A MEDIUM-GRAIN RECONFIGURABLE ARCHITECTURE FOR DIGITAL SIGNAL PROCESSING

By

MITCHELL JOHN MYJAK

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

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2006

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of MITCHELL JOHN MYJAK find it satisfactory and recommend that it be accepted.

____
Chair

____

____

Acknowledgment

This research has been performed in conjunction with the High Performance Computer Systems (HiPerCopS) research group under the direction of Dr. José Delgado-Frias.

The author gratefully acknowledges the financial support received from the EECS Alumni Fellowship at Washington State University, and the U.S. Department of Homeland Security (DHS) Graduate Fellowship. The DHS Scholarship and Fellowship Program is administered by the Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement with the U.S. Department of Energy (DOE). ORISE is managed by Oak Ridge Associated Universities under DOE contract number DE-AC05-00OR22750. All opinions expressed in this work are the author's and do not necessarily reflect the policies and views of DHS, DOE, or ORISE.

I would like to thank the members of the doctoral committee for all their help and support. In particular, Dr. José Delgado-Frias has gone well beyond his role as advisor to become a constant source of inspiration and encouragement. Special appreciation is also due to the fellow students in the HiPerCopS research group for their hard work and dedication to this project: Fredrick Anderson, Katie Blomster, Danny Blum, Seon Kwang Jeon, Jonathan Larson, and Andy Widjaja.

Finally, I would like to convey special gratitude to my parents for always supporting me in my educational pursuits and serving as a role model throughout my life. I also thank David Betowski and Jennifer Streicher for all their help during my graduate school tenure.

A MEDIUM-GRAIN RECONFIGURABLE ARCHITECTURE

FOR DIGITAL SIGNAL PROCESSING

Abstract

by Mitchell John Myjak, Ph.D.
Washington State University
May 2006

Chair: José G. Delgado-Frias

Reconfigurable hardware has become an attractive option for implementing digital signal processing, especially in systems that require both high performance and flexibility. Field-programmable gate arrays use fine-grain cells that implement simple logic functions. Some proposed reconfigurable devices use coarse-grain cells that perform 16-bit or 32-bit operations. A third alternative is to use medium-grain cells with a word length of 4 or 8 bits. This approach combines high flexibility with inherent support for word-length computations.

This dissertation presents a novel medium-grain reconfigurable architecture for digital signal processing. The basic cell contains an array of small lookup tables, or "elements", that operate in two modes. In memory mode, the elements act as a random-access memory. In mathematics mode, the elements perform 4-bit arithmetic. This two-level structure offers good fine-grain flexibility without incurring the overhead of fine-grain devices.

Cells are grouped together to implement larger modules, such as multipliers, adders, and memory units. The proposed architecture features a hierarchical interconnection network that optimizes data transfer both within and between modules. Upper-level switches route data in units of words rather than bits, saving considerable area. The entire system is pipelined to maximize clock rate and throughput.

In all, the proposed architecture encompasses a large design space with many orthogonal axes. Users can control the word length, data format, amount of parallelism, and number

of modules used to implement algorithms. The circuit design can also be customized to focus on a particular class of applications. For example, cells can perform computations in bit-parallel or bit-serial fashion.

Layout simulations in 180-nm CMOS technology indicate that the architecture obtains high performance. Initial prototypes have also been fabricated and tested for functionality. The estimated execution times for several common benchmarks meet or exceed the reported results of other reconfigurable devices in similar technologies.

Contents

Acknowledgment **iii**

Abstract **iv**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Metrics for DSP hardware 1

1.2 Types of DSP hardware 3

1.2.1 General-purpose processors 3

1.2.2 Digital signal processors 3

1.2.3 Custom integrated circuits 4

1.2.4 Fine-grain reconfigurable hardware 4

1.2.5 Coarse-grain reconfigurable hardware 5

1.2.6 Medium-grain reconfigurable hardware 7

1.3 Proposed architecture 8

1.4 Outline 9

2 Elements **11**

2.1 Functionality 11

2.2 Designs 12
2.2.1 Dynamic element 13
2.2.2 Static element 15
2.2.3 Pipelined element 18
2.3 Analysis 18
2.4 Summary 20

3 Cells **21**
3.1 Designs 21
3.1.1 Parallel cell 22
3.1.2 Serial cell 24
3.2 Functions 25
3.2.1 Multiplication 27
3.2.2 Addition and subtraction 28
3.2.3 Bit shifting 29
3.2.4 Memory access 30
3.2.5 Control logic 32
3.2.6 Reconfiguration 33
3.3 Analysis 33
3.3.1 Circuit simulations 33
3.3.2 Layout simulations 35
3.3.3 Comparison of cells 37
3.4 Verification 38
3.4.1 Parallel-dynamic design 38
3.4.2 Parallel-static design 39
3.4.3 Serial-pipelined design 42
3.5 Summary 43

4 Interconnections and Modules **44**
4.1 Interconnection network 45
4.1.1 Local mesh 45
4.1.2 Global H-tree 46
4.1.3 Pipelining 49
4.2 Modules 50
4.2.1 Multiplier 50
4.2.2 Multiply-accumulate unit 51
4.2.3 Adder and subtracter 52
4.2.4 Shifter 53
4.2.5 Memory unit 54
4.2.6 Specialized memory unit 55
4.2.7 Control logic 57
4.2.8 Floating-point adder 57
4.2.9 Floating-point multiplier 58
4.3 Summary 59

5 Hierarchical Multipliers **61**
5.1 Structure 61
5.1.1 Carry-save multiplier 62
5.1.2 Proposed design 63
5.1.3 Proof of functionality 64
5.2 Data formats 65
5.3 Cell functions 68
5.4 Summary 69

6 Algorithms **72**

6.1 Software tools 72
6.2 Configuration 74
6.3 Benchmarks 75
6.3.1 FIR filter 76
6.3.2 CORDIC unit 78
6.3.3 Fast Fourier Transform 79
6.4 Analysis 82
6.4.1 Digital signal processors 82
6.4.2 Fine-grain reconfigurable hardware 83
6.4.3 Coarse-grain reconfigurable hardware 85
6.5 Summary 86

7 Conclusion **87**
7.1 Contributions 88
7.2 Future work 90

Bibliography **91**

A Clock Generator **95**
A.1 NTVW design 96
A.2 Proposed design 97
A.2.1 Oscillator 97
A.2.2 Shift register 98
A.2.3 Decoder 100
A.2.4 Operation 100
A.3 Analysis 102

B Pipeline Registers **106**

B.1 Existing designs 107
B.1.1 Basic differential flip-flop 107
B.1.2 SSTC 108
B.1.3 SAFF1 108
B.1.4 SAFF2 109
B.2 Proposed design 109
B.3 Analysis 112
B.3.1 Methodology 112
B.3.2 Results 115

C Publications 118

C.1 Journal papers 118
C.2 Conference papers 118

List of Tables

1.1 Comparison of DSP hardware. 3

2.1 Comparison of elements. 20

3.1 Examples of cell operations. 26

3.2 Comparison of medium-grain cells. 37

3.3 Test cases to verify prototype. 40

4.1 Latency of cells and interconnection network. 49

5.1 Data formats for two's-complement MAC unit. 68

5.2 Element functions in MAC cells. 69

6.1 Statistics of FIR filter. 77

6.2 Statistics of one CORDIC stage. 79

6.3 Statistics of FFT. 81

6.4 Performance comparison with digital signal processors. 83

6.5 Architecture comparison with fine-grain reconfigurable devices. 84

6.6 Performance comparison with fine-grain reconfigurable hardware. 85

B.1 Simulation results for differential flip-flops. 115

List of Figures

1.1 Granularity of reconfigurable hardware. 7
1.2 Modules and global interconnection network. 8
1.3 Typical module with cells and interconnection network. 9
1.4 Elements inside cell configured for mathematics functions. 10

2.1 Functional diagram of element. 12
2.2 Organization of dynamic element. 13
2.3 Read datapath in dynamic element. 14
2.4 Write datapath in dynamic element. 15
2.5 Organization of static element. 16
2.6 Read datapath in static element. 17
2.7 Write datapath in static element. 17
2.8 Latch used in pipelined element. 18
2.9 Simulation of dynamic element. 19
2.10 Simulation of static element. 19

3.1 Parallel cell in memory mode. 22
3.2 Parallel cell in mathematics mode. 23
3.3 Serial cell in memory mode. 24
3.4 Serial cell in mathematics mode. 25
3.5 Equivalent parallel model of mathematics mode. 26

3.6 Parallel cell used as bit shifter. 29

3.7 Serial cell used as bit shifter. 31

3.8 Parallel cell used to implement control logic. 32

3.9 Simulation of parallel cell with dynamic elements. 34

3.10 Simulation of parallel cell with static elements. 35

3.11 Simulation of serial cell with pipelined elements. 35

3.12 Simulation of parallel cell with pipelined elements. 36

3.13 Layout simulation of parallel cell with static elements. 36

3.14 Layout simulation of serial cell with pipelined elements. 37

3.15 Prototype of parallel cell with dynamic elements. 39

3.16 Verification of parallel cell with dynamic elements. 40

3.17 Prototype of parallel cell with static elements. 41

3.18 Verification of parallel cell with static elements. 41

3.19 Prototype of serial cell with pipelined elements. 42

3.20 Verification of serial cell with pipelined elements. 43

4.1 Steps for implementing DSP on the reconfigurable cell array. 44

4.2 Local interconnection structure. 45

4.3 Interface between cell and interconnection network. 46

4.4 Global interconnection structure. 47

4.5 Typical switch in global interconnection structure. 48

4.6 Link in global switch. 48

4.7 16-bit multiplier. 51

4.8 32-bit adder. 53

4.9 16-bit left shifter. 54

4.10 Dual-port memory unit. 55

4.11 Memory unit for Fast Fourier Transform. 56

4.12 32-bit exchange unit. 57
4.13 Diagram of floating-point adder. 58
4.14 Implementation of floating-point adder. 59
4.15 Diagram of floating-point multiplier. 59
4.16 Implementation of floating-point multiplier. 60

5.1 Carry-save multiplier. 62
5.2 Proposed multiply-accumulate unit. 63
5.3 Two's-complement MAC unit. 66
5.4 Implementation of cell functions. 70

6.1 Screenshot of software tools. 73
6.2 Structure of cell array during reconfiguration. 75
6.3 Diagram of FIR filter. 76
6.4 Implementation of FIR filter. 77
6.5 Diagram of one CORDIC stage. 78
6.6 Implementation of one CORDIC stage. 79
6.7 Diagram of FFT. 80
6.8 Implementation of FFT. 81

A.1 Control signals required by serial cell. 95
A.2 Pulse generator proposed by Nilsson et al. (NTVW) [42]. 96
A.3 Functional diagram of differential clock generator. 97
A.4 Dual ring oscillator. 98
A.5 Shift register used to stop oscillator. 99
A.6 Decoder for initialization. 100
A.7 Layout of clock generator. 101
A.8 Reset and start of pulse train. 101

A.9 End of pulse train and initialization. 102
A.10 Layout simulation showing 2-GHz operation. 103
A.11 Simulation of NTVW clock generator [42]. 103
A.12 NTVW design cuts off pulse at lower frequencies. 104
A.13 Proposed design can operate at lower frequencies. 104

B.1 Basic differential flip-flop. 107
B.2 Static single-transistor clocked (SSTC) flip-flop [44]. 108
B.3 Sense amplifier flip-flop (SAFF1) [45]. 109
B.4 Modified sense amplifier flip-flop (SAFF2) [46]. 110
B.5 Proposed differential flip-flop. 110
B.6 Simulation of proposed flip-flop. 111
B.7 Comparison of differential and single-ended clock buffers. 112
B.8 Simulation of differential clock buffer. 113
B.9 Testbench for flip-flops. 114
B.10 Delay measurements. 114
B.11 Plot of setup time versus output delay. 116
B.12 Plot of total delay versus power consumption. 116

Dedication

Pro Gloria Dei

Chapter 1

Introduction

Many digital systems rely on digital signal processing (DSP) to achieve their functionality. For example, cellular phones use sophisticated compression and encryption algorithms to transmit data securely over a wireless link. Digital multimedia devices such as DVD players translate a stream of bits into images or music. Even hearing aids may contain complex digital filters to enhance speech.

Digital systems may use a variety of components to implement DSP, ranging from custom integrated circuits to general-purpose microprocessors. Reconfigurable hardware has become an attractive option in recent years, especially in applications that must combine high performance and flexibility [1]. In this chapter, we first identify the primary metrics for DSP hardware. We then use these criteria to compare the various components. Finally, we give an overview of the reconfigurable architecture proposed in this dissertation.

1.1 Metrics for DSP hardware

Although DSP encompasses a wide range of applications, DSP hardware can be evaluated with respect to several metrics:

- **Performance:** DSP places great demands on the processing power of the underlying hardware. For example, a 512-point Fast Fourier Transform (FFT) requires around 16,000 multiplications and 9,000 additions [2]. Algorithms typically work with data in vector or matrix form, so the hardware must apply the same basic operation to many data points. Hence, the standard measure of performance is not latency, but rather total execution time, or its reciprocal, throughput. Devices can exploit the inherent parallelism of DSP to achieve high throughput.

- **Flexibility:** For commercial products, the total cost clearly influences the chosen design strategy. Using flexible hardware eliminates the need to design, fabricate, and test custom components. Furthermore, the functionality of the hardware can adapt to changes in system requirements, leading to lower redesign costs.

- **Power consumption:** In recent years, the application space of DSP has shifted to include wireless and mobile computing. Power consumption is a crucial metric for these systems. This evolution requires novel hardware architectures to meet the new demands and challenges.

- **Fault tolerance:** DSP hardware in mission-critical applications, such as communication satellites and real-time monitoring equipment, must contain mechanisms to detect and handle faults. Radiation-induced errors, such as latch-up, burn-out, and single event upsets, are of major concern in space.

Most applications require a balance between two or more of these metrics. Hence, the ability of hardware components to trade off various parameters, such as performance for power consumption, is another key factor that influences the design strategy.

Table 1.1: Comparison of DSP hardware.

Device	Performance	Flexibility	Power	Fault tolerance
General-purpose processors	Poor	Best	Fair	Poor
Digital signal processors	Fair	Good	Good	Poor
Fine-grain reconfigurable hardware	Good	Better	Better	Good
Medium-grain reconfigurable hardware	Better	Better	Better	Good
Coarse-grain reconfigurable hardware	Better	Good	Better	Good
Custom integrated circuits	Best	Poor	Best	Best

1.2 Types of DSP hardware

This section describes the main types of hardware available for DSP. Table 1.1 summarizes these alternatives in terms of the four metrics described in the previous section.

1.2.1 General-purpose processors

General-purpose processors, such as the Intel Pentium 4 or AMD Athlon, can execute a wide variety of software programs. A large number of compilers exist for software written in high-level languages such as C++. Hence, implementing DSP is usually as simple as compiling the appropriate source code.

The main drawback of general-purpose processors is their relatively low performance. Most devices can only perform one multiplication at a time, creating a bottleneck for DSP. The power consumption of general-purpose processors also falls behind other alternatives. Fault tolerance is usually not present in commodity devices, with the possible exception of error checking in cache memory.

1.2.2 Digital signal processors

Digital signal processors resemble general-purpose processors in many respects, but contain specific enhancements to optimize DSP. For example, the instruction set architecture (ISA) of the Texas Instruments TMS320C64 includes instructions to perform dot products and fast

multiplications. The hardware contains several functional units that each support multiply-accumulate operations [3].

Compared to general-purpose processors, the increased performance for DSP incurs a slight penalty in flexibility. The ISA typically lacks instructions such as division that do not occur in most algorithms. However, the optimized datapath reduces the overall power consumption.

1.2.3 Custom integrated circuits

At the other end of the spectrum are application-specific integrated circuits (ASICs). These devices achieve the highest performance but the lowest flexibility, since they are designed for a single algorithm. The hardware can be optimized for low power consumption or fault tolerance as well. Due to their high development costs and limited applicability, ASICs only become feasible for high-volume or very specialized applications.

1.2.4 Fine-grain reconfigurable hardware

In general, reconfigurable hardware contains an array of programmable cells and interconnections. This approach attempts to combine the performance of an ASIC with the flexibility of a microprocessor. The continually expanding capabilities of very large-scale integration (VLSI) have made reconfigurable hardware feasible for DSP [4, 5, 6].

Most reconfigurable hardware today is classified as a field programmable gate array (FPGA). These fine-grain devices contain cells that perform simple logic functions. As a result, FPGAs can implement completely generic operations. The desired functionality may be specified in a hardware description language such as VHDL, or a high-level language such as C. A logic synthesizer then divides the operation into small pieces and maps the result onto a virtual model of the architecture. The synthesizer generates a configuration file used to program the cells and interconnections. The device can be reconfigured at any time, even

after deployment.

Since FPGAs mimic dedicated hardware, they offer higher performance than digital signal processors for computationally intensive algorithms. However, implementing a multiplier on a fine-grain device requires a large number of cells. The resulting structure is hampered by the interconnection delays. For this reason, some FPGAs embed fixed-length multipliers within the reconfigurable fabric [7, 8]. The Xilinx Virtex-II Pro is one example of this trend [9].

Another way to reduce the circuit complexity is to evaluate arithmetic functions such as multiplication in bit-serial fashion. This approach requires fewer cells and interconnection resources on reconfigurable hardware. For example, a bit-serial, n-bit multiplier typically computes the product in $2n$ cycles using a linear chain of processing elements [10]. Recognizing these benefits, researchers have developed FPGAs that perform bit-serial computations [11, 12]. The drawback of these designs is that the entire device runs off a single clock signal. Aside from the problem of clock distribution, the interconnection structures within the critical path limit the maximum clock frequency and hence the processing speed.

FPGAs can employ several measures to reduce the overall power consumption, such as disabling unused portions of the hardware. The devices also contain some degree of fault tolerance, in that the logic synthesizer can remap algorithms to avoid faulty areas. However, some architectural enhancements are necessary to support this feature.

1.2.5 Coarse-grain reconfigurable hardware

Recently, researchers have proposed new types of reconfigurable hardware in which each cell performs 16-bit or 32-bit operations [13]. These coarse-grain devices achieve high performance for DSP. For example, the RaPiD architecture contains a linear array of 16-bit functional units [14]. The KressArray family features a matrix of 32-bit processing units with a hierarchical interconnection structure [15]. The FPOP architecture consists of cells

that perform digit-serial arithmetic, starting from the most significant digit [16]. This technique allows cells to compute simple functions such as addition or multiplication, as well as complex functions such as division and square root.

One characteristic of coarse-grain devices is the limited number of functions available in each cell. This approach does reduce the power consumption compared to fine-grain devices, but can prevent the device from implementing the control logic necessary for DSP. Several proposed designs address this problem by integrating a heterogeneous set of components on the same die. For example, the Pleiades architecture combines a microprocessor with arithmetic/logic units (ALUs), memory modules, and an embedded FPGA [17]. The MONTIUM architecture integrates a microprocessor with an FPGA and an array of coarse-grain cells, each containing several 16-bit ALUs [18].

Coarse-grain reconfigurable hardware has remained an active area of research, stretching back to the systolic array architectures proposed two decades ago [19]. However, the idea has made limited progress in the commercial sector, due to the prevalence of FPGAs and the well-established software tools for logic synthesis. The coarse-grain architecture marketed by PACT XPP Technologies contains one or more arrays of processing elements, each of which holds an ALU or a memory [20]. The Adapt2000 ACM architecture from QuickSilver Technology uses a hierarchy of heterogeneous nodes; different types of nodes can implement 32-bit binary arithmetic, bit-oriented operations, general-purpose code, and memory control [21]. Rather than manufacturing a series of commodity devices like Xilinx, both companies offer intellectual property (IP) that customers can integrate within the target system.

Aside from these novel architectures, many high-performance FPGAs now include coarse-grain components to accelerate DSP. The Xilinx Virtex-4, for example, contains special XtremeDSP slices that can perform 18-bit multiplication and 48-bit addition [22].

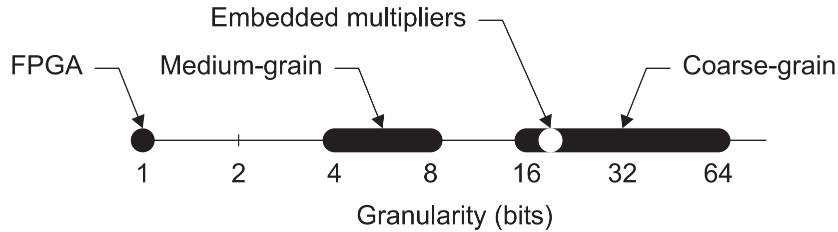


Figure 1.1: Granularity of reconfigurable hardware.

1.2.6 Medium-grain reconfigurable hardware

A third type of reconfigurable hardware contains cells that perform 4-bit or 8-bit functions. These devices use multiple cells to implement 16-bit or 32-bit operations. We call this approach "medium-grain" to distinguish it from architectures that perform word-length operations in one cell. Figure 1.1 compares the granularity of the three types of reconfigurable hardware.

Several examples of medium-grain devices exist in the literature. The CHESS architecture contains a hexagonal array of 4-bit ALUs with embedded 256-byte memory modules [23]. The PipeRench architecture contains an array of 8-bit cells organized into 128-bit stripes [24]. The DReAM architecture places a lookup table and adder inside each cell, together with shifting logic to implement multiplication [25]. In addition, the commercial D-Fabrix architecture from Elixent uses 4-bit cells and switchboxes [26].

Medium-grain reconfigurable hardware achieves a good balance between performance, area, and flexibility. Like coarse-grain devices and digital signal processors, arithmetic functions can be optimized for speed. Unlike these devices, algorithms are not tied to a specific word length. The higher granularity compared to FPGAs also benefits the power consumption. Finally, medium-grain devices have the same potential for fault tolerance as other types of reconfigurable hardware.

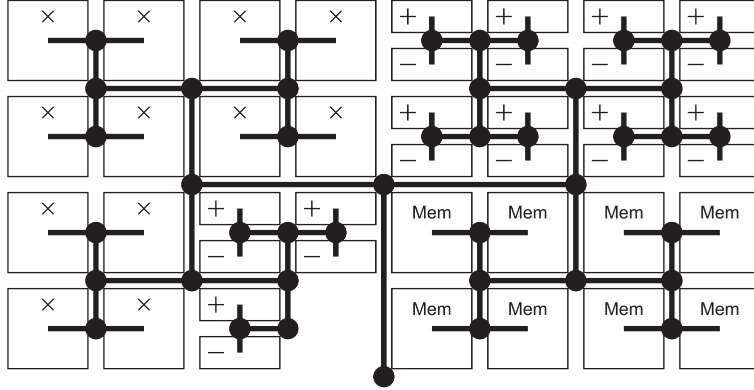


Figure 1.2: Modules and global interconnection network.

1.3 Proposed architecture

This dissertation proposes a novel medium-grain reconfigurable architecture for DSP [27, 28]. This architecture contains an array of 4-bit cells and a hierarchical interconnection network. To implement DSP, the array is partitioned into discrete modules, such as multipliers, adders, and lookup tables. A tree-based interconnection structure transfers data between modules in word-length units. Figure 1.2 gives a top-level overview of the architecture for a sample group of modules.

Each module, such as the 16-bit multiplier in Figure 1.3, occupies a block of cells. The 4-bit granularity allows modules to be constructed for any word length. A local interconnection network passes intermediate results between neighboring cells. The global network connects the inputs and outputs of the module to other modules on the device.

To maximize performance, the interconnection network pipelines all operations into 4-bit portions. This approach allows modules to initiate one operation per clock cycle. Since every module uses the same pipelining scheme, the hardware maintains the maximum clock frequency at all times, irrespective of the current configuration.

Cells in the architecture can perform a wide variety of functions, including multiply-accumulates, addition and subtraction, combinational logic, and memory operations. To

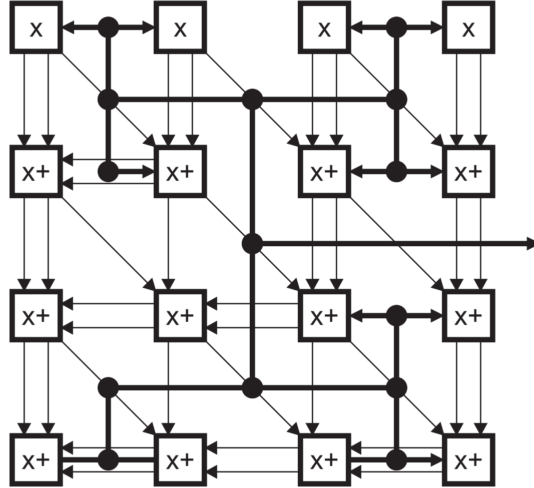


Figure 1.3: Typical module with cells and interconnection network.

implement this functionality, cells contain a small array of 1-bit processing elements. The set of elements can only assume two structures: one optimized for memory operations, and the other for mathematics functions. This innovative two-level architecture combines coarse-grain performance with fine-grain flexibility. Figure 1.4 illustrates the contents of a cell in mathematics mode.

Each element in the cell operates as a random-access memory (RAM) with four inputs and two outputs. In memory mode, the RAM inside the elements combines to form a single, larger memory. In mathematics mode, the RAM specifies a lookup table. In this way, the cell can perform arithmetic or logic functions on signed or unsigned data, just by loading the appropriate truth table into the elements.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 begins by describing three circuit designs for the basic element. Each design uses a different clock approach, but all have the same functionality. Chapter 3 presents two structures for the 4-bit cell: a parallel

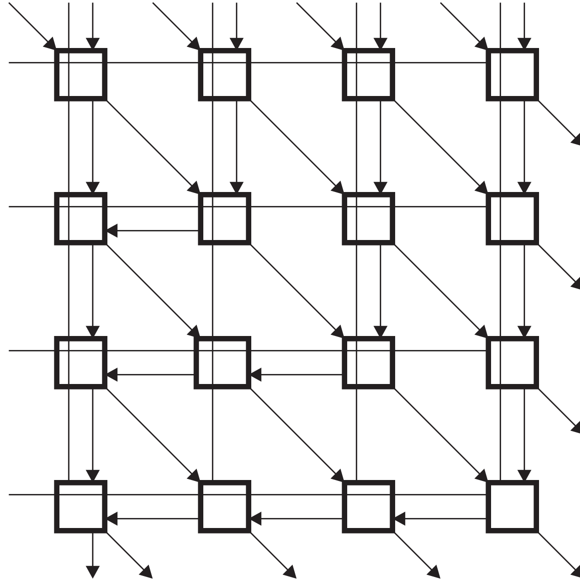


Figure 1.4: Elements inside cell configured for mathematics functions.

version comprised of sixteen elements, and a serial version comprised of five. We compare the overall clock rate and functionality of the two alternatives. Chapter 4 describes how the interconnection network can group cells into word-length modules. The discussion includes several examples, including multipliers, adders, and memory units. Chapter 5 gives more details about mapping multipliers onto the parallel cells. The architecture can support both unsigned and two's-complement arithmetic. Chapter 6 demonstrates how modules can be linked together to implement entire algorithms. We calculate the execution time for several benchmarks and compare the results to other hardware components, including digital signal processors and FPGAs. Finally, Chapter 7 gives some concluding remarks and summarizes the contributions of this research.

The appendices give more details about specific aspects of the proposed reconfigurable architecture. Appendix A describes the clock generator required by the serial cell. Appendix B presents the pipeline register used in the architecture. Finally, Appendix C lists the papers published during the course of this research.

Chapter 2

Elements

A notable characteristic of the medium-grain reconfigurable architecture is the absence of functional units such as adders. Each cell uses an array of elements for memory and mathematics operations. The element itself essentially implements a small memory. This strategy leads to a compact circuit design that achieves high performance.

We begin this chapter by specifying the functionality of an element. We then describe three circuit designs that realize this behavior. Finally, we compare the performance of the alternatives.

2.1 Functionality

Figure 2.1 depicts a functional diagram of an element. The component implements a 32-bit RAM with separate read and write ports. This feature allows the cell to read data from one address and write data to another address simultaneously.

The memory inside the element is divided into two banks of sixteen latches. For read operations, address $ra_{0:3}$ selects the corresponding latch in each bank. Read enable $re_{0:1}$ determines which bank is connected to the output ro. If neither bank is enabled, the element drives ro with the input ri. Write operations proceed in a similar manner: address $wa_{0:3}$

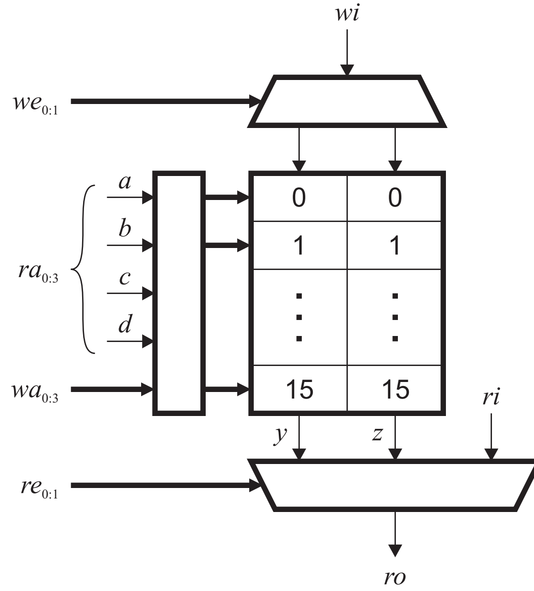


Figure 2.1: Functional diagram of element.

selects one latch in each bank, and write enable $we_{0:1}$ determines which bank, if any, is driven by the input wi.

When configured to perform mathematics functions, the cell uses each element as a 4-input, 2-output lookup table. The four inputs are placed onto the bits of $ra_{0:3}$. For clarity, we also refer to these bits as a, b, c, and d. The outputs of the two selected latches are taken directly to outputs y and z.

