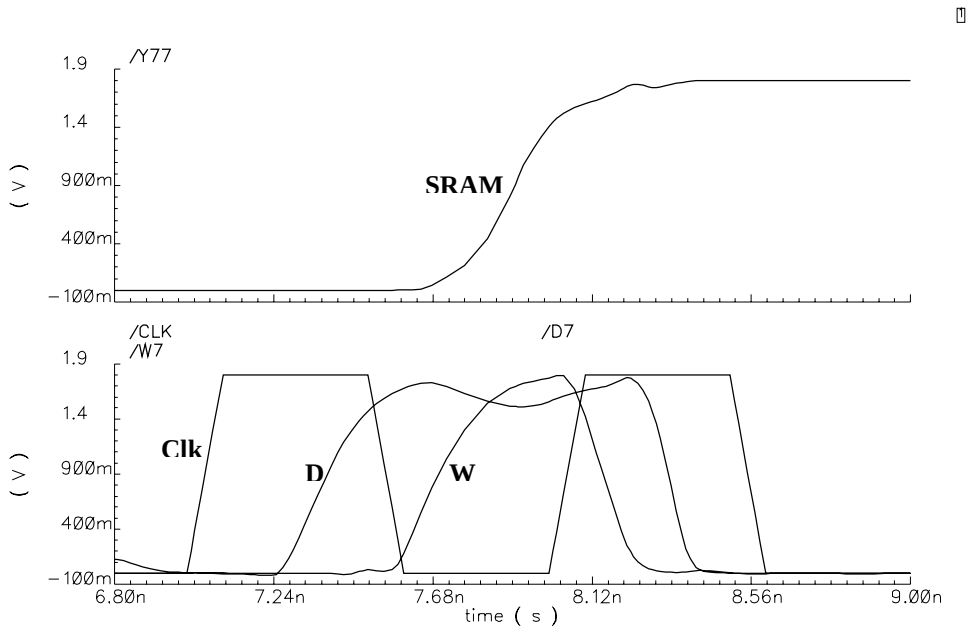


Figure 6.10 Simulation after modification to the row decoder

We saw the need to make the write signal, **W**, more sensitive to the rising-edge of the clock at the end of the write cycle. Therefore, the clock signal was moved to the decoder's gates nearest to its output (see Figure 6.4). That way, when the clock changes from low to high, **W** is deactivated sooner than **D**. Figure 6.10 shows the simulation after the modification to the row decoder. We also note that, since the pull-down of the decoder output is faster than the pull-up, **W**'s logic is reversed such that the write is active high. This further improved the timing of the **W** signal. As can be seen in the figure, the output signal written to the **SRAM** is now significantly cleaner.

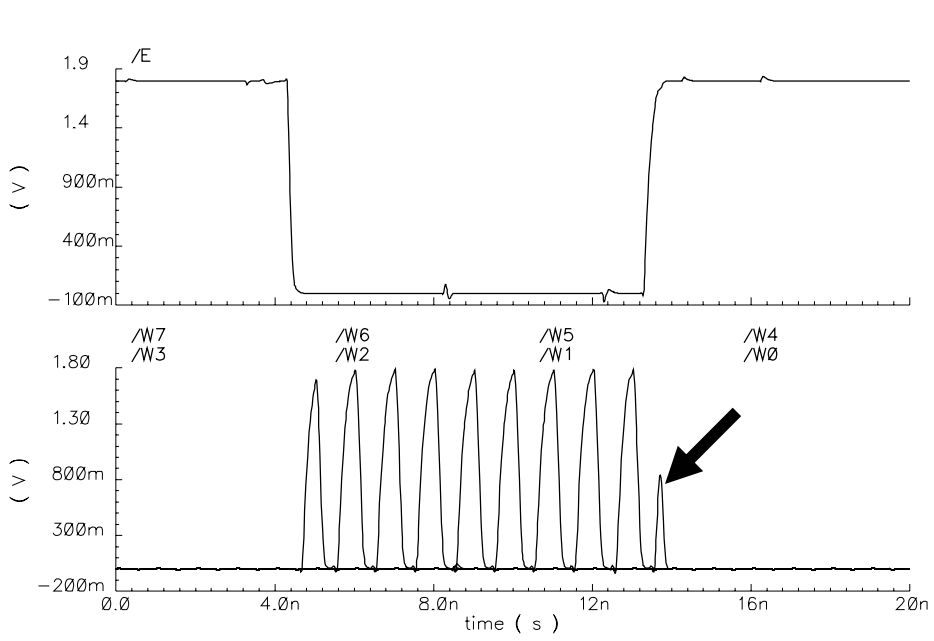


Figure 6.11 Unwanted write signal

The modification, however, did come at a price. Since the rising-edge of the clock closes the last row of gates inside the row-decoder, it also opens these gates to signal changes at the falling-edge of the clock. Figure 6.11 shows the undesired effect on the write signal **W**. The switch is supposed to be opened only for nine write cycles. However, the enable signal **E** travels through more layers of gates than the clock does in the row-decoder. Therefore while it is deactivating the decoder, the clock signal already allows some changes in the data lines to go through.

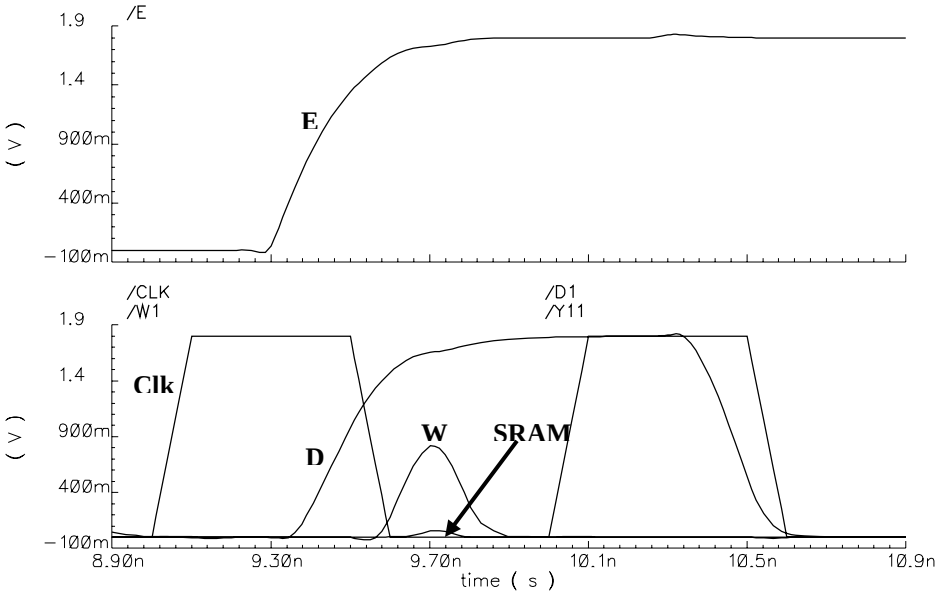


Figure 6.12 A closer look at the unwanted write signal

Figure 6.12 shows a closer look at the unwanted write signal and its effect on the output to the **SRAM**. Before the enable signal **E** (active low) fully deactivates the row-decoder, the falling-edge of the clock allows some residual **W** signal to go through. Although the resulting **W** signal did not alter the data in the SRAM, it did reach an uncomfortable level of around 0.9 V. Therefore, a further improvement is needed in the row-decoder.