2.2 Designs

This section presents three circuit designs for the basic element. Each alternative uses a different clocking scheme to optimize performance.

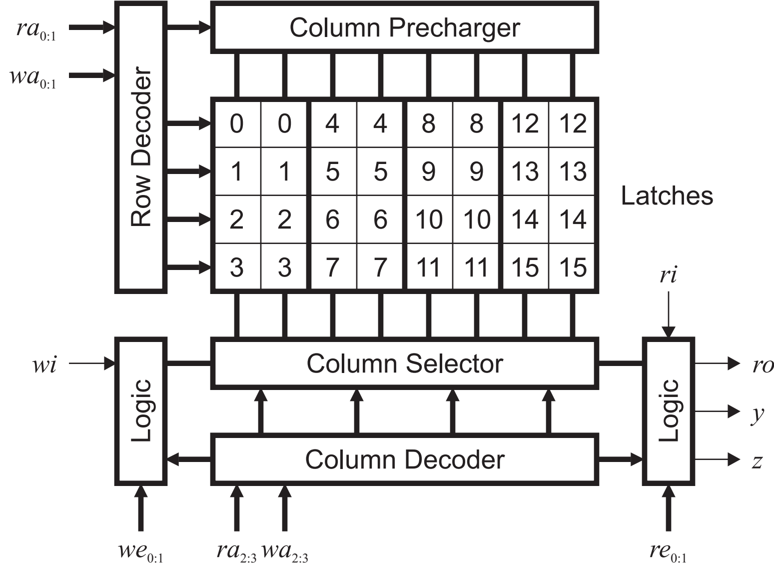


Figure 2.2: Organization of dynamic element.

2.2.1 Dynamic element

The first design relies on dynamic logic with alternating precharge and evaluation phases. All inputs and outputs are represented as complementary lines, such as ra_0 and $\overline{ra}_0$. During the precharge phase, both lines are set high. During the evaluation phase, one line falls low: ra_0 for logic 0 or $\overline{ra}_0$ for logic 1. The circuit does not require a separate clock signal since the inputs themselves contain the necessary timing information.

As shown in Figure 2.2, the dynamic element organizes the internal memory into four rows and four columns. The row decoder uses $ra_{0:1}$ and $wa_{0:1}$ to enable individual rows for reading and writing. Similarly, the column decoder uses $ra_{2:3}$ and $wa_{2:3}$ to enable individual columns. The column selector connects one set of column data lines to the inputs or outputs. The column precharger initializes these lines to high during the precharge phase. Finally, the two logic blocks select which bank the element uses for reads and writes.

Figure 2.3 illustrates the datapath used for read operations. Suppose the element needs to read the latch at address 0 in bank 0. During the precharge phase, all lines in $ra_{0:3}$ and $\overline{ra}_{0:3}$ remain high. The decoders activate a series of p-transistors that pull up all internal lines

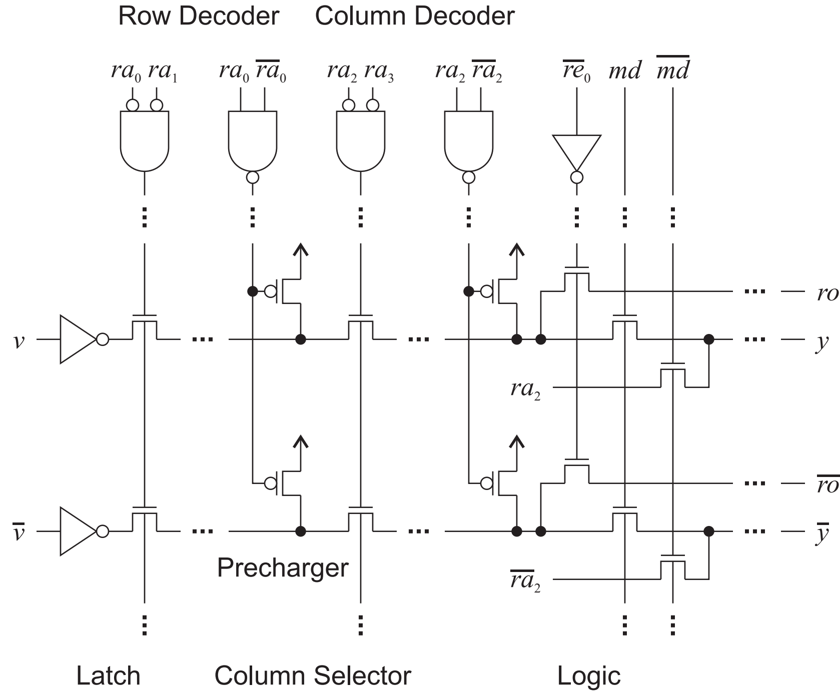


Figure 2.3: Read datapath in dynamic element.

to V_{DD}. When ra_0 and ra_1 fall low, the row decoder enables the n-transistors connected to the latches in the last row. Data from these latches begins to propagate toward the outputs. When ra_2 and ra_3 also evaluate, the column decoder enables the n-transistors connected to the data lines in the last column. Finally, the output logic selects the latch from bank 0 when $\overline{re}_0$ falls low.

The output logic allows the cell to assume a different structure to compute mathematics functions. The four bits in $ra_{0:3}$, also named a, b, c, and d, now specify the four input bits for the function. The selected latch in bank 0 drives y, while the selected latch in bank 1 drives z. These lines may connect to the c and d inputs of neighboring elements. Standard read operations set y and z to default values, usually ra_2 or ra_3.

For write operations, the dynamic element uses the datapath depicted in Figure 2.4. The circuit essentially mirrors the datapath used for read operations, but with $wa_{0:3}$ instead of $ra_{0:3}$. Data propagates from the input wi into the latch at the selected row and column. Each

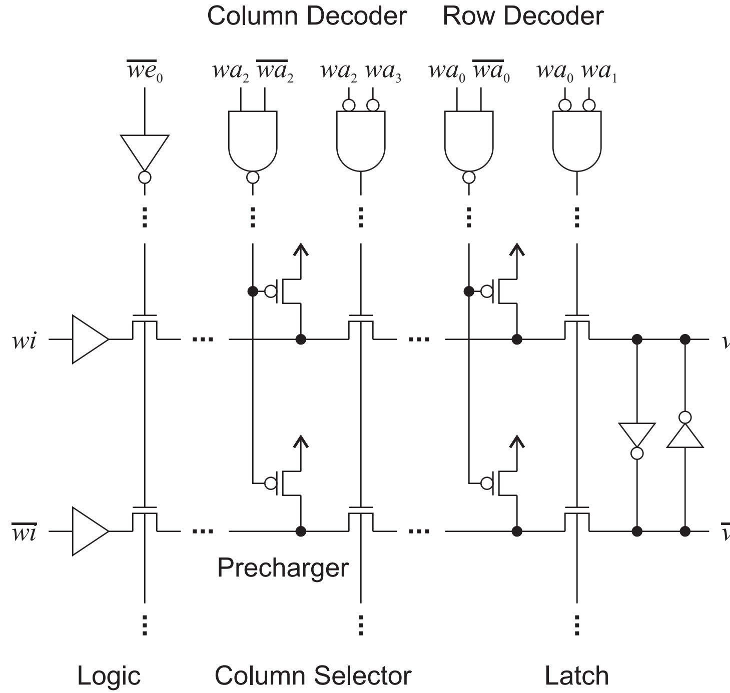


Figure 2.4: Write datapath in dynamic element.

latch contains a simple pair of inverters to store the value, as shown. Since the n-transistors are bidirectional, other latches in the selected row are connected to the internal data lines. Having the element precharge these lines beforehand prevents the latent charge from flipping the state of the latch.

2.2.2 Static element

The second design for the element relies on a static approach with no internal clocking. All inputs and outputs are still complementary signals, but use cross-coupled p-transistors as pull-up devices. As shown in Figure 2.5, the overall organization of the static element remains essentially the same, except for the absence of precharge circuitry. The row decoder still handles the lower two bits of $ra_{0:3}$ and $wa_{0:3}$, while the column decoder handles the upper two bits.

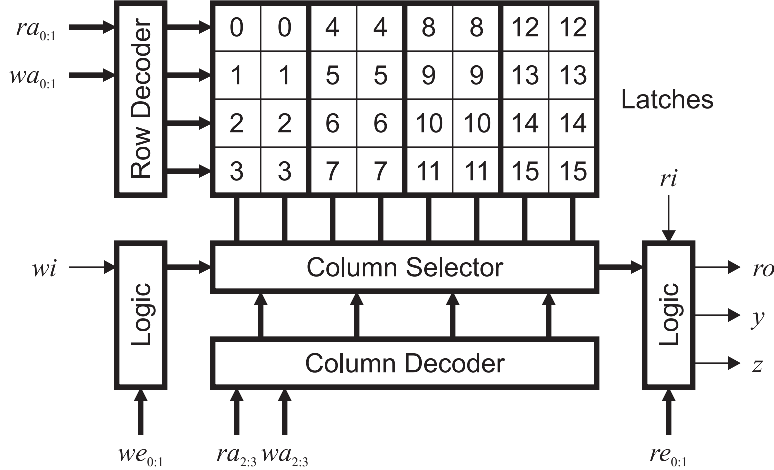


Figure 2.5: Organization of static element.

Figure 2.6 illustrates the read datapath. Data from the selected latch propagates through a series of n-transistors controlled by the row and column decoders. These n-transistors pass a strong low but a weak high. Cross-coupled p-transistors restore the rail-to-rail voltage swing. Notice that the column selector buffers the data lines for improved driving strength. As in the dynamic element, the element connects the selected data lines to y and z rather than ro when the cell is configured for mathematics operations.

The write datapath has a completely new design, as shown in Figure 2.7. Each latch uses a pair of minimum-size inverters to store the logic value. A network of n-transistors implements the set and reset circuitry. The rest of the datapath contains a series of AND-type logic gates that combine the enable signals with wi and $\overline{wi}$. For the latch at address 0 in bank 0, the circuit sets v to low when wi, we_0, and $wa_{0:3}$ are all low. If wi is high, the circuit sets $\overline{v}$ to low instead.

Although read operations execute in a combinational fashion, write operations require some type of clocking to ensure that changes only occur when the inputs are stable. The cell may gate $we_{0:1}$ with a global clock for this purpose.

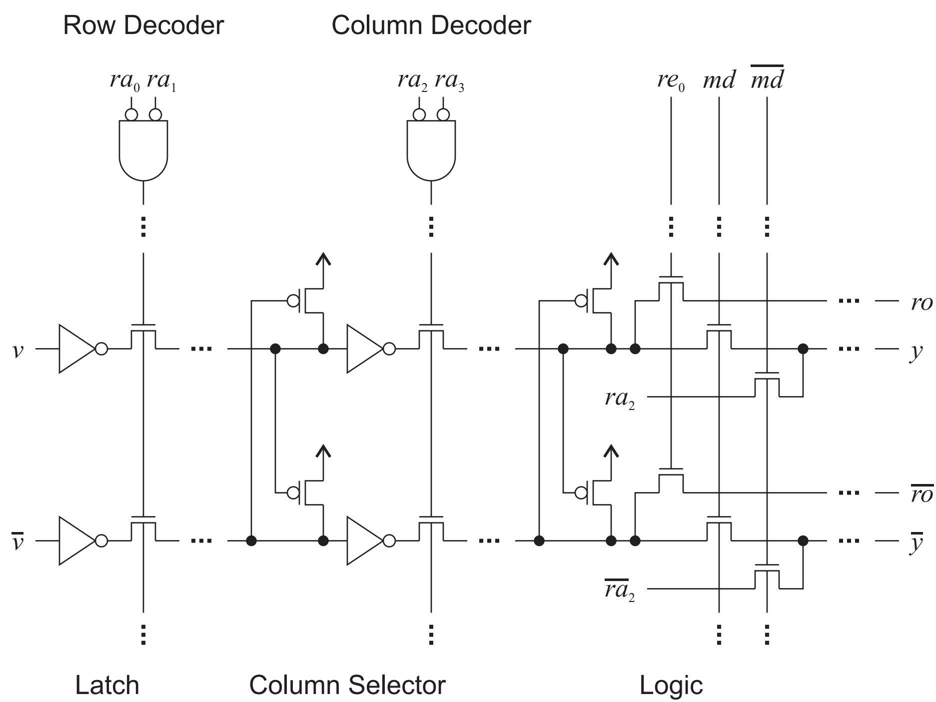


Figure 2.6: Read datapath in static element.

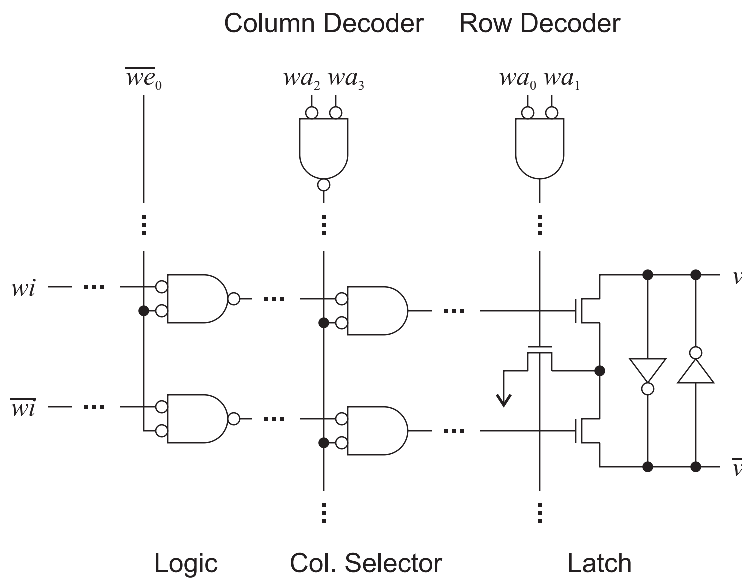


Figure 2.7: Write datapath in static element.

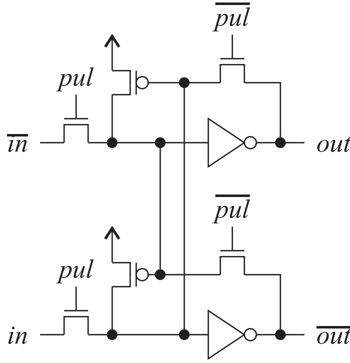


Figure 2.8: Latch used in pipelined element.

2.2.3 Pipelined element

The third design for the element applies bit-level pipelining to maximize throughput. This approach uses the same circuit design as the static element, but places pipeline latches on all the outputs. Having latches rather than full registers cuts down the area and latency. The latches are enabled by the pulse train *pul*. For proper operation, the width of each pulse must be less than the propagation delay through the element.

Figure 2.8 depicts the latch used in the pipelined element. This circuit is essentially half of the pipeline register described in Appendix B. When *pul* is high, data passes from the input to the output. The cross-coupled p-transistors restore the rail-to-rail swing at the inverter inputs. When $\overline{pul}$ is high, the p-transistors combine with additional n-transistors to maintain the logic state.

2.3 Analysis

To verify that the designs implement the correct functionality, we have simulated the circuits in 90-nm CMOS technology. Figure 2.9 illustrates the resulting waveforms for the dynamic element. The circuit is reading logic 0 from a latch in bank 0 and writing logic 1 to a latch in bank 1. Notice how ro and $\overline{ro}$ are precharged to high. The latency from the rising edge

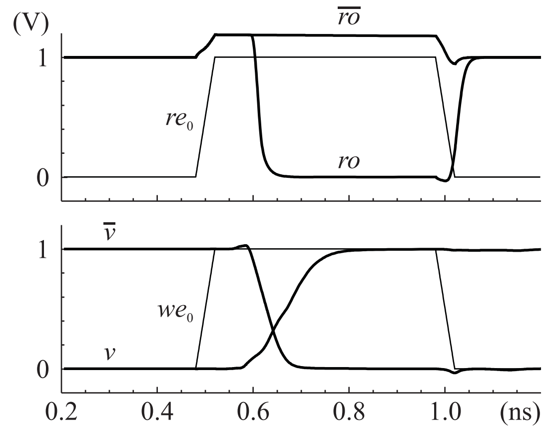


Figure 2.9: Simulation of dynamic element.

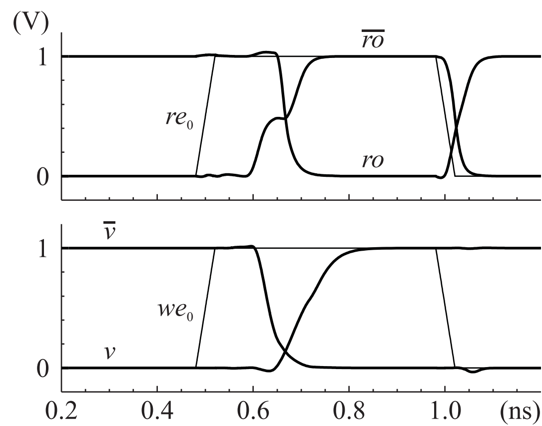


Figure 2.10: Simulation of static element.

of re_0 to the falling edge of ro is 112.1 ps. Similarly, the latency from the rising edge we_0 to the rising edge of v is 169.5 ps.

Figure 2.10 depicts the corresponding simulation for the static element. Now ro and $\overline{ro}$ transition at the same time. The latency from re_0 to ro is 170.8 ps, whereas the latency from we_0 to v is 202.9 ps. Read operations run faster due to the intermediate buffers in the column selector. However, the write datapath operates more slowly, since the circuit only drives one side of the latch.

As described in Chapter 3, the latency of mathematics computations determines the

Table 2.1: Comparison of elements.

Latency	Dynamic	Static	Pipelined
$re_{0:1}$ to ro	112.1 ps	170.8 ps	212.8 ps
$we_{0:1}$ to v	169.5 ps	202.9 ps	202.9 ps
a to y	109.6 ps	168.5 ps	210.7 ps
c to y	61.8 ps	80.8 ps	122.8 ps

overall speed of the cell. We have optimized the read datapath in the three circuit designs to achieve high performance. For example, the lower address bits a and b typically resolve before the upper address bits c and d. Hence, the read datapath decodes the row first, followed by the column. Data can propagate halfway through the element when a and b become available. The most critical path extends from inputs c and d to outputs y and z.

Table 2.1 compares the latencies of the three elements using the results of the circuit simulations. Although the dynamic element offers the lowest latency, it also requires a separate precharge phase. The pipeline latches in the pipelined element add 42.0 ps to read operations, but allow the design to achieve much higher throughput.

2.4 Summary

In this chapter, we have described three circuit designs for the basic element in the medium-grain reconfigurable architecture. Each design implements a 32-bit RAM with separate read and write ports. The dynamic element features a two-phase clock approach that precharges all internal lines during the first phase, and performs the required operations during the second phase. The static element dispenses with the precharge phase and uses differential logic with cross-coupled p-transistors. Finally, the pipelined element adds a pipeline latch to the output of the static element. Chapter 3 illustrates how 4-bit cells can use an array of elements to perform binary arithmetic and memory operations.

Chapter 3

Cells

This chapter presents two designs for the 4-bit cell in the proposed reconfigurable architecture. Each cell contains an array of elements that can assume two structures. In memory mode, the elements collectively implement a RAM. This functionality is useful for specifying constant coefficients and storing intermediate results. In mathematics mode, the elements act as lookup tables so that the cell can evaluate binary arithmetic. The flexible elements allow cells to perform a wide range of functions, ranging from simple combinational logic to powerful multiply-accumulates.

After describing the two designs, we demonstrate how the cells can implement binary arithmetic, bit shifting, and control logic. We then evaluate the two alternatives by means of circuit simulations, layout simulations, and measurements taken from prototype chips.

3.1 Designs

The first design for the cell contains sixteen elements and operates in bit-parallel fashion. The elements may be dynamic, static, or pipelined. This design is very similar to the parallel cell presented in [27, 30, 31], but does implement more functionality in memory mode.

The second design contains five elements and computes binary arithmetic in bit-serial

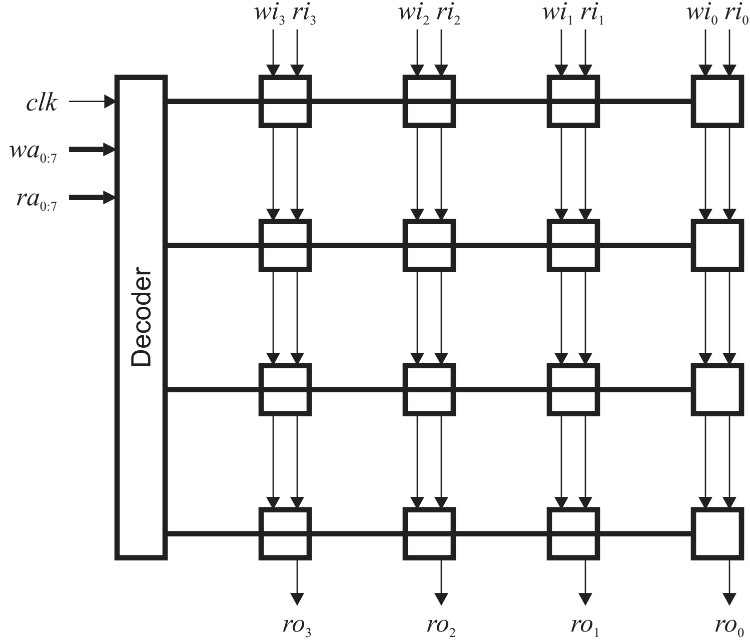


Figure 3.1: Parallel cell in memory mode.

fashion. Only the pipelined element is compatible with this design. We originally presented the serial cell in [32].

3.1.1 Parallel cell

The sixteen elements in the parallel cell are arranged into a 4×4 matrix. In memory mode, shown in Figure 3.1, the cell implements a 128×4-bit RAM with separate read and write ports. Each element manages a 32×1-bit portion of the memory. Input $ra_{0:7}$ specifies the 7-bit read address, with ra_7 serving as the read enable. Similarly, $wa_{0:7}$ specifies the write address and write enable. Lines $wi_{0:3}$ and $ro_{0:3}$ define the input data and output data, respectively. Finally, $ri_{0:3}$ specifies the default value of $ro_{0:3}$ when the read enable is off.

A separate decoder generates the required control signals for the matrix of elements. The lower four bits of $ra_{0:7}$ and $wa_{0:7}$ are passed to each element. The upper four bits are used to produce the read enable and write enable for each row of elements. The component may

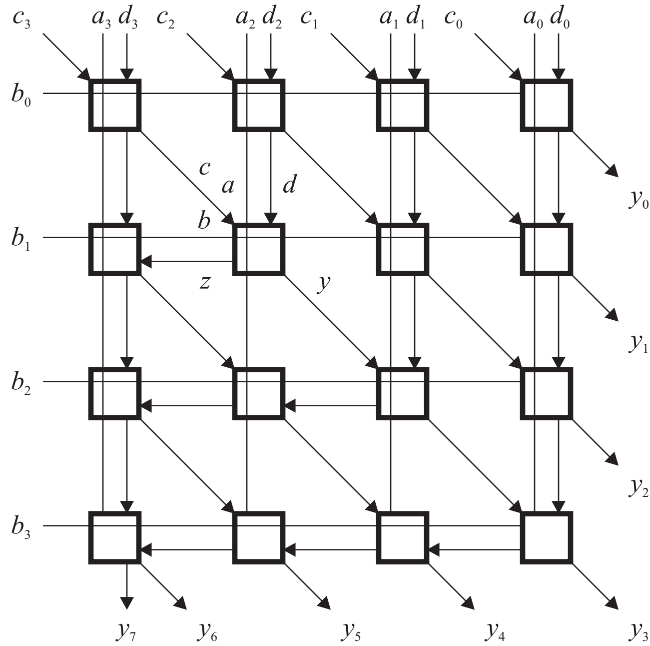


Figure 3.2: Parallel cell in mathematics mode.

gate the write enable with a global clock clk to isolate write operations from changes in the inputs.

In mathematics mode, shown in Figure 3.2, the cell reuses the elements to implement binary arithmetic. The matrix of elements assumes a structure resembling a carry-save multiplier [29]. Each element now acts as a 16×2-bit lookup table. The cell broadcasts inputs $a_{0:3}$ and $b_{0:3}$ across the rows and columns of the structure. Inputs $c_{0:3}$ and $d_{0:3}$ are passed to the four elements in the top row. The cell collects the output $y_{0:7}$ from the seven elements on the right and bottom.

Notice that the y and z outputs of each element usually connect to the c and d inputs of neighboring elements. Hence, elements generally receive the new values of a and b before the new values of c and d. The critical path through the structure consists of seven elements.

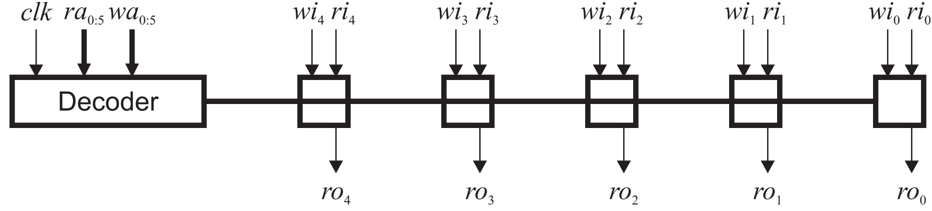


Figure 3.3: Serial cell in memory mode.

3.1.2 Serial cell

The serial cell reduces the area consumption by using five elements rather than sixteen. Figure 3.3 depicts this design in memory mode. The array of elements implements a 32-word RAM with separate read and write addresses. As before, each element manages a 32×1-bit piece of the total memory. With five elements, the memory works with 5-bit words rather than 4-bit words. The motivation behind this additional element will become clear shortly.

In the figure, $ra_{0:5}$ and $wa_{0:5}$ contain the read address and write address, with the most significant bits serving as enable signals. A decoder uses these inputs to generate the control signals for the five elements. The data lines include the input data $wi_{0:4}$, output data $ro_{0:4}$ and default output data $ri_{0:4}$.

Figure 3.4 illustrates the serial cell in mathematics mode. The array of elements now operates in bit-serial fashion with inputs $a_{0:4}$, b_i, $c_{0:4}$, and $d_{0:4}$. Input b_i as well as output y_i have bit-serial format. Like the parallel cell, elements act as a 16×2-bit lookup tables. Each computation consists of an initialization phase and nine execution phases. During initialization, the cell applies the initial values of the inputs to the elements. During execution, the cell takes away $c_{0:4}$ and $d_{0:4}$, and connects the elements into the structure shown. The elements then run for nine cycles to produce the output $y_{0:8}$. The pipeline latches inside the elements, denoted by hash marks in the figure, separate data from adjacent phases.

To clarify how the cell operates in mathematics mode, Figure 3.5 expands the chain

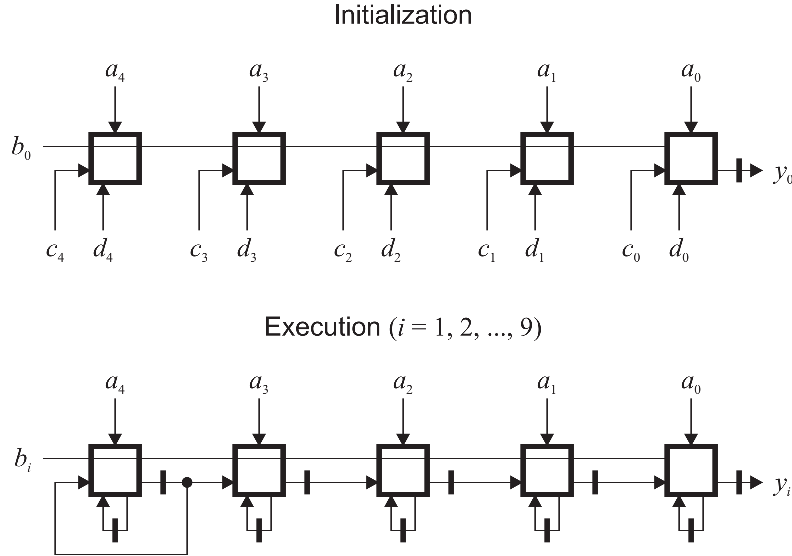


Figure 3.4: Serial cell in mathematics mode.

of elements into an equivalent parallel model. Each row of elements represents one phase. Observe that the structure resembles an array multiplier. The cell collects the first output y_0 when $i = 1$, and the final output y_8 when $i = 9$. The light-colored elements do not contribute to the results.

The serial cell uses the circuit described in Appendix A to subdivide the global clock *ck* into the initialization and execution phases. The control signals generated by this circuit drive the pipeline latches inside the elements.

3.2 Functions

This section demonstrates how the two cells can implement basic operations found in DSP. These functions include multiplication, addition and subtraction, bit shifting, memory access, and control logic. The cells also provide a means for reconfiguration. Table 3.1 summarizes the operations discussed in this section.