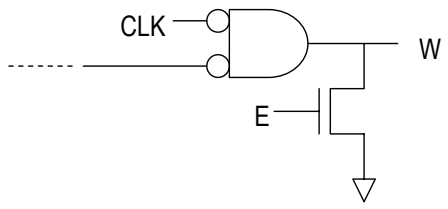


Figure 6.13 Additional transistor to remove unwanted W signal

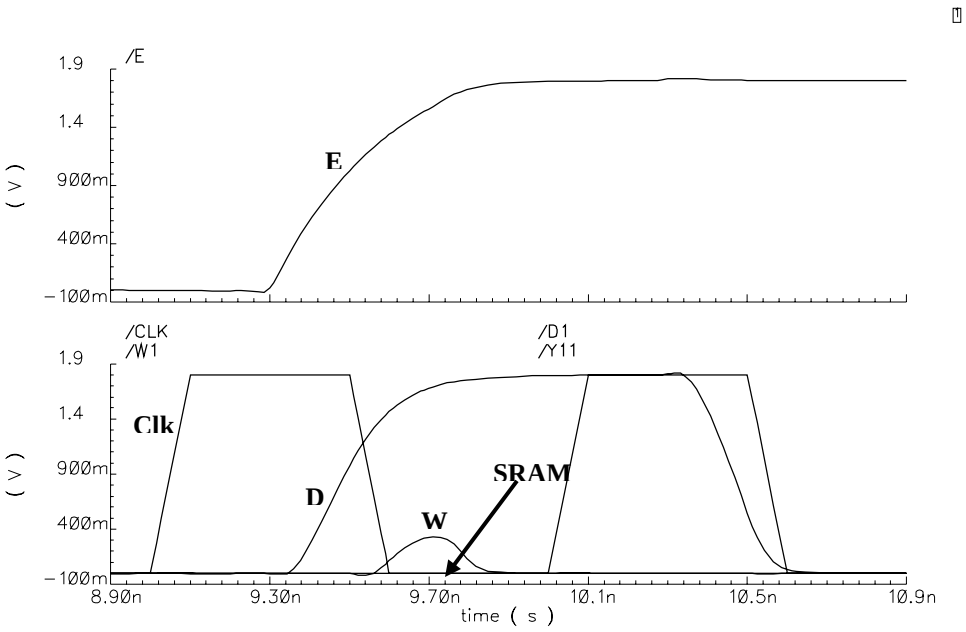


Figure 6.14 Minimizing the unwanted write signal

To minimize the unwanted write signal at the end of the configuration cycle, an additional transistor is attached to every output of the row-decoder (as shown in Figure 6.13). The purpose of this transistor is to ground the row-decoder outputs as soon as possible when the switch reaches the end of its configuration cycles. Since the enable signal **E** is active low, when it deactivates at high voltage, all the outputs of the row-decoder will be grounded. The consequence of this is a slight increase of less than 2% of power consumption during a very active configuration sequence. However, it gives us the assurance of not falsely altering the stored configuration data. Figure 6.14 shows the minimized unwanted **W** signal when **E** is deactivated, and how it no longer affects the SRAM's value.

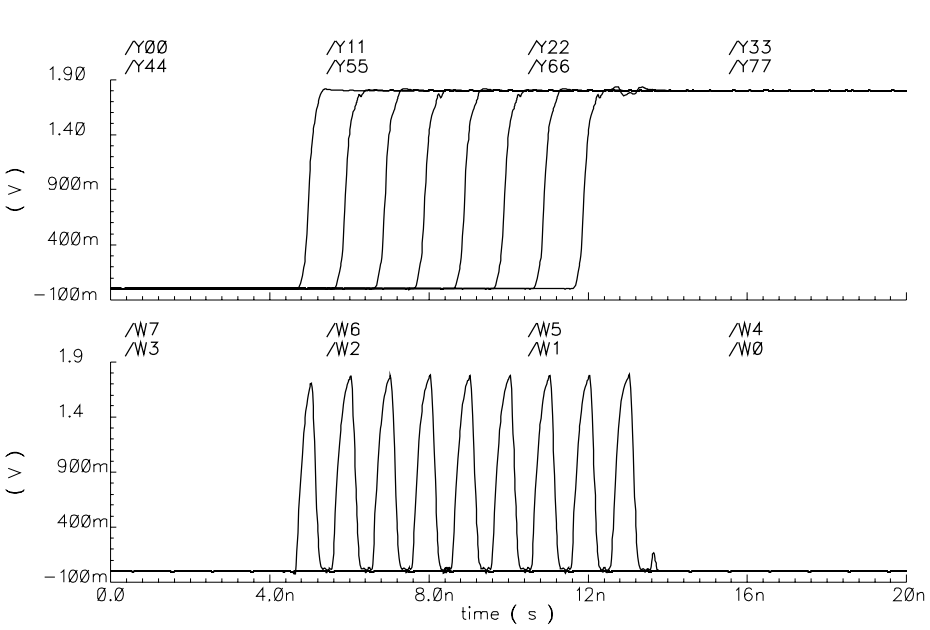


Figure 6.15 Write signals and their respective outputs at the SRAM's

To conclude the simulations section, we show in Figure 6.15 the write signals and their corresponding SRAM outputs during a typical write sequence at a configuration speed of 1 GHz. As can be seen, the unwanted write signal at the end of the cycle has been reduced to approximately 0.3 V maximum. It is not completely removed, but it no longer poses a risk of falsely changing the configuration data at the SRAMs.

Chapter 7

Performance Comparison

There is not a straightforward method of comparison between the performance of different reconfigurable architectures. Despite some works that have been done, like Dehon's in [32] and the *remanence* in [33], a concrete metric for comparison remains elusive. We thus do comparison based on two simple, and yet much sought after factors, in configuration/reconfiguration, i.e. speed and design complexity. This section analyzes the number of configuration bits in a 32x32 cell array, and estimate the number of clock cycles involved using the proposed configuration scheme. We then compare these criteria with a few existing reconfigurable systems currently available in the market.

7.1 Configuration Bits

To analyze the number of clock cycles required for a full configuration, we first total up the number of configuration bits in a 32x32 array of reconfigurable cells.

TABLE 7.1

Number of configuration bits in a 32x32 cell array

Component	Quantity	Bits per component	Configuration bits
Cell cores	1,024	512	524,288
Internal switches	1,024	128	131,072
Local switches	1,984	20	39,680
Global switches	511	96	49,056
		Total	744,096

Table 7.1 shows how the cell cores make up a majority of the configuration bits required in a full configuration. Furthermore, the cores are found in one of the lowest levels in the H-tree network. Reconfiguring the cores would mean reconfiguration of all the switches above them. As such, it motivates us to perform minimal reconfiguration of the cell cores after the initial full configuration. Partial reconfiguration will play an important role in only programming portions of the array while leaving reusable mapped circuits unchanged for the subsequent operations.

7.2 Configuration Cycles

In evaluating the number of configuration cycles, we divide the analysis into two scenarios. The first is the case where the configuration data is not cached within the system. The second assumes a cached memory embedded near the reconfigurable array. Let us first evaluate the first case. Assuming an 8-bit connection to outside the array, the system loads 8-bit configuration words onto the global interconnection lines. One

configuration word is transmitted every clock cycle. In a full configuration, it takes 64 cycles to fill the 64x8 bit memory in the processing cell core. The two internal cell I/O switches each takes 8 cycles to program. An additional 3 cycles are needed for the control words. In total, it takes 64 + 8 + 8 +3 which equals to 83 cycles to program a cell. The estimation of total configuration cycles for a 32x32 array is summarized in Table 7.2.

TABLE 7.2

Number of configuration cycles for a 32x32 array
(assuming 8-bit loading per clock cycle)

Component	Quantity	Cycles per component	Config. cycles
Cell cores	1,024	64 + 1	66,560
Internal switches	1,024	2 x (8 + 1)	18,432
Local switches	1,984	3 + 1	7,936
Global switches	511	12 + 1	6,643
		Total	99,571

7.3 Comparison to Other Systems

The configuration time required by the *unicast* scheme is comparable to FPGAs currently available in the market. The most basic Xilinx Virtex-II device contains 338,976 bits of configuration that can be programmed at 50 MHz in their express mode (i.e. 8-bit loading per clock cycle) [34]. A high-level version has 26,194,208 configuration bits and can be programmed at 200 MHz.

TABLE 7.3

Comparison to other reconfigurable architectures
(assuming 8-bit loading per clock cycle)

Device		Config. bits	Config. cycles	Estimated clock speed	Config. time
Unicast (proposed)		744,096	99,571	1 GHz	99.6 µs
Xilinx Virtex-II	XC2V40	338,976	42,372	50 MHz	847.4 µs
	XC2V2000	6,812,960	851,620	100 MHz	8,516.2 µs
	XC2V8000	26,194,208	3,274,276	200 MHz	16,373.4 µs
Xilinx XC4000	XC4013XLA	393,632	49,204	50 MHz	984.1 µs
	XC4062XLA	1,433,864	179,233	50 MHz	3,584.7 µs

Table 7.3 gives an indication of how the configuration cycles of the proposed system compares with those of the Xilinx Virtex-II series and the more matured but popular XC4000 series. We listed the comparison in terms of clock cycles as well as the estimated time because the actual configuration time depends on the clock speed achievable by the respective technology. The XC4000(XLA) devices use 0.35-µm technology and are programmable at 50MHz. The Virtex-II devices use 0.15-µm technology and are programmable at 50-200MHz. In our simulations, the *unicast* scheme was able to achieve a configuration speed of 1 GHz with a modest 0.18-µm technology. This makes our configuration speed compare very favorably against the systems currently available in the market.