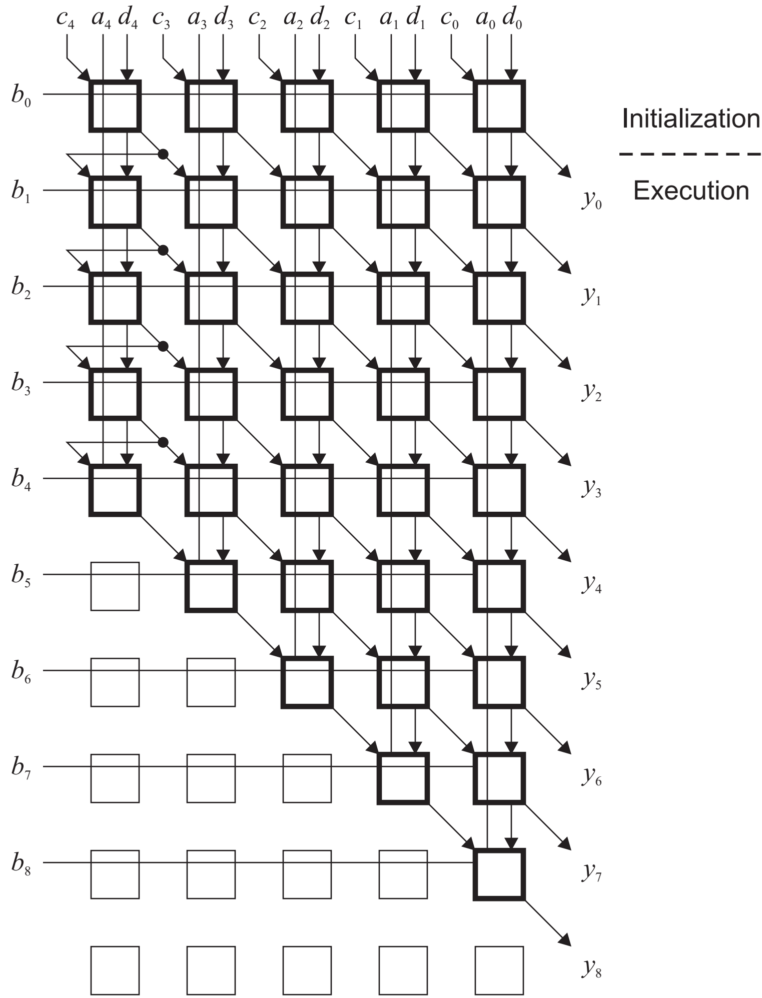


Figure 3.5: Equivalent parallel model of mathematics mode.

Table 3.1: Examples of cell operations.

Operation	Mode	Remarks
MAC	Mathematics	Unsigned or two's-complement
Add/subtract	Mathematics	Unsigned or two's-complement
Shift	Mathematics	Right or left
RAM	Memory	Can read and write simultaneously
Lookup table	Memory	Read operations
Logic	Both	AND-OR, decoders, multiplexers
Reconfiguration	Memory	Write operations

3.2.1 Multiplication

In mathematics mode, both the parallel cell and the serial cell assume a structure that resembles a multiplier. This property is not accidental, as multiplication is one of the most critical operations in DSP. In fact, both cells can evaluate the expression

$$y_{0:7} = (a_{0:3} \times b_{0:3}) + c_{0:3} + d_{0:3}. \tag{3.1}$$

This powerful multiply-accumulate (MAC) allows the reconfigurable architecture to implement word-length multipliers on the array of 4-bit cells.

To perform the MAC on unsigned data, the parallel cell configures each element to calculate

$$(2z + y) = (a \wedge b) + c + d. \tag{3.2}$$

Here, "+" denotes addition, and "$\wedge$" the logical AND. The cell can also compute the MAC on two's-complement data with the following four element configurations:

$$(2z + y) = (a \wedge b) + c + d \tag{3.3}$$

$$(-2z + y) = -(a \wedge b) + c - d \tag{3.4}$$

$$(-2z + y) = -(a \wedge b) - c + d \tag{3.5}$$

$$(-2z + y) = (a \wedge b) - c - d. \tag{3.6}$$

Further details about this subject appear in Chapter 5.

The serial cell implements the MAC in a slightly different manner. Collectively, the five elements compute the 5-bit MAC

$$y_{0:8} = (a_{0:4} \times b_{0:4}) + c_{0:4} + d_{0:4}. \tag{3.7}$$

Each input can encode a 4-bit unsigned number or a 4-bit two's-complement number without changing the function performed by the elements. Thus, the cell to work with a mixture of unsigned and two's-complement 4-bit inputs, a situation that occurs in word-length multiplier modules. Each element evaluates the expression

$$(2z + y) = (a \wedge b) + c + d, \tag{3.8}$$

regardless of the format of the inputs.

3.2.2 Addition and subtraction

Addition and subtraction are special cases of the MAC. In the parallel cell, setting $b_{0:3} = 1$ reduces (3.1) to

$$y_{0:7} = a_{0:3} + c_{0:3} + d_{0:3}. \tag{3.9}$$

This function allows the cell to add two 4-bit terms plus a carry in. For unsigned inputs, the cell configures elements in the top row to evaluate

$$(2z + y) = (a \wedge 1) + c + d = a + c + d, \tag{3.10}$$

and all other elements to evaluate

$$(2z + y) = (a \wedge 0) + c + d = c + d. \tag{3.11}$$

For two's-complement inputs, one can substitute $b_{0:3} = 1$ into the MAC cells presented in Chapter 5. Subtraction follows a similar process with $b_{0:3} = -1$.

Rather than redefining the functionality of each element, the serial cell implements addition and subtraction by fixing $b_{0:8}$ to a constant value in the hardware. This approach is

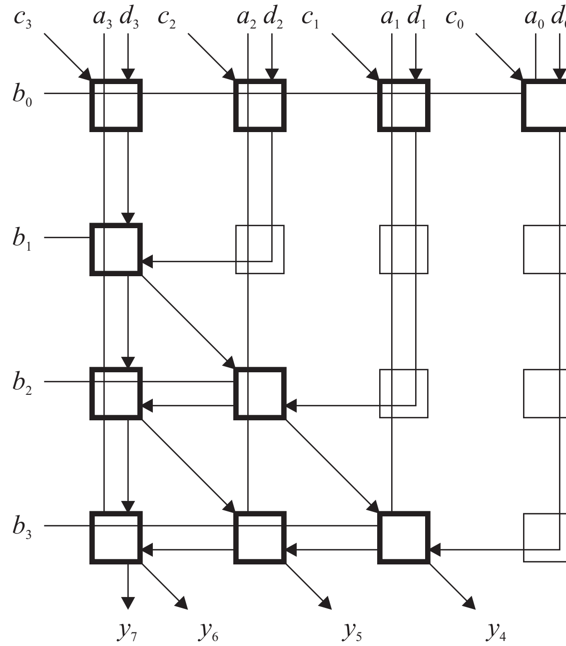


Figure 3.6: Parallel cell used as bit shifter.

necessary because elements must retain the same configuration across all execution phases. Each element computes the same function as in the 5-bit MAC.

3.2.3 Bit shifting

Another operation that appears in DSP is bit shifting. Figure 3.6 illustrates how the parallel cell can implement a 4-bit shifter in mathematics mode. The elements can shift $c_{0:3}$ left and shift in $d_{0:3}$, or equivalently shift $d_{0:3}$ right and shift in $c_{0:3}$. Inputs $a_{0:3}$ and $b_{0:3}$ control the operation of the shifter, and output $y_{4:7}$ contains the result.

To clarify how the shifter works, suppose the cell must shift $c_{0:3}$ two places so that $y_{4:7}$ is assigned to the string $\{c_2, c_3, d_0, d_1\}$. The control signals are set to the following values:

$$\begin{aligned} a_{0:3} &= 0000_2 \\ b_{0:3} &= 1111_2. \end{aligned} \tag{3.12}$$

Each element receives inputs $b = 1$ and $a = 0$. These inputs form a two-bit code that specifies the number of places to shift the inputs. The four elements in the top row act as two-way multiplexers, selecting the bits of $c_{0:3}$ and $d_{0:3}$ that appear in $y_{4:7}$. Thus, the outputs of these elements are d_0, d_1, c_2, and c_3. The six elements at the bottom left route these bits to the proper positions of $y_{4:7}$. The remaining elements in light gray pass data on without modification.

The serial cell offers a more straightforward approach to implement bit shifting. Figure 3.7 uses the expanded parallel model of the cell to depict this operation. Here, $b_{0:8}$ contains the shift data and $y_{4:8}$ the output. Each element acts as a two-way multiplexer controlled by a. When $a = 1$, the element passes the value of b diagonally to the outputs. When $a = 0$, the element uses the output of the previous element instead. With the appropriate bit asserted in $a_{0:4}$, the cell can copy any 5-bit substring of $b_{0:8}$ onto $y_{4:8}$.

3.2.4 Memory access

In memory mode, both medium-grain cells implement a random-access memory. A multi-stage algorithm can reserve a block of cells to store intermediate data. Having separate read and write ports allows the algorithm to load data from memory, process each sample, and store the results back into memory in a pipelined fashion. Notice that the write address need not be the same as the read address. As described in Chapter 4, the memory unit used by the Fast Fourier Transform relies on this functionality.

Memory mode has other uses as well. With appropriate address inputs, a cell can act as a fixed-length queue, delaying the input data for a predetermined number of cycles. A cell can also implement a read-only memory if the system defines the contents during reconfiguration. In this case, the memory might specify a lookup table for a series of constant coefficients.

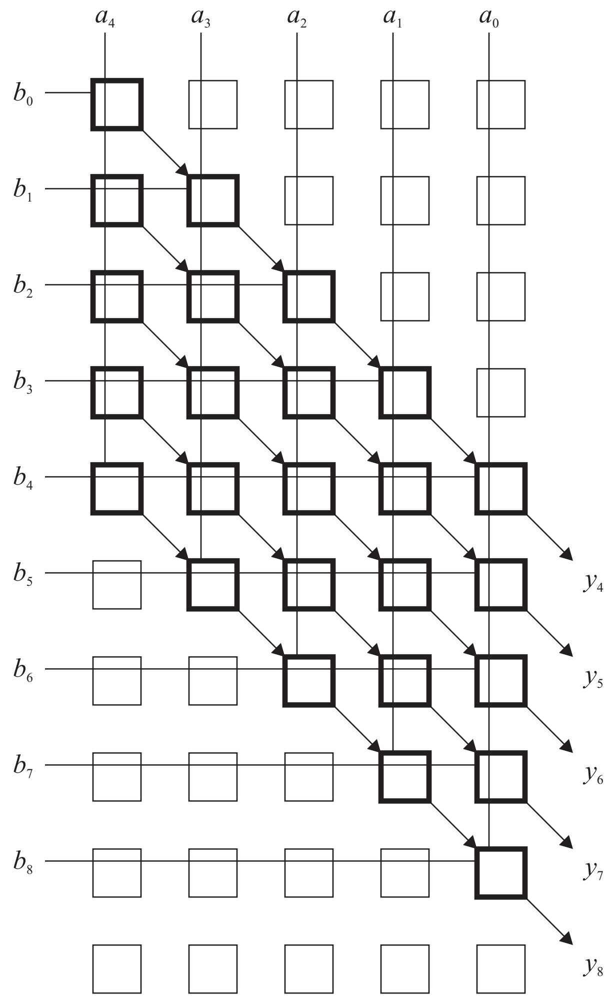


Figure 3.7: Serial cell used as bit shifter.

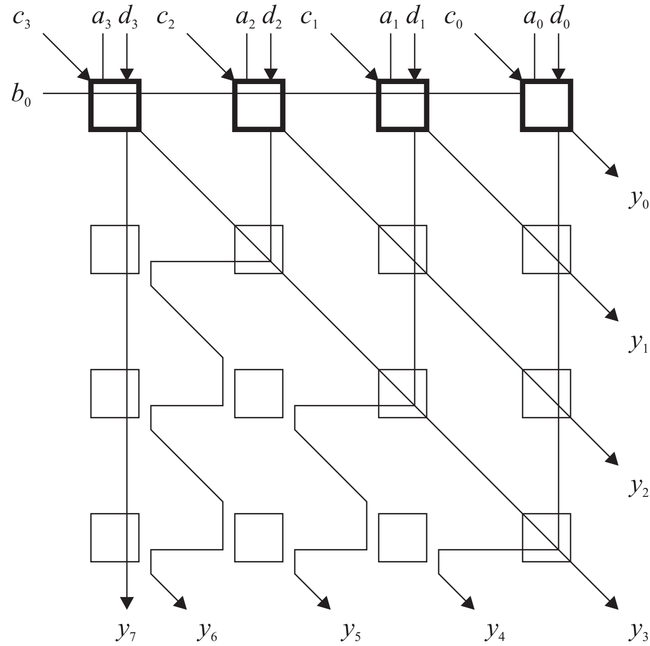


Figure 3.8: Parallel cell used to implement control logic.

3.2.5 Control logic

Besides binary arithmetic and memory access, algorithms require some control logic for proper operation. The parallel cell can evaluate basic logic expressions in both modes. In memory mode, the 128×4-bit memory can act as a lookup table for the desired function. In mathematics mode, the 16×2-bit memory inside each element can define two equations of up to four variables. The cell can configure the first row of elements to implement the desired function, and have the remaining elements pass the results to the outputs without modification. Figure 3.8 depicts this scenario. Possible functions include AND-OR expressions, decoders, and multiplexers.

The simplest way for the serial cell to implement control logic is to specify a lookup table in memory mode. Recall that the array of elements can act as a 32×5-bit memory. Hence, one cell can implement any combinational logic function with up to five inputs and five outputs.

3.2.6 Reconfiguration

Cells default to memory mode during reconfiguration so the system can load new data into the elements. Recall that each element contains 32 bits. The parallel cell requires 513 configuration bits: 512 for the sixteen elements plus 1 to select memory mode or mathematics mode. The serial cell requires 161 configuration bits: 160 for the five elements plus 1 for the mode.

One advantage of the serial cell is that mathematics functions generally use the same lookup table for each element. This property allows the system to write to all five elements simultaneously, saving time. However, the system must sometimes set the initial value of $b_{0:8}$ to use this optimization. Hence, the serial cell requires 42 configuration bits for mathematics mode: 32 for the elements, 1 for the mode, and 9 for $b_{0:8}$.

3.3 Analysis

This section compares the performance and flexibility of the proposed medium-grain cells. Recall that we presented three designs for the basic element in Chapter 2. The parallel cell is compatible with all three designs, whereas the serial cell must use pipelined elements. Hence, this analysis considers a total of four alternatives. We first present a series of circuit simulations detailing a worst-case operation. We then show layout simulations for two alternatives: the parallel cell with static elements, and the serial cell with pipelined elements. Finally, we analyze the functionality and overhead of the proposed cells.

3.3.1 Circuit simulations

For both cells, the worst-case operation occurs in mathematics mode. The critical path involves seven elements in the parallel cell, and nine execution phases in the serial cell. To measure the latency of a worst-case operation, we configured the elements to propagate data

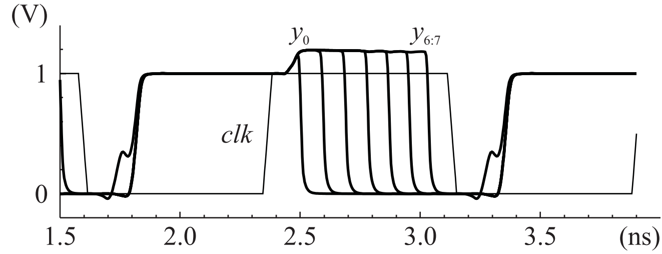


Figure 3.9: Simulation of parallel cell with dynamic elements.

along the critical path. We then performed circuit simulations in 90-nm CMOS.

Figure 3.9 contains the simulated output of the parallel cell with dynamic elements. When *clk* is low, the elements precharge all data lines to high. When *clk* is high, the elements enter the evaluation phase. The outputs initially rise above the supply voltage due to clock feedthrough. The calculated result is zero for this example, so all bits in $y_{0:7}$ fall low. Bit 0 evaluates first, followed by bits 1, 2, 3, and so forth. Bits 6 and 7 evaluate together because the last element generates both simultaneously. The total latency for this operation is 707 ps. Allowing 800 ps for the precharge phase and 800 ps for the evaluation phase, the cell can run at 625 MHz.

Figure 3.10 depicts the corresponding simulation for the parallel cell with static elements. Now the cell is driven by the input data rather than the clock. We configured the elements so that a worst-case operation occurs when the inputs transition from low to high. The inputs are represented by a_0 in the figure. The output evaluates one bit at a time, starting with y_0 and ending with $y_{6:7}$. Measured from a_0 to $y_{6:7}$, the total latency is 828 ps. Allowing 1 ns to account for latching and buffering, the cell can perform computations at 1 GHz.

We next simulated the serial cell with pipelined elements. As shown in Figure 3.11, the circuit generates the pulse train *pul* to demarcate the nine execution phases. This signal has a frequency of approximately 3.4 GHz. When *pul* is high, the cell collects the next bit of $y_{0:8}$. To examine the worst-case behavior, we configured the elements to alternate y_i between high and low. The glitches in y_i occur because the signal drives an output register. This

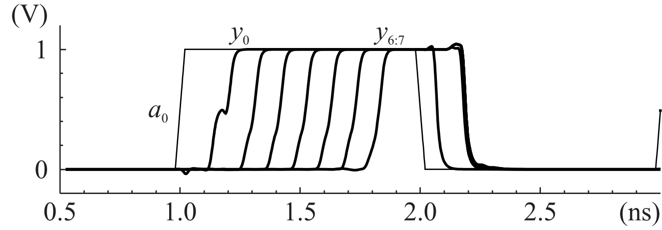


Figure 3.10: Simulation of parallel cell with static elements.

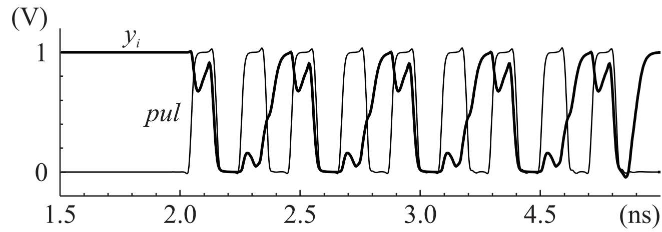


Figure 3.11: Simulation of serial cell with pipelined elements.

component lines up the bits in y_i. Overall, the serial cell requires 3 ns per computation, so the design runs at 333 MHz.

The final alternative combines the parallel cell with the pipelined elements. We proposed this superpipelined approach in [33]. Figure 3.12 illustrates one of the simulations from this paper, performed in 180-nm CMOS. Each pipelined element is configured to toggle the y and z outputs in response to the a, b, c, and d inputs. A complete operation encompasses seven of these pipeline stages, plus one extra stage for buffering. The worst-case latency through the element occurs between the pulse train *pul* and the low-to-high transition of y. In this simulation, *pul* has a frequency of 1.5 GHz. In 90-nm CMOS, *pul* runs at the same speed as the serial cell, or 3.4 GHz.

3.3.2 Layout simulations

Due to its simple clock approach, the static element offers a good counterpart to the parallel cell. We implemented this alternative in 180-nm CMOS and performed layout simulations

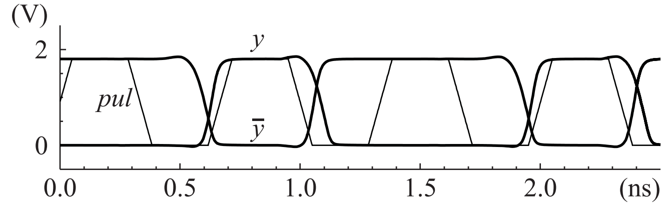


Figure 3.12: Simulation of parallel cell with pipelined elements.

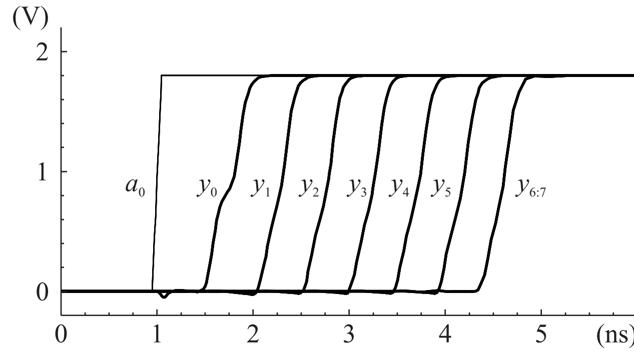


Figure 3.13: Layout simulation of parallel cell with static elements.

with all parasitic capacitances extracted. The simulated output appears in Figure 3.13. As with the circuit simulation, the bits of $y_{0:7}$ transition to high in sequence. The latency from a_0 to $y_{6:7}$ is slightly less than 3.75 ns, so the cell can run at 267 MHz.

Figure 3.14 depicts a similar layout simulation for the serial cell. The cell has been configured to perform the 4-bit MAC. In the simulation, the elements perform the following calculation:

$$(01111_2 \times 01010_2) + 01010_2 + 01010_2 = 01010\ 1010_2. \tag{3.13}$$

These values represent a worst-case situation, since y_i alternates between low and high. As shown, the elements produce the correct output. The pulse train *pul* has a frequency of 1.65 GHz; the cell runs at 167 MHz overall.

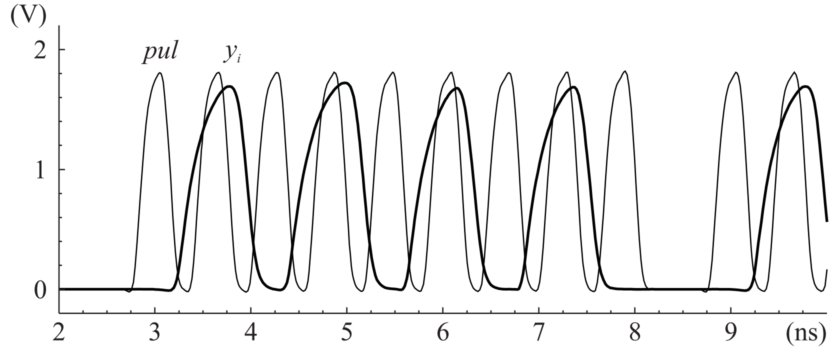


Figure 3.14: Layout simulation of serial cell with pipelined elements.

Table 3.2: Comparison of medium-grain cells.

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Word size (bits)	4	4	4	4–5
Memory per cell (words)	128	128	128	32
Memory configuration bits	513	513	513	161
Math configuration bits	513	513	513	42
Latency (cycles)	1	1	8	1
Frequency of circuit (MHz)	625	1000	3400	333
Frequency of layout (MHz)	∼167	267	∼1650	167
Area of layout (mm^2)		0.040		0.017

3.3.3 Comparison of cells

Table 3.2 compares the performance, area, and overhead of the medium-grain cells. We list the clock frequency for the 90-nm circuit simulations and the 180-nm layout simulations. For the parallel cell with dynamic and pipelined elements, we estimate the frequency of a 180-nm layout based on the results of other simulations. Clearly, the parallel cell achieves higher speed at the expense of area.

In terms of performance, the parallel cell with pipelined elements offers the highest clock frequency, but the latency from the inputs to the outputs is eight clock cycles. The dynamic and static elements allow the cell to compute one operation per cycle. The system can use the precharge phase of the dynamic element to transfer data between cells. In contrast, the

static element offers a higher clock frequency and a simpler clocking scheme.

The functionality of the two medium-grain cells is almost identical, as described in the previous section. However, the parallel cell implements a 512-word RAM, whereas the serial cell implements a 32-word RAM. Having more memory per cell offers an advantage when creating large memory units, but a disadvantage when configuring the cell. The serial cell has a very low configuration overhead for mathematics mode, since all elements generally have the same lookup table. In contrast, the system may have to configure each element separately in the parallel cell.

3.4 Verification

We have created three initial prototypes of the cells through the MOSIS Prototyping Service. The chips used a full-custom design flow with a modest 500-nm technology. Functional testing demonstrated that each design operates correctly. This section briefly describes the three prototypes.

3.4.1 Parallel-dynamic design

The first prototype, shown in Figure 3.15, implements a preliminary version of the parallel cell with dynamic elements. The elements use two datapaths: one for memory reads and writes, and the other for mathematics computations. After fabrication, we found that the current design for the dynamic element achieves better performance. However, the prototype operates in a similar manner. The large block in the center contains the 4×4 matrix of elements, whereas the other small blocks implement control circuitry.

Figure 3.16 contains a series of waveforms that demonstrate the functionality of the prototype. In this test, the circuit is first configured to implement a 4-bit multiplier. Recall that the configuration process uses standard write operations to load values into the lookup

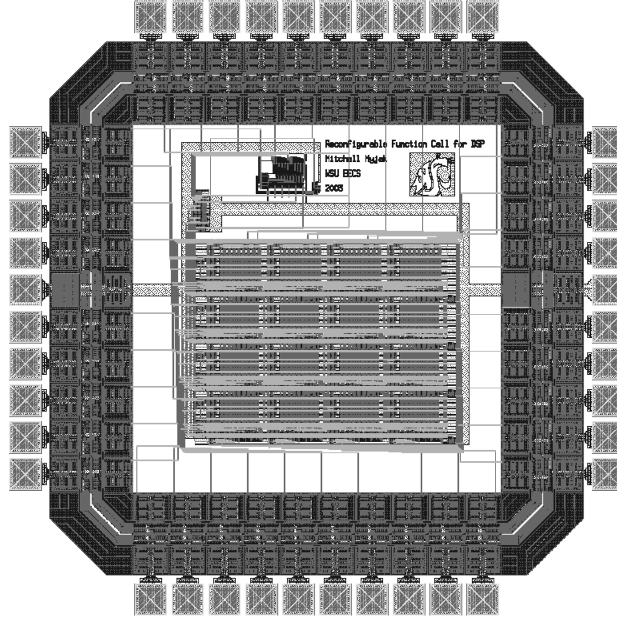


Figure 3.15: Prototype of parallel cell with dynamic elements.

tables. The circuit then calculates all perfect squares from 1×1 to 15×15, as listed in Table 3.3. The device functions correctly for all inputs, verifying both memory mode and mathematics mode.

3.4.2 Parallel-static design

The second prototype, shown in Figure 3.17, contains another version of the parallel cell. This design features the static elements described in Chapter 2. The transistor layout of the cell is very regular. We subjected the prototype to the same functional testing as the previous design. Figure 3.18 depicts the input and output waveforms as the prototype calculates the perfect squares from 1 to 15. Notice that mathematics computations operate in a combinational manner.

Table 3.3: Test cases to verify prototype.

Cycle	Operation	Output
1–65	Configuration	
66	$0 \times 0 = 0$	$0000\ 0000_2$
67	$1 \times 1 = 1$	$0000\ 0001_2$
68	$2 \times 2 = 4$	$0000\ 0100_2$
69	$3 \times 3 = 9$	$0000\ 1001_2$
70	$4 \times 4 = 16$	$0001\ 0000_2$
71	$5 \times 5 = 25$	$0001\ 1001_2$
72	$6 \times 6 = 36$	$0010\ 0100_2$
73	$7 \times 7 = 49$	$0011\ 0001_2$
74	$8 \times 8 = 64$	$0100\ 0000_2$
75	$9 \times 9 = 81$	$0101\ 0001_2$
76	$10 \times 10 = 100$	$0110\ 0100_2$
77	$11 \times 11 = 121$	$0111\ 1001_2$
78	$12 \times 12 = 144$	$1001\ 0000_2$
79	$13 \times 13 = 169$	$1010\ 1001_2$
80	$14 \times 14 = 196$	$1100\ 0100_2$
81	$15 \times 15 = 225$	$1110\ 0001_2$

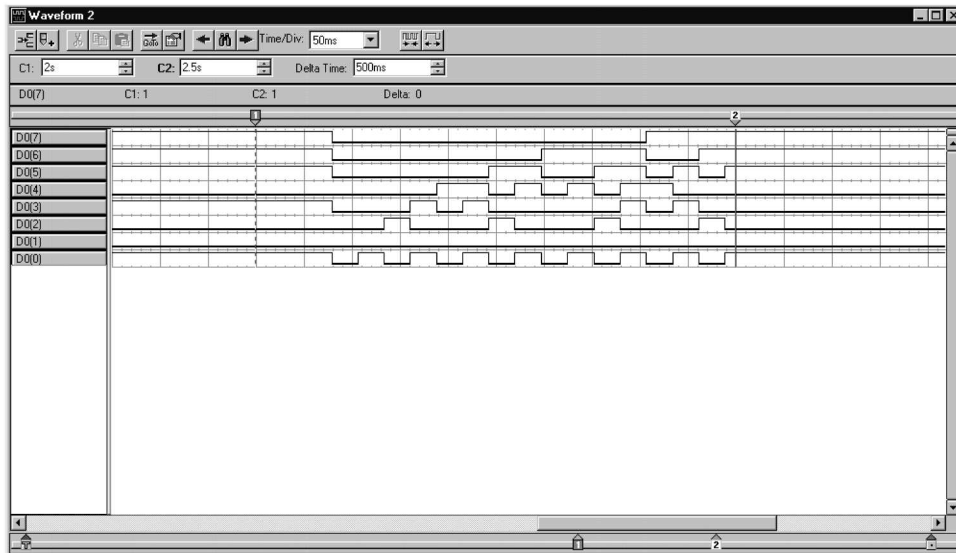


Figure 3.16: Verification of parallel cell with dynamic elements.

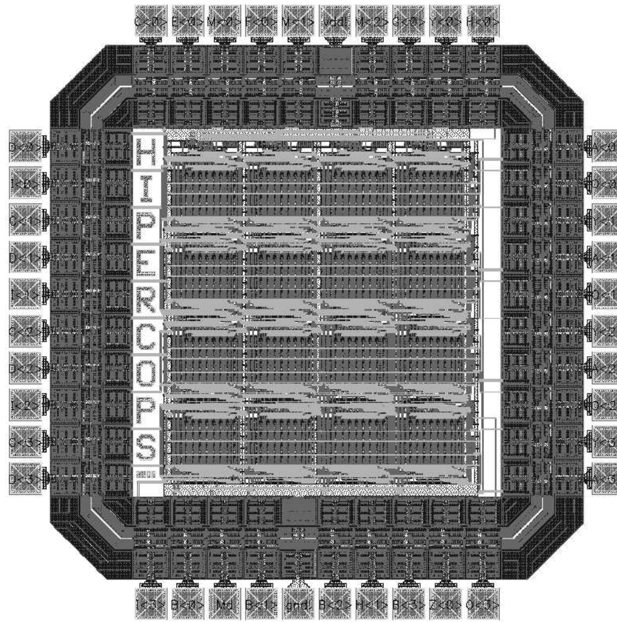


Figure 3.17: Prototype of parallel cell with static elements.

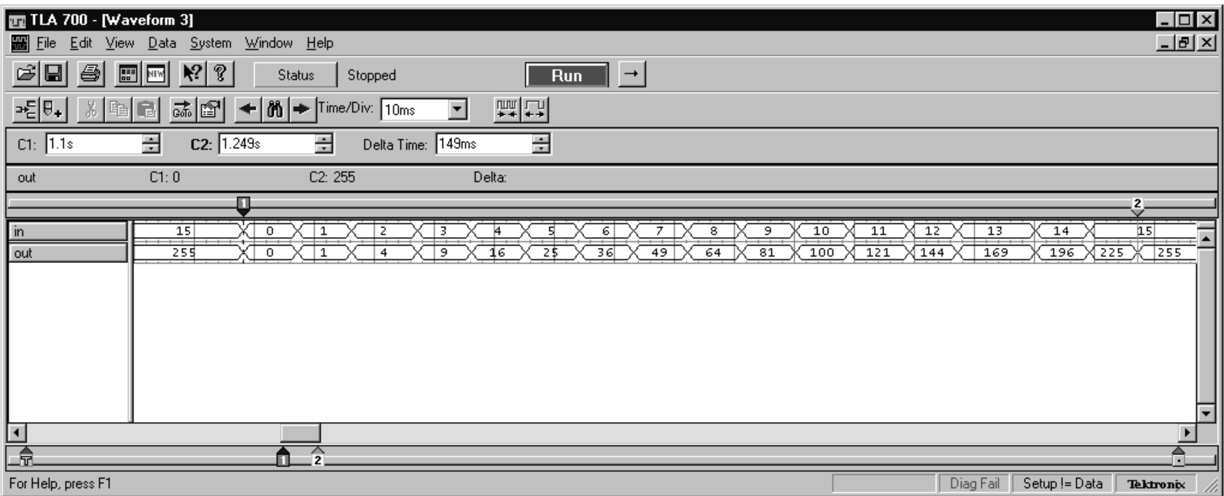


Figure 3.18: Verification of parallel cell with static elements.

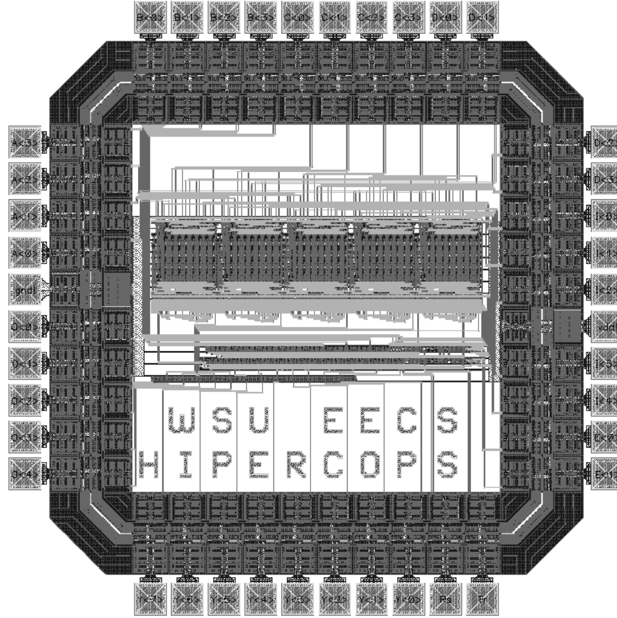


Figure 3.19: Prototype of serial cell with pipelined elements.

3.4.3 Serial-pipelined design

The third prototype, shown in Figure 3.19, implements the serial cell with pipelined elements. We optimized the cell to simplify functional testing. Hence, the chip does not contain all of the clock generator circuitry described in Appendix A. Two external pins, *reset* and *trigger*, specify the timing for mathematics computations. The chip does contain a pulse generator controlled by the rising edge of *trigger*. This circuitry produces the pulse train required to drive the pipeline latches. The latches are designed to hold their current state indefinitely when this signal is low. Thus, *trigger* can run at low frequency.

The test results demonstrated that the serial cell operates correctly. We configured the cell to perform a 4-bit MAC on two's-complement inputs, and had the chip compute a series of perfect squares, as before. Figure 3.20 illustrates the resulting waveforms. The output pins connect to a shift register that receives the bit-serial output one bit at a time. Notice that the mathematics computations require one initialization phase and nine execution phases.

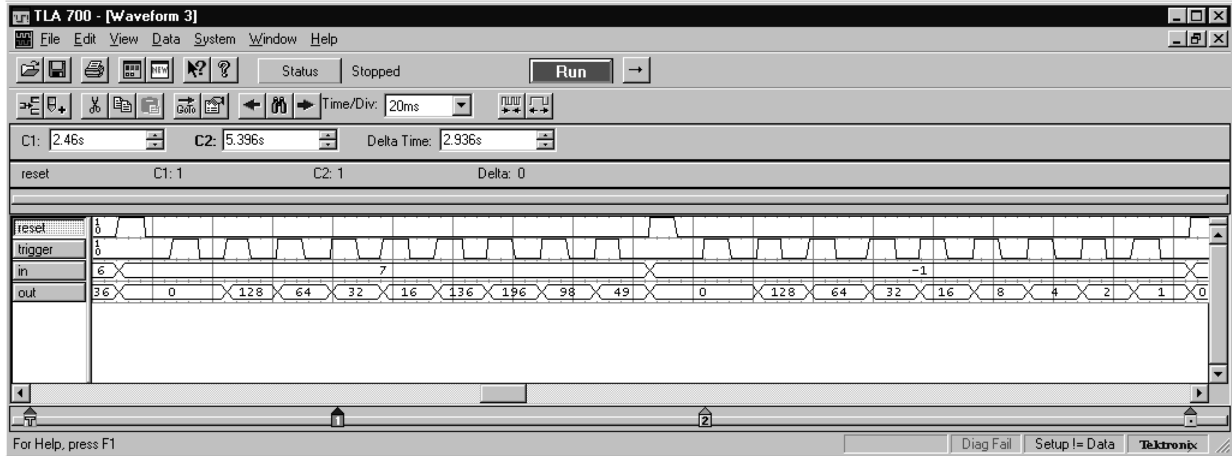


Figure 3.20: Verification of serial cell with pipelined elements.

3.5 Summary

This chapter has presented two novel 4-bit cells for the proposed reconfigurable architecture. The first design, which computes all results in bit-parallel fashion, offers the best performance but consumes a relatively large area. The second design, which computes binary arithmetic in bit-serial fashion, is significantly smaller but incurs a performance penalty. Both designs use an array of elements that can assume two configurations: one optimized for memory operations, and one optimized for binary arithmetic. Layout simulations in 180-nm technology indicate that the parallel cell with static elements runs at 267 MHz, whereas the serial cell with pipelined elements has an overall frequency of 167 MHz. Initial prototypes provide further functional verification of the designs. As described in Chapter 4, cells can be combined into word-length modules such as multipliers.

Chapter 4

Interconnections and Modules

Implementing DSP on the medium-grain reconfigurable architecture follows the three-step process outlined in Figure 4.1. First, the desired algorithm is divided into a series of modules, such as multipliers and adders. Other units may store intermediate data and control the execution of the algorithm. Next, each module is mapped onto a block of 4-bit cells. Cells within a block typically exchange partial results with neighboring cells. Finally, the modules are connected together across the interconnection network. This form of communication naturally occurs in word-length units.

We previously proposed a dual interconnection network that optimizes data transfer both within and between modules [34]. A local mesh of busses connects neighboring cells. Superimposed on the mesh is a global H-tree that routes data between modules. We recently enhanced the design to increase throughput and simplify the mapping process [35]. This chapter describes the interconnection network and illustrates how groups of cells can imple-

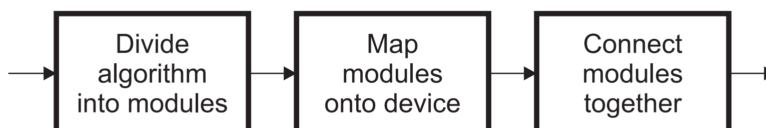


Figure 4.1: Steps for implementing DSP on the reconfigurable cell array.

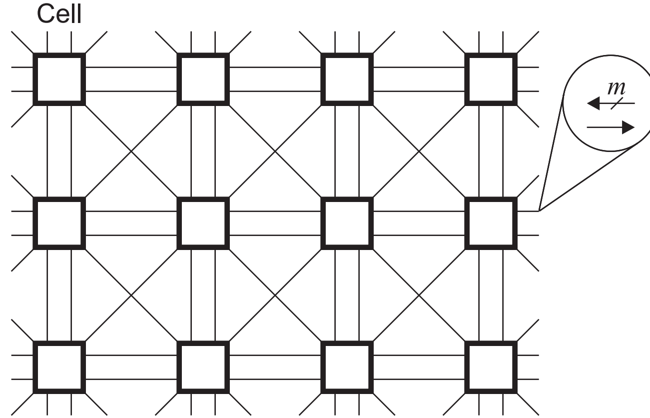


Figure 4.2: Local interconnection structure.

ment various modules. These modules form the building blocks for the algorithms discussed in Chapter 6.

4.1 Interconnection network

This section describes the dual interconnection network used in the medium-grain reconfigurable architecture.

4.1.1 Local mesh

Figure 4.2 depicts the local mesh. A series of m-bit busses allows cells to exchange data with their eight neighbors in the horizontal, vertical, and diagonal directions. The parallel cell uses $m = 4$, and the serial cell uses $m = 5$. All busses are unidirectional. The regularity of the structure supports modules of any size.

Figure 4.3 examines the interface between the cell and the interconnection network. Each input of the cell can connect to any incoming bus in the local mesh or the global H-tree. Similarly, each output of the cell can connect to any outgoing bus. Two crossbar switches perform the required routing. A control unit manages the reconfiguration process for the

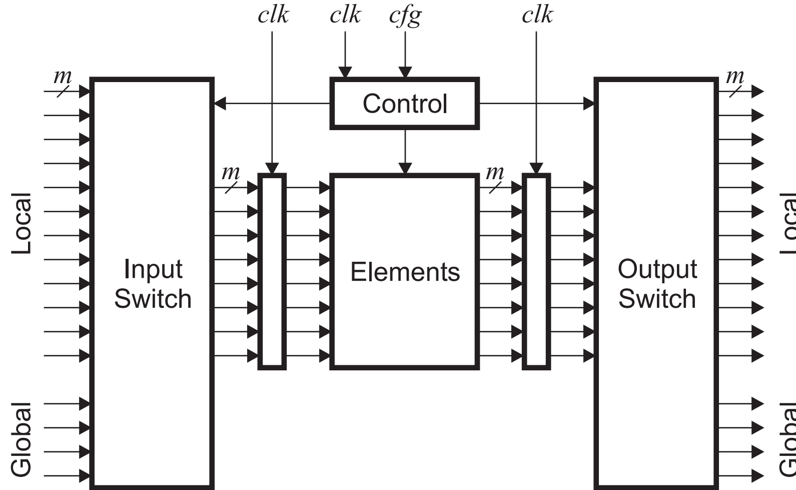


Figure 4.3: Interface between cell and interconnection network.

cell and the crossbar switches. The pipeline registers shown in the figure will be described shortly.

Some output lines of the cell are simply copies of the corresponding inputs. Hence, the local network can route data through a chain of cells to its final destination. This property is useful when multiple cells in a module are driven by the same inputs.

4.1.2 Global H-tree

Figure 4.4 illustrates the global H-tree. This structure can transfer data efficiently between any two cells on the array. Each level of the tree contains four input busses and four output busses. The number of bits per bus starts at m bits and doubles at every level. Thus, the H-tree resembles a fat-tree, which has been recognized as an efficient routing structure for parallel processing applications [36]. Data originating from a cell travels up the output path until it reaches the highest level required. The data then switches to the input path and descends to the destination cell. Hence, data routed between cells A and B traverses eight levels.

Figure 4.5 details a typical switch in the global H-tree. These switches route data in

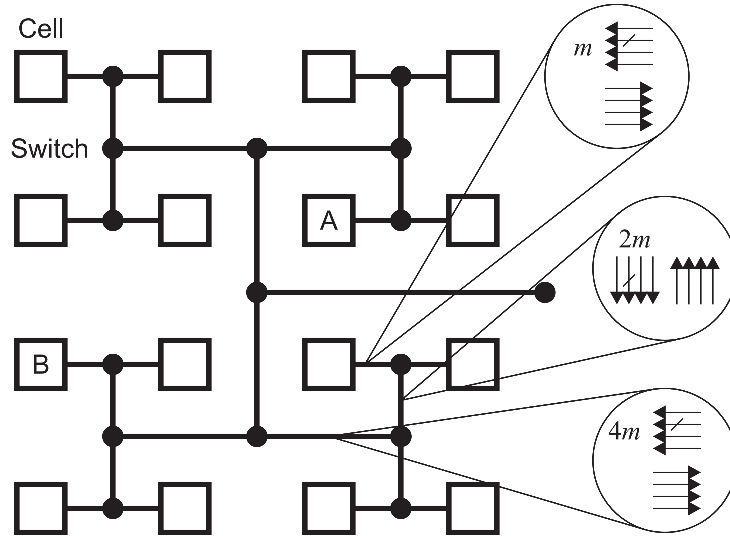


Figure 4.4: Global interconnection structure.

units of n bits, where n is a multiple of m. The architecture of each switch is identical; only the number of bits per bus changes. On the input path, the $2n$-bit busses from the upper level can be routed onto the n-bit busses in the two lower levels. The least significant and most significant n bits are handled separately. On the output path, each n-bit bus from the lower level can be transferred to an outgoing $2n$-bit bus. Alternatively, the switch can transfer data from the output path to the input path on the same level.

Each link between n-bit busses has the structure depicted in Figure 4.6. The link contains a series of programmable connections between the corresponding bits of each bus. Each m-bit portion of the link can be configured separately. This feature allows multiple links to connect to the same n-bit bus, provided that no two links drive the same m-bit portions. Modules sometimes need to merge two busses on the output path in this manner. In most cases, all m-bit portions of a link have the same state, so switches with larger n have about the same configuration time as switches with smaller n.

To save area, one could maintain a constant bus size after a predetermined number of levels. One could also use the techniques similar to those in [37] to fold the H-tree into an

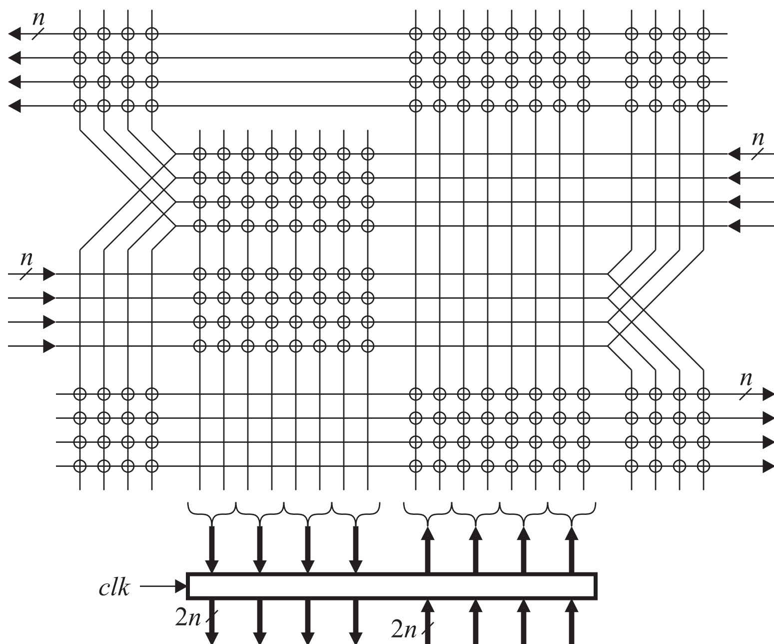


Figure 4.5: Typical switch in global interconnection structure.

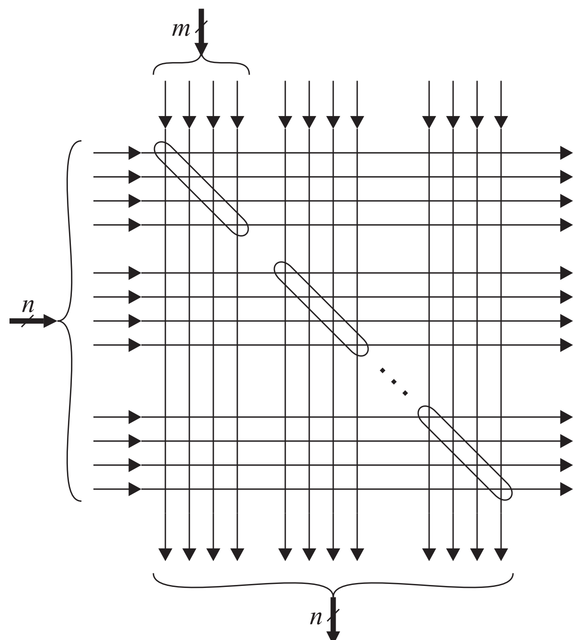


Figure 4.6: Link in global switch.

Table 4.1: Latency of cells and interconnection network.

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Latency of cell (cycles)	1	1	8	1
Latency of local mesh (cycles)	0	1	2	1
Latency of global H-tree (cycles per level)	0.5	0.5	1	0.5

smaller layout.

4.1.3 Pipelining

As shown in Figures 4.3 and 4.5, the interconnection network contains pipeline registers for high throughput. This technique allows the architecture to exploit the inherent parallelism of DSP. The exact pipelining scheme depends on the designs of the cell and element. However, all cell computations and data transfers occupy one or more pipeline stages. Table 4.1 lists the latencies of the cell, local mesh, and global H-tree. Initial simulations indicate that the interconnection network can match the throughput of the cell with these parameters.

For example, consider the parallel cell with static elements. The cell completes all operations within one pipeline stage. The local mesh also consumes one cycle to transfer data between cells. The latency of the global H-tree depends on the number of levels traversed between cells. Here, each level requires half a cycle, so the latency between cells A and B in Figure 4.4 would be four cycles.

The other designs for the cell and element have somewhat different latencies. With the dynamic element, the local mesh can transfer data during the precharge phase of the clock cycle. Hence, the latency is effectively zero. The parallel cell with the pipelined element has increased latency for all operations, due to the increased clock frequency.

The local network contains some extra pipeline registers that cells can use to delay certain inputs or outputs. This feature is useful in some of the modules described next.

4.2 Modules

This section describes how groups of cells can implement modules such as multipliers, adders, memory units, and control logic. We give examples of both fixed-point and floating-point arithmetic.

4.2.1 Multiplier

Most algorithms for DSP require some form of multiplication. Depending on the target application, the algorithm may perform unsigned or signed multiplication of 16-bit, 20-bit, 24-bit, 32-bit, or larger numbers. The 4-bit cells enable applications to implement a multiplier of the required size, while exploiting the inherent parallelism of the operation.

Suppose the reconfigurable device must multiply two 16-bit inputs $A_{0:15}$ and $B_{0:15}$ to generate the output $Y_{0:31}$. The unit is to operate in parallel for maximum performance. Figure 4.7 illustrates one possible approach. The structures shown in (a) and (b) are superimposed onto a 4×4 array of cells. The interconnection scheme scales easily to form n-bit multipliers with $(n/4)^2$ cells (assuming n is a multiple of 4). Notice the similarity between this approach and the parallel cell in Chapter 3.

As shown in part (a), the H-tree passes 4-bit portions of $A_{0:15}$ and $B_{0:15}$ to cells on the top and right. These lines are transferred horizontally and vertically to the $a_{0:3}$ and $b_{0:3}$ inputs of each cell. Typical cells compute the 4-bit MAC

$$y_{0:7} = (a_{0:3} \times b_{0:3}) + c_{0:3} + d_{0:3}. \tag{4.1}$$

The local mesh connects the $y_{0:3}$ and $y_{4:7}$ outputs to the $c_{0:3}$ and $d_{0:3}$ inputs of neighboring cells. As shown in part (b), the H-tree collects the 4-bit portions of $Y_{0:31}$ from the cells on the right and bottom.

The interconnection network automatically pipelines the multiplier into 4-bit portions.

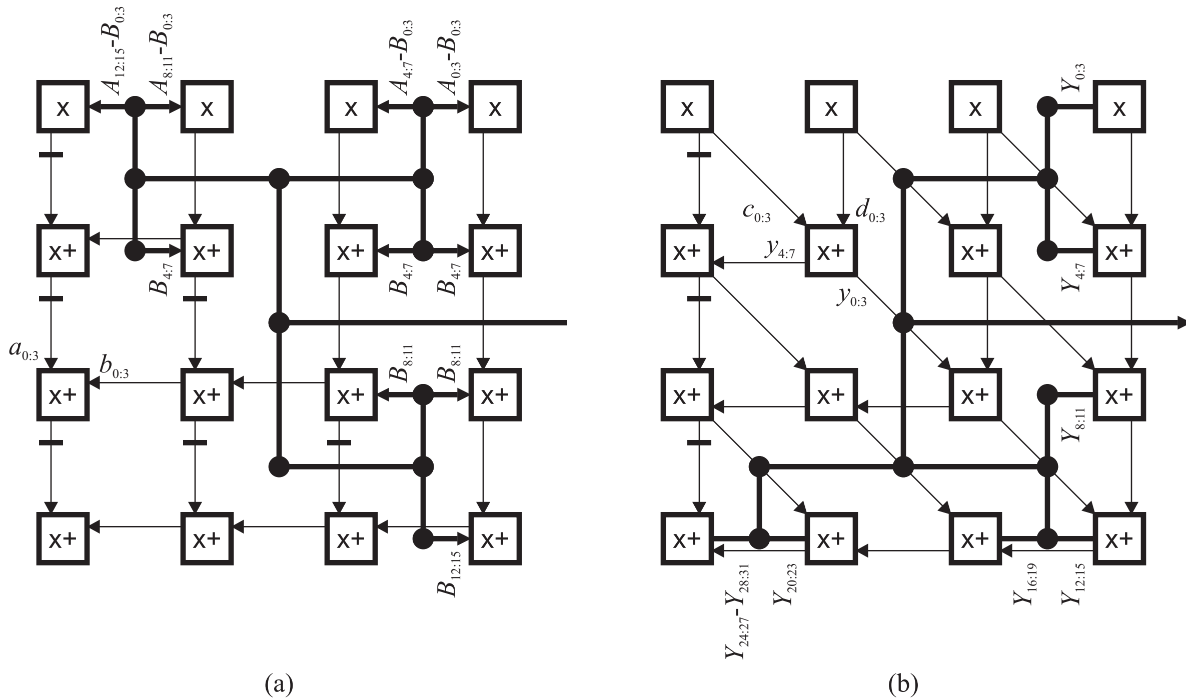


Figure 4.7: 16-bit multiplier.

The hash marks in Figure 4.7 denote places where the cell must delay some outputs to align the data in subsequent pipeline stages. The total latency through the multiplier depends on the design of the cell and the element, but the module can initiate one operation per clock cycle. The critical path involves a chain of 7 cells, or $(n/2 - 1)$ cells in general. Notice that the 4-bit portions of $Y_{0:31}$ are generated across multiple pipeline stages.

Chapter 5 gives more details about mapping multipliers onto the array of cells. With various cell configurations, the module can handle both unsigned and two's-complement inputs.

4.2.2 Multiply-accumulate unit

The cells in the top row of Figure 4.7 perform multiplication but not addition. Connecting the $c_{0:3}$ and $d_{0:3}$ inputs of these cells to additional inputs $C_{0:15}$ and $D_{0:15}$, the module would

evaluate the 16-bit MAC

$$Y_{0:31} = (A_{0:15} \times B_{0:15}) + C_{0:15} + D_{0:15}. \tag{4.2}$$

4.2.3 Adder and subtracter

Most algorithms require addition as well as multiplication. In some cases, an addition may be combined with another multiplication and implemented with the MAC unit described previously. However, other situations require dedicated adders.

The module in Figure 4.8 uses eight cells to add 32-bit inputs $A_{0:31}$ and $B_{0:31}$. Parts (a) and (b) are superimposed, as before. In general, an n-bit adder requires a chain of $(n/4)$ cells. Wrapping the structure into a rectangular shape achieves better utilization of the interconnection network. The H-tree passes 4-bit portions of these inputs to each cell. Cells then add the two terms, along with a carry-in:

$$y_{0:7} = a_{0:3} + b_{0:3} + c_{0:3}. \tag{4.3}$$

The local mesh routes $y_{4:7}$ to the carry-in of the next cell. The H-tree collects $y_{0:3}$ into the final result, $Y_{0:35}$.

As with the multiplier and MAC unit, the interconnection structure pipelines the adder into 4-bit portions. Note that $A_{0:31}$ and $B_{0:31}$ should arrive in a staggered fashion: $A_{0:3}$ and $B_{0:3}$ first, $A_{4:7}$ and $B_{4:7}$ next, and so forth. Many modules described in this section impose similar requirements on the inputs. The output $Y_{0:35}$ is generated in the same order.

With suitable configurations of each cell, the module can add or subtract inputs in unsigned or two's-complement format.

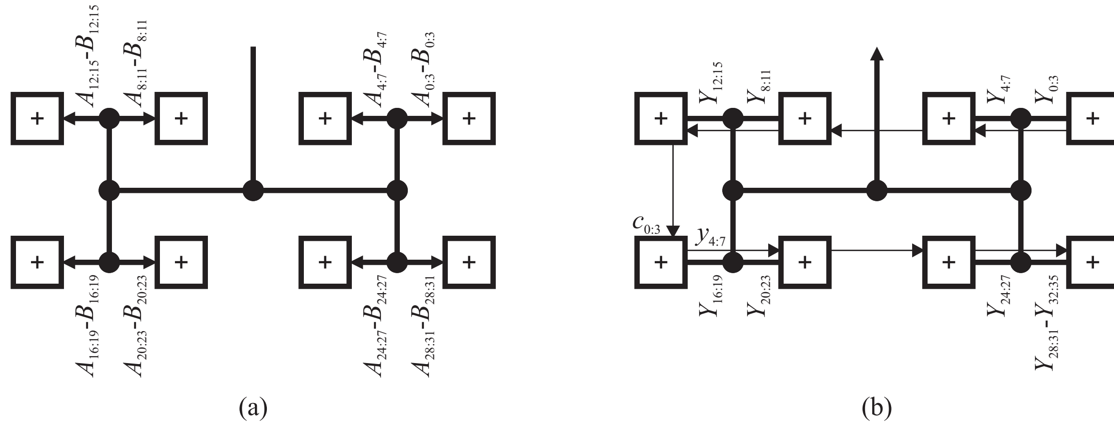


Figure 4.8: 32-bit adder.

4.2.4 Shifter

Shifters are a necessary component of many computations, including floating-point arithmetic and CORDIC rotations. One can implement both linear and logarithmic shifters on the reconfigurable cell array. A linear shifter has a simple structure that resembles the multiplier module. However, a logarithmic shifter usually requires fewer cells, as in the 16-bit left shifter in Figure 4.9. An n-bit module occupies a $(n/4) \times \lceil \log_2(n/2) \rceil$ block of cells.

The logarithmic shifter contains three rows. The first row shifts the input $X_{0:15}$ from zero to four bits. Each cell implements the bit shifter described in Chapter 3. The second and third rows apply optional shifts of four and eight bits, respectively. Here, each cell implements a two-way multiplexer. A control word $S_{0:7}$ determines the number of places to shift.

Part (a) in the figure shows the inputs and outputs of the module, whereas part (b) details the internal connections. Although the local mesh handles most communication within the module, the H-tree does route some lines between the second and third rows. This situation arises because data must travel between non-neighboring cells. The second row delays all other outputs to match the additional latency of the H-tree.

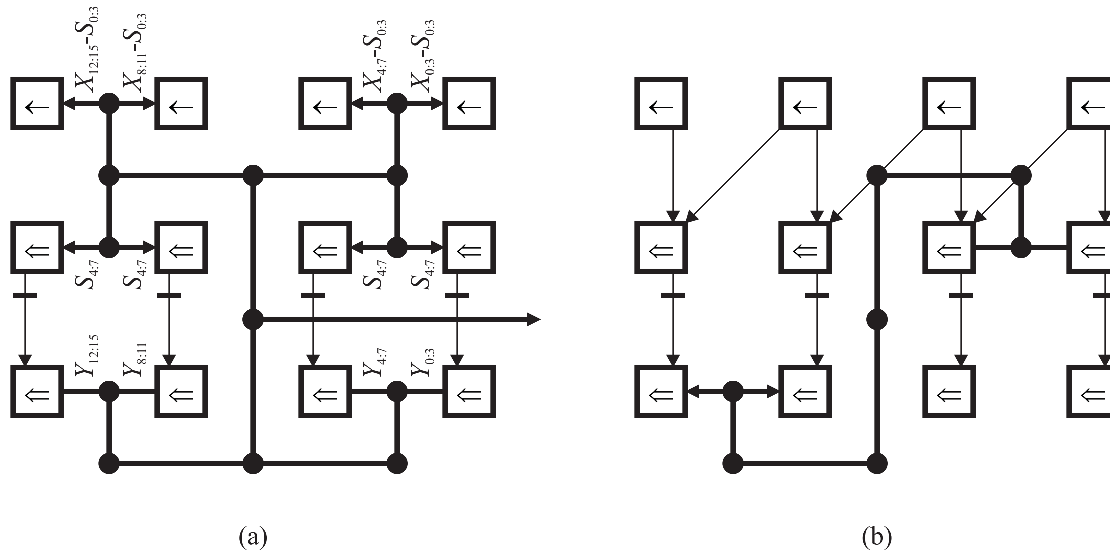


Figure 4.9: 16-bit left shifter.