TABLE 7.4

Configuration cycles and estimated time for a 32x32 array
(assuming 256-bit loading from internal cached memory)

	Scheme	Additional memory required for configuration	Configuration cycles
Full configuration	Unicast	12,152	3,264
	Broadcast	44,920	912
Partial configuration (only the global switches)	Unicast	n/a	352
	Broadcast	n/a	468

In the second scenario, we assume the presence of cached memory within the reconfigurable system. Both proposed schemes will be able to make use of the entire 256-bit data bus going into the global interconnection network. As such we were able to load configuration data to multiple cells that are on the same level in the H-tree at the same time. Specifically, the 256-bit bus is able to feed data into 32 processing cores (i.e. 8-bit per core) every clock cycle. Table 7.4 summarizes the configuration cycles and the number of additional memory required to support the respective configuration schemes. In the worst case, the *unicast* scheme reduces the number of configuration cycles to 3,264 for a full configuration. In a similar situation, our previous scheme, using the *broadcast switches* described in chapter 4, requires only 912 clock cycles. However, it was achieved at the cost of more memory and complex data path controls within the broadcast switches.

Additionally, when only a partial configuration is required, the *unicast* scheme performs better than the *broadcast switches*. We assume a partial configuration where all the global switches are reconfigured and the rest of the components remain unchanged.

The *unicast* scheme reconfigures the global interconnects in 352 clock cycles, whereas 468 cycles are required by the *broadcast switches*. The difference becomes larger if even fewer of the global switches require reconfiguration. This is understandable because the *broadcast* scheme imposes a fixed overhead in first configuring the *broadcast switches* before using them to configure the array components. In applications where runtime reconfiguration does not occur regularly, the *broadcast* scheme offers shorter configuration time. However, in applications where mapped circuits are often reused, the *unicast* scheme quickly pays off its initial full configuration cost with regular partial reconfigurations, and reduces the overall reconfiguration times.

Chapter 8

Conclusion

In this thesis, two efficient H-tree based configuration schemes for a reconfigurable DSP hardware have been presented. The schemes utilize the existing global interconnection network for communication of configuration data in a top-down manner. Configuration is performed in layers, starting at the lowest level of the H-tree and moving up from there. Data paths are managed by differentiating *control words* from *data words* as they make use of the same communication wires. By reusing the existing interconnection wires, the scheme introduces minimal additional hardware to an already crowded architecture, while completing configuration at a fraction of the time required by other reconfigurable systems of comparable size. By including partial configuration features into the system, the scheme saves even more configuration cycles when mapped circuits are reused, and only the upper interconnection structures require reconfiguration. Transistor level simulations indicate that configuration can be performed at a clock frequency of 1 GHz using a modest 0.18-μm technology.

8.1 Contributions

The schemes described in this thesis are aimed at providing high performance for configuration and reconfiguration of a reconfigurable DSP hardware. The major contributions from the research are summarized in the following:

- **Broadcast based configuration scheme:** By setting the switch buses in a broadcast configuration, decision making and/or data decoding are performed at the top level switches in the H-tree. Thus any necessary hardware modification to implement data decoding needs only be done on few switches near the top of the hierarchical tree instead of a large number of switches toward the leafs of the tree. This reduces not only the amount of extra space required to accommodate the modification, but also the complexity of the circuit itself. The scheme provides a time efficient configuration strategy for a large cell array.
- **Unicast based configuration scheme:** Although the broadcast based configuration scheme achieved high configuration speed, the high number of cross-points required in the broadcast switches greatly increased its design complexity and space requirement. Additionally, the time required to configure the H-tree global interconnect switches remains high due to the highly irregular routing in these switches. As the reconfiguration of the global switches is expected to occur more frequently than that of the cells, the speed advantage in configuring the cells alone become less significant in the long run. The unicast scheme achieves high performance by trading off a longer configuration time of the cells with a shorter configuration time for the global interconnects.

Additionally, it alleviates the complexity of the broadcast switch design by having the same global switch design throughout the array.

- **Configuration word implementation:** To perform configuration of the hardware components in hierarchical layers, each component needs to recognize if the next *data word* that is arriving is meant to be stored in its SRAMs or to be rerouted to the next layer of components. To achieve this, a *control word* system is implemented to coordinate the configuration data flow.
- **Configuration circuit:** The configuration schemes are designed with minimal additional hardware requirement beyond the existing structures. The configuration circuit is responsible for managing the *control words* and *data words*. The control word decoder within each reconfigurable component decides whether a component is selected for configuration based on the incoming *control word*. Data word decoders are optimized to achieve high configuration speed when decompressing configuration data.
- **Partial configuration:** The hierarchical nature of the system allows for configurations in clusters of cells or switches. The tree structure also allows for configuring interconnection switches that are at the higher levels without disturbing the lower level switches and cells. Partial configuration is inherent in this architecture. Its implementation requires only a few additional logic gates at the switches. In cases where mapped circuits are often reused, partial configuration greatly reduces the configuration time in the long run.

8.2 Future Work

The expansion of DSP applications in wireless communication has brought greater demands in devices with low power consumption. While we explored low power SRAM briefly in this thesis, we believe there are other components that can yield more improvements in this area. Switches, decoders, and wiring routes will be scrutinized for lower power requirements in future development.

Another direction for this research will involve merging the *broadcast* and *unicast* based schemes to form a hybrid configuration scheme. The *broadcast* scheme will be used to configure the cell core, cell internal switches and local mesh switches as it achieves the highest performance in these components. The *unicast* scheme will be employed solely for the global interconnect switches. This approach will greatly increase the total configuration speed as we capitalize on the strengths of both schemes. However, it is also expected to significantly increase the complexity and size of the broadcast switches as they must be designed to alternate between the two schemes. The tradeoffs of such a hybrid scheme will be explored.

References

[1] J. McClellan, R. Schafer, and M. Yoder, *DSP First: A Multimedia Approach*, Upper Saddle River, NJ, Prentice Hall, 1998, pp. 373-374.

[2] K. Compton and S. Hauck, "Reconfigurable Computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, Jun 2002, pp. 171-210.

[3] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: a survey," in *Programmable Digital Signal Processors*, Y. Hu, ed., Marcel Dekker Inc., 2001.

[4] N. Dutt and K. Choi, "Configurable processors for embedded computing", *IEEE Computer*, vol. 36, no. 1, Jan 2003, pp. 120-123.

[5] M. A. Wahad and D. J. Puckey, "Reconfigurable DSP systems," in *Proc. IEE Colloquium on Applications Specific Integrated Circuits for Digital Signal Processing*, London, UK, Jun 1993, pp. 3/1-3/6.

[6] R. Hartenstein et al, "Mapping applications onto reconfigurable KressArrays," in *Proc. 9th International Workshop on Field Programmable Logic and Applications*, Glasgow, UK, Aug 1999.

[7] N.W. Bergmann and J.C. Mudge, "An analysis of FPGA-based custom computers for DSP applications," in *Proc. 1994 IEEE International Conference on Acoustics, Speech and Signal Processing*, Adelaide, Australia, vol. 2, Apr 1994, pp. 513-516.

[8] K. Rajagopalan and P. Sutton, "A flexible multiplication unit for an FPGA logic block," in *Proc. 2001 IEEE International Symposium on Circuits and Systems*, 2001, pp. 546-549.

[9] R. Hartenstein, "Coarse grain reconfigurable architectures," in *Proc. 6th Asia South Pacific Design Automation Conference*, Yokohama, Japan, 2001, pp. 564-570.

[10] J. Smit et al, "Low cost and fast turnaround: reconfigurable graph-based execution units," in *Proc. 7th BELSIGN Workshop*, Enschede, Netherlands, 1998.

[11] P. Heysters et al, "A reconfigurable function array architecture for 3G and 4G wireless terminals", in *Proc. World Wireless Congress*, San Francisco, CA, 2002, pp. 399-405.