4.2.5 Memory unit

Recall from Chapter 3 that cells can implement a memory with separate read and write addresses. The parallel cell has a 128×4-bit capacity, whereas the serial cell has a 32×5-bit capacity. Each cell divides the address space into two banks. Often, one bank implements a read-only memory, while the other bank implements a write-only memory. This setup allows algorithms to read data from the first bank, perform some calculation, and store results into the second bank. The next stage of the algorithm then reads from the second bank and writes to the first bank.

Figure 4.10 demonstrates one way to combine multiple cells into a dual-port memory. The module contains twelve memory cells, labeled "M", and four decoder cells, labeled "D". Input $RA_{0:7}$ addresses the read memory, and $WA_{0:7}$ addresses the write memory. The decoder cells use $RA_{4:7}$ and $WA_{4:7}$ to enable rows of memory cells for reading and writing. The local mesh transfers the remaining bits to each cell. The output data propagates downwards, while the input data propagates upwards. All operations occur in a pipelined fashion.

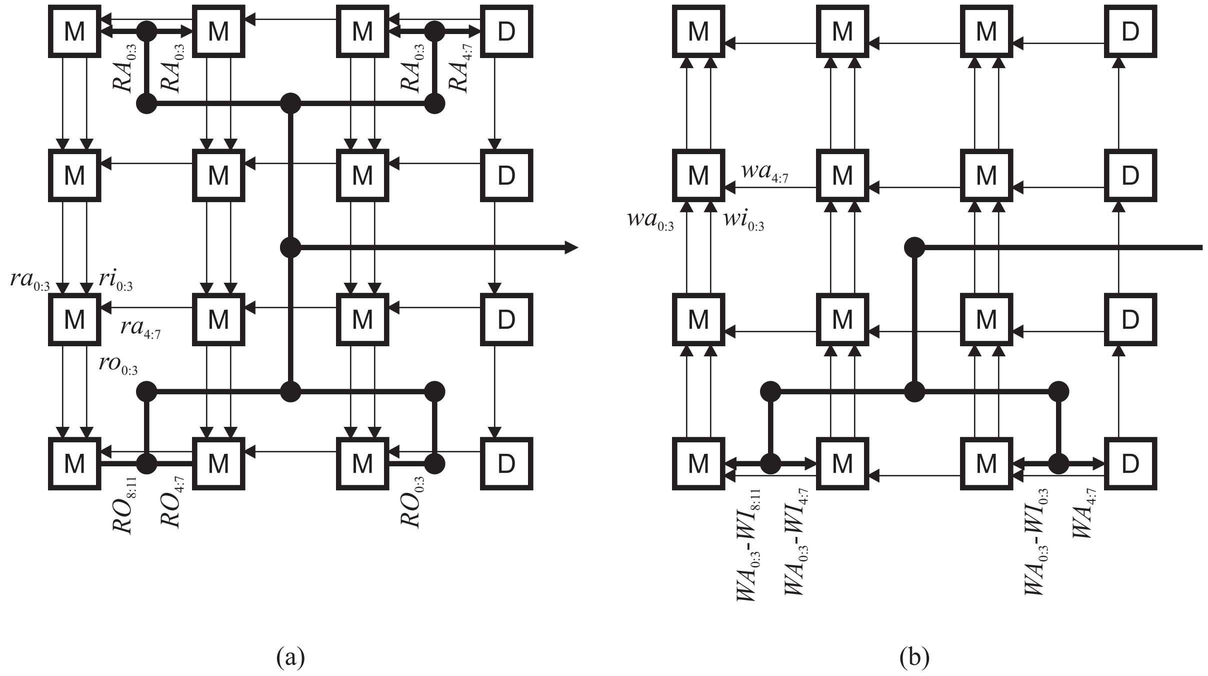


Figure 4.10: Dual-port memory unit.

With the parallel cell, the read and write memories have a 256×12-bit capacity. With the serial cell, the memories have a 64×15-bit capacity. Larger memory units simply require a larger block of cells.

4.2.6 Specialized memory unit

In Chapter 6, we describe a radix-4 implementation of the Fast Fourier Transform (FFT). The kernel of this algorithm simultaneously reads four samples from memory and writes four results back into memory. We developed the module shown in Figure 4.11 to support this operation. A 4×4 block of cells can manage a 4-bit portion of all eight inputs and outputs. As with the previous design, reads and writes operate independently. Each row of cells implements a read memory for one of the inputs to the kernel. Each column, in turn, implements a write memory for one of the outputs. A separate module, not shown, generates the necessary control signals to enable individual rows and columns.

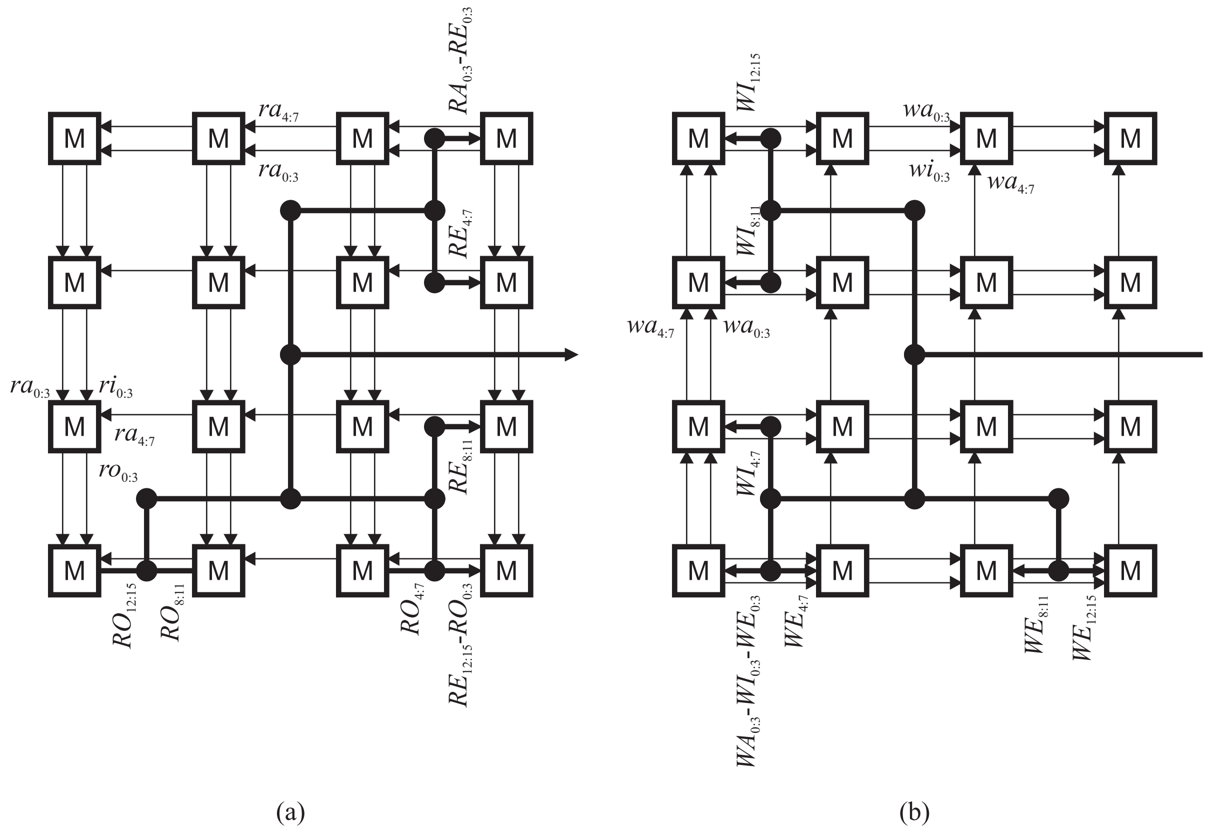


Figure 4.11: Memory unit for Fast Fourier Transform.

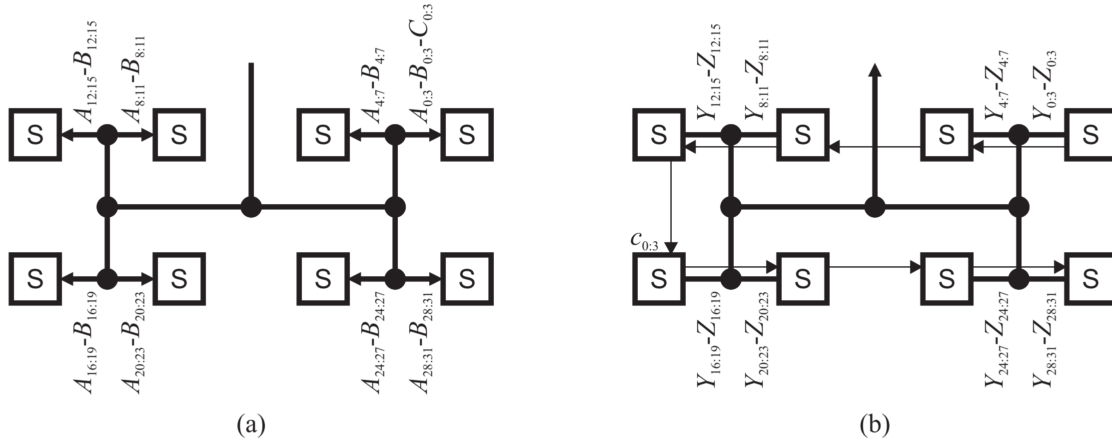


Figure 4.12: 32-bit exchange unit.

4.2.7 Control logic

Control logic presents a problem for many reconfigurable devices that perform DSP. Architectures that place a fixed number of functional units in each cell may not be able to evaluate arbitrary logic expressions efficiently. Some systems work around this problem by supplementing the reconfigurable device with a separate microprocessor: the microprocessor handles the control operations, whereas the reconfigurable device executes the mathematical functions [1]. In contrast, the proposed architecture has both coarse-grain and fine-grain flexibility. Algorithms can map control logic alongside other modules.

To give an example, Figure 4.12 illustrates a unit that exchanges two 32-bit inputs $A_{0:31}$ and $B_{0:31}$ in response to a control signal $C_{0:3}$. The outputs of the module are $X_{0:31}$ and $Y_{0:31}$. A 32-bit multiplexer would have a similar structure.

4.2.8 Floating-point adder

Although the modules described so far use fixed-point arithmetic, the reconfigurable cell array can also implement floating-point arithmetic. For example, Figure 4.13 illustrates a floating-point adder that operates on inputs x and y. A comparator first determines the

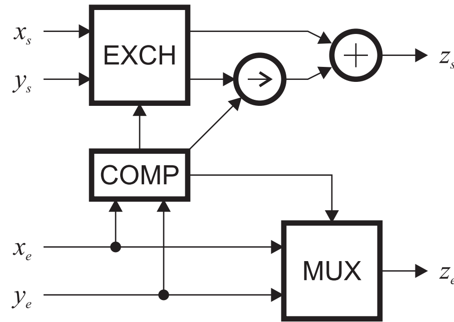


Figure 4.13: Diagram of floating-point adder.

difference between the exponents x_e and y_e. This result is used to align the significands x_s and y_s by shifting one or the other to the right. A fixed-point adder computes the sum, while a multiplexer selects the larger of the two exponents. For completeness, the module should also realign the result, but this operation is best performed after completing all floating-point manipulations.

Although one could design an adder to work with floating-point numbers in IEEE format, a hybrid representation reduces the hardware required for individual operations. Suppose that numbers contain a 28-bit denormalized significand and a 10-bit exponent, both in two's-complement format. Also assume that the last two bits of the exponent are always zero. This property implies that the significand is only shifted in 4-bit units, reducing the size of the shifter. Conversion to and from IEEE single-precision format would be straightforward.

Figure 4.14 shows the resulting implementation of the floating-point adder. The structure requires 52 cells, or the equivalent of a 208-bit fixed-point adder. The critical path runs from x_e and y_e to z_s.

4.2.9 Floating-point multiplier

A diagram of a floating-point multiplier appears in Figure 4.15. This operation also reduces to fixed-point manipulations. The structure multiplies the two significands x_s and y_s, and adds the two exponents x_e and y_e. The final stages realign the result by shifting the signifi-

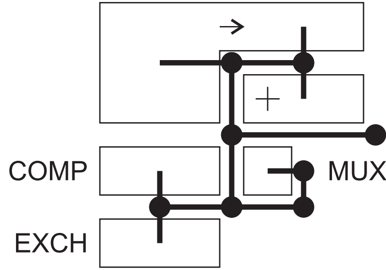


Figure 4.14: Implementation of floating-point adder.

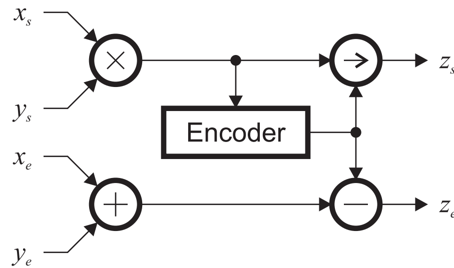


Figure 4.15: Diagram of floating-point multiplier.

cand left and subtracting the appropriate value from the exponent. An encoder determines the appropriate degree of shifting.

Figure 4.16 depicts the implementation of the floating-point multiplier. This example uses the same number format as the floating-point adder. Notice that the outputs of the multiplier connect directly to the encoder for high efficiency. The critical path runs from x_s and y_s to z_s. Much of the latency originates from the dependency of the shift register on the final output of the multiplier and the encoder.

4.3 Summary

In this chapter, we have described how the interconnection network partitions the medium-grain cell array into a hierarchy of modules. Each module performs a word-length operation such as 16-bit multiplication. A local mesh of 4-bit busses connects neighboring cells within

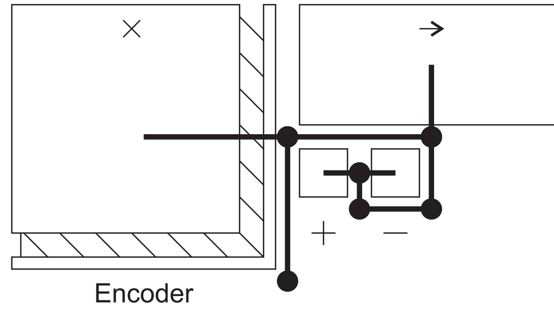


Figure 4.16: Implementation of floating-point multiplier.

a module, and a global H-tree transfers data between modules in word-length units. In addition, we have shown the structure of several modules, including multipliers, adders, and memory units. The interconnection network pipelines all cell computations and data transfers, allowing the architecture to achieve high throughput.

Chapter 5

Hierarchical Multipliers

Multiplication is one of the most critical operations in DSP. Devices that cannot implement this operation effectively will incur significant performance penalties. In this chapter, we demonstrate how to map n-bit multipliers onto an array of m-bit cells. This architecture is a generalization of the proposed reconfigurable cell array, in which $m = 4$. We consider modules that work with unsigned and two's-complement data. We then describe the necessary configurations for each cell. The following sections parallel the discussion in [38], but with some differences in terminology.

5.1 Structure

In this section, we develop an efficient structure to implement an n-bit multiplier on an array of m-bit cells. Denoting the inputs as $A_{0:n-1}$ and $B_{0:n-1}$ and the output as $Y_{0:2n-1}$, the module performs the operation

$$Y_{0:2n-1} = (A_{0:n-1} \times B_{0:n-1}). \tag{5.1}$$

We are not concerned with the data format at this time.

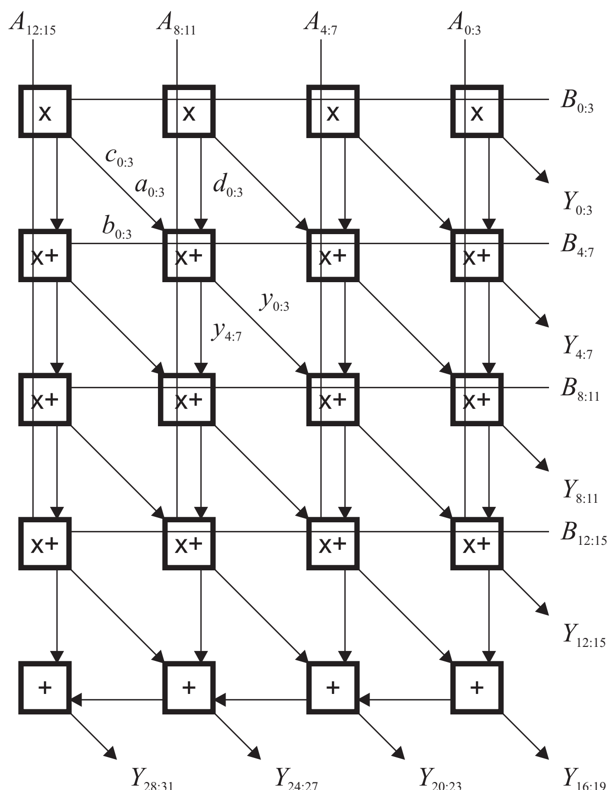


Figure 5.1: Carry-save multiplier.

5.1.1 Carry-save multiplier

As an initial approach, consider the structure of a carry-save multiplier [29]. Figure 5.1 illustrates this approach for $n = 16$ and $m = 4$. The module uses a rectangular array of $\lceil n/m \rceil$ by $\lceil n/m \rceil + 1$ cells. The two inputs are divided into m-bit portions and broadcast across the rows and columns of the array.

Cells in the top row multiply two m-bit portions of $A_{0:n-1}$ and $B_{0:n-1}$, passing the upper and lower portions of result in separate directions. Cells in the middle rows perform the same multiplication and may add one or two m-bit terms to the result. This function is known as a multiply-accumulate (MAC). Finally, cells in the bottom row add two or three m-bit terms together. As a function of n and m, the carry-save multiplier requires $\lceil n/m \rceil^2 + \lceil n/m \rceil$ cells. The critical path contains $2 \lceil n/m \rceil$ cells.

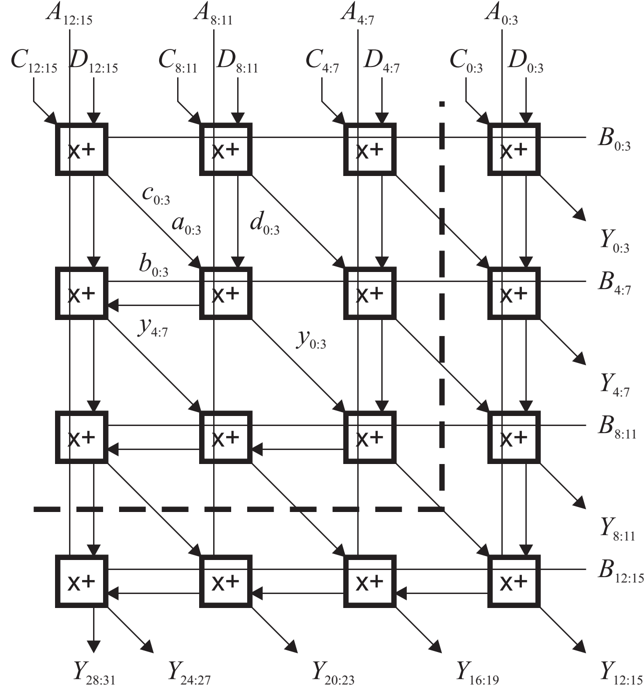


Figure 5.2: Proposed multiply-accumulate unit.

5.1.2 Proposed design

We can reduce the size of the multiplier by having all cells implement the worst-case MAC. This enhancement is beneficial since reconfigurable devices typically contain an array of identical cells. Figure 5.2 depicts the resulting structure for $n = 16$ and $m = 4$. Notice that two additional terms $C_{0:n-1}$ and $D_{0:n-1}$ can be incorporated into the top row of cells. This enhancement allows the module to evaluate the n-bit MAC

$$Y_{0:2n-1} = (A_{0:n-1} \times B_{0:n-1}) + C_{0:n-1} + D_{0:n-1}. \tag{5.2}$$

Each cell performs a similar MAC on m-bit words:

$$y_{0:2m-1} = (a_{0:m-1} \times b_{0:m-1}) + c_{0:m-1} + d_{0:m-1}. \tag{5.3}$$

In general, $a_{0:m-1}$ and $b_{0:m-1}$ represent two m-bit portions of A and B, whereas $c_{0:m-1}$ and $d_{0:m-1}$ denote the two terms added to the result. The design requires $\lceil n/m \rceil^2$ cells and has a critical path of $2\lceil n/m \rceil - 1$ cells.

5.1.3 Proof of functionality

To prove that the proposed MAC unit does implement the n-bit MAC in (5.2), we can use mathematical induction on n. For simplicity, we restrict n to multiples of m. The base case where $n = m$ consists of one cell that computes (5.3), or equivalently (5.2). For the inductive step, we assume that the structure works correctly for inputs of $(n - m)$ bits, and extend this result to n bits.

Suppose we partition the array of cells along the dashed line in Figure 5.2. Notice that the main group of cells has the same structure as an $(n - m)$-bit array. Denote the output of this portion with $Y'_{0:2(n-m)-1}$. By the inductive hypothesis,

$$Y'_{0:2(n-m)-1} = (A_{m:n-1} \times B_{0:n-m-1}) + C_{m:n-1} + D_{m:n-1}. \tag{5.4}$$

Now consider the remaining cells along the right and bottom edges. These cells multiply portions of $A_{0:n-1}$ and $B_{0:n-1}$ to produce the intermediate product

$$\begin{aligned} P_{0:2n-1} = &(A_{0:m-1} \times B_{0:n-m-1}) + 2^{n-m}(A_{0:m-1} \times B_{n-m:n-1}) \\ &+ 2^n(A_{m:n-1} \times B_{n-m:n-1}). \end{aligned} \tag{5.5}$$

The cells then add $P_{0:2n-1}$ to $Y'_{0:2(n-m)-1}$, together with $C_{0:m-1}$ and $D_{0:m-1}$. Collectively, the cells implement a two-input ripple-carry adder with m-bit stages. Hence, we can express

$Y_{0:2n-1}$ as follows:

$$\begin{aligned}Y_{0:2n-1} = (A_{0:m-1} \times B_{0:n-m-1}) + 2^{n-m}(A_{0:m-1} \times B_{n-m:n-1}) \\ + 2^{n}(A_{m:n-1} \times B_{n-m:n-1}) + 2^{m}Y'_{0:2(n-m)-1} + C_{0:m-1} + D_{0:m-1}.\end{aligned} \tag{5.6}$$

Substituting in (5.4),

$$\begin{aligned}Y_{0:2n-1} = (A_{0:m-1} \times B_{0:n-m-1}) + 2^{n-m}(A_{0:m-1} \times B_{n-m:n-1}) \\ + 2^{n}(A_{m:n-1} \times B_{n-m:n-1}) + 2^{m}(A_{m:n-1} \times B_{0:n-m-1}) \\ + 2^{m}C_{m:n-1} + 2^{m}D_{m:n-1} + C_{0:m-1} + D_{0:m-1},\end{aligned} \tag{5.7}$$

which simplifies to

$$\begin{aligned}Y_{0:2n-1} = (A_{0:m-1} + 2^{m}A_{m:n-1}) \times (B_{0:n-m-1} + 2^{n-m}B_{n-m:n-1}) \\ + (C_{0:m-1} + 2^{m}C_{m:n-1}) + (D_{0:m-1} + 2^{m}D_{m:n-1})\end{aligned} \tag{5.8}$$

$$= (A_{0:n-1} \times B_{0:n-1}) + C_{0:n-1} + D_{0:n-1}. \tag{5.9}$$

5.2 Data formats

The proposed MAC unit can work with data in both unsigned and two's-complement format. First we consider the unsigned case. Each cell evaluates the m-bit MAC in (5.3), where $a_{0:m-1}$, $b_{0:m-1}$, $c_{0:m-1}$, and $d_{0:m-1}$ range from 0 to $2^m - 1$. Note that $y_{0:2m-1}$ will not overflow since

$$(2^m - 1)(2^m - 1) + (2^m - 1) + (2^m - 1) = 2^{2m} - 1. \tag{5.10}$$

The two's-complement case is more complex. Consider dividing an n-bit number in two's-complement form into m-bit portions. The most significant portion has two's-complement

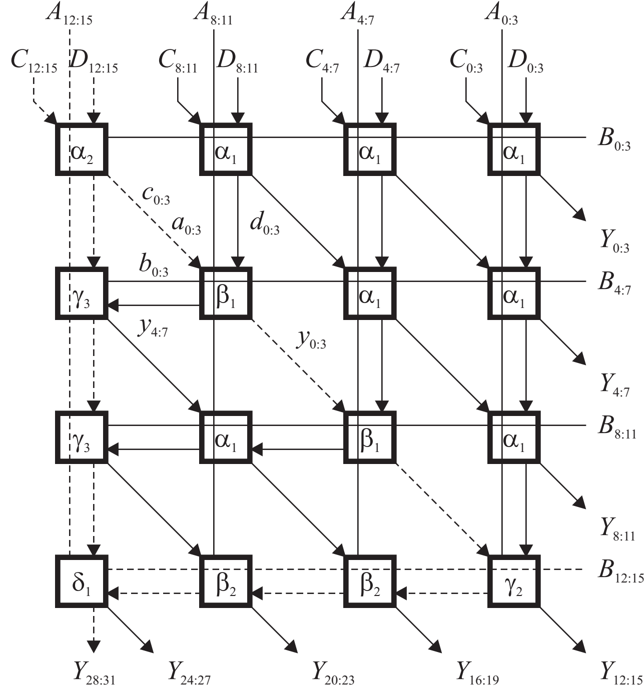


Figure 5.3: Two's-complement MAC unit.

format, but the remaining portions have unsigned format. Since each cell receives m-bit portions of the n-bit inputs, various cells may work with all unsigned data, all two's-complement data, or some combination of the two.

Figure 5.3 depicts the proposed MAC unit with two's-complement inputs. Solid lines denote unsigned data; dashed lines denote two's-complement data. Each cell in the module computes one of seven functions, labeled α_1, α_2, β_1, β_2, γ_2, γ_3, and δ_1. (The reason for this nomenclature will become clear in the next section.) The α_1 cells simply evaluate the unsigned m-bit MAC in (5.3). However, other cells must accept one or more two's-complement inputs.

Consider the β_1 cell in the figure. Three of the inputs ($a_{0:m-1}$, $b_{0:m-1}$, $d_{0:m-1}$) have unsigned format and can range from 0 to $2^m - 1$. However, $c_{0:3}$ has two's-complement format

and can range from -2^{m-1} to $2^{m-1}-1$. The minimum value of the output is

$$(0)(0) + (-2^{m-1}) + 0 = -2^{m-1}, \tag{5.11}$$

whereas the maximum value is

$$(2^m - 1)(2^m - 1) + (0) + (2^m - 1) = 2^m(2^m - 1). \tag{5.12}$$

This range cannot be represented by either an unsigned or two's-complement $2m$-bit number.

The two cells presented in Chapter 3 solve this problem in different ways. The serial cell appends an extra bit to each input and output to prevent overflow. Each cell then evaluates a two's-complement MAC on the $m+1$-bit inputs. In contrast, the parallel cell retains the m-bit words but splits the output into two portions: $y_{0:m-1}$ with two's-complement format, and $y_{m:2m-1}$ with unsigned format. The complete output,

$$y_{0:2m-1} = y_{0:m-1} + 2^m y_{m:2m-1}, \tag{5.13}$$

can represent values from -2^{m-1} to $2^m(2^m-1)$, as required.

Applying this scheme throughout the MAC unit yields the data formats shown in the figure. For example, the α_2 cell produces an output whose lower half and upper half are both two's-complement numbers. Observe that the final output, $Y_{0:2n-1}$, has the correct two's-complement format. Table 5.1 lists the input and output format for the seven functions in the MAC unit, plus a γ_1 function used later. A + sign denotes unsigned format, whereas a − sign denotes two's-complement format.

Table 5.1: Data formats for two's-complement MAC unit.

Name	$a_{0:m-1}$	$b_{0:m-1}$	$c_{0:m-1}$	$d_{0:m-1}$	$y_{m:2m-1}$	$y_{0:m-1}$
α_1	+	+	+	+	+	+
α_2	−	+	−	−	−	−
β_1	+	+	−	+	+	−
β_2	+	−	+	−	−	+
γ_1	+	+	+	−	+	−
γ_2	+	−	−	+	−	+
γ_3	−	+	−	+	−	+
δ_1	−	−	−	−	−	+

5.3 Cell functions

The previous section demonstrated that the proposed MAC unit requires cells to implement seven functions in general, assuming that the architecture does not append an extra bit to each data word. A natural question is how one reconfigurable cell can realize this functionality. Generalizing the parallel cell in Chapter 3, suppose that the cell uses an $m \times m$ array of 1-bit processing elements to compute binary arithmetic. The question then becomes how each element should be configured.

For the α_1 cells, the solution resembles the implementation of the unsigned MAC unit. Each element computes the MAC with unsigned inputs, which in the 1-bit case reduces to

$$(2z + y) = (a \wedge b) + c + d. \tag{5.14}$$

Here y and z denote the lower and upper outputs of the element, and "$\wedge$" the logical AND operation. One can classify this function as type α_1, since all inputs and outputs are unsigned bits.

For the other cells, one can use Table 5.1 to determine the function implemented by each element. Given the format for the four inputs (a, b, c, d), the table defines the format for the two outputs (y, z). It happens that the resulting structure generates $y_{0:2m-1}$ with the

Table 5.2: Element functions in MAC cells.

Name	Expression
α_1, α_2	$(2z + y) = (a \wedge b) + c + d$
β_1, β_2	$(-2z + y) = -(a \wedge b) + c - d$
γ_1, γ_2, γ_3	$(-2z + y) = -(a \wedge b) - c + d$
δ_1	$(-2z + y) = (a \wedge b) - c - d$

correct format in all cases. Figure 5.4 depicts the implementation of the seven functions for $m = 4$.

Now consider the function computed by α_2 elements. From Table 5.1, a, c, d, y, and z all have two's-complement format, such that logic 0 denotes 0 and logic 1 denotes -1. Thus, the elements evaluate the logic expression

$$(-2z - y) = (-a \times b) - c - d, \tag{5.15}$$

which simplifies to

$$(2z + y) = (a \wedge b) + c + d. \tag{5.16}$$

Since this equation is equivalent to (5.14), the truth tables for α_1 and α_2 are identical. Performing a similar analysis on the remaining functions reveals that only four distinct types are required, as listed in Table 5.2. The reconfigurable device can exploit these similarities to reduce the configuration time.

5.4 Summary

This chapter has described a novel scheme for performing n-bit MAC operations on a 4-bit reconfigurable cell array. Each cell computes a 4-bit MAC function with two additive terms. With small changes to the configuration of each cell, the structure can handle unsigned or

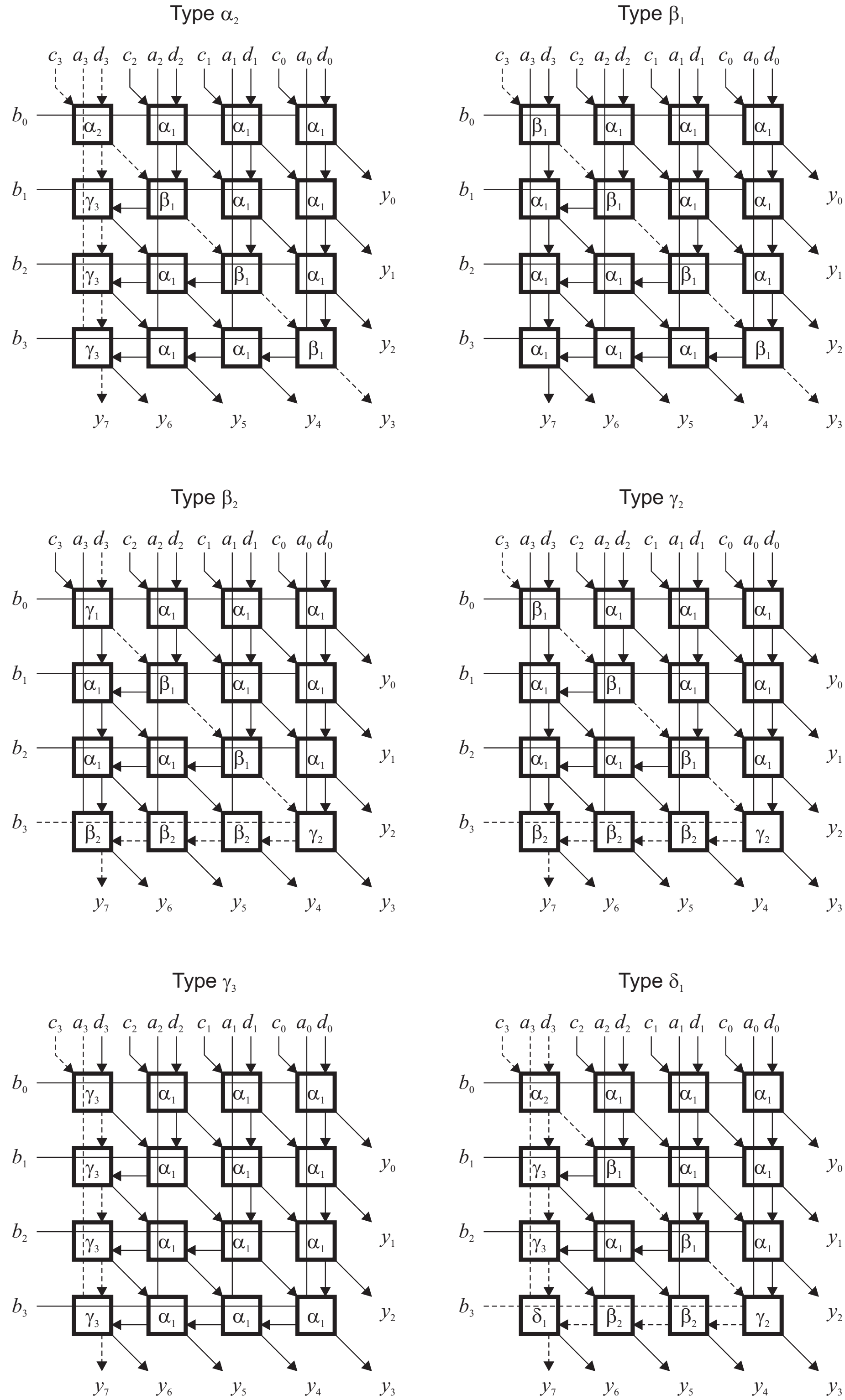


Figure 5.4: Implementation of cell functions.

two's-complement inputs. Each cell, in turn, can use a $4{\times}m$ matrix of 1-bit processing elements to compute the required functions. This approach corresponds to the design of the parallel cell in Chapter 3. Only four element functions are required to construct multipliers of any size.

Chapter 6

Algorithms

This chapter concludes the description of the reconfigurable cell array by evaluating its capabilities for DSP. We illustrate how the device can implement common algorithms such as the FIR filter and the FFT. These algorithms consist of a series of modules, similar to those described in Chapter 4. The structure of the interconnection network makes the mapping process very straightforward. Essentially, the modules are placed onto the array of cells and connected using the global H-tree. We have created software tools to assist the mapping process.

We begin this chapter with a brief description of these computer-aided design (CAD) tools. Next, we summarize how the device configures the cells and interconnection network. We then present the implementations of the selected benchmarks and analyze the performance and flexibility of the proposed architecture.

6.1 Software tools

Mapping DSP onto any reconfigurable platform requires a set of CAD tools. Most commercial software supports design synthesis, timing analysis, and circuit simulations. As an initial step, we have created software tools that allow users to map algorithms by hand. These tools

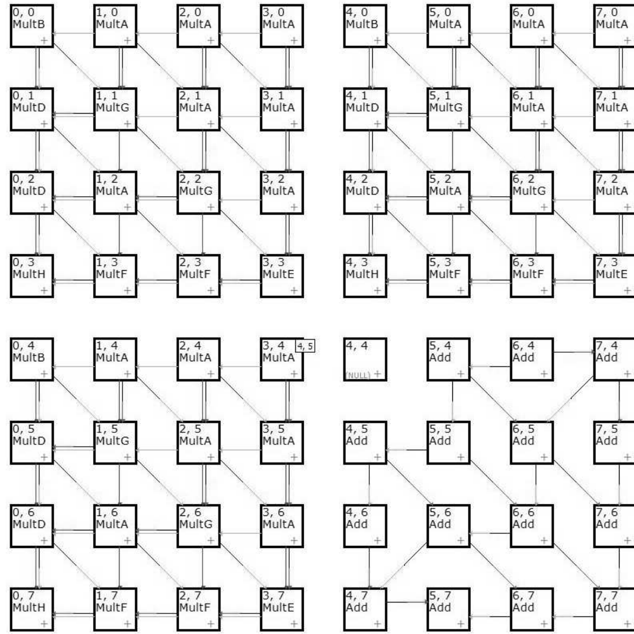


Figure 6.1: Screenshot of software tools.

contain a virtual model of the array of cells and the interconnection network. The software also features a built-in emulator that allows users to feed data into their designs and obtain cycle-by-cycle results.

Figure 6.1 contains a screenshot of the editor used for designing modules such as multipliers. The user can define the configuration of each cell in a separate window and select between memory mode and mathematics mode. The module shown in the figure is part of the FIR filter described later in this chapter. The user can toggle the display of local and global interconnections.

Unlike CAD tools for other reconfigurable platforms, the emulator does not estimate the propagation delays on the physical device to determine the maximum clock rate. This calculation is unnecessary for the proposed architecture, since the pipelined datapath allows the device to run at a fixed frequency. Instead, the emulator allows users to count the number of cycles required by the algorithm. Multiplying the cycle count by the clock period produces the total execution time. Different circuit designs may run at different frequencies,

but all have the same basic architecture.

Reference [39] gives more information about the CAD tools.

6.2 Configuration

Before using the reconfigurable cell array for DSP, the device must be configured to implement the desired algorithm. The architecture reuses the global H-tree to pass the configuration data into the cells. Thus, affected portions of the device cannot perform regular operations during this time. The system configures the cells first, followed by the switches inside the cell interface, followed by the switches in the global network.

The configuration process begins when the system asserts a global cfg signal. In response, all switches inside the architecture revert to a default setting so that the H-tree assumes the structure in Figure 6.2. Each cell switches to memory mode so that new information can be written into the elements. The system then loads configuration commands into the global H-tree, which propagate into the cells along the input busses. Cells are programmed using standard write operations in memory mode, whereas switches are programmed by placing control words on specific data lines.

The time required to completely reconfigure the device depends on a number of factors, including the size of the array, design of the cells, and number of configuration bits supplied per cycle. A 32×32 array contains 1024 cells, 2048 internal switches, and 1023 global switches. Assuming that the system uses the parallel cell, configuring the matrix of elements requires 128 write operations. Adding 4 cycles to select the cell and specify the mode, each cell requires 132 configuration cycles. We estimate that each switch requires 16 configuration cycles in the worst case. Hence, the number of cycles required for complete reconfiguration is

$$1024(132) + 2048(16) + 1023(16) = 184\,304. \tag{6.1}$$

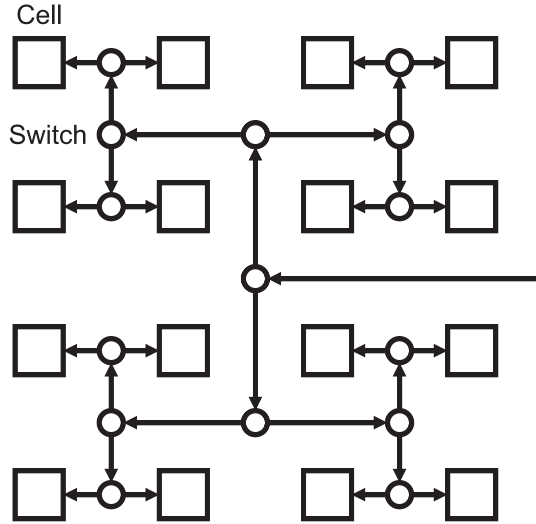


Figure 6.2: Structure of cell array during reconfiguration.

Of course, most algorithms do not use all the resources of the device.

More details about the configuration process appear in [40]. Although the reference uses a slightly different design for the interconnection network, the current architecture can use some of the proposed optimizations. For example, the system can program multiple cells simultaneously by exploiting the hierarchical nature of the H-tree.

6.3 Benchmarks

Assisted by the CAD tools, we have mapped various algorithms onto the reconfigurable cell array. This section presents several examples: a 12-tap FIR filter, a 16-stage CORDIC unit, and a 256-point Fast Fourier Transform (FFT). The filter and FFT in particular are common benchmarks for DSP hardware, whereas the CORDIC unit demonstrates the versatility of the architecture. The algorithms work with inputs in 16-bit fixed-point format, although intermediate data may be calculated to higher precision. We originally presented these results in [35].

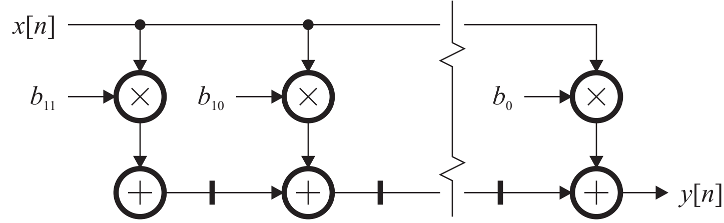


Figure 6.3: Diagram of FIR filter.

6.3.1 FIR filter

Figure 6.3 gives a block diagram of a 12-tap FIR filter. This structure harnesses the power of the reconfigurable cell array by performing all operations in parallel. The input to the filter, x, is passed to twelve multipliers. Each unit i is configured to multiply the data by a fixed coefficient b_i. The outputs of the multipliers are added in pipelined fashion so that the output y becomes

$$y[n] = b_0 x[n] + b_1 x[n-1] + \ldots + b_{11} x[n-11]. \tag{6.2}$$

Figure 6.4 depicts the implementation of the filter. The structure consists of four identical modules, each of which handles three filter coefficients. Each module occupies an 8×8 block of cells and contains three multipliers and three adders. To implement higher-order filters, one would simply add more modules. The current design allows filter coefficients within the range $[-1,\ 1)$. However, the implementation does use 20-bit adders to mitigate rounding errors.

The global interconnection network offers several features that simplify the mapping process. The input lines of the H-tree broadcast x to all modules simultaneously. Each module collects the outputs of the three multipliers into a bundle and passes them to the inputs of the corresponding adders. All three outputs incur the same latency over the H-

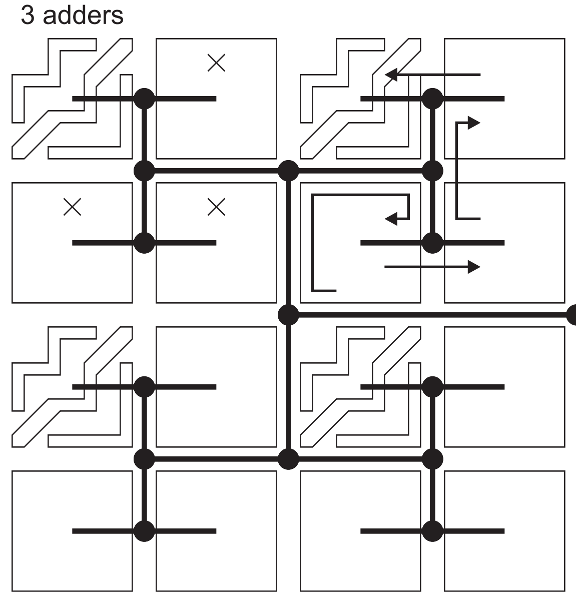


Figure 6.4: Implementation of FIR filter.

Table 6.1: Statistics of FIR filter.

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Cells	256	256	256	256
Area (mm^2)		30.7		24.8
Latency (cycles)	57	72	264	72
Time (cycles)	301	316	508	316
Time (μs)	1.80	1.18	0.31	1.89

tree, simplifying the implementation. The tree structure is also useful for connecting adjacent modules in sequence. The arrows in the figure suggest one method for traversing the tree.

Table 6.1 lists the required area and execution time of the filter for each cell described in Chapter 3. For the area, we estimated that the interconnection network consumes 0.08 mm^2 per cell, including the local mesh and global H-tree. The latency specifies the number of cycles from the arrival of $x[n-11]$ to the computation of $y[n]$. This path involves 20 cells, 15 local busses, and 74 levels of the H-tree. We also list the total execution time for a 256-point data stream. This parameter equals the latency plus 244 cycles.

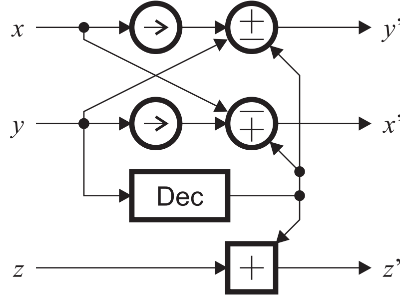


Figure 6.5: Diagram of one CORDIC stage.

6.3.2 CORDIC unit

CORDIC transformations offer one way to perform complex mathematics operations, such as division, square root, trigonometry, and rectangular-to-polar conversion. A typical transformation for n-bit inputs contains n stages, each of which follows the diagram in Figure 6.5. Initially, the system sets x and y to the inputs (such as the rectangular coordinates) and z to zero. Iteration i then updates the data as follows:

$$\begin{aligned} x' &= x \mp 2^{-i}y \\ y' &= y \pm 2^{-i}x \\ z' &= z + f(i) \end{aligned} \tag{6.3}$$

The value of $f(i)$ and the choice of addition or subtraction depends on the sign of y. After n stages, x and/or z will contain the desired results, and y will be essentially zero.

We have mapped a 16-bit CORDIC stage onto the reconfigurable cell array, as shown in Figure 6.6. A 4×8 block of cells contains two adder/subtracters, two shifters, one constant adder, and a small decoder that determines the sign of y. The implementation contains a number of features to improve performance. The two shifters translate data no more than four bits to the right. The remaining shift amount is performed with hardwired connections. In addition, the two values of $f(i)$ needed for the current stage are hardcoded into the

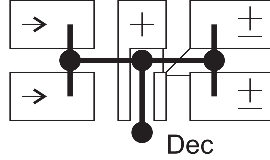


Figure 6.6: Implementation of one CORDIC stage.

Table 6.2: Statistics of one CORDIC stage.

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Cells	32	32	32	32
Area (mm^2)		3.8		3.1
Latency (cycles)	18	23	88	23

constant adder. Finally, all modules work with 24-bit data rather than 16-bit data to alleviate rounding errors.

DSP algorithms could compute CORDIC transformations in two ways. A low-area approach would use a single CORDIC stage to process a group of data points one iteration at a time. After partial reconfiguration, the stage would be ready for the next iteration. A high-performance approach would cascade 16 CORDIC stages to form a 16×32 block. In this case, the throughput of the CORDIC transformation would be the reciprocal of the clock rate. Table 6.2 gives the latency and area of a single CORDIC stage. The critical path consists of 7 cells, 5 local busses, and 22 levels of the H-tree.

6.3.3 Fast Fourier Transform

We previously mapped a 256-point FFT on the reconfigurable architecture using a simple radix-2 decomposition [34]. With the redesigned interconnection network, we can now upgrade to a radix-4 decomposition. Figure 6.7 gives a diagram of this algorithm. The system loads the input data into the specialized memory unit described in Chapter 4. Each sample is represented as a complex number with 16-bit real and 16-bit imaginary portions. The

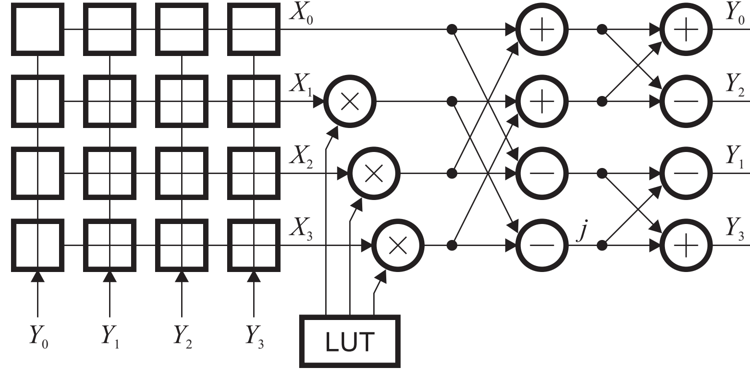


Figure 6.7: Diagram of FFT.

system then executes four computational stages. During each stage, the system reads four samples from memory, calculates a so-called "dragonfly" operation, and stores the results back into memory at different addresses. The process repeats for the remaining groups of samples in the dataset.

The "dragonfly" is described by the matrix equation

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -j \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & j \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} X_0 \\ W_1 X_1 \\ W_2 X_2 \\ W_3 X_3 \end{bmatrix}. \tag{6.4}$$

This operation consists of three complex multiplications and three complex additions or subtractions. Coefficients W_1, W_2, and W_3, called "twiddle factors", are generated by a lookup table. Note that multiplying by a factor of j or $-j$ only requires the system to exchange the real and imaginary portions of the data.

The implementation of the FFT appears in Figure 6.8. Each complex multiplier breaks down into four real multipliers, one real adder, and one real subtracter. The multipliers work with 16-bit data, but the adders work with 24-bit data to avoid rounding errors. With the parallel cell, the implementation occupies a 32×16 block of cells, not including the lookup

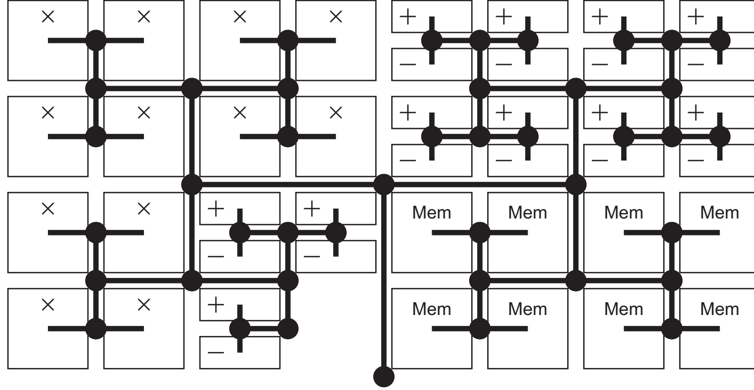


Figure 6.8: Implementation of FFT.

Table 6.3: Statistics of FFT.

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Cells	512	512	512	916
Area (mm^2)		61.4		88.9
Latency per stage (cycles)	55	68	250	87
Time per stage (cycles)	118	131	313	150
Total time (cycles)	472	524	1252	600
Total time (μs)	2.83	1.96	0.76	3.59

table needed for the “twiddle factors”. With the serial cell, the memory unit consumes more space since memory cells have a smaller capacity.

Table 6.3 shows the area and execution time of the FFT. The latency of one stage is the number of cycles between the read port and the write port of the memory unit. With the parallel cell, the critical path involves 19 cells, 13 local busses, and 72 levels of the H-tree. With the serial cell, the memory unit is larger, so the critical path involves 27 cells, 21 local busses, and 78 levels of the H-tree. Since each processing stage handles 64 groups of samples, the time required per stage equals the latency plus 63 cycles. Multiplying this number by 4 yields the total execution time.

6.4 Analysis

In this section, we analyze the medium-grain cell array with respect to other hardware components, including digital signal processors and FPGAs. We compare the characteristics of the underlying architectures, as well as the execution times of selected benchmarks.

6.4.1 Digital signal processors

The proposed medium-grain architecture offers several advantages over digital signal processors. The pipelined array of cells exploits the inherent parallelism of DSP, while allowing designers to tailor the datapath to application requirements. For example, designers can choose the word length and data format of each processing stage. Both fixed-point and floating-point operations are possible. In contrast, digital signal processors generally have a fixed word length and data format. The lack of a customizable parallel datapath also inhibits performance.

Table 6.4 compares the execution times of the FIR filter and FFT to the reported results for several digital signal processors. The devices selected for this comparison represent some of the most advanced processors available today. All components work with a 256-sample dataset with 16-bit fixed-point data. The time required to load the data into memory is not included. As shown, the medium-grain architecture surpasses the performance of digital signal processors in similar technologies. The 180-nm parallel cell even approaches the results of the 90-nm TMS320C64 processor, when combined with the static element. This alternative yields lower execution times than the dynamic element. The parallel pipelined architecture delivers maximum performance, but at the expense of high power consumption. Finally, the serial pipelined architecture offers respectable results. The execution time for the FFT is higher due to the large memory units required.

Table 6.4: Performance comparison with digital signal processors.

Parameter	Analog ADSP-BF533	TI TMS320C62	TI TMS320C64
Technology		150-nm	90-nm
Clock frequency (MHz)	750	300	1000
FIR filter time (μs)	2.39	9.02	1.29
FFT time (μs)	3.10	9.25	1.24

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Technology	180-nm	180-nm	180-nm	180-nm
Clock frequency (MHz)	167	267	1650	167
FIR filter time (μs)	1.80	1.18	0.31	1.89
FFT time (μs)	2.83	1.96	0.76	3.59

6.4.2 Fine-grain reconfigurable hardware

Table 6.5 compares the proposed architecture with a typical FPGA for DSP. Although both alternatives contain a reconfigurable array of cells, each uses a completely different approach to implementing algorithms. On an FPGA, the synthesis tools translate the algorithm into a series of logic expressions, which map onto the 1-bit cells. The interconnection structure routes data between cells in bit-length units. In contrast, the proposed architecture represents algorithms as a set of modules composed of 4-bit cells. The elements inside the cell allow the architecture to maintain 1-bit flexibility. The local mesh connects cells within a module, while the global H-tree routes data between modules in word-length units. Higher levels of the network use a longer word size. Since interconnection structures typically consume the majority of the active area on reconfigurable devices, this optimization has a significant effect.

The method used to implement several important operations also differs from FPGAs. For example, the Xilinx Virtex-II features embedded 18-bit multipliers and fast carry logic within the basic cells. The Xilinx Virtex-4 uses special DSP blocks that can perform 18-bit multiplication and 48-bit addition. In contrast, the medium-grain cell array features

Table 6.5: Architecture comparison with fine-grain reconfigurable devices.

Parameter	FPGA	Proposed
Cell granularity	1-bit	4-bit with 1-bit flexibility
Interconnection structure	Mesh with global lines	Combined mesh and H-tree
Interconnection granularity	1-bit	4-bit to word-length
Multipliers	Embedded	Integrated
Memory units	Integrated	Integrated dual-port
Operating frequency	Depends on algorithm	Fixed at maximum

a homogeneous structure that can integrate multipliers and adders with other modules. Each cell can perform mathematics functions or memory operations, so designers can create powerful memory units as well. These modules work with separate read and write addresses, simplifying multi-stage algorithms. As always, designers can customize the word length and number of modules to the needs of the application. A 32×32 array of cells could contain 64 multipliers for 16-bit inputs or 16 multipliers for 32-bit inputs, for example.

The pipelined interconnection network unifies all computations on the medium-grain reconfigurable architecture. Each module runs at the same clock frequency and initiates one operation per cycle. With FPGAs, the ultimate clock rate depends on the complexity of each module as well as the sophistication of the routing tools. The medium-grain cell array always maintains the maximum clock rate, regardless of the current configuration.

Table 6.6 compares the performance of the proposed architecture with several high-performance FPGAs. We present the execution time of the 256-point FFT, since results for the FIR filter were not available. For the Xilinx FPGAs, the execution time corresponds to a radix-4, burst-mode algorithm. The 90-nm Virtex-4 does outperform the proposed architecture, but this gap narrows significantly with the 150-nm Virtex-II. Also recall that preliminary circuit simulations have the parallel static architecture running at 1 GHz in 90-nm technology. At this speed, the FFT would complete in 0.52 μs, although this result ignores parasitic interconnect capacitances.

Although not shown in Table 6.6, the reconfiguration time required by the two-level

Table 6.6: Performance comparison with fine-grain reconfigurable hardware.

Parameter	Xilinx Virtex-II	Xilinx Virtex-II Pro	Xilinx Virtex-4
Technology	150-nm	130-nm	90-nm
Clock frequency (MHz)	195	223	421
FFT time (μs)	1.48	1.30	0.69

Parameter	Parallel Dynamic	Parallel Static	Parallel Pipelined	Serial Pipelined
Technology	180-nm	180-nm	180-nm	180-nm
Clock frequency (MHz)	167	267	1650	167
FFT time (μs)	2.83	1.96	0.76	3.59

architecture is comparable to FPGAs. The most basic Virtex-II device has 338 976 bits of configuration memory that can be programmed at 50 MHz in 8-bit units. The reconfiguration time for this device is thus 847 μs. As shown earlier, completely reprogramming a 32×32 array of parallel cells requires approximately 184 304 cycles, or 690 μs at 267 MHz.

6.4.3 Coarse-grain reconfigurable hardware

The proposed architecture shares many of the benefits of coarse-grain reconfigurable devices, but also has several distinct characteristics. Coarse-grain cells generally perform a limited number of 16-bit or 32-bit operations. To implement the control logic necessary for DSP, some architectures integrate the coarse-grain array with a separate fine-grain array or microprocessor. Another option is to use a heterogeneous array of cells, as with the PACT XPP and QuickSilver Adapt2000. In contrast, the proposed architecture features a homogeneous reconfigurable fabric that uses the same cells to perform binary arithmetic, memory operations, and control logic. The 4-bit granularity gives designers more flexibility over the word length.

6.5 Summary

In this chapter, we have discussed the implementation of several benchmarks on the medium-grain cell array. We estimated the execution times with the assistance of custom CAD tools. The proposed architecture achieves comparable performance to advanced digital signal processors and FPGAs in the same technology. In addition, the homogeneous design offers maximum versatility.

Chapter 7

Conclusion

This dissertation has presented a novel medium-grain reconfigurable cell array for DSP. The architecture features an array of 4-bit cells and a hierarchical interconnection network. Each cell uses a smaller array of 1-bit processing elements to perform mathematics and memory operations. The interconnection scheme allows cells to be grouped into discrete modules, such as adders, multipliers, and memory units. Modules can then be connected to implement entire algorithms.

We have evaluated two designs for the 4-bit cell and three designs for the basic element. An initial prototype of the cell has been fabricated and tested for functionality. Layout simulations indicate that a bit-parallel cell that uses static elements achieves a clock frequency of 267 MHz in a modest 180-nm technology. A bit-serial cell that uses pipelined elements runs at 167 MHz in the same technology. In addition, we have mapped several benchmarks onto the reconfigurable architecture and calculated the execution times. A 32×32 array of parallel cells can perform a 16-bit, 256-point FFT in 1.96 μs. The architecture achieves comparable performance to FPGAs in similar technology.

The remainder of this chapter discusses the contributions of this research, and proposes directions for future work.

7.1 Contributions

This research encompasses a variety of architectural innovations, including the following:

- **Two-level array:** The reconfigurable architecture contains a two-level array of 4-bit cells and 1-bit elements [30, 31, 28]. This approach allows the design to achieve the coarse-grain performance required for binary arithmetic, as well as the fine-grain flexibility required for control logic. Cells can implement all necessary operations, including multiplication, addition, bit shifting, control logic, and data storage. Elements permit the cell to handle various number formats, such as unsigned and two's-complement. The 4-bit granularity allows the device to match the word length of the application.

- **Two-mode cell configurations:** Traditional fine-grain reconfigurable hardware suffers from complex interconnection structures. In contrast, the cell only assume two configurations: one optimized for memory operations, and the other for mathematics functions [30, 31, 28]. Hence, the design requires minimal routing resources at the fine-grain level. Mathematics mode is optimized for the 4-bit MAC, and thus encompasses multiplication and addition as well. Memory mode allows embedded random-access memory and lookup tables to be distributed throughout the array of cells.

- **Multiplication with 4-bit cells:** This research also incorporates a novel parallel multiplier structure that uses medium-grain processing elements [38]. In this way, large multipliers and MAC units can be mapped onto the array of 4-bit cells. Unsigned multipliers require one universal element configuration; two's-complement multipliers require three additional element types. The number, word length, and location of these multipliers is only limited by the size of the array. This property makes the proposed architecture a promising choice for cryptography and other applications that use multiplication extensively.

- **Hierarchical interconnection structure:** The interconnection network used in the medium-grain architecture recognizes that algorithms are composed of discrete modules, such as multipliers and adders [34, 35]. Hence, the architecture provides a mesh of busses for data transfer within a module, as well as a global H-tree for connecting modules together. The higher levels of the H-tree manipulate data in word-length units, allowing the inputs and outputs of modules to be routed together. All 4-bit portions incur the same latency between modules. This approach contrasts with FPGAs, which may route each bit on a different path.

- **Pipelined execution:** This is the first known study that applies deep pipelining to reconfigurable cell arrays. Each 4-bit cell pipelines all input and output data. This approach allows modules to initiate one operation per clock cycle, dramatically increasing throughput. The upper levels of the H-tree contain pipeline latches as well so that interconnection latencies do not adversely affect the maximum clock frequency. Unlike FPGAs, the clock rate does not depend on the complexity of each module, but remains constant at all times.

- **Alternative cell designs:** One alternative design for the cell carries pipelining even further, down to the bit level [33]. This approach achieves performance comparable to hardware in vastly superior technologies. Another alternative design computes operations in a bit-serial fashion [32]. This approach maintains the same functionality while reducing the area. Serial computations are localized to the cell. The global clock runs at moderate frequency to mitigate clock distribution problems.

Taken as a whole, the proposed reconfigurable architecture supports a large application space with a number of orthogonal axes. Designers can customize the word length, amount of parallelism, and number of modules to meet the needs of the application. In this manner, systems can balance performance and flexibility while minimizing development costs.

7.2 Future work

Further research on the reconfigurable cell array will focus on several areas. On the hardware level, we will migrate the cells to 90-nm technology to permit more accurate comparison with current DSP hardware. We will also implement the interconnection network and consider ways to improve its efficiency. The successful fabrication of additional prototype chips would give important evidence in favor of the design.

On the software level, we will continue to develop the CAD tools that allow users to map algorithms onto the architecture. Support for automatic placement and routing is the ultimate goal of this effort. We will calculate the execution times of further benchmarks to enable a more detailed assessment of the architecture. Further work can also focus on improving the hardware-software interface in the form of the configuration process.

Finally, we will explore various enhancements to the basic design. The application space of DSP has expanded in recent years to include devices with specialized requirements, such as low power consumption and high reliability. Low power consumption is vital to wireless communication devices, whereas high reliability is crucial for many real-time monitoring systems. The development of methods to lower the power requirements and increase the resilience of the device remains an important avenue for future work.

Bibliography

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

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

[3] Texas Instruments, Inc., "TMS320C6000 CPU and Instruction Set Reference Guide," literature number SPRU189F, Oct. 2000.

[4] 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, pp. 3/1–3/6, Jun. 1993.

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

[6] 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.

[7] S.D. Haynes and P.Y.K. Cheung, "Configurable multiplier blocks for embedding in FPGAs," *Electronics Letters*, vol. 34, iss. 7, pp. 638–639, Apr. 1998.

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

[9] Xilinx, Inc., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," literature number DS083, v. 4.5, Oct. 2005.

[10] P.T. Balsara and D.T. Harper III, "Understanding VLSI bit serial multipliers," *IEEE Trans. Education*, vol. 39, iss. 1, pp. 19–28, Feb. 1996.

[11] T. Isshiki et al., "High density bit-serial FPGA with LUT embedding shift register function," in *Proc. 2002 Asia-Pacific Conf. on Circuits and Systems*, vol. 1, pp. 475–480, Oct. 2002.

[12] S.A. Rahim and L.E. Turner, "A field programmable bit-serial digital signal processor," in *Proc. 4th IEEE Int. Workshop on System-on-Chip for Real-Time Applications*, pp. 295–298, Jul. 2004.

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

[14] C. Ebeling, D. Cronquist, P. Franklin, and C. Fisher, "RaPiD—a configurable computing architecture for compute-intensive applications," University of Washington Department of Computer Science & Engineering Tech Report TR-96-11-03, Nov. 1996.

[15] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Using the KressArray for reconfigurable computing," *Proc. SPIE*, vol. 3526, pp. 150–161, Oct. 1998.

[16] A. Tisserand, P. Marchal, and C. Piguet., "An on-line arithmetic based FPGA for low-power custom computing," in *Proc. 9th Int. Workshop on Field Programmable Logic and Applications*, London, England, vol. 1673 of LNCS, pp. 264–273, Sep. 1999.

[17] H. Zhang et al, "A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing," *IEEE J. Solid-State Circuits*, vol. 35, iss. 11, pp. 1697–1704, Feb. 2000.

[18] P. Heysters and G. Smit, "Mapping of DSP algorithms on the MONTIUM architecture," in *Proc. Int. Parallel and Distributed Processing Symposium*, pp. 180–185, Apr. 2003.

[19] A. Gunzinger, S. Mathis, and W. Güggenbuhl, "A reconfigurable systolic array for real-time image processing," in *Proc. 1988 Int. Conf. on Acoustics, Speech, and Signal Processing*, New York, NY, vol. 4, pp. 2054–2060, Apr. 1988.

[20] PACT Informationstechnologie GmbH, "The XPP white paper," v. 2.1, Mar. 2002.

[21] B. Plunkett and J. Watson, "Adapt2400 ACM Architecture Overview," QuickSilver Technology, Inc., white paper, 2004.

[22] Xilinx, Inc., "Virtex-4 Family Overview," literature number DS112, v. 1.5, Feb. 2006.

[23] A. Marshall et al., "A reconfigurable arithmetic array for multimedia applications," in *Proc. 7th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, pp. 135–143, 1999.

[24] S.C. Goldstein et al., "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, iss. 4, pp. 70–77, Apr. 2000.

[25] J. Becker, T. Pionteck, C. Habermann, and M. Glesner, "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture," in *Proc. IEEE Computer Soc. Workshop on VLSI*, Orlando, FL, pp. 41–46, Apr. 2001.

[26] Elixent, Inc., "Applications of the D-Fabrix Array," white paper WP0001, 2001.

[27] M.J. Myjak, "A two-level reconfigurable cell array for digital signal processing," M.S. thesis, Washington State University, May 2004.

[28] M.J. Myjak and J.G. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," *Microelectronic Engineering*, in press.

[29] J. Rabaey, A. Chandrakasan, and B. Nikolić, *Digital Integrated Circuits: A Design Perspective*, 2nd ed., Upper Saddle River, NJ: Pearson Education, Inc., 2003, pp. 591–592.

[30] J.G. Delgado-Frias, M.J. Myjak, F.L. Anderson, and D.R. Blum, "A medium-grain reconfigurable cell array for DSP applications," in *Proc. 3rd IASTED Int. Conf. on Circuits, Signals, and Systems*, Cancun, Mexico, pp. 231–236, May 2003.

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

[32] M.J. Myjak and J.G. Delgado-Frias, "A bit-serial cell for reconfigurable DSP hardware," in *Proc. 2005 IEEE Int. Midwest Symposium on Circuits and Systems*, Cincinnati, OH, pp. 960–963, Aug. 2005.

[33] M.J. Myjak and J.G. Delgado-Frias, "Superpipelined reconfigurable hardware for DSP," in *Proc. 2006 IEEE Int. Symposium on Circuits and Systems*, Kos, Greece, May 2006, to be published.

[34] M.J. Myjak, F.L. Anderson, and J.G. Delgado-Frias, "H-tree interconnection structure for reconfigurable DSP hardware," in *Proc. 2004 Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, pp. 170–176, Jun. 2004.

[35] M.J. Myjak, J.K. Larson, and J.G. Delgado-Frias, "Mapping and performance of DSP benchmarks on a medium-grain reconfigurable architecture," in *Proc. 2006 Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, Jun. 2006, to be published.

[36] C. Leiserson, "Universal networks for hardware efficient supercomputing," *IEEE Trans. on Computers*, vol. 34, iss. 10, 1985, pp. 892–901.

[37] A. DeHon, "Compact, multilayer layout for butterfly fat-tree," *Proc. 12th ACM Symposium on Parallel Algorithms and Architectures*, Bar Harbor, ME, 2000, pp. 206–215.

[38] M.J. Myjak and J.G. Delgado-Frias, "Pipelined multipliers for reconfigurable hardware," in *Proc. 11th Reconfigurable Architectures Workshop*, Santa Fé, NM, pp. 150–154, Apr. 2004.

[39] J.K. Larson, "CAD tool emulation for a two-level reconfigurable cell array for digital signal processing," M.S. thesis, Washington State University, Dec. 2005.

[40] A. Widjaja, "H-tree based configuration schemes for a reconfigurable DSP architecture," M.S. thesis, Washington State University, May 2005.

[41] M.J. Myjak and J.G. Delgado-Frias, "A symmetric differential clock generator for bit-serial hardware," in *Proc. 2005 Int. Conf. on Computer Design*, Las Vegas, NV, pp. 159–164, Jun. 2005.

[42] P. Nilsson, M. Torkelson, M. Vesterbacka, and L. Wanhammar, "CMOS on-chip clock for digital signal processors," *Electronics Letters*, vol. 29, iss. 8, pp. 669–670, Apr. 1993.

[43] D.E. Duarte, N. Vijaykrishnan, and M.J. Irwin, "A clock power model to evaluate impact of architectural and technology optimizations," *IEEE Trans. VLSI Systems*, vol. 10, no. 6, pp. 844-855, Dec. 2002.

[44] J. Yuan and C. Svensson, "New single-clock CMOS latches and flipflops with improved speed and power savings," *IEEE J. Solid-State Circuits*, vol. 32, iss. 1, pp. 62-69, Jan. 1997.

[45] B. Nikolić, V. Stojanovic, V.G. Oklobdzija, J. Wenyan, J. Chiu, and M. Leung, "Sense amplifier-based flip-flop," in *Proc. 1999 Solid-State Circuits Conf.*, San Francisco, CA, pp. 282-283, Feb. 1999.

[46] V. Stojanovic and V.G. Oklobdzija, "Flip-flop," U.S. Patent 6,232,810, May 2001.

[47] V.G. Oklobdzija, V.M. Stojanovic, D.M. Markovic, and N.M. Nedovic, *Digital System Clocking: High-Performance and Low-Power Aspects*, John Wiley and Sons, 2003.

[48] V. Stojanovic and V.G. Oklobdzija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems," *IEEE J. Solid-State Circuits*, vol. 24, iss. 4, pp. 536-548, Apr. 1999.

Appendix A

Clock Generator

Bit-serial architectures trade off physical size for time complexity by representing data words as a stream of bits on a single line. The serial cell proposed in Chapter 3 consumes less area than the parallel design described in the same chapter. However, computations in mathematics mode consist of an initialization phase and nine execution phases. Figure A.1 illustrates the control signals required for this operation. A pulse train *pul* controls the pipeline latches that separate each execution phase. After nine pulses, *clr* initializes the cell again. The rising edge of the global clock *clk* triggers the next series of computations.

This appendix describes the circuit we developed to generate the control signals for the

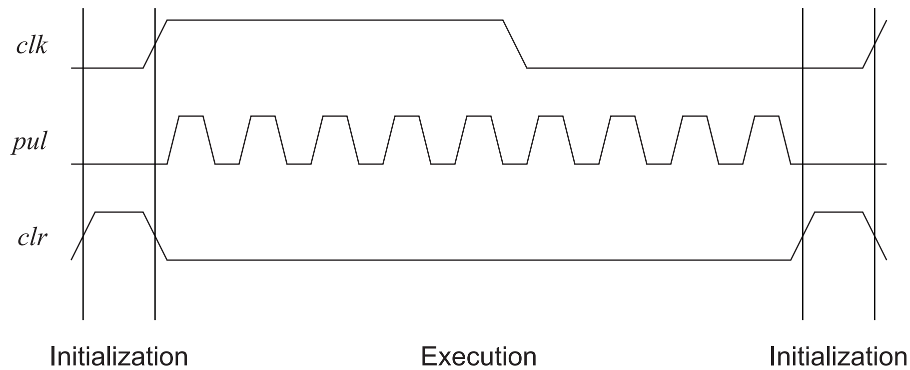


Figure A.1: Control signals required by serial cell.

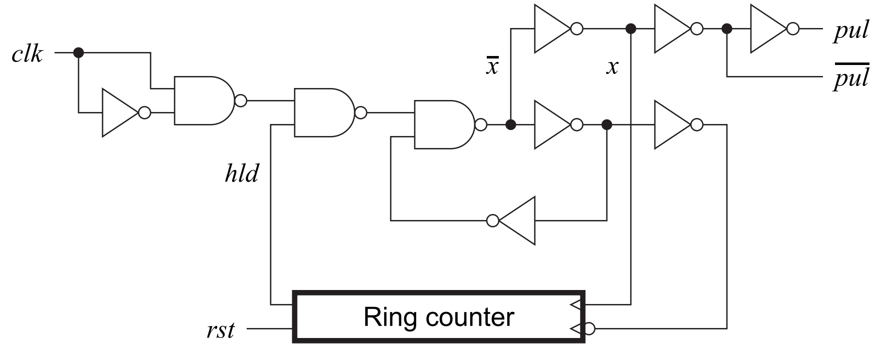


Figure A.2: Pulse generator proposed by Nilsson et al. (NTVW) [42].

serial cell. The design is quite simple, allowing each cell to use a separate circuit. This approach saves power by reducing the number of global signals on the reconfigurable device. In addition, the circuit produces differential outputs that can drive the complementary inputs of the serial cell. The remainder of this appendix parallels the discussion in [41]. We first review a simple pulse generator found in the literature. Then, we present the proposed design along with layout simulations that verify its functionality. A comparison of the two alternatives concludes the discussion.

A.1 NTVW design

Figure A.2 illustrates a circuit proposed by Nilsson, Torkelson, Vesterbacka, and Wanhammar (NTVW) to control a bit-serial module [42]. A ring oscillator generates the pulse train *pul*. The output of the oscillator controls a ring counter, which is reset to 1000... on power-on. After a given number of pulses, the ring counter asserts *hld* and stops the oscillator. The rising edge of *clk* creates a negative pulse that enables the ring oscillator again, repeating the cycle. This design is quite straightforward and functions properly at high frequencies. However, *pul* is not aligned with $\overline{pul}$. Having symmetric outputs would improve the performance of the serial cell and mitigate potential problems with clock overlap.

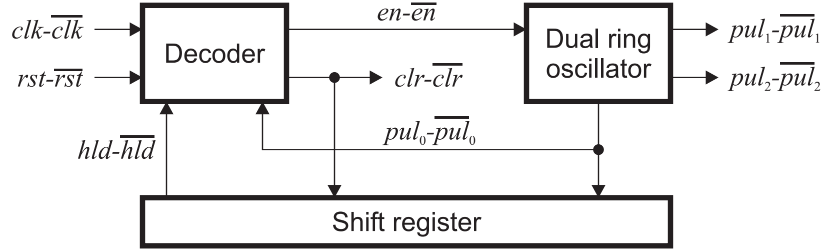


Figure A.3: Functional diagram of differential clock generator.

A.2 Proposed design

Figure A.3 depicts a functional block diagram of the differential clock generator. The design contains a dual ring oscillator to generate the pulse train, a shift register to stop the oscillator after the required number of execution phases, and a decoder to compute the initialization signals. All input and output signals are differential.

The remainder of this section describes each component of the clock generator, and gives a series of simulations to validate the design.

A.2.1 Oscillator

A schematic of the dual ring oscillator appears in Figure A.4. The circuit provides three differential outputs: pul_0-$\overline{pul}_0$ for the shift register, pul_1-$\overline{pul}_1$ for the serial cell, and pul_2-$\overline{pul}_2$ for any parts of the cell that require delayed clocks. Due to variations in parasitic capacitance, the two ring oscillators will likely have slightly different natural frequencies. However, the weak cross-coupled p-transistors ensure that the internal nodes remain in anti-phase. Although only one pair of p-transistors is necessary for this purpose, two pairs are used for extra assurance. The numbers in the figure denote the transistor sizes, in units of the minimum width.

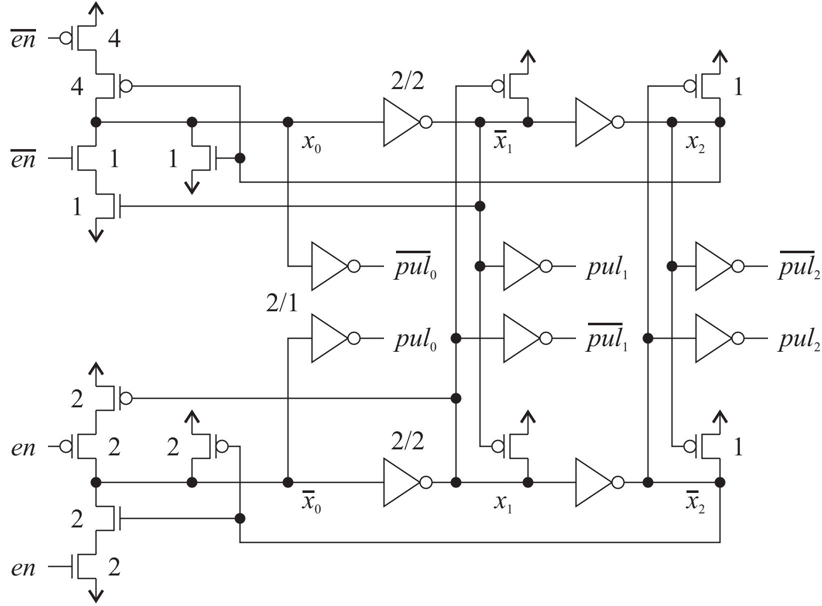


Figure A.4: Dual ring oscillator.

A.2.2 Shift register

As shown in Figure A.5, the shift register contains a chain of dynamic latches. The number of latches determines the number of pulses generated by the oscillator. All control inputs are buffered so that the register can be of any length. The initialization signals clr-$\overline{clr}$ connect to the data input of the register. When clr is high, all the latches reset asynchronously. When clr is low, the internal data shifts to the left through the register. The hld output remains at high during the initialization phase, transitions to low at the first execution phase, and transitions back to high during the last execution phase.

The shift register uses several techniques to increase performance. The pulse train pul_0-$\overline{pul}_0$ connects directly to the latches driving hld-$\overline{hld}$ so that the oscillator can be stopped quickly. For the same reason, the circuit asserts hld on the rising edge of the last pulse rather than the falling edge. Finally, notice that the shift register initializes by propagating the reset data through each latch from the input end to the output end. To expedite this process, some of the latches have an extra reset input.

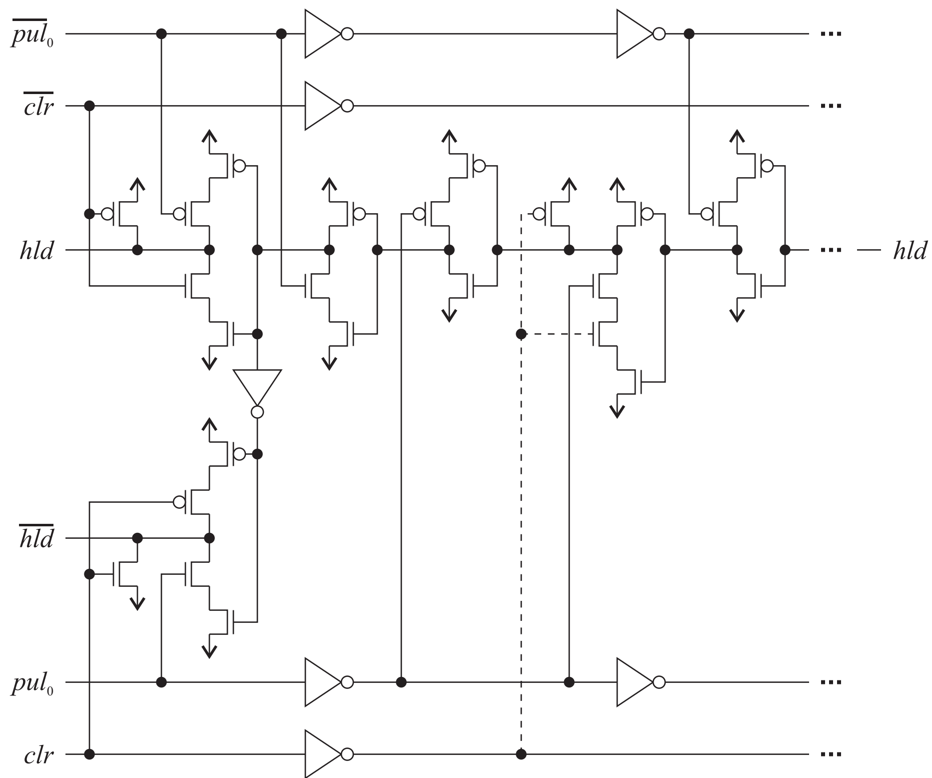


Figure A.5: Shift register used to stop oscillator.

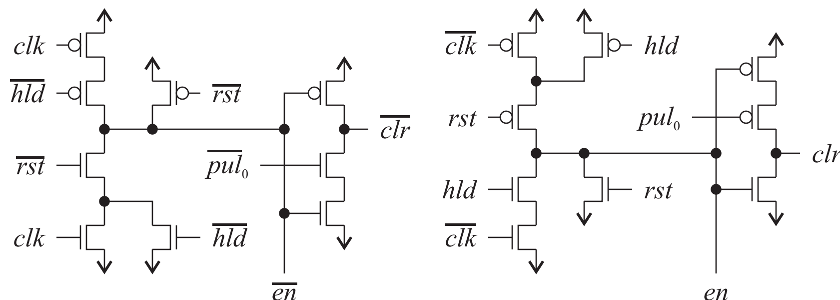


Figure A.6: Decoder for initialization.

A.2.3 Decoder

The last component of the design, the decoder, appears in Figure A.6. This circuit reads the *hld*-$\overline{hld}$ outputs of the shift register and generates the *en*-$\overline{en}$ signals to enable the oscillator. The decoder is also responsible for producing the initialization signals *clr*-$\overline{clr}$.

A.2.4 Operation

We implemented the differential clock generator in 180-nm CMOS. Figure A.7 gives a view of the transistor layout, which occupies an area of 690 μm^2. Each serial cell uses a separate circuit to drive internal computations. Thus, the system does not have to distribute *pul*-$\overline{pul}$ and *clr*-$\overline{clr}$ to each cell on the device, but only the relatively low-frequency global clock. This approach saves considerable power and area. In addition, generating the control signals locally allows the serial cell to achieve high performance. The frequency of *pul*-$\overline{pul}$ does not depend on the propagation delay between cells, but rather the latency of the elements in mathematics mode.

Figure A.8 depicts the initial state of the clock generator. To reset the circuit at power-on, the system asserts rst and leaves clk low. The compound gates on the left of Figure A.4 assert $\overline{en}$ to disable the oscillators. Hence, internal node x_1 is held low. The initialization signal *clr* is also asserted, resetting the contents of the shift register. However, the *hld* signal is set to high. When the system releases rst, $\overline{en}$ remains high so the circuit remains in

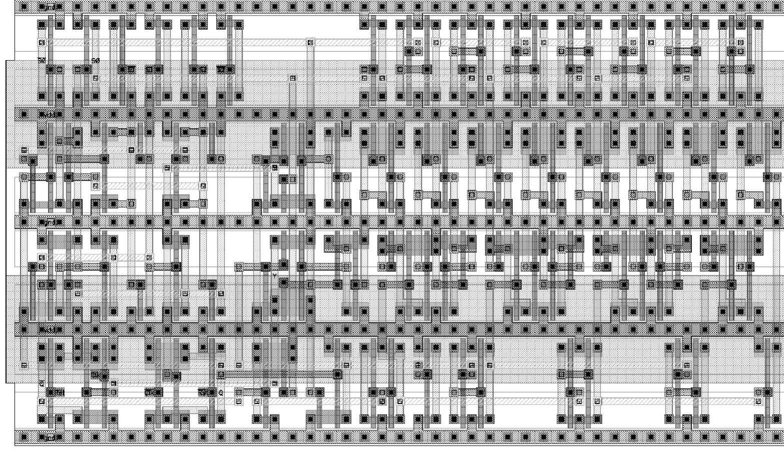


Figure A.7: Layout of clock generator.

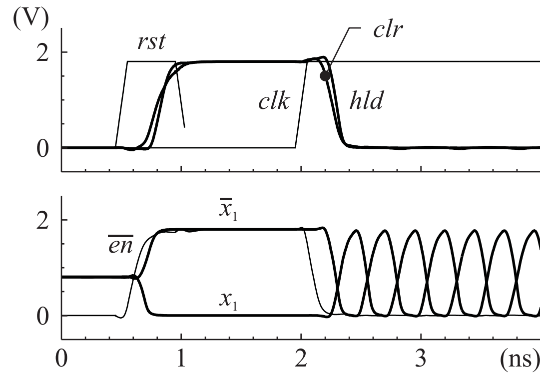


Figure A.8: Reset and start of pulse train.

initialization state.

The rising edge of clk causes the decoder to deactivate $\overline{en}$ and clr, turning on the two oscillators. In the shift register, hld falls low as the first internal values are shifted to the output. The cross-coupled p-transistors in the oscillators align x_1 and $\overline{x_1}$ very well, even at this high frequency.

Figure A.9 illustrates how the clock generator returns to initialization state after producing nine pulses. Assume that clk has transitioned back to low. When pul_0 rises for the last time, the shift register sets hld to high. This event pulls $\overline{en}$ high as well, but the circuit does not enter into initialization state yet. Observe in Figure A.4 that the p-transistor path to

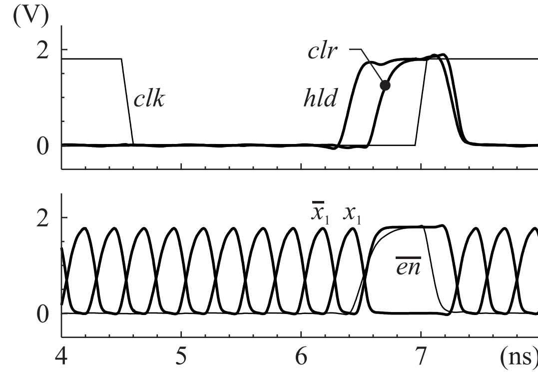


Figure A.9: End of pulse train and initialization.

node x_0 is cut off, preventing any more positive transitions, but the n-transistor path cannot truncate the output pulse prematurely since $\overline{x_1}$ remains low. After a short time, x_2 rises and pulls down x_0. Then $\overline{x_1}$ rises and holds x_0 low. The decoder only asserts *clr* after x_0 transitions low to prevent conflicts in the shift registers. The initialization state lasts until *clk* rises again, repeating the process.

A.3 Analysis

Figure A.10 depicts an overall layout simulation for the differential clock generator. The serial clock contains nine pulses at a frequency of 2 GHz. The system clock runs at 200 MHz, providing some leeway for parameter variations. To increase the oscillation frequency, the n-transistors within the ring oscillators could be enlarged. This change would augment the driving capability of the pull-down stages compared to the cross-coupled p-transistors. Other simulations have suggested that the design performs correctly up to 2.5 GHz, at which point the internal signals no longer swing from rail to rail. To decrease the oscillation frequency, the transistor widths could be decreased, or more stages added to the oscillators.

When scaled to the same 180-nm technology, the NTVW design can run above 2 GHz but does not produce symmetric outputs. Figure A.11 contains simulated waveforms for this

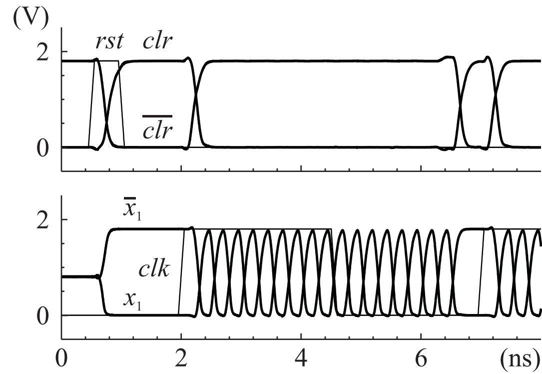


Figure A.10: Layout simulation showing 2-GHz operation.

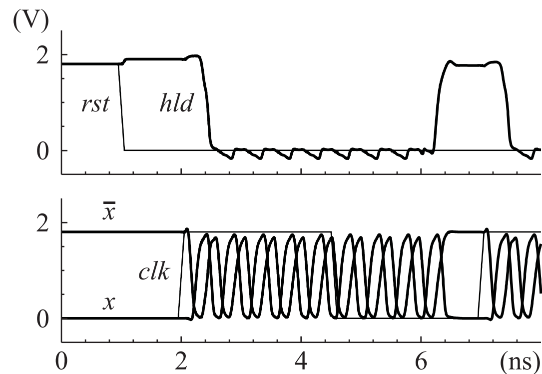


Figure A.11: Simulation of NTVW clock generator [42].

design at 2 GHz. The outputs *pul* and $\overline{pul}$ are misaligned by approximately one-sixth of a cycle. Using these outputs to drive data latches might lead to race conditions. Having a symmetric differential clock allows the module to use high-performance elements such as transmission gates and dynamic circuitry.

The NTVW design also has a problem generating the serial clock at lower frequencies. In Figure A.12, the frequency of the ring oscillator has been reduced to about 1 GHz by adding more inverter stages. The output H of the ring counter rises at the beginning of the last pulse. This event reinitializes the circuit, cutting off the pulse prematurely. One solution to this problem would be to modify the ring counter so that it asserted H at the falling edge

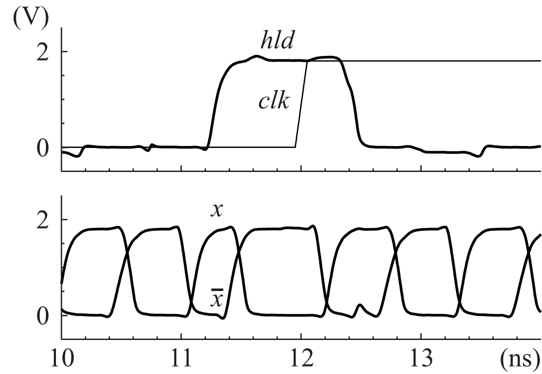


Figure A.12: NTVW design cuts off pulse at lower frequencies.

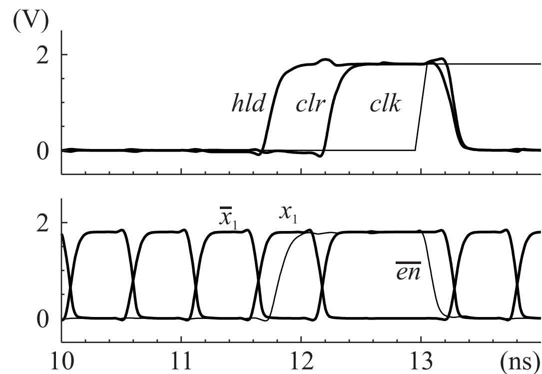


Figure A.13: Proposed design can operate at lower frequencies.

of the last pulse. However, this change would prevent the design from operating at high frequencies, due to the propagation delay between the ring counter and the oscillator.

Figure A.13 demonstrates that the proposed design supports lower frequencies with no changes. As described earlier, stopping the oscillator requires a two-step process whereby the serial clock must complete the last pulse before the circuit initializes. This property also increases the resilience of the clock generator to parasitic capacitances and parameter variations.

The average power consumption of the clock generator is 1.66 mW when running at 2 GHz. Although the NTVW design is not fully delineated in [42], circuit simulations

suggest that its power consumption is around 1.3 mW in the same situation. This result does not include parasitic interconnect capacitances, so the actual power consumption would be closer to the proposed design. Notice that both designs save power by turning off the ring oscillator when not in use.

Future work could consider several improvements to the design. Currently, the system clock must transition from high to low while the serial clock is still running; otherwise, the oscillator does not stop correctly. Modifying the logic to correct this problem would be straightforward. Another potential limitation of the circuit is that it has no mechanism to change the oscillation frequency at runtime. This capability is not essential for the serial cell, but would give the design more robustness against parameter variations. A tuning element such as a current-starved inverter could be added into the oscillation loop. The special initialization circuitry already permits the design to operate over a wide frequency range.

Appendix B

Pipeline Registers

Deeply pipelined systems such as the proposed reconfigurable architecture face several challenges: numerous pipeline registers, extensive clock buffers, and increased power consumption [43]. The delay overhead of the pipeline registers also becomes more significant as the number of stages increases and clock frequencies scale upward. The register delay may actually approach the pipeline stage delay, greatly reducing the time available for computations. For all these reasons, high-speed and low-power flip-flops are essential to sustaining high throughput in deeply pipelined systems.

This appendix describes a novel differential flip-flop that we developed for the reconfigurable architecture. The circuit achieves high energy efficiency by using cross-coupled p-transistors as pull-up devices. We begin by reviewing a number of existing designs found in the literature. We then present the proposed flip-flop and give several circuit simulations that validate its functionality. Finally, we compare the delay and power consumption of the proposed flip-flop with the existing designs.

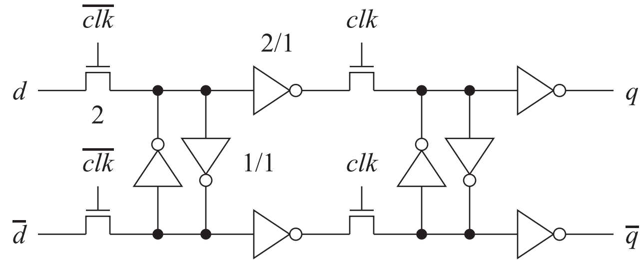


Figure B.1: Basic differential flip-flop.

B.1 Existing designs

Most flip-flops only generate a single-ended output, using an inverter to supply the complement if required. The extra inverter carries a speed penalty and places the two signals out of alignment. On the other hand, differential flip-flops generate both outputs simultaneously. This alternative works well with the complementary data lines used throughout the reconfigurable cell array.

The remainder of this section describes five differential flip-flops: one basic design, and four designs proposed in the literature. We limit our focus here to circuits that generate the outputs at nearly the same time, and operate in a static fashion for the best noise tolerance.

B.1.1 Basic differential flip-flop

Figure B.1 depicts a basic differential flip-flop. The numbers in the figure denote the transistor widths with respect to the minimum size. The circuit works with input data d-$\overline{d}$, output data q-$\overline{q}$, and differential clock clk-$\overline{clk}$. Two pairs of minimum-size inverters serve as memory elements. When clk is low, data from the inputs overwrites the first memory element. The rising edge of clk causes the data to overwrite the second memory element and propagate to the outputs. We use this simple design in later analysis as a baseline for comparison.

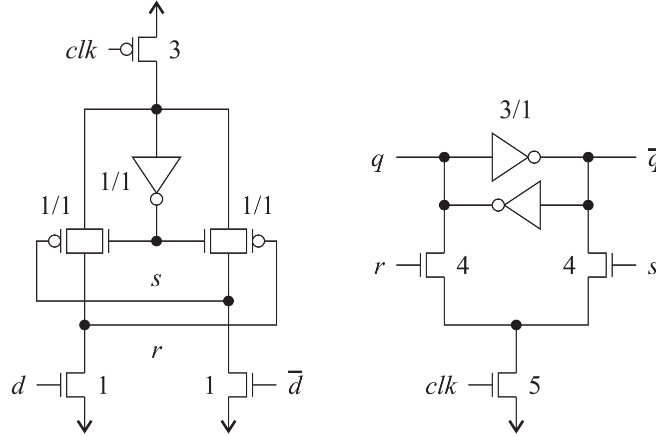


Figure B.2: Static single-transistor clocked (SSTC) flip-flop [44].

B.1.2 SSTC

Researchers have proposed a number of flip-flops that provide complementary outputs. One such design is the static single-transistor clocked (SSTC) flip-flop proposed in [44]. Figure B.2 illustrates how this design consists of a master portion and a slave portion. The role of the master portion is to assert the set signal s or the reset signal r when clk is low. The slave portion uses these signals to change the outputs when clk is high. The extra inverter and n-transistors in the master portion reset s and r if the inputs change while clk is high. Notice that the slave portion maintains the current state when s and r are both low. Unlike the basic flip-flop, the SSTC does not require a differential clock.

B.1.3 SAFF1

Another design taken from the literature is the sense amplifier flip-flop (SAFF1) from [45]. As shown in Figure B.3, this design also contains a master portion and a slave portion. The master portion generates a differential set signal s-$\overline{s}$ and a differential reset signal r-$\overline{r}$. Both $\overline{s}$ and $\overline{r}$ are charged to high when clk is low. The rising edge of clk causes one of these lines to fall to ground. The slave portion then uses these signals to set the outputs of the flip-flop.

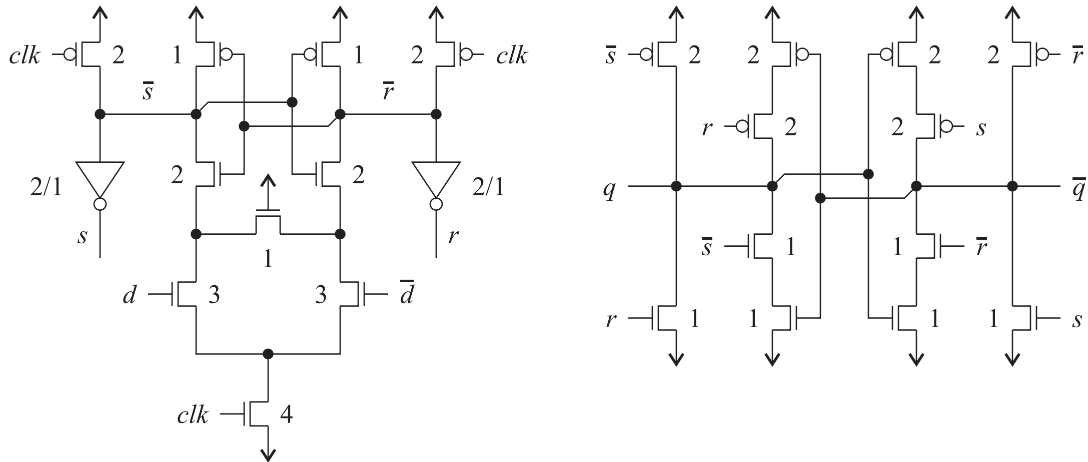


Figure B.3: Sense amplifier flip-flop (SAFF1) [45].

A feedback loop maintains the current state of the outputs when $\overline{s}$ and $\overline{r}$ are high.

B.1.4 SAFF2

A slightly different sense-amplifier flip-flop (SAFF2) appears in [46]. This version uses the same master portion, but reduces the number of transistors in the slave portion. The slave portion now contains a memory element implemented by a pair of inverters. The s-$\overline{s}$ and r-$\overline{r}$ signals control four transistors that set the state the memory element. As with the SAFF1, the SAFF2 only requires a single clock. Figure B.4 depicts the design.

B.2 Proposed design

Figure B.5 illustrates the proposed differential flip-flop. The circuit consists of identical master and slave latches. When *clk* is low, the input data overwrites the contents of the master latch. When *clk* is high, the master latch overwrites the contents of the slave latch. The design is fully static, so *clk* can run at any frequency up to the maximum.

As shown, the master latch contains minimum-size n-transistors that overwrite the stored value when *clk* is low, and maintain the current state when *clk* is high. The slave latch

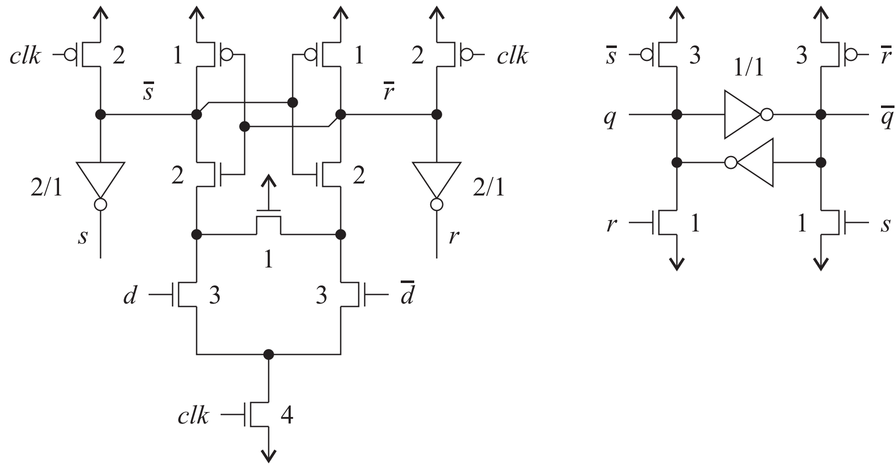


Figure B.4: Modified sense amplifier flip-flop (SAFF2) [46].

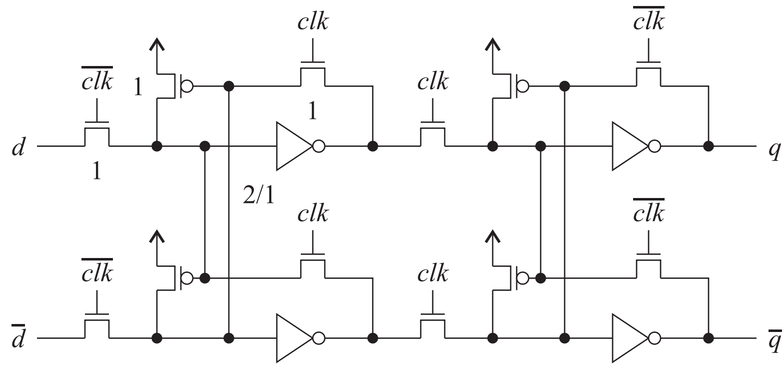


Figure B.5: Proposed differential flip-flop.

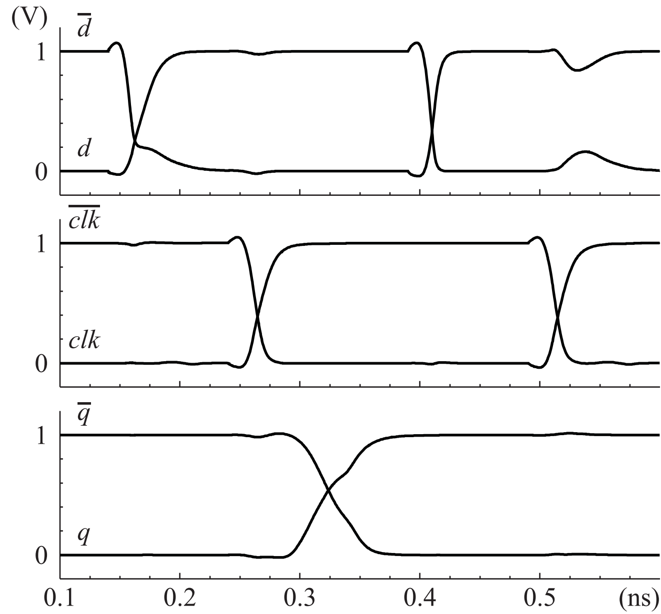


Figure B.6: Simulation of proposed flip-flop.

operates in a complementary manner. The latches also include cross-coupled p-transistors as pull-up devices. These transistors improve noise tolerance by providing full-rail swing at the inverter inputs. Since the minimum-size p-transistors are several times weaker than the n-transistors in the write path, writing new data into the latches does not consume much power.

Figure B.6 contains a circuit simulation of the proposed flip-flop in 90-nm CMOS technology. The input data propagates to the outputs on the rising edge of clk. Changing the data again while clk is high has no effect on q and $\overline{q}$. However, the voltage levels of d and $\overline{d}$ deteriorate slightly when clk falls low and the external circuitry changes the state of the master latch.

Unlike most of the other designs described in the previous section, the proposed flip-flop uses a differential clock. This property might seem disadvantageous, since distributing a global signal requires significant power. However, the system can use the circuit in Figure B.7(a) to generate the differential clock locally. Two cross-coupled NAND gates translate

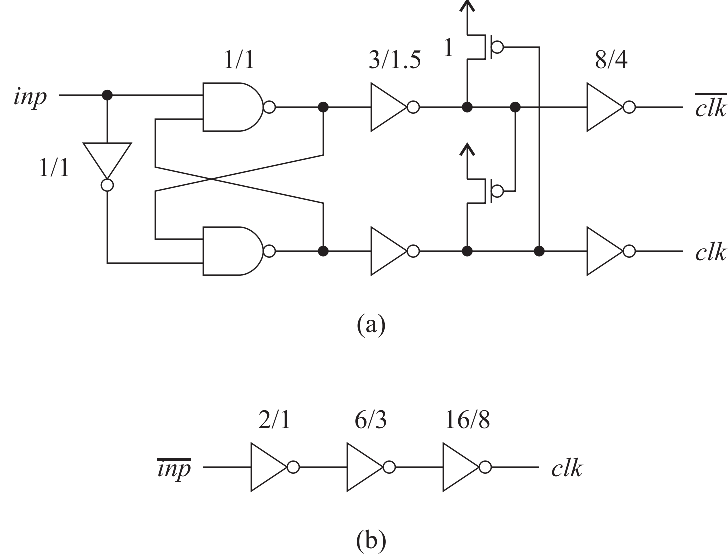


Figure B.7: Comparison of differential and single-ended clock buffers.

the clock input *inp* into relatively symmetric waveforms. Two chains of inverters then buffer the outputs. The cross-coupled p-transistors after the first stage align the differential signals further. This technique resembles the clock generator circuit described in Appendix A. Figure B.7(b) illustrates a single-ended clock buffer for comparison.

A circuit simulation of the differential clock buffer appears in Figure B.8.

B.3 Analysis

In this section, we compare the proposed differential flip-flop to the other alternatives. We begin by defining the parameters we measured in the circuit simulations. Then, we discuss the results of the study.

B.3.1 Methodology

We simulated the five flip-flops in 90-nm CMOS technology at a clock frequency of 2 GHz. To ensure a fair assessment of each circuit, we used the testbench shown in Figure B.9. The

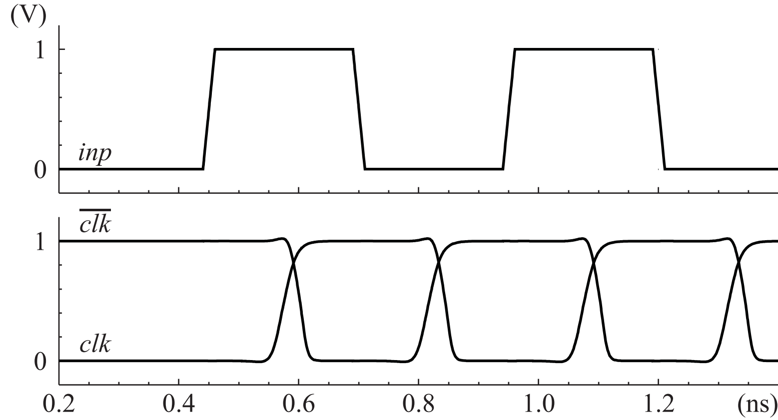
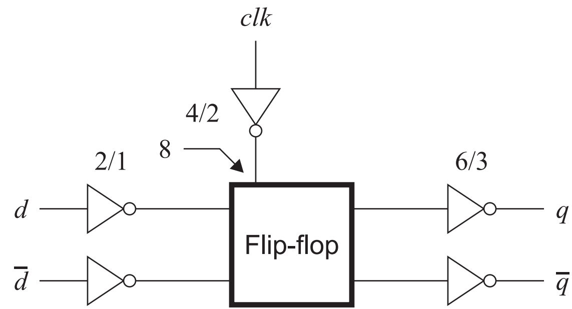


Figure B.8: Simulation of differential clock buffer.

differential inputs are driven by standard inverters for greater realism. Designs that only require a single-ended clock have a double-size inverter as the clock driver. The total load presented to the clock inputs equals 8 units in all cases. The differential outputs drive a pair of triple-size inverters. This configuration models the basic setup in a high-performance pipelined datapath.

We measured a number of delay parameters to characterize the flip-flops using the methodology described in [29] and [47]. Referring to Figure B.10, the clock-output delay t_{ck-q} depends on t_{d-clk} and t_{clk-d}, which describe how long the input data remains stable before and after the clock edge. Decreasing the window of stability increases the clock-output delay until the flip-flop no longer captures the correct value. The *setup time* is the value of t_{d-clk} that minimizes the sum $t_{d-clk} + t_{clk-q}$. We call the corresponding value of t_{ck-q} the *output delay*, and the sum the *total delay*. The total delay places an upper limit on the clock rate in a pipelined system. The *hold time* is the value of t_{clk-d} that minimizes the sum $t_{clk-d} + t_{clk-q}$. The hold time is often negative, meaning that the input data can transition before the clock edge without changing the sampled value.

We also measured the power consumption of the five flip-flops during the simulations. We included the contributions of the input and output buffers in the testbench (Figure B.9),



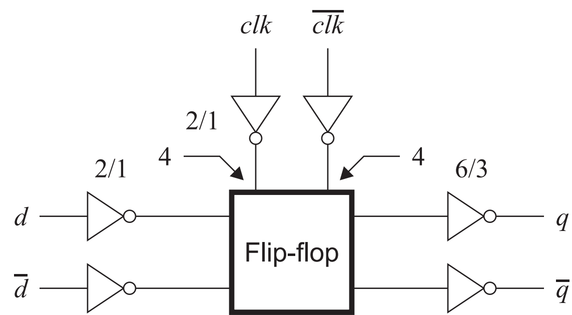


Figure B.9: Testbench for flip-flops.

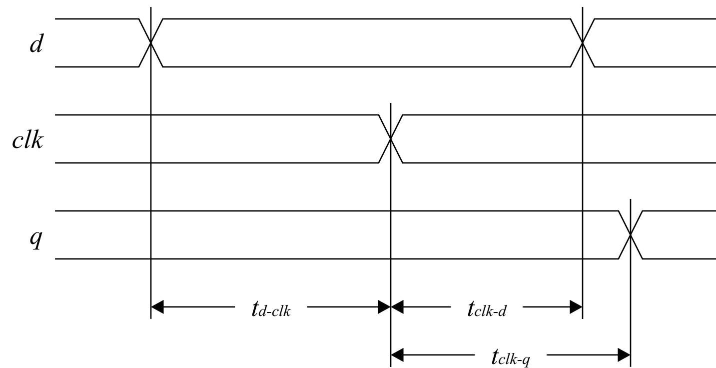


Figure B.10: Delay measurements.

Table B.1: Simulation results for differential flip-flops.

Parameter	Basic	SSTC	SAFF1	SAFF2	Prop.
Transistor count	20	16	26	22	20
Differential clock?	Y	N	N	N	Y
Setup time (ps)	50.5	99.5	−4.8	−9.0	42.4
Hold time (ps)	−20.9	45.1	30.7	30.7	−18.4
Output delay (ps)	77.9	74.6	83.6	85.1	66.7
Total delay (ps)	128.4	174.1	78.8	76.1	109.1
0% power (μW)	4.0	3.8	14.0	13.7	4.6
50% power (μW)	13.8	15.2	19.9	19.8	12.5
100% power (μW)	23.5	26.6	25.8	26.0	20.4
Power-delay (fJ)	1.78	2.64	1.57	1.51	1.36

since different designs have different internal loads. Now the power consumption of a flip-flop depends on the *utilization*, or the probability that the data changes within a given clock cycle. We determined the power consumption at 0% and 100% utilization, and averaged the two values to find the power consumption for random data. Finally, we multiplied this number by the total delay to compute the *power-delay product*. This parameter measures the energy efficiency of the circuit.

B.3.2 Results

Table B.1 presents the results of the simulations. As shown, the proposed flip-flop achieves above-average to excellent results in almost all parameters. For example, only the SSTC requires fewer transistors.

The setup time varies widely between the five alternatives: from −9.0 ps for the SAFF2 to 99.5 ps for the SSTC. The proposed design falls in the middle at 42.4 ps. The hold time ranges from −20.9 ps for the basic flip-flop to 45.1 ps for the SSTC. The proposed design has a hold time of −18.4 ps, very close to the basic flip-flop. The output delay does not show as much variation as the setup time or the hold time, although the proposed flip-flop has the best value at 66.7 ps. As a result, the SAFF2 has the lowest total delay, the SSTC

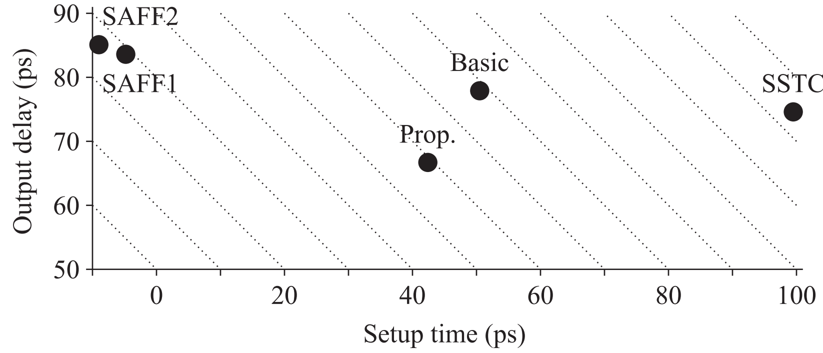


Figure B.11: Plot of setup time versus output delay.

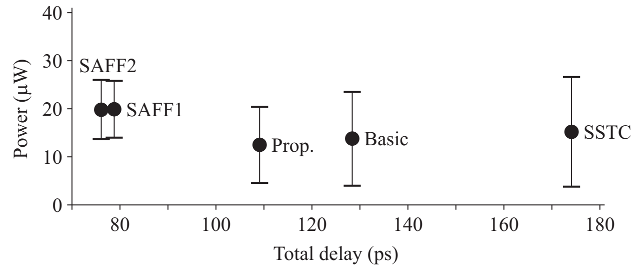


Figure B.12: Plot of total delay versus power consumption.

has the highest, and the proposed flip-flop again falls in the middle. Figure B.11 plots the setup time versus the output delay for each design. The diagonal lines in the figure represent equal values of the total delay.

The experimental results show a clear tradeoff between delay and power. The power consumption at 0% utilization is the lowest for the SSTC, and the highest for the SAFF1 and SAFF2. At 100% utilization, the proposed flip-flop has the lowest power consumption at 20.4 μW. This design also emerges on top for the case of 50% utilization. A plot of the total delay versus the power consumption appears in Figure B.12. Each vertical bar denotes the range of power consumption from 0% to 100% utilization.

We also measured the power consumption of the single-ended and differential clock buffers in Figure B.7. Driving an equivalent load of eight flip-flops, the single-ended circuit consumed 36.8 μW, whereas the differential circuit consumed 48.6 μW. The difference, 11.8 μW, is small

when divided among the eight flip-flops. Hence, the differential clock buffer carries only a small penalty in incremental power consumption.

As a final remark, one difference between this study and related work such as [48] is that the flip-flops do not drive a large output load. Thus, we used small transistors throughout the five circuits. Many other studies assume that the flip-flops will be part of a standard-cell library, and hence size the transistors for greater driving capacity. It is possible that the relative performance of the five designs might be different in this case. However, the cited study also found that the SAFF achieved better performance than the SSTC.

Appendix C

Publications

This appendix gives a list of papers published and submitted for publication during the course of this research.

C.1 Journal papers

1. M.J. Myjak and J.G. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," *Microelectronic Engineering*, in press.
2. M.J. Myjak and J.G. Delgado-Frias, "Medium-grain cells for reconfigurable DSP hardware," *IEEE Trans. on Circuits and Systems*, under second review.
3. M.J. Myjak and J.G. Delgado-Frias, "Array multipliers for reconfigurable DSP hardware," *IEEE Trans. on Computers*, submitted for publication.
4. M.J. Myjak and J.G. Delgado-Frias, "A medium-grain reconfigurable architecture for DSP: VLSI design, benchmark mapping, and performance," *IEEE Trans. on VLSI Systems*, submitted for publication.

C.2 Conference papers

1. J.G. Delgado-Frias, M.J. Myjak, F.L. Anderson, and D.R. Blum, "A medium-grain reconfigurable cell array for DSP applications," in *Proc. 3rd IASTED Int. Conf. on*

Circuits, Signals, and Systems, Cancun, Mexico, pp. 231–236, May 2003.

2. M.J. Myjak and J.G. Delgado-Frias, "A two-level reconfigurable architecture for digital signal processing," in *Proc. 2003 Int. Conf. on VLSI*, Las Vegas, NV, pp. 21–27, Jun. 2003.

3. M.J. Myjak and J.G. Delgado-Frias, "Pipelined multipliers for reconfigurable hardware," in *Proc. 11th Reconfigurable Architectures Workshop*, Santa Fé, NM, pp. 150–154, Apr. 2004.

4. M.J. Myjak, F.L. Anderson, and J.G. Delgado-Frias, "H-tree interconnection structure for reconfigurable DSP hardware," in *Proc. 2004 Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, pp. 170–176, Jun. 2004.

5. M.J. Myjak, D.R. Blum, and J.G. Delgado-Frias, "Enhanced fault-tolerant CMOS memory elements," in *Proc. 2004 IEEE Int. Midwest Symposium on Circuits and Systems*, Hiroshima, Japan, Jul. 2004.

6. D.R. Blum, M.J. Myjak, and J.G. Delgado-Frias, "Enhanced fault-tolerant data latches for deep submicron CMOS," in *Proc. 2005 Int. Conf. on Computer Design*, Las Vegas, NV, pp. 28–34, Jun. 2005.

7. M.J. Myjak and J.G. Delgado-Frias, "A symmetric differential clock generator for bit-serial hardware," in *Proc. 2005 Int. Conf. on Computer Design*, Las Vegas, NV, pp. 159–164, Jun. 2005.

8. M.J. Myjak and J.G. Delgado-Frias, "A bit-serial cell for reconfigurable DSP hardware," in *Proc. 2005 IEEE Int. Midwest Symposium on Circuits and Systems*, Cincinnati, OH, pp. 960–963, Aug. 2005.

9. M.J. Myjak and J.G. Delgado-Frias, "Superpipelined reconfigurable hardware for DSP," in *Proc. 2006 IEEE Int. Symposium on Circuits and Systems*, Kos, Greece, May 2006, to be published.

10. M.J. Myjak, J.K. Larson, and J.G. Delgado-Frias, "Mapping and performance of DSP benchmarks on a medium-grain reconfigurable architecture," in *Proc. 2006 Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, Jun. 2006, to be published.

11. M.J. Myjak, J.G. Delgado-Frias, and S.K. Jeon, "An energy-efficient differential flip-flop for deeply pipelined systems," *2006 IEEE Int. Midwest Symposium on Circuits and Systems*, San Juan, Puerto Rico, Aug. 2006, submitted for publication.