[12] A. Gunzinger, S. Mathis, and W. Guggenbuhl, "A reconfigurable systolic array for real-time image processing," in *Proc. 1988 International Conference on Acoustic, Speech, and Signal Processing,* New York, NY, Apr 1998, vo. 4, pp. 2054-2060.

[13] J. G Delgado-Frias, M. Myjak, F. Anderson, and D. Blum, "A medium-grain reconfigurable cell array for DSP application," in *Proc. 3rd iasted International Conference on Circuits, Signals, and Systems*, Cancun, Mexico, May 2003, pp. 231-236.

[14] M. J. Myjak and J. G. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," in *Proc. 2003 International Conference on VLSI*, Las Vegas, NV, Jun 2003, pp. 21-27.

[15] M. J. Myjak and J. G. Delgado-Frias, "Pipelined multipliers for reconfigurable hardware," in *Proc. 11th Reconfigurable Architectures Workshop,* Santa Fé, NM, Apr 2004.

[16] M. J. Myjak, F. L. Anderson, and J. G. Delgado-Frias, "H-Tree interconnection structure for reconfigurable DSP hardware," in *Proc. 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA),* Las Vegas, NV, Jun 2004.

[17] M. J. Myjak, "A two-level reconfigurable cell array for digital signal processing", *Master Thesis*, School of EECS, Washington State University, May 2004.

[18] K. Leijten-Nowak and A. Katoch, "Architecture and implementation of an embedded reconfigurable logic core in CMOS 0.13-μm," in *Proc. 15th Annual IEEE International ASIC/SOC Conference,* Sep 2002, pp. 3-7.

[19] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *IEEE Symposium on FPGAs for Custom Computing Machines,* 1995, pp. 99-107.

[20] M. J. Wirthlin and B. L. Hutchings, "Sequencing run-time reconfigured hardware with software," in *ACM/SIGDA International Symposium on FPGAs*, 1996, pp. 122-128.

[21] W. H. Mangione-Smith, "ATR from UCLA", *Personal Commun.*, 1999.

[22] D. Deshpande et al, "Configuration caching vs data caching for striped FPGAs," in *ACM/SIGDA International Symposium on FPGAs*, 1999, pp. 206-214.

[23] Z. Li, K. Compton and S. Hauck, "Configuration caching for FGPAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines,* 2000, pp. 22-36.

[24] K. Compton et al, "Configuration relocation and defragmentation for FPGSs," *Northwestern University Technical Report*; http://www.ece.nwu.edu/~kati/publications.html

[25] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath. A Configurable Computing Architecture for Computer-Intensive Applications, Dept. of Computer Science and Engineering, University of Washington, Nov 1996.

[26] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Páez-Monzón, I. Rahardja, "The design of an SRAM-based Field-Programmable Gate Array – Part I: Architecture," *IEEE Trans. VLSI Syst. 7*, 2, 1999, pp. 191-197.

[27] S. Trimberger, K. Duong, and B. Conn, "Architecture issues and solutions for a high-capacity FPGA. *ACM/SIGDA International Symposiumon FPGAs*," 1997, pp. 3-9.

[28] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," *ACM/SIGDA International Symposium on FPGAs*, 1999, pp. 69-78.

[29] A. Aggarwal and D. Lewis, "Routing architectures for hierarchical field programmable gate arrays," in *Proc. IEEE International Conference on Computer Design*, Cambridge, MA, Oct 1994, pp. 475-478.

[30] Y. Lai and P. Wang, "Hierarchical interconnection structures for field programmable gate arrays," *IEEE Transations on VLSI Systems*, vol. 5, iss. 2, Jun 1997, pp. 186-196.

[31] E. Tan and W. B. Heinzelman, "DSP architectures: past, present and future," *Computer Architecture News*, vol. 31, no. 3, Jun 2003, pp. 6-19.

[32] A. DeHon, "Comparing Computing Machines," *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, Nov 1998, pp. 2-3.

[33] P. Benoit, G. Sassatelli, L. Torres, D. Demigny, M. Robert and G. Cambon, "Metrics for reconfigurable architectures characterization: Remanence and Scalability," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[34] Xilinx Product Specifications, "FPGA Devices Families," http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp