MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report No. 1586 October, 1996

Reconfigurable Architectures for General-Purpose Computing

André DeHon
andre@mit.edu

Abstract: General-purpose computing devices allow us to (1) customize computation after fabrication and (2) conserve area by reusing expensive active circuitry for different functions in time. We define *RP*-space, a restricted domain of the general-purpose architectural space focussed on reconfigurable computing architectures. Two dominant features differentiate reconfigurable from special-purpose architectures and account for most of the area overhead associated with *RP* devices: (1) *instructions* which tell the device how to behave, and (2) *flexible interconnect* which supports task dependent dataflow between operations.

We can characterize *RP*-space by the allocation and structure of these resources and compare the efficiencies of architectural points across broad application characteristics. Conventional FPGAs fall at one extreme end of this space and their efficiency ranges over two orders of magnitude across the space of application characteristics. Understanding *RP*-space and its consequences allows us to pick the best architecture for a task and to search for more robust design points in the space.

Our DPGA, a fine-grained computing device which adds small, on-chip instruction memories to FPGAs is one such design point. For typical logic applications and finite-state machines, a DPGA can implement tasks in one-third the area of a traditional FPGA. TSFPGA, a variant of the DPGA which focuses on heavily time-switched interconnect, achieves circuit densities close to the DPGA, while reducing typical physical mapping times from hours to seconds.

Rigid, fabrication-time organization of instruction resources significantly narrows the range of efficiency for conventional architectures. To avoid this performance brittleness, we developed MATRIX, the first architecture to defer the binding of instruction resources until run-time, allowing the application to organize resources according to its needs. Our focus MATRIX design point is based on an array of 8-bit ALU and register-file building blocks interconnected via a byte-wide network. With today's silicon, a single chip MATRIX array can deliver over 10 Gop/s (8-bit ops). On sample image processing tasks, we show that MATRIX yields 10-20$\times$ the computational density of conventional processors.

Understanding the cost structure of *RP*-space helps us identify these intermediate architectural points and may provide useful insight more broadly in guiding our continual search for robust and efficient general-purpose computing structures.

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

Acknowledgments

This effort grew out the intellectual backdrop of the Transit and Abacus projects. Years prototyping Transit machines with Tom Simon and his specialization philosophy set the stage for my initial interest in FPGAs for computing. The initial ideas for the DPGA grew out of dialogs with Mike Bolotski in which we tried to reconcile Abacus, his SIMD architecture which he described as "a bunch of one-bit processors, " with FPGAs, which looked to me like "a bunch of one-bit processors."

Tom Knight has been my research advisor since I was a junior. He has always encouraged me to focus on the big idea and has been supportive as I explored sometimes radical points of view. He gave me plenty of freedom to do the right thing, and hopefully, I have lived up to the confidence and trust implied by that autonomy.

The efforts of Jeremy Brown, Derrick Chen, Ian Eslick, Ethan Mirsky, and Edward Tau during and after 6.371 made the DPGA prototype possible. Ian's perseverance to finalize the layout and verification was particular responsible for the completion of that effort. Ed and Ian both helped see the DPGA prototype through its final postmortem.

TSFPGA and MATRIX were both possible only because of Derrick Chen and Ethan Mirsky, the Master of Engineering students who respectively took ownership of the microarchitecture and VLSI portions of those designs. We were largely able to complement each other's efforts in our attempts to understand and develop these architectures.

Discussion with Rich Lethin, Russ Tessier, and Jonathan Babb at MIT were useful in focusing in on the key issues which needed addressing.

Regular interaction with the emerging reconfigurable computing community was valuable for encouragement and for identifying key problems and issues. Notably, discussions with Brad Taylor, Mike Butts, Brad Hutchings, Bill Magione-Smith, John Villasenor, Phil Kuekes, Steve Trimberger, Mike Smith, and Carl Ebling have been helpful in identifying the questions which need answers and cleaning up ideas for presentation.

Thomas McDermott provided valuable feedback on the early chapters of this work.

The availability of high-quality, experimental CAD tools in source form from universities made the experimental mapping work done here feasible. University of Toronto's Chortle provided a clean basis for several early experiments in DPGA synthesis. UC Berkeley's SIS was used for standard, technology independent circuit mapping. UC Berkeley's mustang was the workhorse behind multicontext FSM mapping.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

Contents

List of Figures

List of Tables

Part I

Introduction and Background

1. Overview and Synopsis

1.1 Evolution of General-Purpose Computing with VLSI Technology

General-purpose computers have served us well over the past couple of decades. Broad applicability has led to wide spread use and volume commoditization. Flexibility allows a single machine to perform a multitude of functions and be deployed into applications unconceived at the time the device was designed or manufactured. The flexibility inherent in general-purpose machines was a key component of the computer revolution.

To date, processors have been the driving engine behind general-purpose computing. Originally dictated by the premium for active real estate, processors focus on the heavy reuse of a single or small number of functional units. With Very Large Scale Integration (VLSI), we can now integrate complete and powerful processors onto a single integrated circuit, and the technology continues to provide a growing amount of real estate.

As enabling as processors have been, our appetite and need for computing power has grown faster. Despite the fact that processor performance steadily increases, we often find it necessary to prop up these general-purpose devices with specialized processing assists, generally in the form of specialized co-processors or ASICs. Consequently, today's computers exhibit an increasing disparity between the general-purpose core and its specialized assistants. High performance systems are built from a plethora of specialized ASICs. Even today's high-end workstations dedicate more active silicon to specialized processing than to general-purpose compute. The general-purpose processor will be only a small part of tomorrow's multi-media PC. As this trend continues, the term "general-purpose computer" will become a misnomer for modern computer systems. Relatively little of the computing power in tomorrow's computers can be efficiently deployed to solve any problem.

The problem is not with the notion of general-purpose computing, but with the implementation technique. For the past several years, industry and academia have focussed largely on the task of building the highest performance processor, instead of trying to build the highest performance general-purpose computing engine. When active area was extremely limited, this was a very sensible approach. However, as silicon real estate continues to increase far beyond the space required to implement a competent processor, it is time to re-evaluate general-purpose architectures in light of shifting resource availability and cost.

In particular, an interesting space has opened between the extremes of general-purpose processors and specialized ASICs. That space is the domain of *reconfigurable computing* and offers all the benefits of general-purpose computing with greater performance density than traditional processors. This space is most easily seen by looking at the *binding time* for device function. ASICs bind function to active silicon at fabrication time making the silicon useful only for the designated function. Processors bind functions to active silicon only for the duration of a single cycle, a restrictive model which limits the amount the processor can accomplish in a single cycle while requiring considerable on-chip resources to hold and distribute instructions. Reconfigurable devices allow functions to be bound at a range of intervals within the final system depending on the needs of the application. This flexibility in binding time allows reconfigurable devices to make better use of the limited device resources including instruction distribution.

Consequently, reconfigurable computing architectures offer:

- More application-specific adaptation than conventional processors
- Greater computational density than conventional processors
- More and broader reuse of silicon than ASICs
- Better opportunities to ride hardware and algorithmic technology curves than ASICs
- Better match to current technology costs than ASICs or processors

1.2 This Thesis

This thesis characterizes a class of reconfigurable computing architectures and relates them broadly to the more well understood conventional alternatives. Since technology costs dictate the architectural tradeoffs involved, this characterization is performed in the context of MOS VLSI implementations. The convergence of process technologies along with the large amount of silicon real-estate available on a single die these days allows us to perform broad comparisons based primarily on silicon area.

The thesis provides:

1. A *high level characterization* of a reconfigurable processing space which includes reconfigurable architectures such as FPGAs. This characterization helps us understand the key characteristics of reconfigurable devices, including when and what level of performance we can extract from various architectural points.

2. *Empirical relations* on the key building blocks in CMOS VLSI taken from existing designs in the literature and our own experimental designs, include:
   - sizes (*e.g.* How big is a 4-LUT?)
   - performance density (operations per unit space-time)
   - relative feature sizes (*e.g.* interconnect versus configuration memory versus active computing)
   - first order modeling of key area factors

3. *Architecture designs and implementations* which explore new points in the identified design space based on the empirical characterization.
   - architectures which exploit the identified cost structure to provide greater functional density for reconfigurable devices
   - architectures which allow diversity/density tradeoffs based on application characteristics

4. *Lessons and observations* for future device architects and systems designers

The major contributions of this thesis include:

1. *RP*-space model for reconfigurable processing architectures – While many loose taxonomies exist for general-purpose computing, none are as systematic as the one presented here.

By focusing on the *RP*-space domain, this model provides size estimates and facilitates pedagogical efficiency comparisons of architectures within the space.

2. DPGA – A novel bit-level architecture with multiple, on-chip instructions per compute element, including the theory and concepts behind the architecture, an implementation, experimental CAD to support it, and validation of efficiency using standard circuit benchmarks. For typical logic circuits and finite-state machines, the DPGA implementation is one-third the size of the FPGA implementation.

3. TSFPGA – A model, possible implementation, and CAD for fine-grained, time-switched interconnect with demonstrated fast physical mapping capabilities. TSFPGA exploits the observations that most of the area benefits in DPGAs come from sharing the interconnect and that most of the difficulty in mapping to traditional FPGAs is their limited interconnect. By sharing interconnect resources in time, TSFPGA extracts more interconnect functionality from less active switching resources. For typical applications, quick mapping can be done in seconds. The mapped design area is smaller than comparable FPGAs and slightly larger than comparable DPGAs.

4. MATRIX – The first architecture to allow run-time binding of instruction resources. Focusing on a design point using an array of 8-bit ALU and register-file building blocks interconnected via a byte-wide network, MATRIX yields 10-20$\times$ the computational density of conventional processors on sample image processing tasks. With today's silicon, we can place hundreds of these 8-bit functional units on a large die operating at 100MHz, making it possible to deliver over 10 Gop/s (8-bit ops) per component.

The remainder of this chapter provides a synopsis of the key results and relationships developed in the thesis. This introductory part of the thesis continues with Chapter 2 which defines the terminology and metrics used throughout the thesis. Chapter 3 reviews and highlights the existing evidence for the high performance potential of reconfigurable computing architectures.

Part II sets the stage by examining the computational capabilities of existing general-purpose computing devices. This starts with a broad, empirical, review of general-purpose architectures in Chapter 4. In Chapter 5, we compare hardwired and general-purpose multiplier implementations as a case study bridging general-purpose and application-dedicated architectures. In Chapter 6, we review processor architectures implemented on top of reconfigurable architectures to broaden the picture and to see one way in which conventional reconfigurable architectures deal with high operational diversity.

Part III takes a more compositional view of reconfigurable computing architectures. Chapter 7 looks at building blocks, sizes, and requirements for interconnect. Chapter 8 looks at resource requirements for instruction distribution. Finally in Chapter 9, we bring the empirical data, interconnect, and instruction characteristics together, providing a first order model of *RP*-space, our high-level model for reconfigurable processing architectures.

Part IV includes three new architectures: DPGA (Chapters 10 and 11), TSFPGA (Chapter 12), and MATRIX (Chapter 13), which are highlighted below in Section 1.6. The final chapters in Part V, review the results and identify promising directions for the future.

1.3 Reconfigurable Device Characteristics

Broadly considered, reconfigurable devices fill their silicon area with a large number of computing primitives, interconnected via a configurable network. The operation of each primitive can be programmed as well as the interconnect pattern. Computational tasks can be implemented *spatially* on the device with intermediates flowing directly from the producing function to the receiving function. Since we can put thousands of reconfigurable units on a single die, significant data flow may occur without crossing chip boundaries. To first order, one can think about turning an entire task into hardware dataflow and mapping it on the reconfigurable substrate. Reconfigurable computing generally provides spatially-oriented processing rather than the *temporally*-oriented processing typical of *programmable* architectures such as microprocessors.

The key differences between reconfigurable machines and conventional processors are:

- **Instruction Distribution** – Rather than broadcasting a new instruction to the functional units on every cycle, instructions are locally configured, allowing the reconfigurable device to *compress* instruction stream distribution and effectively deliver more instructions into active silicon on each cycle.

- **Spatial routing of intermediates** – As space permits, intermediate values are routed in parallel from producing function to consuming function rather than forcing all communication to take place in time through a central resource bottleneck.

- **More, often finer-grained, separately programmable building blocks** – Reconfigurable devices provide a large number of separately programmable building blocks allowing a greater range of computations to occur per time step. This effect is largely enabled by the compressed instruction distribution.

- **Distributed deployable resources, eliminating bottlenecks** – Resources such as memory, interconnect, and functional units are distributed and deployable based on need rather than being centralized in large pools. Independent, local access allows reconfigurable designs to take advantage of high, local, parallel on-chip bandwidth, rather than creating a central resource bottleneck.

1.4 Configurable, Programmable, and Fixed-Function Devices

To establish an intuitive feel for the design point and role of configurable devices, we can take a high-level look at conventional devices. Ignoring, for the moment multiplies, floating-point operations, and table lookup computations, the modern processor has a peak performance on the order of 256, 3-LUT gate-evaluations per clock cycle (*e.g.* two 64-bit ALUs). A modern FPGA has a peak performance on the order of 2,048, 4-LUT gate-evaluations per clock cycle. The basic clock cycle time is comparable giving the FPGA at least an order of magnitude larger raw capacity.

Note that both the processor ALUs and FPGA blocks are typically built with additional gates which serve to lower the *latency* of word operations without increasing the raw throughput (*e.g.* fast carry chains which allow a full 64-bit wide add to complete within one cycle time). This latency reduction may be important to reducing the serial path length in tasks with limited parallelism, but is not reflected in this raw capacity comparison.

The FPGA can sustain its peak performance level as long as the *same* 2K gate-evaluation functionality is desired from cycle to cycle. Wiring and pipelining limitations are the primary reason the FPGA would achieve lower than peak performance, and this is likely to account for, at most, a 20-50% reduction from peak performance. If more diverse functionality is desired from a single FPGA than the 1-2K gate-evaluations provided by the FPGA, performance drops considerably due to function reload time.

The processor is likely to provide a much lower peak performance and the effect is much more application specific. Due to the bitwise-SIMD nature of traditional ALUs, work per cycle can be as low as a couple of gate-evaluations on compute operations. Since processors perform all "interconnect" using shifts, moves, loads, and stores, many cycles yield no gate-evaluations, only movement of data. The lower peak performance of processors comes from the fact that the processor ALU occupies only a small fraction of the die, with substantial area going to instruction flow control and on-chip memory to support large sequences of diverse operations without requiring off-chip instruction or data access.

A comparably sized, dedicated piece of hardwired functionality, with no memory could provide a capacity of 200,000-300,000 4-LUT gate-evaluations per clock cycle, at potentially higher clock rates. While the raw gate delay on the hardwired logic can be $10\times$ smaller than on the FPGA, reasonable cycle times in equivalent logic processes are closer to $2\times$ since it makes sense to pipelined the FPGA design at a more shallow logic depth than the custom logic. Returning to the multiplier, for example, such a chip might provide 64K multiply bit operations per cycle (*e.g.* a 256×256 multiply pipelined at the byte level). The dedicated hardware provides 100-300 times the capacity of the FPGA on the one task it was designed to solve. To first order, the dedicated hardware can deliver very little capacity to significantly different applications. It is also worthwhile to note that the fixed granularity of hardwired devices often causes them to sacrifice much of their capacity advantage when used on small data items. For instance, performing an 8×8 multiply on a 64×64 hardwired multiplier makes use of only $\frac{1}{64}$'th of the multiplier's capacity, removing much of its $300\times$ capacity advantage.

Combining these observations, we can categorize the circumstances under which the various structures are prefered.

- **Fixed Function, Limited Operation Diversity, High Throughput** – When the function and data granularity to be computed are well-understood and fixed, and when the function can be economically implemented in space, dedicated hardware provides the most computational capacity per unit area to the application.

- **Variable Function, Low Diversity** – If the function required is unknown or varying, but the instruction or data diversity is low, the task can be mapped directly to a reconfigurable computing device and efficiently extract high computational density.

- **Space Limited, High Entropy** – If we are limited spatially and the function to be computed has a high operation and data diversity, we are forced to reuse limited active space heavily and accept limited instruction and data bandwidth. In this regime, conventional processor organization are most effective since they dedicate considerable space to on-chip instruction

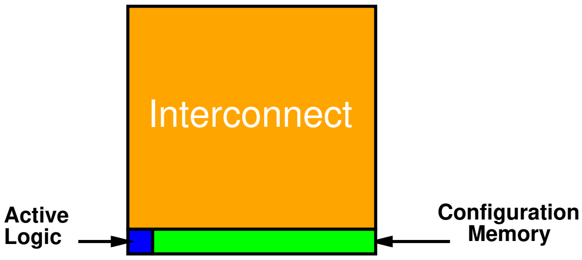


Figure 1.1: First Order Size Comparison for Configurable Designs

storage in order to minimize off-chip instruction traffic while executing descriptively complex tasks.

Reconfigurable devices have become increasingly interesting as aggregate IC capacity has grown large enough to adequately hold the computational diversity of many computing tasks or, at least, the key kernels of these tasks. As the area available for general-purpose computing devices increases, more tasks will fit conveniently on reconfigurable devices, increasing the range of applications where the reconfigurable solution yields higher performance per unit area.

In Chapter 2 we define our evaluation and comparison metrics more carefully. Chapters 4 and 5 provide an empirical review of conventional general-purpose and specialized architectures, focusing on their performance density.

1.5 Key Relations

While reconfigurable devices have, potentially, 100$\times$ less performance per unit area than hardwired circuitry, they provide 10-100$\times$ the performance density of processors. As noted above, FPGAs offer a potential 10$\times$ advantage in raw, peak, general-purpose functional density over processors. This density advantage comes largely from dedicating significantly less instruction memory and distribution resources per active computing element. At the same time this lower memory ratio allows reconfigurable devices to deploy active capacity at a finer grained level, allowing them to realize a higher yield of their raw capacity, sometimes as much as 10$\times$, than conventional processors. It is these two effects taken together which give reconfigurable architectures their 10-100$\times$ performance density advantage over conventional processor architectures in many situations.

From an empirical review of conventional, reconfigurable devices, we see that 80-90% of the area is dedicated to the switches and wires making up the reconfigurable interconnect. Most of the remaining area goes into configuration memory for the network. The actually logic function only accounts for a few percent of the area in a reconfigurable device. This interconnect and configuration overhead is responsible for the 100$\times$ density disadvantage which reconfigurable devices suffer relative to hardwired logic.

To a first order approximation, this gives us:

$$Area_{net} = 10 \times Area_{config} = 100 \times Area_{logic} \tag{1.1}$$

It is this basic relationship (Shown diagrammatically in Figure 1.1) which characterizes the *RP* design space.

- Since $Area_{config} << Area_{net}$, devices with a single on-chip configuration, such as most reconfigurable devices, can afford to exert fine-grained control over their operations – any savings associated with sharing configuration bits would be small compared to the network area.

- Since $Area_{config} << Area_{net}$, to pack the most functional diversity into a part, one can allocate multiple configurations on chip. With the order-of-magnitude relative sizes given in Relation 1.1, up to a $10\times$ increase in the functional diversity per unit area is attainable in this manner.

- However, since $Area_{net}$ is only $10 \times Area_{config}$, if the number of configurations is large, say 100 or more, the configuration memory area will become the dominant size factor. Processors are essentially optimized into this regime and that partially accounts for their $10\times$ lower raw performance density compared to reconfigurable devices.

- Once we go to a large number of contexts, such that the total configuration memory space begins to dominate interconnect area, fine-granularity becomes costly. In this regime wide-word operations allow us to amortize instruction area across multiple bit processing elements. This simultaneously allows machines with wide (*e.g.* 32 bit) datapaths to hold 1000's of configurations on chip while making them only $10\times$ less computationally dense than fine-grained, single context devices.

After reviewing implementations in Chapter 4, Chapters 7 and 8 examine interconnect and instruction delivery issues in depth. Chapter 9 brings these together, yielding a slightly more sophisticated model than the one above to explain the primary tradeoffs in the design of reconfigurable computing architectures.

1.6 New General-Purpose Architectures

From the general relationships above, we see that conventional conventional Field Programmable Gate Arrays (FPGAs) represent one extreme in our *RP*-space. The space is large, leaving considerable space for interesting architectures in middle. Exploiting the relative area properties identified above and common device usage scenarios, we have developed three new general-purpose computing architectures. By judicious allocation of device resources, these architectures offer higher yielded capacity over a wide range of applications.

DPGA The Dynamically Programmable Gate Array (DPGA) is a multicontext FPGA, formed by associating memory for several configurations with each active LUT and interconnect switch (See Figure 1.2). From Relation 1.1, we see that the area associated with context memory is small and can be replicated several times without substantially impacting active device capacity. The multicontext design allows the device to reuse its active capacity to provide additional functionality rather than additional throughput. For the operations required by an application which are not

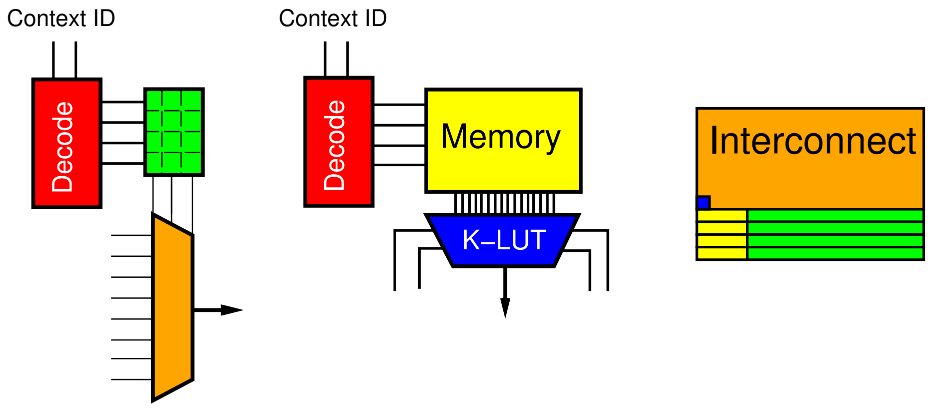


Figure 1.2: LUT and Interconnect Primitives for Multicontext FPGA

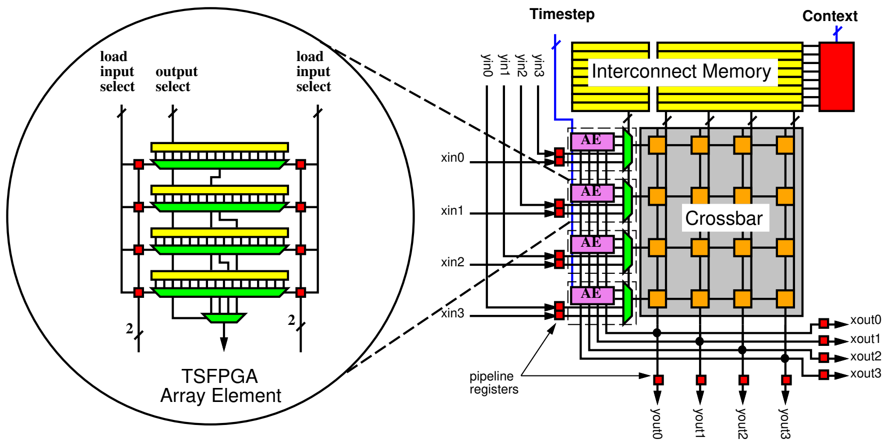


Figure 1.3: TSFPGA Organization

the throughput bottleneck, the multicontext device yields higher device capacity than conventional FPGA architectures. Chapter 10 describes our 4-context DPGA design and implementation, identifies several common usage scenarios, and details experimental mapping techniques for circuits and finite-state machines. Chapter 11 extends the basic DPGA model and circuit mapping tools to include input retiming registers. The resulting architecture achieves 3$\times$ the density of conventional FPGAs without sacrificing performance on typical applications.

TSFPGA A careful review of the DPGA implementation and Relation 1.1, reminds us that the active logic portion of a reconfigurable design comprises only a small fraction of the space while the programmable network is the key area consumer. The Time-Switched FPGA (TSFPGA) focuses on reuse of the critical switch and wire resources (See Figure 1.3). By pipelining the switching

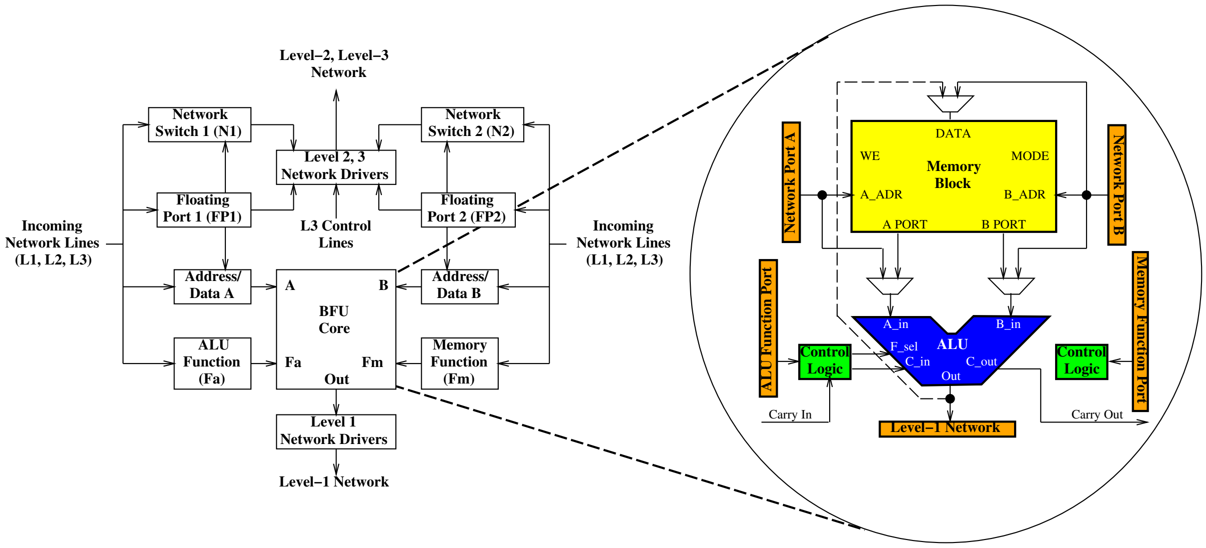


Figure 1.4: MATRIX Basic Functional Unit

operations TSFPGA allows us to extract higher capacity from the available switches and wires. At the same time, the switched interconnect allows each individual switching element to play a number of different roles. Consequently, TSFPGA compresses switching requirements, providing more effective switching capacity with less physical interconnect. The greater yielded switching capacity allows physical design mapping to occur rapidly – in seconds rather than the hours typical of conventional FPGA architectures. Chapter 12 details the TSFPGA design, implementation, and experimental mapping software.

MATRIX All prior general-purpose computing architectures, including processors, FPGAs, and the two previous architectures make a rigid distinction between instruction and control resources which manage computation and the computing resources which perform computations for an application. Consequently, one must make a fabrication time decision about the device's control structure and the deployment of resources for control. We see in Chapters 8 and 9 that this decision has a large impact on the distribution of dedicated instruction resources in the design and the range of applications where the device is efficiently employed. MATRIX is a novel, coarse-grained, computing architecture which uses a multilevel configuration scheme to defer this binding to the application (See Figure 1.4). Our focus implementation uses an 8-bit primitive datapath element for the basic functional unit. Rather than separate the resources for instruction storage and distribution from the resources for data storage and computation and fix them at fabrication time, the MATRIX architecture unifies these resources. Once unified, traditional instruction and control resources are decomposed along with computing resources and can be deployed in an application-specific manner. Chip capacity can be deployed to support active computation or to control reuse of computational resources depending on the needs of the application and the available hardware resources. As a result, MATRIX can be efficiently employed across a broader range of computational characteristics than conventional architectures. Chapter 13 introduces the MATRIX architecture and shows how it obtains these unique characteristics.

1.7 Prognosis for the Future

Ultimately, reconfiguration is a technique for compressing the resources dedicated to instruction stream distribution while maintaining a general-purpose architecture. As such, it is an important architectural tool for extracting the highest performance from our silicon real estate. Characteristics of an application which change slowly or do not change can be configured rather than broadcast. The savings in instruction control resources result in higher logic capacity per unit area.

With CMOS VLSI we have reached to the point where we are no longer so limited by the aggregate capacity of a single IC die that the device must be optimized exclusively to maximize the number of distinct instructions resident on a chip. Beyond this point spatial implementation of all or portions of general-purpose computations is both feasible and beneficial. From this point on we will see:

1. More applications and kernels fit for spatial implementations on reconfigurable substrates
2. Reconfigurable techniques find their way into general-purpose and flexible computing devices, changing the way we design even "nominally" conventional architectures

Reconfigurable architectures and techniques should be added to the modern computer architect's repertoire of design techniques, alongside more venerable ones such as microprogramming, translation, and caching.

The thesis closes in Part V by reviewing the key lessons from reconfigurable designs and their implications for future general-purpose architectures.

2. Basics and Terminology

In this chapter we introduce much of the terminology used throughout the document. We start with a high-level review of *general-purpose* computing. We define the distinction between *programmable* and *configurable* devices and the various components of configurable devices. Much of the discussion will take Field-Programmable Gate Arrays as a basis, so we introduce an initial, conceptual FPGA model. Finally, we define metrics for capacity, density, and diversity which will be used when characterizing the various architectures reviewed. A glossary summarizing terminology follows Chapter 16.

2.1 General-Purpose Computing

General-purpose computing devices are specifically intended for those cases where, economically, we cannot or need not dedicate sufficient spatial resources to support an entire computational task or where we do not know enough about the required task or tasks prior to fabrication to hardwire the functionality. The key ideas behind general-purpose processing are:

1. Defer binding of functionality until device is employed – *i.e.* after fabrication
2. Exploit *temporal reuse* of limited functional capacity

Delayed binding and temporal reuse work closely together and occur at many scales to provide the characteristics we now expect from general-purpose computing devices.

We are quite accustomed to exploiting these properties so that unique hardware is not required for every task or application. This basic theme recurs at many different levels in our conventional systems:

- **Market Level** – Rather than dedicating a machine design to a single application or application family, the design effort may be utilized for many different applications.
- **System Level** – Rather than dedicating an expensive machine to a single application, the machine may perform different applications at different times by running different sets of instructions.
- **Application Level** – Rather than spending precious real estate to build a separate computational unit for each different function required, central resources may be employed to perform these functions in sequence with an additional input, an *instruction*, telling it how to behave at each point in time.
- **Algorithm Level** – Rather than fixing the algorithms which an application uses, an existing general-purpose machine can be reprogrammed with new techniques and algorithms as they are developed.

- **User Level** – Rather than fixing the function of the machine at the supplier, the instruction stream specifies the function, allowing the end user to use the machine as best suits his needs. Machines may be used for functions which the original designers did not conceive. Further, machine behavior may be upgraded in the field without incurring any hardware or hardware handling costs.

In the past, processors were virtually the only devices which had these characteristics and served as general-purpose building blocks. Today, many devices, including reconfigurable components, also exhibit the key properties and benefits associated with general-purpose computing. These devices are economically interesting for all of the above reasons.

2.2 General-Purpose Computing Issues

There are two key features associated with general-purpose computers which distinguish them from their specialized counterparts. The way these aspects are handled plays a large role in distinguishing various general-purpose computing architectures.

2.2.1 Interconnect

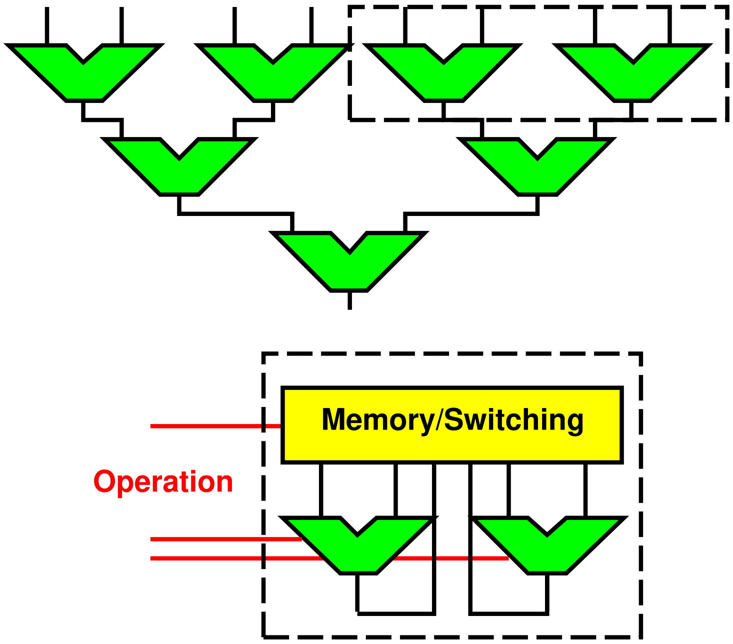
In general-purpose machines, the datapaths between functional units cannot be hardwired. Different tasks will require different patterns of interconnect between the functional units. Within a task individual routines and operations may require different interconnectivity of functional units. General-purpose machines must provide the ability to direct data flow between units. In the extreme of a single functional unit, memory locations are used to perform this routing function. As more functional units operate together on a task, spatial switching is required to move data among functional units and memory. The flexibility and granularity of this interconnect is one of the big factors determining yielded capacity on a given application.

2.2.2 Instructions

Since general-purpose devices must provide different operations over time, either within a computational task or between computational tasks, they require additional inputs, **instructions**, which tell the silicon how to behave at any point in time. Each general-purpose processing element needs one instruction to tell it what operation to perform and where to find its inputs. As we will see, the handling of this additional input is one of the key distinguishing features between different kinds of general-purpose computing structures. When the functional diversity is large and the required task throughput is low, it is not efficient to build up the entire application dataflow spatially in the device. Rather, we can realize applications, or collections of applications, by sharing and reusing limited hardware resources in time (See Figure 2.1) and only replicating the less expensive memory for instruction and intermediate data storage.

2.3 Programmables and Configurables

The distinction between *programmable* devices and *configurable* devices is mostly artificial – particularly since we show in Part III that these architectures can be viewed in one unified



In general, we cannot embed the entire dataflow for a computational task (top) in hardware. Consequently, we must reuse the limited active silicon resources available in time (bottom), using additional control inputs, *instructions*, to tell the active silicon how to behave at each point in time to realize the desired computational task.

Figure 2.1: Temporal Reuse of Limited Active Silicon on General-Purpose Computing Devices

design space. Nonetheless, it is useful to distinguish the extremes due to their widely varying characteristics.

Programmable – we will use the term "programmable" to refer to architectures which heavily and rapidly reuse a single piece of active circuitry for many different functions. The canonical example of a programmable device is a processor which may perform a different instruction on its ALU on every cycle. All processors, be they microcoded, SIMD, Vector, or VLIW are included in this category.

Configurable – we use the term "configurable" to refer to architectures where the active circuitry can perform any of a number of different operations, but the function cannot be changed from cycle to cycle. FPGAs are our canonical example of a configurable device. Once the instruction has been "configured" into the device, it is not changed during an operational epoch.

One of the key distinction, then, is the balance between a piece of (1) active logic and its associated interconnect, and (2) the local memory to configure the operation of the logic and interconnect. We define one **configuration context** as the collection of bits which describe the behavior of a general-purpose machine on one operation cycle. One programming stream for a conventional FPGA containing instructions for every array element along with interconnect composes a "configuration context." One might also think of each of the following as a configuration

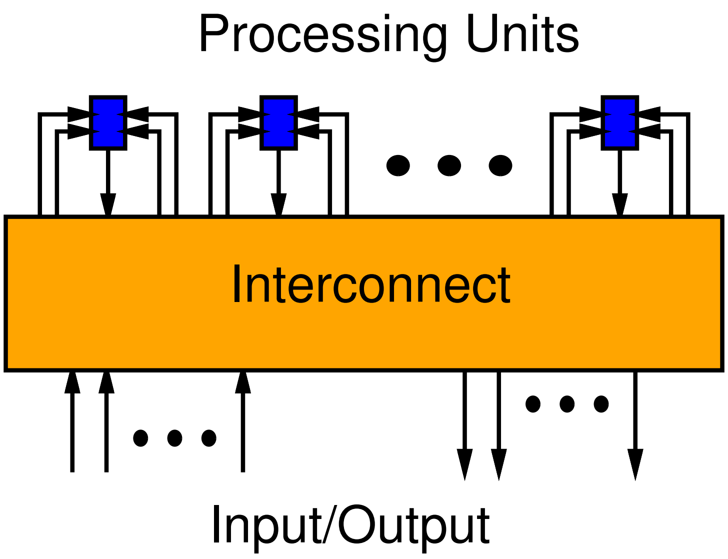


Figure 2.2: High-Level FPGA Abstraction

context.

- one instruction for scalar processor
- one VLIW word composed of instructions for all the parallel functional units
- one line of horizontal microcode

2.4 FPGA Introduction

A Field-Programmable Gate Array (FPGA) is a collection of configurable processing units embedded in a configurable interconnection network. In the context of general-purpose computing, we concern ourselves here primarily with devices which can be *reprogrammed*. Figure 2.2 shows the basic model.

From the high-level view shown in Figure 2.2, the FPGA looks much like a network of processors. A conventional FPGA, however, differs from a conventional multiprocessor in several ways:

- **Granularity** – Conventional FPGAs have single bit processing elements, each of which is controlled independently.

- **Instruction Control** – Conventional FPGAs are configurable with a single instruction resident per processing element. Changing instructions is slow compared to the rate at which the processing element can operate on data.

- **Static Interconnect** – With conventional FPGAs, interconnect is purely static, connecting sources and sinks by locking down a path through the switching network.

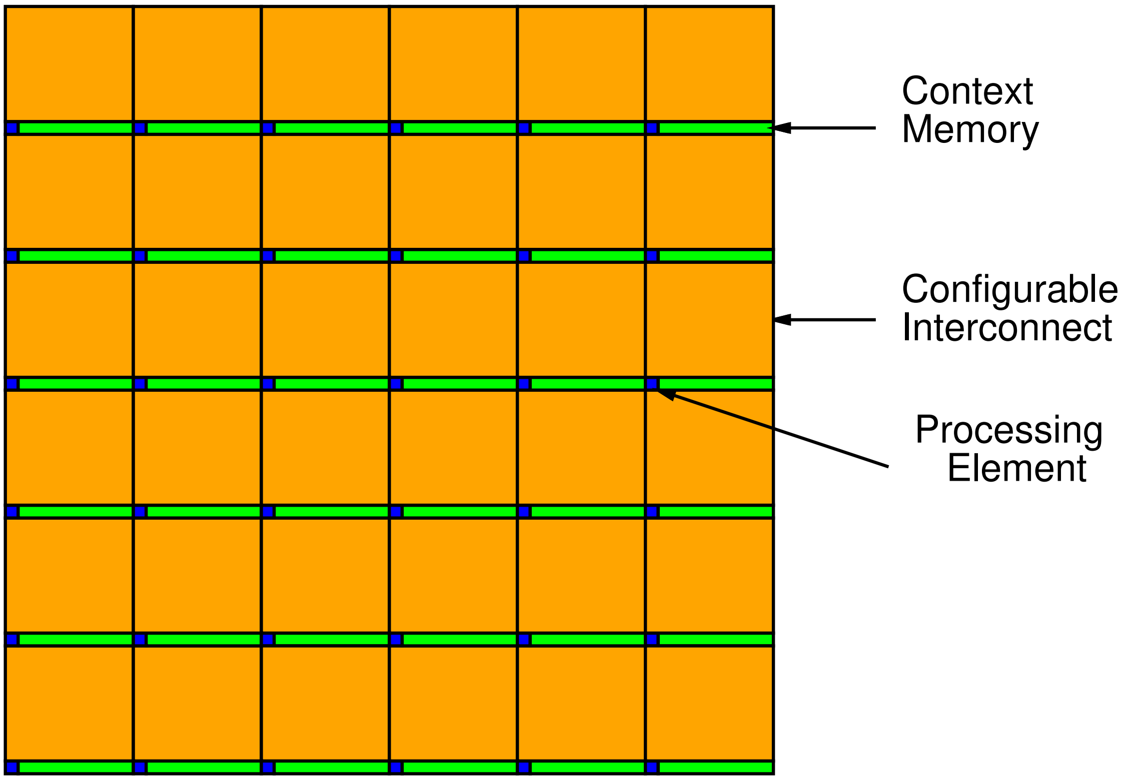


Figure 2.3: FPGA Array

Processing elements are generally organized in an array on the IC die with less than complete interconnect (See Figure 2.3). Since full connectivity would grow as $O(n^2)$, FPGAs employ more restricted connection schemes to limit the resources required for interconnect which are, nonetheless, the dominant area component in conventional devices. When processing elements are homogeneous, as is typically the case, device placement can be used to improve interconnect locality and accommodate the limited interconnect. The interconnect is typically arranged in a hierarchical mesh.

The processing elements, themselves, are simple functions combining a small number of inputs to produce a single output. The most general such function being a Look-Up Table (LUT). We have already noticed that the active logic function typically makes up only a small portion of the area. Further, it turns out that most of the configuration memory goes into describing the programmable interconnect. Consequently, for processing elements with a small number of inputs, the cost of using a full look-up table for the programmable logic function, versus some more restricted programmable element, is small.

We will use 4-input lookup tables (4-LUTs), as the canonical FPGA processing element for the purpose of discussion and comparisons. To first order, reconfigurable FPGAs from Xilinx [CDF+86, Xil94a], Altera [Alt95], and AT&T [ATT95] use 4-LUTs as their basic, constituent

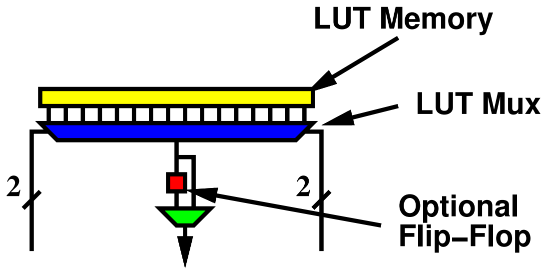


Figure 2.4: Canonical 4-LUT Processing Element

processing elements. Research at the University of Toronto [RFLC90] indicates that four input LUTs yield the most area efficient designs across a collection circuit benchmarks. An optional flip-flop is generally associated with each 4-LUT. Figure 2.4 shows the canonical, 4-LUT processing element.

Throughout the thesis we make comparisons between processors and FPGAs. At times it is convenient to equate small LUTs (2 to 4-LUTs) and ALU bits. It is therefore, worthwhile to note that a 2-LUT can perform any logical operation including those provided by typical ALUs (*e.g.* AND, OR, XOR, INVERT). A 3-LUT can act as a half-adder. A pair of 3-LUTs can serve as an adder or subtracter bit-slice with one bit providing the carry out the other the data bit output. Together these cover all the standard arithmetic and logic operations in a typical ALU, such that one or two 3-LUTs, with appropriate interconnect, can subsume any single ALU bit function.

2.5 Regular and Irregular Computing Tasks

Computing tasks can be classified informally by their regularity. **Regular** tasks perform the same sequence of operations repeatedly. Regular tasks have few data dependent conditionals such that all data processed requires essentially the same sequence of operations with highly predictable flow of execution. Nested loops with static bounds and minimal conditionals are the typical example of regular computational tasks. **Irregular** computing tasks, in contrast, perform highly data dependent operations. The operations required vary widely according to the data processed, and the flow of operation control is difficult to predict.

2.6 Metrics: Density, Diversity, and Capacity

The goal of general-purpose computing devices is to provide a common, computational substrate which can be used to perform any particular task. In this section, we look at characterizing the amount of computation provided by a given general-purpose structure.

To perform a particular computational task, we must extract a certain amount of computational work from our computing device. If we were simply comparing tasks in terms of a fixed processor instruction set, we might measure this computational work in terms of instruction evaluations. If we were comparing tasks to be implemented in a gate-array we might compare the number of gates required to implement the task and the number of time units required to complete it. Here, we want to consider the portion of the computational work done for the task on each cycle of execution of diverse, general-purpose computing devices of a certain size. The ultimate goal is to roughly quantify the computational capacity per unit area provided to various application types by the computational organizations under consideration. That is, we are trying to answer the question:

"What is the *general-purpose* computing capacity provided by this computing structure."

We have two immediate problems answering this questions:

1. How do we characterize general-purpose computing tasks?

2. How do we characterize capacity?

The first question is difficult since it places little bounds on the properties of the computational tasks. We can, however, talk about the performance of computing structures in terms of some general properties which various important subclasses of general-purpose computing tasks exhibit. We thus end up addressing more focussed questions, but ones which give us insight into the properties and conditions under which various computational structures are favorable.

We will address the second question – that of characterizing device capacity – using a "gate-evaluation" metric. That is, we consider the minimum size circuit which would be required to implement a task and count the number of gates which must be evaluated to realize the computational task. We assume the collection of gates available are all k-input logic gates, and use $k = 4$. This models our 4-LUT as discussed in Section 2.4, as well as more traditional gate-array logic. The results would not change characteristically using a different, finite value as long as $k \geq 2$.

2.6.1 Functional Density

Functional capacity is a space-time metric which tells us how much computational work a structure can do per unit time. Correspondingly, our "gate-evaluation" metric is a unit of space-time capacity. That is, we can get 4 "gate-evaluations" in one "gate-delay" out of 4 parallel **and** gates (Figure 2.5) or in 4 "gate-delays" out of a single **and** gate (Figure 2.6).

That is, if a device can provide capacity D_{cap} gate evaluations per second, optimally, to the application, a task requiring N_{ge} gate evaluations can be completed in time:

$$T_{task} = \frac{N_{ge}}{D_{cap}} \tag{2.1}$$

In practice, limitations on the way the device's capacity may be employed will often cause the task to take longer and the result in a lower **yielded capacity**. If the task takes time T_{task_actual}, to perform N_{task_ge}, then the yielded capacity is:

$$D_{yielded\_cap} = \frac{N_{task\_ge}}{T_{task\_actual}} \tag{2.2}$$

The capacity which a particular structure can provide generally increases with area. To understand the relative benefits of each computing structure or architecture independent of the area in a particular implementation, we can look at the capacity provided per unit area. We normalize area in units of λ, half the minimum feature size, to make the results independent of the particular

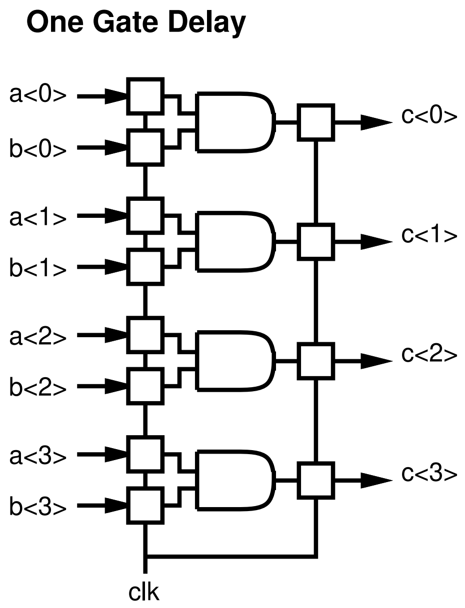


Figure 2.5: Parallel **and**

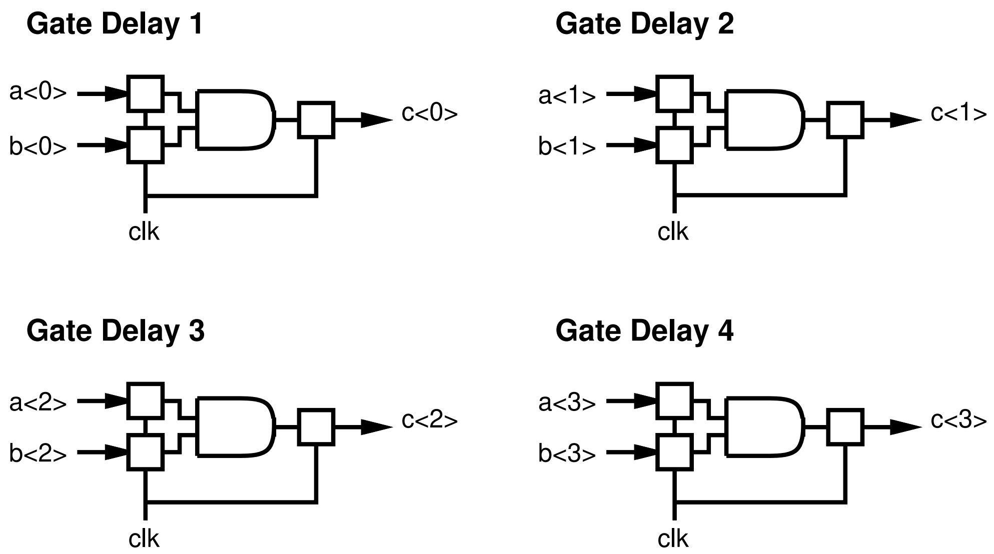


Figure 2.6: Serial **and**

process feature size. Our metric for **functional density**, then, is capacity per unit area and is measured in terms of gate-evaluations per unit space-time in units of gate-evaluations/λ^2s. The general expression for computing functional density given an operational cycle time t_{cycle} for a unit of silicon of size A evaluating N_{ge} gate evaluations per cycle is:

$$F_{density} = \frac{N_{ge}}{t_{cycle} \times A} \tag{2.3}$$

This capacity definition is very logic centric, not directly accounting for interconnect capacity. We are treating interconnect as a generality overhead which shows up in the additional area associated with each compute element in order to provide general-purpose functionality. This is somewhat unsatisfying since interconnect capacity plays a big role in defining how effectively one uses logic capacity. Unfortunately, interconnect capacity is not as cleanly quantifiable as logic capacity, so we make this sacrifice to allow easier quantification.

As noted, the focus here is on functional density. This density metric tells us the relative merits of dedicating a portion of our silicon budget to a particular computing architecture. The density focus makes the implicit assumption that capacity can be composed and scaled to provide the aggregate computational throughput required for a task. To the extent this is true, architectures with the highest capacity density that can actually handle a problem should require the least size and cost. Whether or not a given architecture or system can actually be composed and scaled to deliver some aggregate capacity requirement is also an interesting issue, but one which is not the focus here.

Remember also that resource capacity is primarily interesting when the resource is limited. We look at computational capacity to the extent we are compute limited. When we are i/o limited, performance may be much more controlled by i/o bandwidth and buffer space which is used to reduce the need for i/o.

2.6.2 Functional Diversity – Instruction Density

Functional density alone only tells us how much raw throughput we get. It says nothing about how many different functions can be performed and on what time scale. For this reason, it is also interesting to look at the **functional diversity** or **instruction density**. Here we use functional diversity to indicate how many distinct function descriptions are resident per unit of area. This tells us how many different operations can be performed within part or all of an IC without going outside of the region or chip for additional instructions. We thus define functional diversity as:

$$I_{density} = \frac{N_{instruction}}{A} \tag{2.4}$$

To first order, we count instructions as native processor instructions or processing element configurations assuming a nominally 4-LUT equivalent logic block for the logic portion of the processing element.

2.6.3 Data Density

Space must also be allocated to hold data for the computation. This area also competes with logic, interconnect, and instructions for silicon area. Thus, we will also look at **data density** when examining the capacity of an architecture. If we put N_{bits} of data for an application into a space, A, we get a data density:

$$D_{density} = \frac{N_{bits}}{A} \tag{2.5}$$

3. Reconfigurable Computing Background

This chapter briefly reviews reconfigurable computing including:

- Modern successes
- Intellectual lineage
- Technology trends which determine the circumstances when reconfigurable architectures are viable and advantageous

3.1 Successes of Reconfigurable Computing

FPGAs first became available in the middle of the 1980's (*e.g.* [CDF+86]). In the late 80's and early 90's we began to see reconfigurable computing engines enabled by these new devices. In this section we highlight the early reconfigurable computing "successes."

3.1.1 Programmable Active Memories

DEC PRL's Programmable Active Memory (PAM) was one of the earliest platforms for reconfigurable computing. PAM is an array of Xilinx 3K components connected to a host workstation [BRV89]. The Perle-1 board contained 23 XC3090's – roughly 15,000 4-LUTs. Using this component as an accelerator, DEC PRL was able to speedup many application by an order of magnitude and, in some cases, provide performance in excess of conventional supercomputers or custom VLSI implementations. Highlights from [BRV92]:

- Large number multiply 16$\times$ faster than Cray-II
- 600kbit/s, 512-bit RSA decoding – fastest implementation in existence at time of development – 10$\times$ best software implementation on DEC Alpha
- String matching within a factor of two of custom implementation requiring 28 VLSI ICs
- Convolution and 3-D geometry at 200-300 MIPs
- Laplace equation at 25 GIPs
- DCT at 15 GIPs

The total silicon in the Perle-1 board was comparable to the total silicon in the host workstation – but the combination ran these applications and others 10$\times$ faster than the workstation alone. The difference being that almost all of the silicon on the Perle-1 board was general-purpose and capable of being deployed to the problem at hand.

3.1.2 Splash

SRC's Splash is a systolic array composed of 32 Xilinx XC3090's, $\approx$20K 4-LUTs. On DNA sequence matching Splash achieved over 300$\times$ the performance of a Cray-II or over 200$\times$ the performance of a 16K-processor CM-2 [GHK+91].

3.1.3 PRISM

Brown's PRISM architecture coupled a single Xilinx XC3090, 640 4-LUTs, with a Motorola 68010 node processor. The coupled FPGA could compute fine-grained, bitwise functions (*e.g.* Hamming distance, bit reversal, ECC, logic evaluations, find first one), 20$\times$ faster than the 68010 host microprocessor [AS93].

3.1.4 Logic Emulation

Perhaps the most commercially significant application of "reconfigurable logic" to date has been in the business of logic emulation. One of the earliest FPGA-based logic emulators was the Realizer [VBB93] which was a precursor to Quickturn System's Enterprise Emulation System. The Realizer, with 42 XC3090's ($\approx$27K 4-LUTs) and 160 XC2018's serving exclusively for interconnect, was able to emulate $\approx$10K gate designs at a rate of several million clock cycles per second.

3.2 Lineage

While reconfigurable architectures have only recently begun to show significant application viability, the basic ideas have been around almost as long as the idea of programmable general-purpose computing.

John von Neumann, who is generally credited with developing our conventional model for serial, programmable computing, also envisioned spatial computing automata – a grid of simple, cellular, building blocks which could be configured to perform computational tasks [vN66].

As computing implementation technology improved from vacuum tubes to diodes and transistors to integrated circuits, research continued into cellular computation. In [Min67] Minnick reviewed the state of the art in microcellular computational arrays, suggesting a role for "programmable arrays." Minnick's own cutpoint cellular array in 1964 housed 48 cells less powerful than a 2-LUT in a 6$\times$8 cellular array with only right and down nearest neighbor connections in the space of a suitcase. In 1971, Minnick reported a programmable cellular array which used flip-flops to hold the configuration context which customized the array [Min71].

Jump and Fitsche detail the workings of a programmable cellular array [JF72] without describing a possible technology realization.

Schaffner developed one of the earliest "general-purpose," "programmable hardware" machines in 1969 [Sch71, Sch78]. Shaffner's machine used ALU's with reconfigurable interconnect for his reconfigurable building blocks, including the facilities to swap in "hardware" pages. The machine was employed primarily for real-time signal processing for radar and weather.

The early eighties saw considerable interest in systolic computing architectures [Kun82]. While much of the research was concerned with deriving hardwired, application-specific arrays, this research also spawned the development of programmable systolic components (*e.g.* [FKM83] [HS84]). These components were some of the first "reconfigurable computing" devices built in VLSI. Owing to the application focus and the silicon real estate available at the time, the programmable systolic building blocks were more coarse-grained than the cellular arrays or FPGAS, placing a single 8-bit ALU per chip and relying predominantly on large, multichip or wafer-scale arrays to build up significant spatial computations.

The most direct descendent of the programmable cellular array research is the Configurable Array Logic (CAL) IC from Tom Kean and Algotronix [Kea89, GK89, Alg90]. CAL used a minimal 2-LUT for the basic cellular element and mostly nearest-neighbor connections for interconnect. This gives it a much finer grain than the contemporary FPGAs from Xilinx which use 4-LUTs and richer interconnect.

3.3 Technological Enablers

The basic idea of configurable array computation has been around as long as the ideas for central processor, stored program execution. So, why have programmable processors become the mainstream of general-purpose processing while "reconfigurable computing" is only now emerging as a competitive, general-purpose computing technology?

The answer lies with technology costs and application requirements. Active computing resources have been a premium since the days of the vacuum tube. To realize general-purpose computers, it took thousands of tubes to build a general-purpose computer – making it infeasible to implement large, spatial computations. With the advent of core-memory, memory became moderately dense compared to computing elements. To implement large, complex, computational tasks, it was more efficient to store large programs densely in memory and reuse a small amount of fixed logic.

The beginning of the MOS VLSI era reinforced these costs. Dense memories could be implemented on silicon ICs. Because of high off-chip i/o costs, the critical unit became the amount of logic or computation which could be placed on a single IC. The driving force has been to localize computation to one or a small number of ICs to reduce costs and interchip communications. The microprocessor was made successful by minimizing the amount of compute logic to the point where it would fit onto a single IC. The critical turning point in processor development was when it became possible put a competent processor on a single IC. The RISC structure became so successful because it enabled early integration of such capable processors. Once single-chip processors became possible, they rapidly rose to dominate multichip implementations. While silicon area was a premium, exploiting the higher density of memories to store programs and reuse the limited space on the processor die was necessary. Today, we still see some premium to fitting the kernel task descriptions and their data into the limited memory available on the processor die.

The turning point for configurable hardware came when it was possible to place hundreds of programmable elements on a single IC. At that point it became possible to realize regular computations in space, dedicating each active computing element to a single task. Reconfigurable computing began to take off as we could put 500-1,000 such programmable elements on a single IC. Today we look at thousands of such elements per IC and that number continues to increase with the silicon capacity. At thousands to tens of thousands of programmable elements, tight application kernels can be spatially configured on one or a few configurable ICs without the need to share active resources. This, in effect, caches the kernel not just in on-chip memory for use by a limited amount of active processing elements, but right with the active processing elements such that a large number may operate simultaneously.

There will always be some premium for dense task representation to handle the most complicated tasks. However, as the silicon real-estate becomes larger, the premium for dense task packing subsides making it more and more beneficial to increase the on-chip silicon available for active processing and remove the on-chip bottleneck between memory and processing elements. This transition moves us to reconfigurable architectures.

Part II

Empirical Review

Here we review various general-purpose computing architectures by taking an empirical look at their implementations during the past decade. In this section we draw from the whole realm of general-purpose architectures – not just those which fit directly into our *RP*-space. This makes a larger set of design points avilable for review, but also introduces considerably more variation in architectures than we will focus on in later chapters. We look primarily at general-purpose capacity in this section, generally ignoring the effects of specialized functional units. The following chapter will look at the effects of custom multipiers, the most common specialized functional unit added to nominally general-purpose computing devices. The focus here is on integrated, single IC, computational building blocks to keep the comparison as consistent as possible across such a wide variety of architectures. Additionally, we focus entirely on MOS VLSI implementations since most of these architecture have had multiple MOS VLSI implementations and the effects of MOS feature size device scaling are moderately well understood.

4.1 Processors

We start by looking at a simple RISC-style processor.

Model The pedagogical processor model (Figure 4.1) is composed of:

- n-bit ALU (two read, one write port from register file) [for the sake of comparison, we do include multiple ALU processors in the empirical review]
- memory (Register File, Instruction/Data Cache)
- control (sequencer, cache control, load/store unit, etc.)

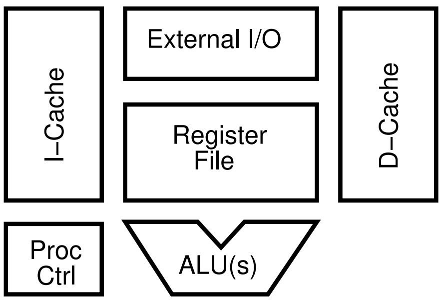


Figure 4.1: Basic Organization for a Processor

Instruction	Gate Evaluation Capacity	Explanation
LD, ST, MOVE	0	overhead allowing reuse and interconnect
ADD, SUB, CMP	$2n$	Each full adder bit is 2 gate evaluations
AND, OR, XOR,	n	One gate evaluation per bit
BEQ,BNE	$\approx \frac{n}{3}$	n-bit AND reduction $\frac{n}{4} + \frac{n}{16} + \cdots + 1$
B, CALL, RETURN	0	flow control overhead
SHIFT	0	interconnect

Table 4.1: Basic ALU Operations and Capacities

The ALU is the sole source of general-purpose capacity. Table 4.1 shows the traditional ALU operations provided by the ALU along with the computational capacity provided by each operation. An n-bit ALU provides n ALU bit operations. For this simple processor, no multiplier or specialized coprocessor is included. We will look at hardwired multiply implementations separately in Chapter 5 as an example of such specialized coprocessors. Each ALU operation operates in one processor clock cycle.

Capacity Provided We extract a maximum of $2n$ gate evaluations (n ALU bit operations) per cycle. Modern processors are achieving cycle times as low as 2-5ns with $n = 128$. The fastest, single-ALU processors today thus offer a peak capacity around 84 gate-evaluations/ns. Table 4.2 compares several processor implementations over the past decade. Results are summarized there in terms of ALU bit ops since that is the native, and hence most accurate, unit for processors. From Table 4.2, we see that conventional processors have provided a peak functional density of 3-9 ALU bit operations/λ^2s over the past decade. We see from Table 4.1 and some simple weightings below that an ALU bit op is somewhere between one half and two 3-LUT gate evaluations.

It is interesting, and perhaps a bit unexpected, to note how consistent this capacity density has been over time. We might have expected:

1. delays to improve with process such that $\lambda^2\tau$ would be a better measure of process-normalized capacity than λ^2s [τ is the delay parameter for a process which nominally amounts to the intrinsic RC delay for gates. One τ is the delay required for one inverter to drive a single inverter of equal size.]

2. architectural or circuit design improvements to have increased functional density over time

Given the displayed consistency, we may be seeing compensating effects from:

1. velocity saturation in the CMOS devices, especially submicron CMOS prevents the expected τ scaling

2. decreasing relative performance of on-chip interconnect with scaling

Year	Design	Organization	Die Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1984	RISC II [SKPS84]	1×32 (100%)	4.3mm$\times$7.7mm	1.5μ	15M	330 ns	6.5
1984	MIPS [RPJ+84]	1×32 (100%)	5.5mm$\times$6.1mm	1.5μ	15M	250 ns	8.5
1987	MIPS-X [HHC+87]	1×32 (100%)	8mm$\times$8.5mm	1.0μ	68M	50 ns	9.4
1987	PA-RISC [YFJ+87]	1×32 (100%)	8.4mm$\times$8.4mm	0.75μ	125M	66 ns	3.9
1988	SPARC [QC88, TFT+85]	1×32 (100%)	12.1mm$\times$12.7mm	0.75μ	273M	60 ns	2.0
1990	PA-RISC [TLB+90]	1×32 (100%)	14mm$\times$14mm	0.5μ	784M	11 ns	3.7
1990	SPARC [MMN+90]	1×64 (75%) IEEE FPU (25%)	14.9mm$\times$15.1mm	0.4μ	1.4G	25 ns	2.4
1992	SuperSparc [ANAB+92]	2×32 (82%) IEEE FPU (18%)	16mm$\times$16mm	0.4μ	1.6G	25 ns	2.0
1992	Alpha [DWA+92]	1×64 (81%) IEEE FPU (19%)	16.8mm$\times$13.9mm	0.38μ	1.7G	5 ns	9.5
1994	PA-RISC [RDB+94]	2×64 (88%) IEEE FPU (12%)	14mm$\times$15mm	0.28μ	2.8G	7 ns	7.4
1994	MIPS [SYN+94]	1×32 (100%)	7.9mm$\times$8.8 mm	0.2μ	1.7G	2 ns	9.1
1995	PowerPC [BBB+95]	2×64 (87%) IEEE FPU (13%)	18.2mm$\times$17.1mm	0.25μ	5G	7.5 ns	3.9
1995	UltraSparc [CDd+95]	2×64 (84%) $2\times$FP/GFU (16%)	17.7mm$\times$17.8mm	0.25μ	5G	6 ns	5.0
1995	SPARC V9 [SPA+95] [CDF+95]	4×64 (80%) IEEE FPU (20%)	297mm^2	0.2μ	7.4G	6.5 ns	6.6
1995	Alpha [BAB+95]	2×64 (90%) $2\times$IEEE FPU (10%)	16.5mm$\times$18.1mm	0.25μ	4.8G	3.3 ns	9.0
1996	MIPS [KDS+96]	1×32	9.1mm$\times$8.3mm	0.25μ	1.2G	10 ns	2.6
1996	PA-RISC [LLNK96]	2×64 (80%) IEEE FPU (20%)	17.7mm$\times$19.1mm	0.25μ	5.4G	4 ns	7.4
1996	ARM [MWA+96]	1×32	7.8mm$\times$6.4mm	0.18μ	1.6G	5 ns	4
1996	Alpha [GBB+96]	2×64 (95%) $2\times$IEEE FPU (5%)	14.5mm$\times$14.4mm	0.18μ	6.8G	2.3 ns	8.6

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

$P_{fraction}$ is the fraction of the die not including any specialized coprocessors – primarily omitting any FPUs.

Table 4.2: Survey of Processor Capacity

Year	**Design**	**Ref.**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1984	RISC II	[SKPS84]	6.5	0	3.0×10^{-4}
1984	MIPS	[RPJ+84]	8.5	0	3.4×10^{-5}
1987	MIPS-X	[HHC+87]	9.4	7.5×10^{-6}	1.5×10^{-5}
1987	PA-RISC	[YFJ+87]	3.9	0	8.2×10^{-6}
1988	SPARC	[QC88]	2.0	0	1.9×10^{-5}
1990	PA-RISC	[TLB+90]	3.7	0	1.3×10^{-6}
1990	SPARC	[MMN+90]	2.4	1.5×10^{-6}	1.9×10^{-5}
1992	SuperSparc	[ANAB+92]	2.0	3.9×10^{-6}	1.0×10^{-4}
1992	Alpha	[DWA+92]	9.5	1.2×10^{-6}	4.1×10^{-5}
1994	PA-RISC	[RDB+94]	7.4	0	7.5×10^{-6}
1994	MIPS	[SYN+94]	9.1	1.5×10^{-7}	5.3×10^{-6}
1995	PowerPC	[BBB+95]	3.9	1.9×10^{-6}	6.0×10^{-6}
1995	UltraSparc	[CDd+95]	5.0	9.7×10^{-7}	3.3×10^{-5}
1995	SPARC V9	[SPA+95]	6.6	0	1.1×10^{-5}
1995	Alpha	[BAB+95]	9.0	$4.8\text{-}57\times10^{-7}$	$1.5\text{-}18\times10^{-5}$
1996	MIPS	[KDS+96]	2.6	8.4×10^{-7}	2.8×10^{-5}
1996	PA-RISC	[LLNK96]	7.4	0	4.7×10^{-7}
1996	ARM	[MWA+96]	4	2.5×10^{-6}	8.0×10^{-5}
1996	Alpha	[GBB+96]	8.6	$3.1\text{-}38\times10^{-7}$	$1.0\text{-}12\times10^{-5}$

Table 4.3: Processor Capacity Summary

3. increasing chip size implies increasing wire runs – the area occupied by long interconnect wires is not scaling with λ^2

4. increasing use of CAD – designers are giving up some density in order to manage the complexity associated with the larger and larger designs

5. increasing gap between on-chip and off-chip performance necessitates dedicating more on-chip area to non-active resources, particularly memory, to compensate for the i/o bottleneck

6. increasing area being dedicated to control and state management to prevent control and data dependent stalls in the instruction stream

This peak computational density assumes that every operation on each n-bit ALU performs an n-bit compute operation and the processor completes one instructions per ALU on every cycle. In practice, a significant number of processor cycles are not spent executing compute operations.

- **Limited instruction and data bandwidth** – coupled with long latencies to off-chip resources, limited bandwidth can cause the processor to stall waiting on the information which it needs to determine the course of the computation or the data it needs to operate upon. As a result, the number of instructions completed per cycle is always less than the number of ALUs available.

- **Abstraction overhead and data movement consume capacity** – procedure calls, data marshaling, traps, and data conversion consume capacity without, directly, providing capacity

Application	$\frac{\text{Gate Evaluations}}{\text{Datapath Bit}}$
GCC	0.5
TeX	0.5
US Steel	0.8
Average	0.6

Table 4.4: Average Gate Evaluations/Datapath Bit

to the problem. Operations which simply move data around do not provide capacity to the problem either. Since we have a mix of the instructions shown in Table 4.1, we will never achieve the peak condition where every operation provides $2n$ gate-evaluations to the computational task.

Quantitative studies tell us, for given systems or application sets, average values for the number of gate evaluations per instruction and the number of instructions executed per clock cycle. This gives us an expected computational capacity:

$$\text{E(Functional Density)} = \\ \frac{\text{Gate Evaluations}}{\text{Datapath Bit}} \times \frac{\text{Datapath Bits}}{\text{Instruction}} \times \frac{\text{Instructions}}{\text{Issue Slot}} \times \frac{\text{Issue Slots}}{\text{Clock Cycle}} \times \frac{1}{\text{area} \times t_{cycle}}$$

For example, while HaL's SPARC64 should be able to issue 4 instructions per cycle, in practice it only issues 1.2 instructions per cycle on common workloads [EG95]. Thus $\frac{\text{Instructions}}{\text{Issue Slot}} \approx 30\%$, resulting in a 70% reduction from expected peak capacity.

Assuming the integer DLX instructions mixes given in Appendix C of [HP90] are typical, we can calculate $\frac{\text{Gate Evaluations}}{\text{Datapath Bit}}$ by weighting the instructions by their provided capacities from Table 4.1. In Table 4.4 we see that one ALU bit op in these applications is roughly 0.6 gate-evaluations.

If this effect is jointly typical with the instructions per issue slot number above, then we would yield at most $0.3 \times 0.6 \approx 0.2$ of the theoretical, functional density. For the HaL case, this reduces 6.6 ALU Bit Ops$/\lambda^2$s to 1.3 gate-evaluations$/\lambda^2$s.

There are, of course, several additional aspects which prevent most any application from achieving even this expected peak capacity and which cause many applications to not even come close to it.

1. **Coarse-grained datapath/network** – the word-oriented datapaths in conventional processors prevents efficient interaction of arbitrary bits – *e.g.* A simple operation like XOR-ing together two fields in a data word requires a mask, shift, and XOR operation even though the whole operation may effect only a few gate evaluations. Returning to the HaL example above, we note that the processor has a 64-bit datapath. When running code developed for 32-bit SPARCs, half of the datapath is idle, further reducing the yielded capacity by 50% to 0.7 gate-evaluations$/\lambda^2$s.

//R1 x_i base pointer	
//R2 avg_i base pointer	
//R3 working sum	
//R6 buffer top	
ld -32[R1],R4	*//value dropping out of window*
sub R4,R3,R3	*//subtract out*
ld 0[R1],R4	*//value entering window*
add R4,R3,R3	*//add in*
asr R4,R5,#3	*//divide by 8*
st [R2],R5	*//save new average*
addi R1,#4,R1	*//increment x ptr*
ble R1,R6,top	*//branch to top*
addi R2,#4,R2	*//increment avg ptr (delay slot)*

Figure 4.2: Inner Loop of Processor Implementation for Windowed Average

2. **Limited control of ALU capacity** – the n-bit ALU mostly functions as a collection of n-bit processors controlled in SIMD fashion. Full capacity is only extracted when all n bits arranged in a dataword need the same operation. Data with smaller ranges (less than 2^n) do not make full use of the capacity. Inhomogeneous operations (*e.g.* ADD the low 8-bits, XOR bits 10-8, use this constant for bits 15-11, AND in bits 19-16, and OR together the remaining bits) must be decomposed into sets of homogeneous operations.

Example: Average Calculation Consider a windowed average calculation performed on a processor:

$$avg_i = \left(\frac{1}{8}\right) \cdot \left(x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4}\right)$$

Figure 4.2 shows a possible inner loop of the windowed average calculation on a standard RISC processor. Assuming one instruction per instruction slot, this sequence takes 9 instruction slots to perform two potentially 32 bit adds – for a total of 128 gate evaluations. The loads, stores, and shifts are all data movement operations and do not contribute to the actual computational task at hand. The branch and increments are control overhead. Assuming this operation is performend on a MIPS-X processor at 1 CPI, we yield a functional density:

$$F_{d_{yield}} = \frac{128 \text{ gate evaluations}}{68\text{M}\lambda^2 \times 9 \text{ cycles} \times 50\text{ns}} = 4.2 \left(\frac{\text{gate evaluations}}{\lambda^2 \text{s}}\right)$$

Example: Parity Calculation Consider calculating the parity of a 32-bit word.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

//R1 d input	
//R3 parity output	
asr R1,R2,#16	//align half words
xor R2,R1,R3	//16b xor
asr R3,R2,#8	//align bytes
xor R2,R3,R3	//8b xor
asr R3,R2,#4	//align nibbles
xor R2,R3,R3	//4b xor
asr R3,R2,#2	//align 2 bits
xor R2,R3,R3	//2b xor
asr R3,R2,#1	//align final bits
xor R2,R3,R3	//final xor

Figure 4.3: Processor Implemention for Parity Computation

Year	**Design**	**Organization**	λ	**Die Size**	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1990	LIFE-1 [LS90]	2×32	0.75μ	78mm^2	139M	20 ns	23
1993	VIPER [GNAB93]	4×32	0.6μ	12.9mm$\times$9.1mm	326M	40 ns	9.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 4.5: Survey of VLIW Capacity

In 10 operations (See Figure 4.3), the processor can perform the 32b XOR required for the parity calculation – 32 2-input gate evaluations or 11 4-input gate evaluations. Again, assuming a MIPS-X like processor and 1 CPI, we yield:

$$F_{d_{yield}} = \frac{11 \text{ gate evaluations}}{68\text{M}\lambda^2 \times 10 \text{ cycles} \times 50\text{ns}} = 0.32 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.2 VLIW Processors

Very Long Instruction Word (VLIW) machines are processors with multiple, parallel functional units which are exposed at the architectural level. A single, wide, instruction word controls the function of each functional unit on a cycle-by-cycle basis. Pedagogically, a VLIW processor looks like a processor with multiple, independent functional units. At this level, the VLIW processor does not look characteristically different from the modern superscalar processors included at the end of the processor table.

Table 4.5 summarizes the characteristics of two VLIW processors. With only two datapoints it is not possible to assess general trends. These examples seem to have about 2$\times$ the peak capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1990	LIFE-1 [LS90]	23	?	?
1993	VIPER [GNAB93]	9.8	0	1.0×10^{-4}

Table 4.6: VLIW Capacity Summary

of processors. To the extent this may be characteristic of VLIW designs, it may arise from the fact that the separate functional units share instruction control and management circuitry more than in superscalar processors.

VLIW processors may fail to achieve their peak for the same reasons as processors. In addition, they may suffer from:

- **Scheduling granularity** – instructions must be statically scheduled in blocks at the VLIW width. When this packing does not match the needs of the application, functional units may go idle.
- **Mismatch in Functional Unit Mix** – some VLIWs have a mix of functional units which perform different operations. If the mix of operations in the application does not match the functional unit mix at a fine-grained level, functional units may go idle, lowering yielded capacity.
- **Data Transfer** – one way VLIWs may achieve a higher functional density than superscalar processors is to segregate the register files, reducing the interconnect required to deliver data from the register file to the functional units and back. Some cycles will be required to transfer data between register files as necessary to allow the various functional units to cooperate on a task.

4.3 Digital Signal Processors (DSPs)

Digital Signal Processors (DSPs) are essentially specialized microprocessors which:

- Integrate a hardwired Multiplier Accumulator (MAC) unit
- Includes specialized datapaths allowing
  - zero overhead loops
  - parallel increment of counters and pointers

The net effect of these additions is generally to increase the percentage of yielded capacity and particularly to increase the yielded capacity on tight loop multiply operations.

Table 4.7 reviews several DSP implementations. On non-multiply operations, the peak performance is generally lower than the processors. For the kinds of operations typical of DSPs, these processors will generally yield much closer to their peak capacity than processors.

Year	Design	Organization	Die Size	λ	λ^2 area	cycle	ALU bit ops / λ^2s
1985	[WDW+85]	1×16	15.5mm^2	1.0μ	16M	100 ns	10
1986	[vMWvW+86] [Gol87]	1×16	88.5mm^2	1.0μ	90M	125 ns	1.4
1987	[KNK+87]	1×16	11.5mm×12.9mm	0.65μ	350M	50 ns	0.9
1987	[CBBF87]	1×32	6mm×8.5mm	0.5μ	200M	60 ns	1.3
1989	[PML+89]	1×16	71mm^2	0.6μ	200M	100 ns	0.8
1992	[SKYH92]	1×16	9.5mm×10.5mm	0.6μ	275M	50 ns	1.2
1993	[USO+93]	1×16	9.3mm×9.1mm	0.4μ	530M	93 ns	0.33
1995	[NHK95]	1×32	8.5mm×8.5mm	0.25μ	1.2G	10 ns	2.8
1996	[KOT+96]	2×16	10mm×9.7mm	0.25μ	1.6G	25 ns	0.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Sizes above include MACs, which generally amount to 5-10% of the total DSP die area.

Table 4.7: Survey of DSP Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1985	[WDW+85]	10	0	1.5×10^{-5}
1986	[vMWvW+86]	1.4	0	4.9×10^{-5}
1987	[KNK+87]	0.9	2.9×10^{-6}	9.3×10^{-5}
1987	[CBBF87]	1.3	1.0×10^{-5}	4.1×10^{-5}
1989	[PML+89]	0.8	1.6×10^{-5}	4.1×10^{-5}
1992	[SKYH92]	1.2	1.4×10^{-5}	3.0×10^{-5}
1993	[USO+93]	0.33	1.5×10^{-5}	1.5×10^{-4}
1995	[NHK95]	2.8	8.9×10^{-7}	2.9×10^{-5}
1996	[KOT+96]	0.8	?	?

Table 4.8: DSP Capacity Summary

4.4 Memories

Most general-purpose devices use memories to store instructions and data. A memory can also be used directly to implement complex computational functions. For complicated functions, a memory lookup can often provide high, programmable computational capacity. For just a few examples see [Has87, HT95, RS92, Lev77].

Model We characterize a memory by:

- depth – either in terms of address bits (a) or total number of memory words (2^a)
- width – w, the number of bits read out for each a bits of address put into the memory

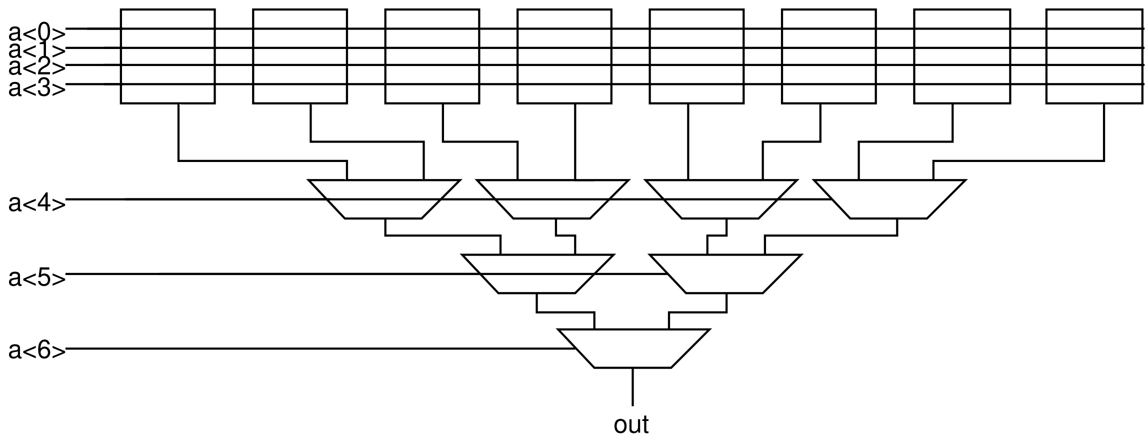


Figure 4.4: Gate Implementation of any Function Computed by 7-input Lookup Table

- t_{cycle} – the minimum time required between successive read operations

Capacity Provided The capacity provided by a memory is highly dependent on the inherent complexity of the logic function being computed. The lower bound is trivially zero since we could program the identify function into a memory.

We can use a counting argument to determine how complicated the functions can get. We start by observing that an a-input, one-output lookup table can implement 2^{2^a} different functions. We then consider how many gates it requires to implement any of these functions. Each gate can be any function of four inputs, so each gate can implement 2^{2^4} functions. A collection of n gates can thus implement at most $\left(2^{2^4}\right)^n$ functions (less due to overcounting). In order to implement any of the 2^{2^a} functions provided by the table, we need at least:

$$\begin{aligned} 2^{2^a} &\leq \left(2^{2^4}\right)^n = 2^{\left(n \cdot 2^4\right)} \\ 2^a &\leq n \cdot 2^4 \\ 2^{(a-4)} &\leq n \end{aligned} \tag{4.1}$$

Conversely, by construction, we can show that any function computed by the a input lookup table can be computed with $2^{(a-3)} - 1$ gate evaluations. As suggested in Figure 4.4, we can use $2^{(a-4)}$ gates to select the correct functional value based on the low four bits of the address. We then build a binary mux reduction tree to select the final output based on the remaining address bits. This tree requires $2^{(a-4)} - 1$ muxes. Together, the $2^{(a-3)} - 1$ gates compute any function computable by the a-input lookup table.

An a input by one output table lookup can thus provide between $2^{(a-4)} + 1$ and $2^{(a-3)} - 1$ gate evaluations per cycle for the most complicated a input functions. Since the bounds are essentially a factor of two apart, we can approximate the peak as $2^{(a-4)}$ gate evaluations per cycle. If the table is

w bits wide, the table provides at most w times as many gate evaluations. Putting all this together, we get:

$$C_{memory\_peak} \approx \frac{w \cdot 2^{(a-4)}}{\text{area} \cdot t_{cycle}} \tag{4.2}$$

Tables 4.9, 4.10 and 4.11 reviews memory implementations, showing the peak functional density for each memory array. For the most complex functions, memories provide the highest capacity of any general-purpose architecture. For less complex operations, however, memories are inefficient, yielding very little of their potential capacity.

For example, an 8-bit add operation with carry output requires 16 gate evaluations. Performed in a $2^{16} \times 9$ memory, such as a 9-bit version of the 64K$\times$18 memory from [SMK$^+$94], this provides only 2.4 gate-evaluations/λ^2s. The inefficiency of the memory-based adder increases with operand size since the number of gate evaluations in an n-bit add increases as $2n$ whereas the memory area increases as $2^{2n} \times (n+1)$.

For all the memories listed, the capacity is based on continuous cycles of random access. In particular, nibble, fast page, or synchronous access in DRAMs is not exploited. For example, [TNK$^+$94] achieves 13,500 gate-evaluations/λ^2s on random access. In sequential access mode, the part can output 18 bits every 8ns. For large sequential access, this means an effective cycle time of 8ns instead of the 48ns quoted – a factor of six improvement in cycle time and capacity. Used in this mode, the peak performance is 81,000 gate-evaluations/λ^2s.

It is also worth noting that, unlike processors, the capacity of memories has increased over time. This is likely due to:

- increased specialization of the fabrication processes to memories – especially the introduction of three-dimensional structures in DRAMs, local interconnect, and high poly film resistors for SRAMs
- increased pipelining of memory access

Modern processors actually dedicate a significant portion of their area to memory. Table 4.12 summarizes the peak capacity the processor can extract by using table lookups in its D-cache. The area used in calculating this capacity is the entire processor for the processors listed in Table 4.2. This peak capacity can be thought of as the peak capacity one could extract from each load operation when using the on-chip D-cache for table lookup operations.

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$
1984	[BDN84]	16K×1	16.3mm^2	1.0μ	16M	35 ns	1800
1984	[SMI+84]	32K×8	6.7mm×8.9mm	0.6μ	164M	46 ns	2200
1984	[YKK+84]	64K×1	4.7mm×6.6mm	0.75μ	55M	25 ns	3000
1984	[SCLB84]	4k×16	32.6mm^2	0.75μ	58M	30 ns	2400
1984	[MSM+84]	8K×8	6.0mm×6.8mm	0.6μ	113M	28 ns	1300
1984	[OKH+84]	2K×8	2.7mm×3.5mm	0.75μ	17M	16 ns	3800
1984	[CH84]	4K×4	11mm^2	1.05μ	10M	18 ns	5700
1984	[MMS+84]	64K×1	3.2mm×6.0mm	0.65μ	45M	20 ns	4600
1985	[YTN+85]	32K×8	5.0mm×9.2mm	0.65μ	110M	45 ns	3400
1985	[SAI+85]	32K×8	49.6mm^2	0.65μ	120M	45 ns	3100
1985	[SGS+85]	8K×8	31.8mm^2	0.75μ	57M	35 ns	2100
1985	[KEK+85]	32K×8	40.7mm^2	0.6μ	110M	55 ns	2600
1986	[CC86]	8K×8	45mm^2	0.75μ	80M	35 ns	1500
1986	[KIK+86]	256k×1	4.5mm×10.6mm	0.5μ	190M	25 ns	3500
1986	[FRV+86]	64K×1	3.4mm×8.9mm	0.75μ	54M	13 ns	5900
1987	[WBS+87]	32K×8	6.8mm×9mm	0.6μ	170M	21 ns	4600
1987	[KTO+87]	128K×8	8mm×13.7mm	0.5μ	440M	35 ns	4300
1987	[WHS+87]	128K×8	5.5mm×14.8mm	0.4μ	510M	34 ns	3800
1987	[MOT+87]	128K×8	6.9mm×15.4mm	0.4μ	660M	25 ns	4000
1987	[GHS+87]	32K×8	9.5mm×7.8mm	0.65μ	175M	40 ns	2300
1988	[SKI+88]	128K×8	7.6mm×12.4mm	0.4μ	590M	44 ns	2500
1988	[WBEK+88]	1M×1	10.6mm×8.5mm	0.35μ	730M	29 ns	3100
1988	[CDH+88]	128K×8	12.2mm×7.7mm	0.35μ	770M	25 ns	3400
1988	[STT+88]	256K×4	7.5mm×12mm	0.35μ	730M	18 ns	5000
1988	[SHU+88]	256K×4	6.2mm×15.2mm	0.4μ	580M	15 ns	7500
1988	[KWA+88]	1M×1	5.5mm×15.7mm	0.35μ	710M	14 ns	6600
1988	[ONN+88]	32K×8	4.4mm×9.5mm	0.4μ	260M	7.5 ns	8400
1989	[VPP+89]	256K×1	3.9mm×9.5mm	0.35μ	300M	14 ns	3900
1989	[MMK+89]	512K×8	7.5mm×17.4mm	0.25μ	2.1G	25 ns	5000
1989	[SIY+89]	1M×1	5.3mm×10.3mm	0.25μ	870M	9 ns	8300
1990	[FPH+90]	256K×1	11.6mm×3.7mm	0.5μ	170M	8 ns	12000
1990	[ASO+90]	4M×1	7.7mm×18.6mm	0.28μ	1.9G	15 ns	9200
1990	[HKM+90]	4M×1	8.4mm×18.0mm	0.3μ	1.7G	20 ns	7800
1990	[SIS+90]	512K×8	7.2mm×16.9mm	0.25μ	1.9G	23 ns	5900
1990	[OHK+90]	512K×8	7.8mm×17.4mm	0.25μ	2.2G	23 ns	5200
1991	[CCS+91]	32K×16	11.1mm×10.1mm	0.4μ	700M	2 ns	23500
1992	[SSN+92]	32K×8	6mm×9mm	0.4μ	340M	200 ns	240
1992	[GOK+92]	2M×8	18.3mm×12.5mm	0.2μ	5.7G	12 ns	15300
1992	[MKS+92]	4M×4	10.4mm×21.5mm	0.2μ	5.6G	15 ns	12500
1992	[SIU+92]	256K×4	4mm×7.4mm	0.15μ	1.3G	7 ns	7200
1993	[SKS+93]	4M×4	9.7mm×21.9mm	0.18μ	6.9G	9 ns	16800
1993	[SUT+93]	4M×4	10.4mm×10.6mm	0.13μ	7.1G	20 ns	7400
1994	[IKM+94]	4M×4	10.3mm×20.9mm	0.2μ	5.4G	30 ns	6500

Table 4.9: Survey of Peak Memory Logic Capacity (SRAM)

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$
1984	[MKS+84]	256K×1	6.3mm×6.3mm	0.75μ	70M	140 ns	1700
1984	[BCH+84]	256K×1	50mm^2	1.0μ	50M	150 ns	2200
1984	[MKM+84]	256K×1	30.2mm^2	0.8μ	47M	116 ns	3000
1984	[KFO84]	256K×1	46.8mm^2	0.6 μ	130M	100 ns	1250
1984	[SNT+84]	128K×8	9.4mm×8.1mm	0.5μ	300M	120 ns	1800
1985	[KCE+85]	1M×1	5.5mm×10.5mm	0.5μ	230M	160 ns	1800
1985	[KFM+85]	1M×1	5mm×13mm	0.6μ	180M	260 ns	1400
1985	[SFO+85]	1M×1	5mm×12.5mm	0.6μ	170M	190 ns	2000
1985	[TJ85]	256K×4	6.0mm×11.4mm	0.6μ	190M	190 ns	1800
1986	[FSO+86]	1M×1	4.4mm×12.3mm	0.6μ	150M	190 ns	2300
1986	[TTS+86]	4M×1	6.2mm×16.0mm	0.4μ	620M	300 ns	1400
1986	[FOW+86]	4M×1	7.8mm×17.5mm	0.5μ	550M	200 ns	2200
1986	[KAI+86]	64K×4	3.1mm×6.9mm	0.6μ	59M	200 ns	1400
1986	[HOW+86]	1M×1	4.8mm×13.2mm	0.6μ	180M	260 ns	1400
1987	[MNA+87]	4M×1	4.9mm×14.9mm	0.4μ	450M	220 ns	2600
1987	[KSE+87]	4M×1	6.4mm×17.4mm	0.4μ	690M	230 ns	1600
1987	[OFW+87]	4M×1	6.9mm×16.1mm	0.45μ	550M	220 ns	2200
1987	[MYM+87]	256K×4	4.7mm×13.8mm	0.5μ	260M	220 ns	1100
1988	[YKMI88]	256K×16	7.5mm×12.7mm	0.4μ	600M	200 ns	2200
1988	[LCwH+88]	128K×4	8.1mm×9.6mm	0.5μ	310M	36 ns	2900
1988	[ANH+88]	16M	8.2mm×17.3mm	0.3μ	1.6G	180 ns	3700
1988	[IYK+88]	16M×1	5.4mm×17.4mm	0.25μ	1.5G	120 ns	5800
1989	[WOI+89]	4M×4	17.5mm×12mm	0.35μ	1.7G	190 ns	3200
1989	[FOS+89]	16M×1	7.9mm×17.4mm	0.3μ	1.5G	150 ns	4600
1989	[CTK+89]	16M×1	8.0mm×16mm	0.28μ	1.7G	150 ns	4100
1989	[AFM+89]	16M×1	7.7mm×17.5mm	0.25μ	2.2G	120 ns	4100
1989	[CKC+89]	16M×1	8.5mm×18.4mm	0.3μ	1.7G	190 ns	3200
1989	[LBK+89]	256K×4	6.8mm×12.3mm	0.5μ	335M	56 ns	3500
1990	[TTK+90]	16M×1	8.2mm×15.9mm	0.28μ	1.7G	150 ns	4100
1990	[KDK+90]	4M×1	5.6mm×15.2mm	0.35μ	700M	120 ns	3100
1990	[KSB+90]	16M×1	7.8mm×18.1mm	0.25μ	2.3G	150 ns	3100
1991	[NTT+91]	16M×4	9.7mm×20.3mm	0.15μ	8.8G	180 ns	2700
1991	[MMM+91]	64M×1	12.5mm×18.7mm	0.2μ	5.8G	120 ns	6000
1991	[TTU+91]	64M×1	19.9mm×11.3mm	0.2μ	5.6G	120 ns	6200
1991	[OTW+91]	64M×1	9.2mm×19.1mm	0.2μ	4.4G	90 ns	11000
1991	[YNH+91]	4M×16	234mm^2	0.2μ	5.6G	95 ns	7500
1991	[NNO+91]	4M×1	4.8mm×11.1mm	0.3μ	592M	60 ns	7400
1992	[HAH+92]	16M×1	8mm×16.6mm	0.3μ	1.5M	120 ns	6000
1992	[KDK+92]	512K×8	13.2mm×6.4mm	0.4μ	525M	60 ns	8300
1993	[STN+93]	16M×16	13.6mm×24.5mm	0.13μ	21G	60 ns	13100
1993	[KHK+93]	64M×4	14.4mm×33.2mm	0.13μ	30.6G	100 ns	5500
1994	[TNK+94]	1M×18	17.1mm×6.6mm	0.25μ	1.8G	48 ns	13500
1994	[AOT+94]	32M×8	13.3×22.8mm	0.13μ	19.5G	90 ns	9600
1994	[TTT+94]	32M×8	13.2mm×25.9mm	0.13μ	21.9G	56 ns	13700
1995	[SMK+94]	32K×9	1.7mm×5.0mm	0.4μ	53M	50 ns	7000
1995	[SMK+94]	64K×18	2.1mm×4.9mm	0.25μ	165M	40 ns	11200

Table 4.10: Survey of Peak Memory Logic Capacity (DRAM)

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2\text{s}}$
Pseudostatic							
1984	[KSY+84]	32K×9	55mm^2	1μ	55M	125 ns	2700
1991	[SKK+91]	512K×8	6.5mm×14.2mm	0.4μ	580M	116 ns	3900
Virtually Static							
1986	[NSS+86]	128K×8	6mm×13.8mm	0.5μ	330M	150 ns	1300

Table 4.11: Survey of Peak Memory Logic Capacity (Hybrid)

Year	Design	Ref.	D-Cache $\frac{\text{gate-evaluations}}{\lambda^2\text{s}}$
1984	RISC II	[SKPS84]	0
1984	MIPS	[RPJ+84]	0
1987	MIPS-X	[HHC+87]	0
1987	PA-RISC	[YFJ+87]	0
1990	PA-RISC	[TLB+90]	0
1990	SPARC	[MMN+90]	39
1992	SuperSparc	[ANAB+92]	250
1992	Alpha	[DWA+92]	610
1994	PA-RISC	[RDB+94]	0
1994	MIPS	[SYN+94]	150
1995	PowerPC	[BBB+95]	510
1995	UltraSparc	[CDd+95]	330
1995	SPARC V9	[SPA+95]	0
1995	Alpha	[BAB+95]	580
1996	MIPS	[KDS+96]	170
1996	PA-RISC	[LLNK96]	0
1996	ARM	[MWA+96]	1000
1996	Alpha	[GBB+96]	550

Table 4.12: Survey of Processor On-Chip Memory Capacity

4.5 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) are composed of a collection of programmable gates embedded in a programmable interconnect. Programmable gates are often implemented using small lookup tables. The small lookup tables with programmable interconnect allow one to take advantage of the structure inherent in many computations to reduce the amount of memory and space required to implement a function versus the full memory arrays of the previous section. Ultimately, this allows FPGA space required for an application to scale with the complexity of the application rather than scaling exponentially in the manner of pure memories.

Model For pedagogical purposes, we consider an FPGA composed of:

- n, four-input lookup tables (4-LUTs) for gates with an optional flip-flop on the output of each LUT which can be used for pipelining or data storage
- "adequate" programmable interconnect to wire up functions using the 4-LUTs
- a minimum operating cycle time, t_{cycle}, which accounts for the time to travel through one LUT and its associated interconnect.

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2\text{s}}$
1986	Xilinx 2K [CDF+86]	1 CLB (4-LUT)	$693\mu \times 715\mu$	1μ	500K	20 ns	100
1988	Xilinx 3K [Xil89, HDJ+88]	64 CLBs (2×4-LUT/CLB)	5mm×6mm (XC3020 die)	0.6μ	83M	13 ns	120
1991	UTFPGA [CSA+91]	3 4-LUTs	$900\mu\times800\mu$	0.6μ	2M	7 ns	210
1992	Xilinx 4K [Xil94b]	49 CLBs (2×4-LUT/CLB)	4.8mm×4.6mm (XC4005 Quadrant)	0.6μ	61M	7 ns	230
1994	LEGO [Seo94]	4 4-LUTs	$1240\mu\times1184\mu$	0.6μ	4M	4.1 ns	240
1995	DPGA [TEC+95]	16 4-LUTs	$1500\mu\times1750\mu$	0.5μ	10.5M	7 ns†	210
1995	Xilinx 5K [Xil91]	49 CLBs (4×4-LUTS/CLB)	3mm×3.3mm (XC5206 Quadrant)	0.3μ	110M	6 ns	290
1995	Altera Flex 8K [Alt95]	1008 LEs (4-LUT/LE)	8mm×10.5mm (81188A die)	0.3μ	930M	7.5 ns	144
1995	ORCA 2C [ATT95]	256 PLCs (4×4-LUT/PLC)	10mm×9.8mm (ATT2C10 die)	0.3μ	$1.1\text{G}\lambda^2$	7 ns	134

† No context switch – 10 ns cycle for context switch

$$\text{gate-evals}/\lambda^2\text{s} = \frac{N_{4LUT}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.13: Survey of FPGA Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1986	Xilinx 2K [CDF+86]	100	2.0×10^{-6}	2.0×10^{-6}
1988	Xilinx 3K [Xil89, HDJ+88]	120	1.5×10^{-6}	1.5×10^{-6}
1991	UTFPGA [CSA+91]	210	1.5×10^{-6}	1.5×10^{-6}
1992	Xilinx 4K [Xil94b]	230	1.6×10^{-6}	1.6×10^{-6}
1994	LEGO [Seo94]	240	9.8×10^{-7}	9.8×10^{-7}
1995	Xilinx 5K	290	1.8×10^{-6}	1.8×10^{-6}
1995	Altera 8K [Alt95]	144	9.3×10^{-7}	9.3×10^{-7}
1995	ORCA 2C [ATT95]	134	1.1×10^{-6}	1.1×10^{-6}

Table 4.14: FPGA Capacity Summary

Capacity Provided Running at full capacity and minimum operating cycle, the FPGA provides $\frac{n}{t_{cycle}}$ gate evaluations per cycle. Modern FPGAs can hold on the order of 2000 4-LUTs and run at cycle times on the order of 5-10ns. Table 4.13 computes the normalized capacity provided by a few representative FPGAs. From these numbers we see that an FPGAs provide a peak capacity on the order of 200-300 $\frac{\text{gate-evaluations}}{\lambda^2\text{s}}$.

FPGA capacity has not change dramatically over time, but the sample size is small. There is a slight upward trend which is probably representative of the relative youth of the architecture.

This peak, too, is not achievable for every application. Some effects which may prevent an application from achieving this peak include:

- **Limited interconnect** – When the network connectivity is inadequate, all of a device's capacity cannot be used. This may require either that cells buffer and transmit data or simply that cells go unused in the area. Conventional FPGA interconnect routes most applications with over 80% utilization.

- **Pipeline efficiency limits** – In heavily pipelined systems, capacity can be required to pass data across pipeline stages to all of its points of consumption. This transit capacity consumes device capacity without contributing to the evaluation capacity required for the application.

- **Limited ability to pipeline operations** – Some tasks have cyclic dependencies where a result is required before the next round of computation can begin. Unless several, orthogonal tasks are interleaved on the FPGA, the cyclic path length limits the rate at which resource can be reused and, in turn, prevents the application from fully utilizing the FPGAs capacity.

- **Limited I/O Bandwidth** – The I/O cycle time on most FPGAs is higher than the logic cycle time shown in Table 4.13. Data transfer to and from the FPGA may limit the capacity which can actually be applied to a problem.

- **Limited need for this functionality** – If a piece of functionality implemented in an FPGA is not required at the rate and frequency achievable on the FPGA, the FPGA can be employed far below its available capacity.

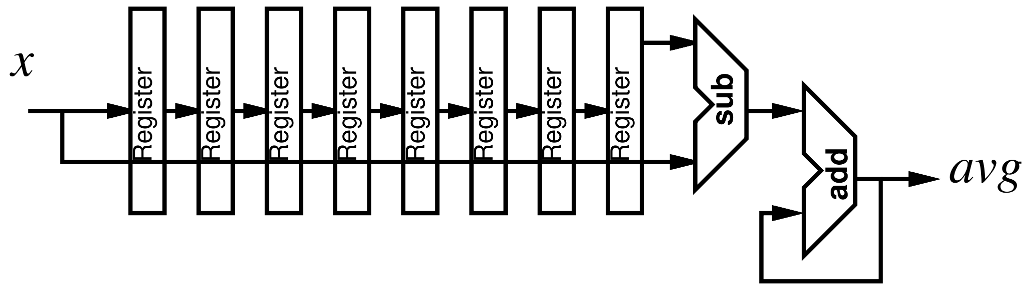


Figure 4.5: Windowed Average – Pipelined FPGA Implementation

- **Need for additional functionality** – When the FPGA cannot hold the required functional diversity for a task and must be reprogrammed in order to complete the task, the device goes partially or entirely unused during the reprogramming cycle.

As one example of pipelining, i/o, and functionality limitations, DEC's Programmable Active Memories ran from 15-33MHz for several application [BRV92]. At these rates, the peak functional density extracted from the XC3090's employed was 13-26 $\frac{\text{gate-evaluations}}{\lambda^2\text{s}}$, only about 10-20% of the potential functional density.

Example: Average Calculation Consider, again, our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot \left(x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4}\right)$$

Figure 4.5 shows a pipelined datapath to compute this windowed average. A 16-bit add on an XC4000 part is operates in 21ns. Thus a cycle time of 42ns should be achievable if the x_i's are 28 bits each – if they are 12-bit entities the 21ns cycle would be feasible. With the 32-bit datapath, the impementation requires:

- 8 pipeline registers of 16 CLBs each (128 CLBs)
- 1 adder of 17 CLBs
- 1 subtractor of 17 CLBs

$$F_{d_{yield}} = \frac{\text{128 gate evaluations}}{1.25\text{M}\lambda^2 \times 162 \times 42\text{ns}} = 15\left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

The cycle time can easily be cut in half by pipelining the two halves of each 32b operation, effectively doubling yielded functional density.

Example: Parity Calculation Consider also the FPGA implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

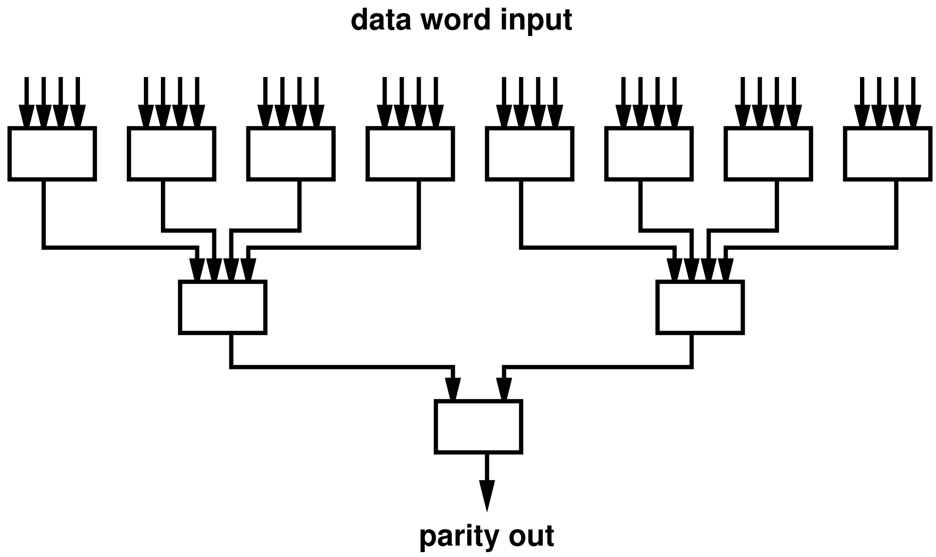


Figure 4.6: 32-bit Parity – 4-LUT Implementation

The FPGA can build the 11 gate parity reduction (See Figure 4.6). The path is three gates long. At $\approx$7ns/gate, the unpipelined version operates in roughly 21ns.

$$F_{d_{yield}} = \frac{11\text{gate evaluations}}{1.25\text{M}\lambda^2 \times 11 \times 21\text{ns}} = 38\left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Pipelining the XOR-reduction, we can reduce the cycle time and increase yield. If we pipeline at the gate level and assume that we can only cycle the part at a 10ns cycle due to clocking limitations, we yield:

$$F_{yield_d} = \frac{11\text{ gate evaluations}}{1.25\text{M}\lambda^2 \times 11 \times 10\text{ns}} = 80\left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.6 Vector and SIMD Processors

Single-Instruction, Multiple-Data (SIMD) machines are composed of a number of processing elements performing identical operations on different data items. Vector processors perform identical operations on a linear ensemble of data items. At a pedagogical level vector processors are essentially SIMD processors, though in practice the two architectures have traditionally been optimized for different usage scenarios.

Model For pedagogical purposes, we consider a SIMD/Vector array composed of:

- n processing elements (or vector units) all of which perform the same operation on each cycle
- w-bits wide processing element

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2 \text{s}}$
1987	DEC MP [Gro87]	32×4	10.4mm×9.4 mm	1μ	98M	100 ns	13
1990	MP1 [Nic90]	32×4	11.6mm×9.5mm	0.8μ	170M	70 ns	11
1990	SLAP [FHR94]	4×16	7.9mm×9.2mm	1μ	73M	100 ns	8.8
1990	BLITZEN [HBD94]	128×1	11mm×11.7mm	0.5μ	514M	50 ns	5
1993	MP2 [KT93]	32×32	14mm×14mm	0.5μ	780M	80 ns	16
1994	IMAP [YKF+94]	64×8	15.5mm×15.6mm	0.28μ	3G	25 ns	6.6
1995	MIT Abacus [BSV+95]	1000 PEs (2 3-LUTs/PE)	6.5mm×7.3mm	0.5μ	190M	8 ns	660
1995	MGAP-2 [GOI95]	2×2	0.4mm^2	0.4μ	2.5M	10 ns	160
1996	Sony [KHN+96]	4320 PEs	15.1mm×15mm	0.2μ	5.7G	20 ns	38
1996	PIP [AKY+96]	128×8	18.8mm×16.7mm	0.19μ	8.7G	33 ns	3.6

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 4.15: Survey of SIMD Processor Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1987	DEC MP [Gro87]	13	0	3.4×10^{-4}
1990	MP1 [Nic90]	11	0	2.4×10^{-4}
1990	SLAP [FHR94]	8.8	0	?
1990	BLITZEN [HBD94]	5	0	2.5×10^{-4}
1993	MP2 [KT93]	5.4	0	5.2×10^{-5}
1994	IMAP [YKF+94]	6.6	0	6.7×10^{-4}
1995	MIT Abacus [BSV+95]	660	0	3.0×10^{-4}
1995	MGAP-2 [GOI95]	160	0	2.6×10^{-5}
1996	Sony [KHN+96]	38	7.4×10^{-7}	2.0×10^{-4}
1996	PIP [AKY+96]	3.6	0	1.9×10^{-3}

Table 4.16: SIMD Processor Capacity Summary

- instruction control and distribution logic
- a minimum operating cycle time, t_{cycle} – the rate at which new instructions can be initiated in the array

Capacity Provided The SIMD/Vector array provides provides a peak of $\frac{n\times w}{t_{cycle}}$ ALU bit operations per cycle or $\frac{2n\times w}{t_{cycle}}$ gate-evaluations per cycle. Abacus, a modern, fine-grained SIMD array, supports 1000 1-bit PEs and can operate at 125MHz. Abacus thus provides 660 $\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$. Table 4.15 computes the normalized capacity provided by several SIMD arrays of varying granularity, and Table 4.17 shows the composition of a modern vector microprocessor.

SIMD/Vector arrays only achieve their peak capacity when every PE/VU is computing a useful

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1995	[ABI+95]	1×32+16×32	16.75mm×16.75mm	0.5μ	1.1G	22 ns	22

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_{sd} \times N_{scalar} W_{vd} \times N_{vector\ units}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.17: Example Vector Processor Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	[ABI+95]	22	2.3×10^{-7}	1.6×10^{-5}

Table 4.18: Vector Processor Capacity Summary

logic operation on every cycle. Limitations to achieving this peak include:

- **Limited, local interconnect** – PEs are typically connected only to a few neighbors. Every communication operation occupies a PE without providing any gate-evaluation capacity. On SIMD arrays, when data is moved into the array, around in the array, or out of the array, PEs can be occupied for several cycles without performing any logical operations.
- **Inhomogeneous operation** – All PEs are only usefully employed when the same operation is required on every data bit. When this is not the case, many PEs sit idle or perform no useful work. PEs are often masked out of operation in order to perform computations selectively on data bits.

Flynn [Fly72] summarizes some of the limitations associated with SIMD processing.

Example: Average Calculation Returning to our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Here, we assume the data is resident in PE memory on the array. It could have been loaded via a background load operation during a previous operation if it started off chip. Groups of 32 PEs are assigned to each word. To perform the average we shift the target data across the array the 8 times and accumulate at each group of 32 PEs as shown in Figure 4.7. The average takes a total of 30 cycles on 32 PEs to perform what we determined earlier to be 128 gate evaluations:

$$F_{d_{yield}} = \frac{128 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 30 \text{ cycles} \times 8\text{ns}} = 89 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Operation	cycles
initialize result with local value	1 (assumed)
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
store result	1 (assumed)

Figure 4.7: Abacus (SIMD) Implementation of Windowed Average

Example: Parity Calculation Consider also the Abacus implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

The SIMD array can perform a series 31 bitwise shift and **xor** operations to effect an XOR-scan. The **xor** can be folded into the shift such that scan operation only takes 31 cycles for the shift-XOR plus one to configure the operation. At the end of the scan operation, the partiy result is in the high (or low) processor of each 32-bit word.

$$F_{d_{yield}} = \frac{11 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 32 \text{ cycles} \times 8\text{ns}} = 14.5 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

The data memory can be preloaded with a sequence of bypass operations to allow faster accumulation. The scan can then be performed in $\log_2(32) = 5$ operations, where each operation is 3 cycles long.

$$F_{d_{yield}} = \frac{11 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 15 \text{ cycles} \times 8\text{ns}} = 31 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.7 Multimedia Processors

Multimedia processors are a recent hybrid of microprocessors, DSP, and Vector/SIMD processors. Aimed at processing video, graphics, and sound, these processors support efficient operation on data of various grain sizes by segmenting up their wide-word ALUs to provide SIMD parallel

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2 \text{s}}$
1995	[Sla95]	128 bits	290mm^2	0.25μ	4.6G	3.3 ns	8
1995	[Sla95]	128 bits	100mm^2	$0.25\mu^\dagger$	1.6G	1 ns	80
1996	[Eps95, TNH+96]	4×72	$12.8\text{mm}\times14\text{mm}$	0.25μ	2.9G	16 ns	6.3

† Experimental BiCMOS process

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_{sd} \times N_{scalar} W_{vd} \times N_{vector\ units}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.19: Multimedia Processor Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	[Sla95]	8	1.8×10^{-6}	5.6×10^{-5}
1995	[Sla95]	80	5.1×10^{-6}	1.6×10^{-4}
1996	[Eps95, TNH+96]	6.3	$2.2\text{-}8.9\times10^{-8}$	$0.96\text{-}1.2\times10^{-5}$

Table 4.20: Summary of Multimedia Processor Capacity

operation on the bytes within the word. This segmentation combats the increasing inefficiency associated with processing small data values on wide-word processors.

From Table 4.19, we see the CMOS multimedia processor have the same peak functional density as processors. The major difference is that the segmentation allows these processor to operate on 16-bit and byte-wide data without discarding a factor of 4-8× in performance. Of course, this is true only as long as these finer-grained operations can be performed efficiently in a SIMD manner.

The BiCMOS multimedia processor promised by MicroUnity would have a significantly higher performance density by exploiting a novel process. The comparison between their architecture in CMOS and BiCMOS makes it clear that this functional density advantage comes primarily from the process and not from the architecture.

4.8 Multiple Context FPGAs

Like FPGAs, multicontext FPGAs are composed of a collection of programmable gates embedded in a programmable interconnect. Unlike FPGAs, multicontext devices store several configurations for the logic and the interconnect on the chip. The additional area for the extra contexts decreases functional density, but it increases functional diversity by allowing each LUT element to perform several different functions.

Table 4.21 summarizes the capacities of some experimental, multiple-context FPGAs. Like FPGAs, these devices may suffer from limited interconnect or application pipelining limits. The additional context memory makes them less susceptible to functionality limits than traditional components. Chapter 10 details the usage of multicontext devices including their relative capacity yield compared to single context devices.

Year	Design	Composition	Per Cycle	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2\text{s}}$
1995	VEGA [JL95]	2048 4-LUT	1	144mm^2 (1 PE)	0.6μ	400M	10 ns	0.25
1995	DPGA [TEC$^+$95]	64 4-LUTs	16	1500$\mu\times$1750μ (subarray)	0.5μ	10.5M	10 ns	150
1996	TSFPGA [CD96]	64 4-LUTs	2-8	1.1mm$\times$1.2mm (subarray)	0.25μ	21M	5 ns	19-76

$$\text{gate-evals}/\lambda^2\text{s} = \frac{N_{4LUT}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.21: Survey of Multi-Context FPGA Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	VEGA [JL95]	0.25	5.1×10^{-6}	5.1×10^{-6}
1995	DPGA [TEC$^+$95]	150	6.1×10^{-6}	1.5×10^{-6}
1996	TSFPGA	19-76	3.0×10^{-6}	3-12$\times10^{-6}$

Table 4.22: Multi-Context FPGA Capacity Summary

Year	Reference	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1983	[LRSS84]	1$\times$16	6mm$\times$6mm	2.0μ	9M	$\approx$4$\times$140 ns	3
1991	[D$^+$92]	1$\times$32	150 mm^2	0.5μ	600M	62.5 ns	0.9
1991	[FKS91]	2$\times$32	18.85mm$\times$9.85mm	0.5μ	740M	20 ns	4.3
1992	[Sei92]	1$\times$32	9.25mm$\times$10.0mm	0.6μ	257M	33 ns	3.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}} \tag{4.3}$$

Table 4.23: Survey of MIMD Processor Capacity

4.9 MIMD Processors

Contemporary MIMD processors have largely been built from collections of microprocessors. As such, the functional density of these multiprocessors is certainly no larger than that of the microprocessors employed for the compute nodes. Since these machines typically require additional components for routing between processor and to connect processors into the routing network, the average functional density is actually much lower.

Table 4.23 samples a few processors which were designed explicitly for multiprocessor implementation. These processor integrate the basic network interface and, in some cases, a portion of the routing network, onto the device. While the sample size is too small to draw any strong

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1992	[CR92]	8×16	6.8mm$\times$6.7mm (core)	0.6μ	126M	40ns	25
1995	[YR95]	48×16	11.5mm$\times$11.2mm (core)	0.5μ	515M	20ns	75
1996	[MD96]	1×8	1.5mm$\times$1.2mm (PE)	0.25μ	29M	10ns	28

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.24: Survey of Reconfigurable ALU Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1992	[CR92]	25	5.1×10^{-7}	1.2×10^{-5}
1995	[YR95]	75	7.5×10^{-7}	8.9×10^{-6}
1996	[MD96]	28	$0.14\text{-}1.1\times10^{-6}$	7.1×10^{-5}

Table 4.25: Survey of Reconfigurable ALU Capacity

conclusions, the highest capacity implementations show only about half the functional density of the microprocessors we reviewed in Section 4.1.

4.10 Reconfigurable ALUs

Reconfigurable ALUs are composed of a collection of coarse-grain ALUs embedded in a programmable interconnect. Their word orientation and limitation to ALU operations distinguishes them from FPGAs.

Model For pedagogical purposes, a reconfigurable ALU contains:

- n, w-bit ALUs
- "adequate" programmable interconnect to wire up functions of the ALUs
- a minimum operating cycle time, t_{cycle}, which accounts for the time to operate in one ALU or traverse the interconnect between ALUs.
- optionally, a small instruction store associated with each ALU

Capacity Provided Running at full capacity and minimum operating cycle, the reconfigurable ALU provides $\frac{n\times w}{t_{cycle}}$ ALU bit operations per cycle. Experimental reconfigurable ALUs achieve roughly 50 ALU bit operations/λ^2s.

Like a processor D-cache, the memory on MATRIX can be used as a large lookup table. Using the MATRIX 256$\times$8 memory for function lookup, MATRIX can achieve up to 440 4-LUT gate-evaluations/λ^2s.

Like processor, reconfigurable ALUs may suffer lower yield due to:

- **Mismatched grain-size and limited ALU control** – When fine-grain operations are required, the word-wide interconnect limits which bits may interact with each other. ALU operations are word-wide SIMD making sub-word operations awkward and inefficient.

Unlike processors, the reconfigurable interconnect allows these architectures to avoid much of the data movement overhead necessary on processors. Like FPGAs, pipelining, interconnect, and functionality limits may prevent full utilization.

Example: Average Calculation Returning to our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Figure 4.8 shows a pipelined datapath to compute this windowed average on MATRIX. In this scheme, 4 BFUs (See Figure 1.4 and Chapter 13) are used to serve as an 8 value delay register, and 4 are used to perform the addition and subtration. Two cycles are required for each result so that a single datapath can be used for the add and subtract and so that the single memory can provide one read and one write cycle. The implementation yields:

$$F_{d_{yield}} = \frac{\text{128 gate evaluations}}{28.8\text{M}\lambda^2 \times 9 \times 2 \text{ cycles} \times 10\text{ns}} = 25 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Example: Parity Calculation Consider also the MATRIX implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

The most straightforward implemenation, uses the memory as an 8-LUT to calculate the parity of 8 bit data chunks. A total of 5 such chunks will perform the entire calculation (See Figure 4.9). Assuming pipelined operation of the first four and final reductions:

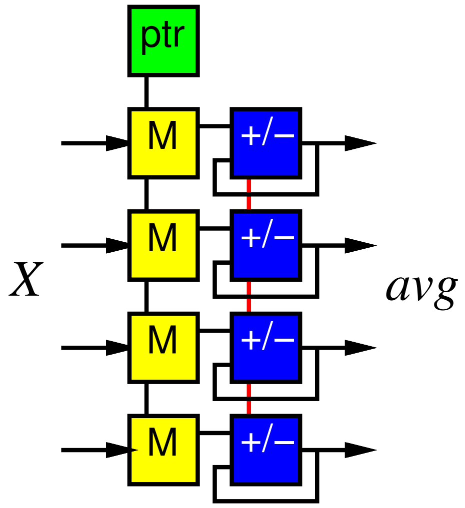
$$F_{yield_d} = \frac{\text{11 gate evaluations}}{28.8\text{M}\lambda^2 \times 5 \times 10\text{ns}} = 7.6 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.11 Summary

Table 4.26 summarizes the observed computational densities for the general-purpose architecture classes reviewed in this section.

Memories provide the highest programmable capacity of any of the devices reviewed. However, they only yield this capacity on the most complex functions – those whose complexity is, in fact, exponential in the number of input bits. The capacity they provide is not *robust* in the face of less complex tasks.

Reconfigurable devices provide the highest general-purpose capacity which can be deployed to application needs. Unlike memories capacity consumption scales along with problem complexity. Their peak performance is 10$\times$ all non-reconfigurable architectures, with the exception of large,



ptr	memory	calculate
	read <ptr>	avg←avg+new
increment ptr modulo 8	write <ptr>←new	avg←avg-old

Figure 4.8: Windowed Average – MATRIX Implementation

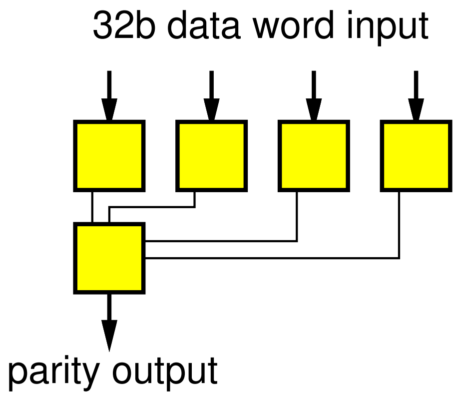


Figure 4.9: 32-bit Parity – MATRIX Implementation

Architecture	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$	Limitations
Memory	1500-15000	most complicated functions only
SIMD (1000's of PEs)	60-1200	highly homogenous computations only
FPGA	100-300	regular, highly pipelined, computations only
RALUs	50-150	semi-regular, word-wide operations
Few context DPGAs	30-150	semi-regular operations
Vector/VLIW	20-50	coarse-grain, semi-regular operations
SIMD (100's of PEs)	10-30	homogenous computations
Processors/Multimedia	4-20	word-wide operations
DSPs	2-20	word-wide operations
Highly multicontext FPGAs	0.25	

Table 4.26: General-Purpose Computational Capacity Summary

well engineered SIMD arrays. Fine-grained devices, such as FPGAs, are robust to grain-size variation, as well. Reconfigurable architectures are not, however, robust to tasks with functional diversity larger than the aggregate device capacity. Multicontext devices, such as the DPGA, sacrifice a portion of the peak FPGA capacity density to partially mitigate this problem – providing support for much higher on chip functional diversity.

Large SIMD or vector arrays have high peak performance because they ammortize a single stream of instruction control, bandwidth, and memory among a large number of active computing elements. They handle high diversity with the ability to issue a new instruction on each cycle. However, they require very large granularity operations in order to efficiently use the computational resources in the array.

Processors are robust to high functional diversity, but achieve this robustness at a large cost in available capacity – $10\times$ below reconfigurable devices. They also give up fine-grain control of operations, creating a potential for another $10\times$ loss in performance when irregular, fine-grained operations are required. Vector and VLIW structures provide slightly higher capacity density for very stylized usage patterns, but are less robust to tasks which deviate from their stylized control paradigm.

Here we see distinctions in granularity, operation diversity, and yieldable capacity. The key issues we used to classify architectures was the way the devices store and distribute instructions to processing elements. Characterizing instructions and interconnect issues with a focus on *RP*-space is the goal of Part III.

5. Case Study: Multiply

In this segment we review hardwired, programmable, and configurable multiply implementations. The custom multiplier implementations show us the functional density achievable by custom hardware on its intended task for comparison with the general-purpose structures reviewed in Chapter 4.

We use the multiply operation for this comparison because it is relatively simple and important to many computing tasks including signal processing. Because of its importance and regularity, it has received much attention over the years including many, high quality, custom implementations. Multiply is probably one of the first computational operators to be implemented in most new VLSI processes. *Considering the amount of attention given to custom multiply implementations, the comparison between custom multiplies and configurable implementations represents an upper bound on the performance disparity between custom and configurable implementations.* Few functions, if any, should show a larger disparity, and most show a significantly smaller disparity. Multiply is also interesting since it is the first piece of custom logic added to "general-purpose" processors.

In this section we use a domain specific metric for functional capacity, the multiply bit operation (MPY bit op). To allow us to compare multiplies of various sizes, we assume each $n \times m$ multiply requires $n \cdot m \cdot$ MPY bit ops. As such, we metric multiply functional density in MPY Bit Ops$/\lambda^2$s and compute it as shown in Equation 5.1.

$$D_{mpy} = \frac{m \times n}{\frac{Area}{\lambda^2} \times t_{cycle}} \tag{5.1}$$

An $n \times n$ multiply can be done in less than $O(n^2)$ operations (see for example [Knu81]), but, for the multiplies reviewed here, all of the circuits and algorithms do scale as $O(n^2)$.

5.1 Custom Multipliers

Table 5.1 summarizes the performance of numerous custom multipliers according to Equation 5.1. Implementations range from sub 1000 to almost 9000 MPY Bit Ops$/\lambda^2$s with 2000-4000 MPY Bit Ops$/\lambda^2$s representing the range of typical, high-performance, custom multipliers. Like processors there is no clear trend for improvement with time or decreasing feature size. The latest designs, if anything, show a tendency to emphasize latency over throughput resulting in lower functional density.

5.2 Semicustom Multipliers

Table 5.2 shows a few, sample, semicustom multiplier implementations. At 330 and 560 MPY Bit Ops$/\lambda^2$s, the gate array and standard cell implementations provide a factor of 5-10 less functional density than the custom implementations.

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
1984	[LGC84]	8×8	1.25mm^2	1.5μ	0.56M	120 ns	960
		16×16	5mm^2	1.5μ	2.2M	120 ns	960
1984	[UKY84]	24×24	3.8mm×3.8mm	1.0μ	14.4M	71 ns	560
1985	[GGA+85]	32×32	5.3mm×5.7mm	1.0μ	30M	56 ns	600
1985	[HFML85]	16×16	1.7 mm×1.7mm	0.75μ	5.1M	40 ns	1250
1986	[NSLKE86]	8×8	1.5mm×0.4mm	0.5μ	2.4M	3 ns	8900
1987	[LGS87]	8×8	0.61mm×0.58mm	0.5μ	1.4M	9.5 ns	4800
1988	[KKHY88]	32×32	3.2mm×5.2mm	1.0μ	17M	59 ns	1000
1988	[SJ88]	4×4	1.37mm^2	1.0μ	1.4M	16 ns	730
1989	[SH89]	64×64	3.8mm×6.5mm	0.8μ	39M	47 ns	2300
1989	[SLM+89]	16×16	1.55mm×1.44mm	0.25μ	36M	6.75 ns	1100
1990	[ADD90]	32×32	9880 mil^2	0.5μ	25.5M	35 ns	1150
		24×16	3819 mil^2	0.5μ	9.9M	28 ns	1400
		16×16	2888 mil^2	0.5μ	7.5M	22 ns	1600
1990	[YYN+90]	16×16	1.3mm×3.1mm	0.25μ	64M	3.8 ns	1000
1990	[SA90]	56×56	3.4mm×6.5mm	0.5μ	88M	30 ns	1200
1991	[MNH+91]	54×54	3.62mm×3.45mm	0.25μ	200M	10 ns	1500
1992	[FHT+92]	24×24	3.42mm×4.5mm	0.6μ	43M	30 ns	450
1992	[GSNS92]	54×54	3.36mm×3.85mm	0.4μ	81M	13 ns	2800
1993	[LS93]	12×12	2.5mm×3.7mm	0.5μ	37M	5 ns	780
1993	[SV93]	8×8	1.5mm×1.4mm	0.8μ	3.3M	4.3 ns	4500
1994	[KHANW94]	11×11	1.53mm^2	1.0μ	1.5M	22 ns	3600
		11×16	0.9mm^2	0.6μ	2.5M	19 ns	3700
1995	[OSS+95]	54×54	3.77mm×3.41mm	0.125μ	823M	4.4 ns	810
1995	[IIF+95]	16×16	0.77mm×0.72mm	0.125μ	35M	10 ns	730
1996	[HKKM96]	54×54	17mm^2	0.15μ	760M	2.5 ns	1500
1996	[LE96]	4×4	0.224mm^2	0.5μ	0.90M	17 ns	1100
1996	[MNS+96]	54×54	3.1mm×3.1mm	0.25μ	150M	8.8 ns	2200
1996	[MYO+96]	32×32	2.35mm^2	0.2μ	59M	18 ns	980

Table 5.1: Survey of Multiplier Capacity

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
Gate Array							
1987	[BMNW87]	16×16	14.4mm^2	0.75μ	26M	30 ns	330
Standard Cell							
1993	[FA93]	16×16	3mm^2	0.63μ	7.7M	60 ns	560
Layout Generator							
1993	[FA93]	16×16	1mm^2	0.63μ	2.6M	40 ns	2500

Table 5.2: Sample Semi-Custom Multiplier Capacity

Architecture	Reference	Multiply Op	area and time	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
Processor				
(basic ALU ops)	[SKPS84]	8×8	41 instructions	0.3
		16×16	81 instructions	0.7
(w/`mstep`)	[Cho89]	8×8	10 instructions	2
		16×16	18 instructions	4
		32×32	34 instructions	9
(w/ booth step)	[RPJ+84]	16×16	9 instructions	4
(w/ multiplier)	[BBB+95]	64×64	2 per cycle	250
DSP (16×16 MAC)	[WDW+85]	16×16	1 cycle	165
DSP (16×16 MAC)	[Gol87]	16×16	1 cycle	23
DSP (16×16 MAC)	[PML+89]	16×16	1 cycle	13
DSP (2×16×16 MAC)	[USO+93]	16×16	0.5 cycles	10
DSP (32×32 MAC)	[NHK95]	32×32	1 cycles	89
Memory	[SMK+94]	8×8	1 64K×18 block	10
	[SKS+93]	11×11	6 ICs	0.3
SIMD	[BSV+95]	8×8	8 PEs, 66 cycles	80
		16×16	16 PEs, 126 cycles	84
		32×32	32 PEs, 235 cycles	90
(ALU only)	[YKF+94]	8×8	1 PE, 40 cycles	1.3
(w/ lookup)		8×8	1 PE, 11 cycles	4.8
Vector				
(w/ 16×16 mpy)	[ABI+95]	16×16	8 per cycle	82
FPGA	[ATT94]	8×8	27 PLCs, 19ns	30
	[Alt96]	8×8	164 LEs, 49ns	8.6
	[LE94]	8×8	66 CLBs, 102ns	7.6
		16×16	102 CLBs, 152ns	13
		32×32	174 CLBs, 254ns	18.5
		200×200	930 CLBs, 1320ns	26
	[ID95]	16×16	316 CLBs, 26ns	25
	[ID95]	16×16	88 CLBs, 120ns	19
PADDI2	[YR95]	16×8	4 PEs, 50MHz	150
MATRIX		8×8	1 BFU, 20 ns	110
		16×16	6 BFU, 20 ns	74

Table 5.3: Survey of Programmable Multiply Capacity

5.3 General-Purpose Multiply Implementations

For comparison, Table 5.3 summarizes the capacity density of several configurable and programmable implementations. Processors without specialized multiply support show a factor of 10,000× lower performance density than hardwired multipliers. Processors, with multiply or booth step operations have only a factor of 1,000× lower performance density. FPGAs are a factor of 100-300× less dense than custom hardware. Processors, DSPs, and reconfigurable ALUs with integrated multipliers are only a factor of 10-20× lower in performance density. Figure 5.1 shows these basic relationships.

//R1,R2 hold inputs		
	ADD R0,R0,R3	
//repeated for number of bits in R2 input		
	AND R1,#1,R4	*//mask low bit*
	JUMP lequ,ZBITn	*//skip add if zero*
	SLL R1,#1,R1	*//delayed branch slot*
	ADD R3,R1,R3	*//add in scaled term*
ZBITn:	SRA R2,#1,R2	*//scale for next add*
//result in R3		

Table 5.4: Multiply Using Standard ALU Operations

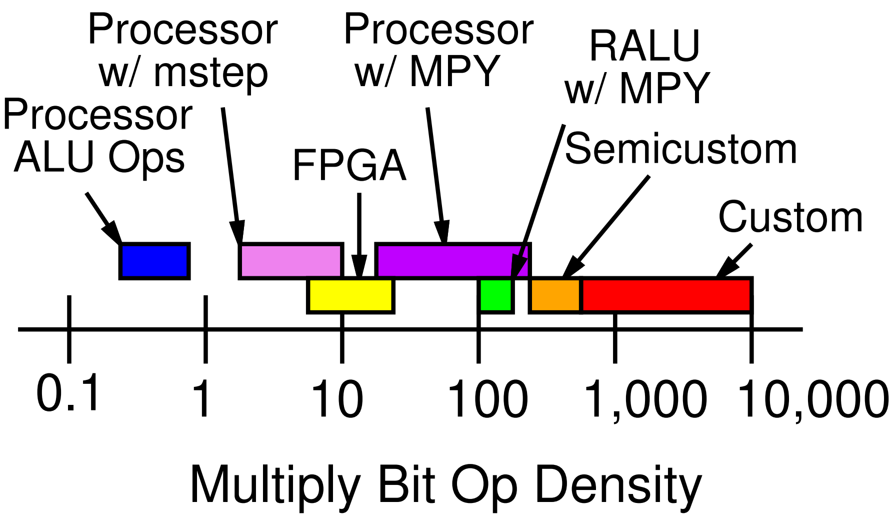


Figure 5.1: Comparison of Programmable and Custom Multiply Functional Densities

5.4 Hardwired Functional Units in "General-Purpose Devices"

One thing we note from Table 5.3 is that processors with integrated multipliers provide roughly 10% of the performance density of a custom multiplier. This comes about simply by dedicating $\approx$10% of the processor real-estate to hold a custom multiplier. Because of the importance of the multiply function in many applications and the 100-1,000$\times$ performance density differential achievable by setting aside this 10%, many processors and all DSPs augment the general-purpose core with a hardwired multiplier. Custom multiply and floating-point logic are the two main piece of custom logic which have been regularly integrated onto conventional "general-purpose" computing devices for this reason.

Structure	Reference	64×64	54×54	32×32	16×16	8×8	4×4
Custom 64×64	[SH89]	2300	1600	560	140	35	9
Custom 54×54	[GSNS92]		2800	970	240	60	15
Processor							
(w/multiplier)	[BBB+95]	250	180	63	16	4	1
(w/mstep)	[Cho89]			9	4	2	0.8
(ALU Ops)	[SKPS84]				0.7	0.3	0.2
FPGA							
	[LE94]	23	21	19	13	8	3.5

Table 5.5: Yielded Multiply Capacity as a Function of Granularity

Architecture	Reference	Multiply Op	area and time	$\frac{\text{MPY bit ops}}{\lambda^2 \text{s}}$
Processor	[Cho89]	8×8	8 instructions	2
		16×16	16 instructions	5
Memory	[SMK+94]	16×16	2 64K×18 block	19
	[SKS+93]	22×22	11 ICs	0.7
FPGA	[Cha93]	8×8	22 CLBs, 25 ns	93
		16×16	84 CLBs, 40 ns	61

Table 5.6: Survey of Specialized Programmable Multiply Capacity

5.5 Multiplication Granularity

A custom multiplier is often called upon to perform multiplies for a variety of data sizes. When multiplying operands smaller than the native multiply size, the custom multiplier yields lower multiply functional density than indicated in Table 5.1. Table 5.5 compares the yielded capacity of the various custom and programmable multipliers reviewed above.

5.6 Specialized Multiplication

In many applications, one of the operands in the multiply is a constant – or changing slowly. In these case, the operation complexity is slightly reduced, in general, and may be greatly reduce in particular circumstances. Hardwired, 2-operand, multipliers cannot take advantage of this reduced complexity whereas programmable and configurable devices can. Table 5.6 summarizes the multiply capacity provided on specialized multiplies. For comparison with the previous tables, the multiply capacity density is calculated as if it is performing a full $n \times m$ multiply. It might be more accurate to say the complexity of the problem decreased rather than the density of multiply bit ops increased, but the ratio of the performance density numbers is the same whichever way we view it. Note that the densities shown in Table 5.6 apply for *any* constant operand. Particular operands may admit to much tighter implementations.

5.7 Summary

In general, reconfigurable devices achieve 100-300$\times$ lower capacity density than their custom multiply counterparts. At the same time, they achieve 10-30$\times$ better performance than a processor building a multiply out of ALU operations. For this particular operation, most processors include a specialized multiply-step operation, which brings them closer to parity with the reconfigurable devices, or integrate a custom multiplier, which gives them a 10$\times$ advantage over the reconfigurable devices. Reconfigurable devices which also include custom multiply support achieve about the same multiply density as processor with integrated, custom, multipliers. When large, custom multiplier arrays are used on small data, the gap between the custom devices and the reconfigurable devices narrows. Similarly, when a multiply operand is constant or slowly changing, reconfigurable devices may exploit the reduction in operation complexity to narrow the density gap.

We have already noted that conventional FPGAs are poor at handling a functional diversity which is larger than the aggregate functional capacity provided by a single device (Section 4.5). Handling larger diversity may require reloading the FPGA programming, a slow process for conventional FPGAs. During the reload time, the device goes largely unused. Alternately, a more generic processing unit can be built on top of the FPGA and microsequenced like a processor. In the most extreme case of spatial limitations, we might end up building a processor-like design on top of the FPGA. Table 6.1 summarizes the capacity density provided by several processors which have been built on top of FPGAs.

From Table 6.1, we see that such processors, when optimized for the FPGA, have a peak capacity of about 2 ALU bit operations/λ^2s, or about one fourth the capacity of a custom processor. The architecture for R16 and jr16 are moderately straight RISC processor architectures, and are likely to yield about the same fraction of this capacity as most other RISC processors.

At a 4$\times$ penalty from custom processors, for high diversity operations, one would certainly be better off using, or building, a custom processor. As the commonality in the computational task increases and the area available to the FPGA increases, the FPGA can build more application specialized structures, realizing higher capacity density. This suggests there is a continuum between the most highly diverse functional operations, where FPGAs are 4$\times$ less dense than processors, to the most regular operations, where FPGAs provide 10-100$\times$ more performance density.

It is also interesting to note that the performance density penalty for handling these highly diverse operations on an FPGA is much less than the performance density penalty associated with implementing a multiplication on the FPGA.

With only a 4$\times$ performance density penalty, an FPGA processor is roughly equivalent to a

Year	**Design**	**Organization**	**Design Size**	**λ^2 area**	**cycle**	ALU bit ops / λ^2s
1991	Fliptronics R16 [Fre94]	1$\times$16	150 XC4K CLBs	190M	50 ns	1.7
1994	nP [WHG94]	1$\times$8	40 XC3K CLBs	52M	3$\times$30 ns	1.7
1994	MacDLX [Dur94]	1$\times$32	$\approx$1000 XC4K CLBs	1.2G	500 ns	0.05
1994	jr16 [Gra94]	1$\times$16	$\approx$200 XC4K CLBs	250M	25 ns	2.6
1996	j32 [Gra96]	1$\times$32	$\approx$ 250 XC4K CLBs	310M	63 ns	1.6
1996	Hokie [GHH+96]	1$\times$16	$\approx$ 140 XC4K CLBs	175M	63 ns	1.5

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 6.1: Survey of FPGA-Implemented Processor Capacity

$4\times$ smaller processor. From table 4.2, we have seen aggregate processor capacity increase from $15\text{M}\lambda^2$ in 1984 to $5\text{G}\lambda^2$ in 1995, or about 70% per year. The $4\times$ capacity density thus puts a processor implemented on an FPGA implemented in a modern processes roughly equivalent to a 2.5-3 year old processor. As such, FPGA processors – which can ride the FPGA technology to track technology advances – may be an attractive option for running legacy assembly code.

Part III

Structure and Composition of Reconfigurable Computing Devices

7. Interconnect

Programmable interconnect is the dominant contributor to die area and cycle time in configurable devices. To support their large, active functional density, the computational units must be richly interconnected and support highly parallel data routing. FPGAs, more than other general-purpose devices, place most of their area into interconnect.

We review interconnect issues in the context of on-chip networks for reconfigurable architectures. We establish typical size and delay contributions by analyzing conventional FPGA implementations, then we look at how resource requirements grow with increasing array size. Understanding conventional sizes and growth factors help us characterize the design space. It also serves as background context for the architectural developments described in Part IV.

In this chapter, we:

1. Decompose FPGA area into three component parts and establish the relative areas of each: fixed logic, configuration memory, interconnect resources
2. Review issues in configurable network design
3. Establish growth rates for interconnect and description requirements as a function of network size
4. Establish relationships between network size and richness of network interconnect
5. Examine the efficiency of device utilization when viewed in relation to network resource utilization rather than programmable gate utilization
6. Examine the effects of multibit granularity on interconnect resource requirements

7.1 Dominant Area and Delay

7.1.1 Fixed Area

Reviewing LUT-based FPGA implementations from Table 4.13, and calculating the area per 4-LUT (Table 7.1), we see that each 4-LUT is roughly 600Kλ^2. The flip-flop and 16:1 LUT multiplexor make up very little of this area, easily less than 20Kλ^2. [BFRV92] estimates the area of the 4-LUT multiplexor with flip-flop as 13Kλ^2. In our own DPGA implementation these items occupied 15Kλ^2 (See Chapter 10). The majority of the area associated with each 4-LUT (97%), goes into programmable interconnect and configuration memory

This breakdown, alone, shows us one reason why a full 4-input lookup table is often used as the programmable logic element, rather than a more restricted gate. The area required for the full LUT, including its configuration memory, is less than 10% of the area of the 4-LUT cell, such that there is little advantage to reducing the cell's functional size.

Year	Design	λ^2 **area** / 4-LUT
1986	Xilinx 2K [CDF+86]	500K
1988	Xilinx 3K [Xil89, HDJ+88]	650K
1991	UTFPGA [CSA+91]	670K
1992	Xilinx 4K [Xil94b]	630K
1994	LEGO [Seo94]	1020K
1995	DPGA [TEC+95]	660K
1995	Xilinx 5K [Xil91]	560K
1995	Altera 8K [Alt95]	930K
1995	Orca 2C [ATT95]	1060K

Table 7.1: FPGA 4-LUT Size

Part	Approximate Bits/4-LUT
Xilinx xc2k	160
Xilinx xc3k	100
Xilinx xc4k	200
Xilinx xc5k	120
UTFPGA	48
LEGO	120
DPGA	4×40
Altera 8k	190
Orca 2C	120

Table 7.2: Bits per 4-LUT

7.1.2 Interconnect and Configuration Area

The number of programming bits per 4-LUT for these devices is summarized in Table 7.2. Using a rather large memory cell ($\approx$4.5Kλ^2/bit), the memory accounted for 35% of the area on UTFPGA. With 4-contexts and 600λ^2 3T-DRAM memory cells, memory only occupied 33% of the area on the DPGA. If we assume 1000λ^2 static memory cells, for the Xilinx parts, memory accounts for about 15-30% of that area ($\frac{160\text{K}\lambda^2}{500\text{K}\lambda^2}$ (32%), $\frac{100\text{K}\lambda^2}{650\text{K}\lambda^2}$, (15%), $\frac{200\text{K}\lambda^2}{630\text{K}\lambda^2}$, (32%), $\frac{120\text{K}\lambda^2}{560\text{K}\lambda^2}$, (21%)). Making similar assumptions, memory accounts for 21% of an Altera 8K part ($\frac{190\text{K}\lambda^2}{930\text{K}\lambda^2}$) and 11% ($\frac{120\text{K}\lambda^2}{1060\text{K}\lambda^2}$) of an Orca 2C part. Interconnect and routing occupies the balance of the area (70-90%).

7.1.3 Delay

Most vendors lump interconnect timing in with lookup table evaluation, making it difficult to distinguish the components of delay. Table 7.3 summaries interconnect and LUT logic delay for Altera's 8K series [Alt95] and our own experience with the DPGA (Chapter 10). From here, we

Design	Path	Total Delay	LUT delay	Interconnect
Altera 8K	LUT-local-LUT	2.5 ns	2 ns	20%
[Alt95]	LUT-row-local-LUT	7.5 ns	2 ns	73%
	LUT-row-column-local-LUT	10.5 ns	2 ns	81%
MIT DPGA	LUT-LUT (in subarray)	3.5 ns	1.5 ns	60%
[TEC+95]	LUT-xbar-LUT	7 ns	1.5 ns	80%

Table 7.3: FPGA Delay Breakdown

see that interconnect typically accounts for 80% of the path delay.

7.2 Problems with "Simple" Networks

FPGA networks, which already need to interconnect thousands of independent processing elements, do not, typically, look like conventional multiprocessor networks. In particular, a number of conceptually "simple" network structures commonly used as the basis for multiprocessor networks do not scale properly for use in FPGAs. In this section, we review three typical organizations and highlight their shortcomings on the scale required for FPGA networks.

1. crossbars
2. multistage networks
3. mesh networks

This review helps identify and motivate important design issues for reconfigurable interconnect which we will address in the following section.

7.2.1 Crossbars

To guarantee arbitrary, full, connectivity among elements, we could build a a full crossbar for the interconnection network. In such a scheme we would not have to worry about whether or not a given network could be mapped onto the programmable interconnect nor would we have to worry about where logic elements were placed. Unfortunately, the cost for this full interconnect is prohibitively high.

For an n element array where each element is a k-input function (*e.g.* k-LUT), the crossbar would be an $n \times k \cdot n$ crossbar. Arranged in a roughly square array, each input and output must travel $O(\sqrt{n})$ distance, before we account for saturated wire density. Since interconnect delay is proportional to interconnect distance, this implies the interconnect delay grows at least as $O(\sqrt{n})$. However, the bisection bandwidth for any full crossbar is $O(n)$. For sufficiently large n, this bisection bandwidth requires that the side of an array be $O(n)$ to accommodate the wires across the bisection. In turn, the $O(n)$ bisection bandwidth dictates an area $O(n^2)$. This also dictates input and output wires of length $O(n)$. For large crossbars, wire size dominates the areas. These growth rates are not acceptable even at the level of thousands of LUTs. If we were to build devices using a single monolithic crossbar for interconnect:

- area growth would be as the square of the number of LUTs supported
- cycle time would slow down linearly with the number of LUTs in the network

Consider, for the sake of illustration, the size of a crossbar required to interconnect a 2,500 4-LUT device. We will assume the minimum wire pitch is 8λ and the crossbar is implemented with two layers of dense metal routed at this minimum wire-pitch. The area of such an array, as dictated simply by the wiring would be:

$$(8\lambda \times 4 \times 2500) \times (8\lambda \times 2500) = 1.6\text{G}\lambda^2$$

Making for an area of $\frac{1.6\text{G}\lambda^2}{2500} = 640\text{K}\lambda^2$ per 4-LUT just to handle the requisite wiring. Conventional FPGAs use a single SRAM cell to configure each of the crosspoints in the crossbar. If this were done, the area would be memory bit dominated rather than wire dominated and take up:

$$1000\lambda^2 \times (4 \times 2500) \times 2500 = 25\text{G}\lambda^2$$

Which results in $10\text{M}\lambda^2$ per 4-LUT just to hold the configuration memory. The area per LUT, of course, continues to grow linearly in the number of LUTs for larger networks.

7.2.2 Multistage Networks

Multistage interconnection networks (*e.g.* butterfly, omega, CLOS, Benes) can reduce the total number of switches required from $O(n^2)$ to $O(n\log(n))$, but have the same bisection bandwidth problem. Between any two pair of stages in a butterfly network, the total bisection bandwidth is $O(n)$, such that the wiring requirements dictate that area grows at $O(n^2)$.

7.2.3 Mesh Interconnect

At the opposite interconnect extreme, we can use only local connections within the array between adjacent, or close, array elements. By limiting all the connections to fixed distances, the link delay does not grow as the array grows. Further, the bisection bandwidth in a mesh configuration is $O(\sqrt{n})$ and hence, never dominates the logical array element size. However, communicating a piece of data between two points in the array requires switching delay proportional to the distance between the source and the destination. Since switching delay through programmable interconnect is generally much larger than fanout or wire propagation delay along a fixed wire, this makes distant communication slow and expensive. For example, in a topology where direct connections are only made between an array element and its north, east, south, and west neighbors (typically called a NEWS network), a signal must traverse a number of programmable switching elements proportional to the Manhattan distance between the source and the destination ($O(\sqrt{n})$). For the interconnect network topologies typically encountered in logic circuits, this can make interconnect delay quite high – easily dominating the delay through the array element logic.

7.3 Issues in Reconfigurable Network Design

With this background, we can begin to formulate the design requirements for programmable interconnect:

1. **Provide adequate flexible** – The network must be capable of implementing the interconnection topology required by the programmed logic design with acceptable interconnect delays.

2. **Use configuration memory efficiently** – Space required for configuration memory can account for a reasonable fraction of the array real-estate, as we saw in Section 7.2.1. However, as we will see in Section 7.8, configuration encodings can be tight and do not have to take up substantial area relative to that required for wires and switches.

3. **Balance bisection bandwidth** – As discussed above, interconnect wiring takes space and can, in some topologies, dominate the array size. The wiring topology should be chosen to balance interconnect bandwidth with array size and expected design interconnect requirements.

4. **Minimize delays** – The delay through the routing network can easily be the dominant delay in a programmable technology (See Section 7.1.3). Care is required to minimize interconnect delays. Two significant factors of delay are:

   (a) **Propagation and fanout delay** – Interconnect delay on a wire is proportional to distance and capacitive loading (fanout). This makes interconnect delay roughly proportional to distance run, especially when there are regular taps into the signal run. Consequently, small/short signal runs are faster than long signal runs.

   (b) **Switched element delay** – Each programmable switching element in a path (*e.g.* crossbar, multiplexor) adds delay. This delay is generally much larger than the propagation or fanout delay associated with covering the same physical distance. Consequently, one generally wants to minimize the number of switch elements in a path, even if this means using some longer signal runs.

   Switching can be used to reduce fanout on a line by segmenting tracks, and large fanout can be used to reduce switching by making a signal always available in several places. Minimizing the interconnect delay, therefore, always requires technology dependent tradeoffs between the amount of switching and the length of wire runs.

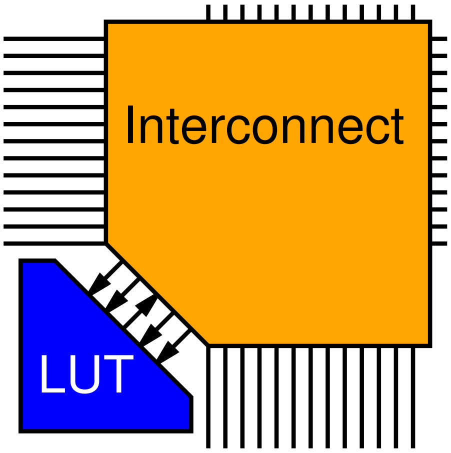


Figure 7.1: Conventional FPGA Interconnect Topology

7.4 Conventional Interconnect

Conventional FPGA interconnect takes a hybrid approach with a mix of short, neighbor connections and longer connections. Figure 7.1 shows a canonical FPGA LUT tile. Full connectivity is not supported even within the interconnect of a single tile. Typically, the interconnect block includes:

- A hierarchy of line lengths – some interconnect lines span a single cell, some a small number of cells, and some an entire row or column
- Limited, but not complete, opportunity for corner turns
- Limited opportunity to link together shorter segments for longer routes
- Options for the value generated by the LUT to connect to some lines of each hierarchical length in each direction – perhaps including some local interconnect lines dedicated to the local LUT output
- Opportunity to select the k-LUT inputs from most of the lines converging in the interconnect block

The amount of interconnect in each of the two dimensions is not necessarily the same. Figure 7.2 shows these features in a caricature of conventional FPGA interconnect.

The University of Toronto has performed a number of empirical interconnect studies aimed at establishing basic FPGA interconnect characteristics, including:

- How densely to populate the interconnect with switches and the number of routing tracks required to route representative circuits [RB91]
- The merits of hierarchical interconnect [AL94]

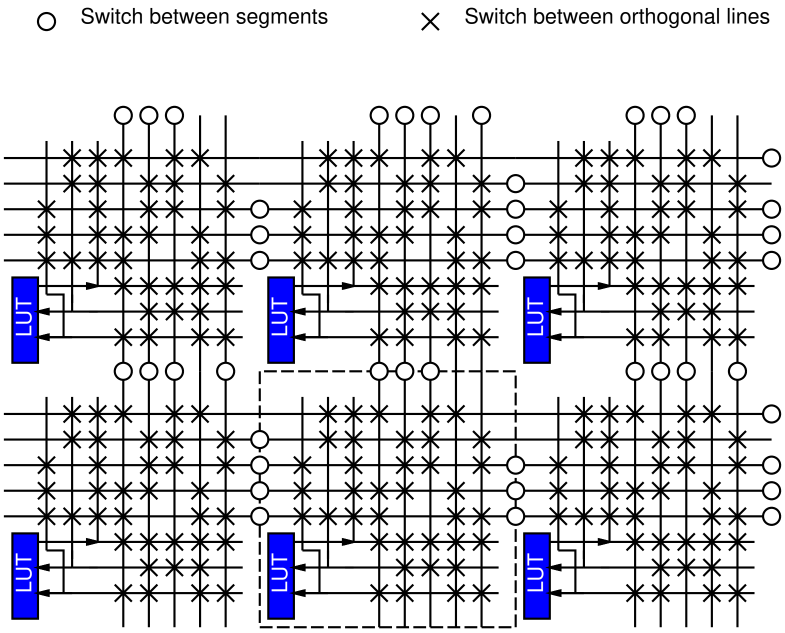


Figure 7.2: FPGA Interconnect Caricature

- The distribution of line lengths [Seo94]

One of the key differences between FPGAs and traditional "multiprocessor" networks is that FPGA interconnect paths are locked down serving a single function. The FPGA must be able to simultaneously route all source-sink connections using unique resources to realize the connectivity required by the FPGA. Another key difference is that the interconnection pattern is known *a prior* to execution, so offline partitioning and placement can be used to exploit locality and thereby reduce the interconnect requirements.

7.5 Switch Requirements for FPGAs with 100-1000 LUTs

Before we examine how network requirements scale with connectivity and network size, in this section, we briefly review the number of switches conventionally employed by networks supporting 100 to 1000 4-LUTs. Brown and Rose [RB91, BFRV92] suggest each 4-LUT in a moderate sized FPGA with 100's of 4-LUTs will require 200-400 switches. Agarwal and Lewis suggest approximately 100 switches per LUT for hierarchical FPGAs [AL94] with some reduction in logic utilization. Conventional, commercial FPGAs do little or no encoding on their interconnect bit streams – that is, each interconnect switch is controlled by a single configuration bit. From the configuration bit summary in Table 7.2, we see that commercial devices also exhibit on the order of 200 switches per 4-LUT. The fact that conventional FPGAs can, with difficulty, route most all designs using less than 80-90% of the device LUTs, suggests that they chose a number of switches which provides reasonably "adequate" interconnect for the current device sizes – hundreds to a couple of thousand 4-LUTs.

7.6 Channel and Wire Growth

In Sections 7.1 and 7.5, we have empirically established the size of conventional interconnect. However, as we glimpsed in Section 7.2, the area which these resources occupy is not necessarily independent of the number of LUTs interconnected. In this section we look at how interconnect requirements will grow with the number of LUTs supported.

The best characterization to date which empirically meters interconnect requirements is Rent's Rule [LR71, Vil82]:

$$N_{io} = CN_{gates}^{p} \tag{7.1}$$

N_{io} is the number number of interconnection in/out of a region containing N_{gates}. C and p are empirical constants. For logic functions $0.5 < p < 0.7$, typically.

El Gamal used a stochastic model to estimate the interconnection requirements for channeled gate arrays [Gam81]. He found that each routing channel requires $O(\overline{R})$ tracks if the average wire length, $\overline{R}$, grows faster than $O\left(\log(n)\right)$. n here is the total number of circuits in the array, generally arranged in an $\sqrt{n} \times \sqrt{n}$ array. Brown used El Gamal's routing model for FPGAs and found good correspondence between it and FPGA interconnect requirements [Bro92]. For large numbers of gates (N_{gates}) and $p > 0.5$, Donath finds that $\overline{R} \propto N_{gates}^{p-0.5}$ [Don79]. Together this means the channel width grows as $O(N_{gates}^{p-0.5})$. From which we can derive the interconnect requirements growth:

$$\begin{aligned}
N_{channels} &\propto \sqrt{N_{gates}} \\
N_{interconnect_width} &= N_{channels} \times N_{channel_width} \\
N_{interconnect_area} &= N_{interconnect_width}^{2} \\
N_{interconnect_area} &\propto (\sqrt{N_{gates}} \times N_{gates}^{p-0.5})^{2} \\
N_{interconnect_area} &\propto N_{gates}^{2p}
\end{aligned} \tag{7.2}$$

$p = \frac{2}{3}$ is often considered a good, conservative, value for p to handle most interconnect requirements.

For $p < 0.5$, Donath finds that $\overline{R}$ grows as $O(\log(N_{gates}))$ or smaller. For $\overline{R} \leq O(\ln(N_{gates}))$, El Gamal's model suggests the the track width grows as $O(\ln(N_{gates}))$. In this case, total interconnect requirements grow as $O(N_{gates}\log^{2}(N_{gates}))$.

7.6.1 Rent's Rule Based Hierarchical Interconnect Model

To make this size estimate more concrete, let us consider a specific structure built according to Rent's Rule. We build a fully hierarchical interconnect with inter-level signaling bandwidth growing according to Rent's Rule. To simplify analysis, we consider only unidirectional signal wires.

The gates are recursively partitioned into n equally sized sets at each level of the hierarchy. The principal interconnect occurs at each node of convergence in the hierarchy (See Figure 7.3). At a level l in the hierarchy, each node has a fan-in from below of $n \times n_{out_{l-1}}$ signals and a fan-in from above of n_{in_l}. Similarly, it has a fan-out of $n \times n_{in_{l-1}}$ toward the leaves and n_{out_l} towards

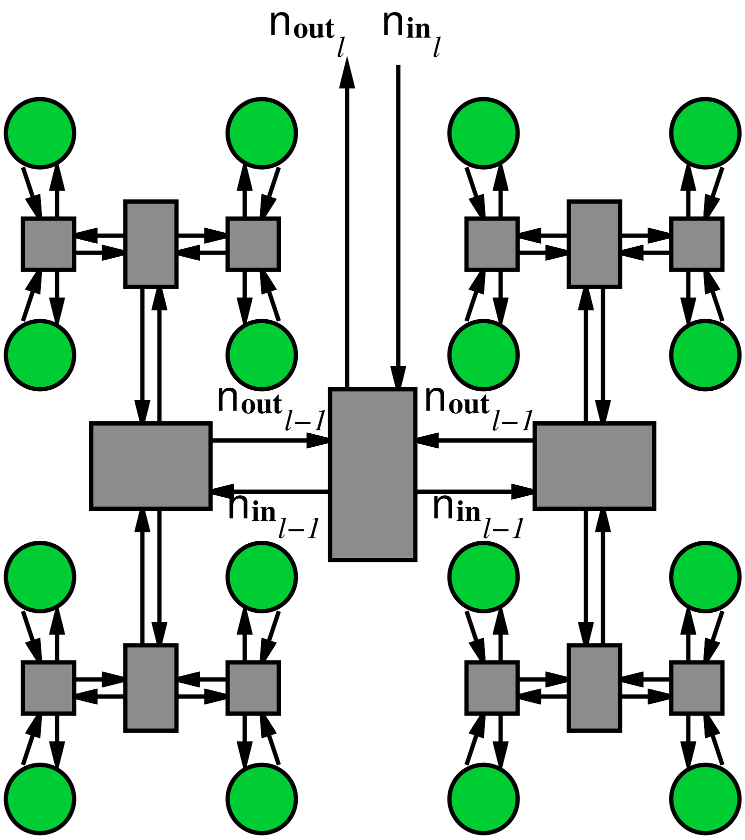


Figure 7.3: Logical Structure of Hierchical Interconnect

the root. At each level l, we have n_{LUT_l} LUTs, n_{in_l} external inputs, and n_{out_l} external outputs. According to the hierarchical combining and Rent's Rule growth, we have:

$$\begin{aligned} N_{LUT_l} &= n^l \\ n_{in_l} &= C\left((n)^l\right)^p \\ n_{out_l} &= C\left((n)^l\right)^p \end{aligned} \tag{7.3}$$

We take $C = K$, the number of LUT inputs. When $\left(C\left((n)^l\right)^p\right) > n^l$, which will be true for small l, we take $n_{out_l} = n^l$ – that is, all outputs are passed out of the region when this Rent bandwidth permits.

Logically, we have $n+1$ distinct output directions from each node of convergence in the interconnect – n for the n leaves, plus one for the root. Allowing full connectivity within each tree node, each of the n leaves picks its n_{in} inputs from the $(n-1) \times n_{out}$ outputs from its siblings and from the n_{in} inputs from the parent node. The n_{out} outputs of this node are selected from

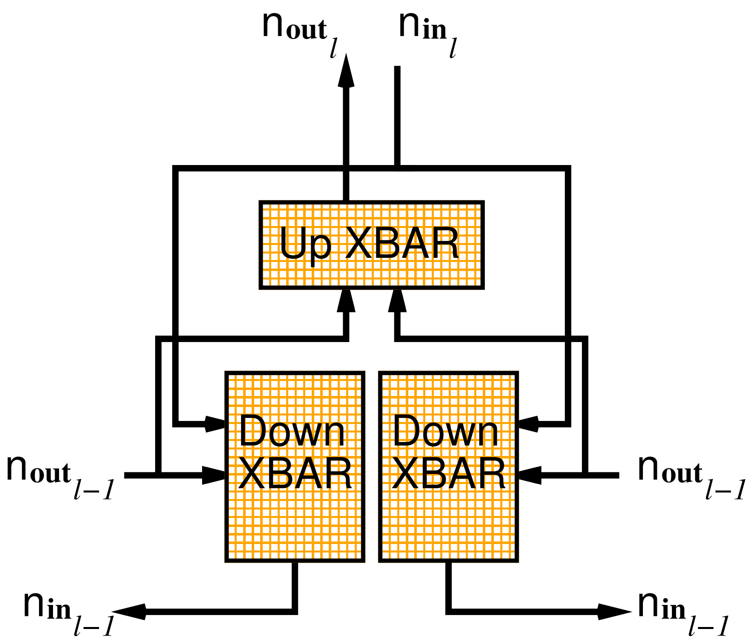


Figure 7.4: Switching node in 2-ary Hierarchical Interconnect

the $n \times n_{out}$ outputs from all n subtrees converging at this point. Figure 7.4 shows this basic arrangement for $n = 2$.

7.6.2 Wire Growth in Rent Hierarchy Model

First, let us consider how wiring resources grow in this structure. At each stage of the hierarchy, there are $n_{in_l} = n_{out_l} = C\left((n)^l\right)^p$ wires coming and leaving each subarray. This makes the bisection width of $O\left(\left(n^l\right)^p\right) = O\left((N_{LUT_l})^p\right)$. For a two-dimensional network layout, this bisection width must cross out of the subarray through the perimeter. Thus the perimeter of each subarray is $O\left((N_{LUT_l})^p\right)$. The area of the subarrays will be proportional to the square of its perimeter, making:

$$A_{subarray} \propto \left(N_{LUT}^p\right)^2$$

The area required for each LUT based on wiring constraints, then, goes as:

$$A_{LUT} = \frac{A_{subarray}}{N_{LUT}} \propto \frac{\left(N_{LUT}^p\right)^2}{N_{LUT}}$$

$$A_{LUT} \propto \left(N_{LUT}^{(2p-1)}\right) \tag{7.4}$$

Not unsurprisingly, this matches the interconnect growth we derived in Equation 7.2. Of course, if $p < 0.5$, wiring is not the dominant resource constraining LUT area. A_{LUT} may be $O(1)$ for $p \leq 0.5$ as far as strict wiring requirements are concerned.

7.6.3 Switch Growth in Rent Hierarchy Model

We can also look at the number of switches required if each of the logical switching units is a fully-populated crossbar. At each level, l, the total number of switches is:

$$\begin{aligned}
n_{SW-total_l} &= \left(n \times \underbrace{\left(\overbrace{\left((n-1) \cdot n_{out_{l-1}} + n_{in_l}\right)}^{\text{inputs to down xbar}} \times n_{in_{l-1}} \right)}_{\text{each down xbar}} \right) + \underbrace{\left(n_{out_l} \times \left(n \cdot n_{out_{l-1}}\right)\right)}_{\text{up xbar}} \\
&= \left(n \times \left(\left((n-1) \cdot K n^{p(l-1)} + K n^{pl}\right) \times K n^{p(l-1)}\right)\right) + K n^{pl} \times \left(n \cdot K n^{p(l-1)}\right) \\
&= \left(n \times \left(\left((n-1+n^p) \cdot K n^{p(l-1)}\right) \times K n^{p(l-1)}\right)\right) + K n^{pl} \times \left(n \cdot K n^{p(l-1)}\right) \\
&= \left(n \cdot (n-1+n^p) \cdot K^2 n^{2p(l-1)}\right) + \left(n^p \cdot n \cdot K^2 n^{2p(l-1)}\right) \\
&= \left(n \cdot K^2 n^{2p(l-1)}\right) \cdot \left((n-1+n^p) + n^p\right) \\
&= \left(n \cdot K^2 n^{2p(l-1)}\right) \cdot (n-1+2n^p) \qquad (7.5)
\end{aligned}$$

Amortizing across the number of LUTs supported at level l, we can count the number of switches per LUT at each level:

$$\begin{aligned}
n_{SW_l} &= \frac{\left(n \cdot K^2 n^{2p(l-1)}\right) \cdot (n-1+2n^p)}{n^l} \\
&= \left(K^2 n^{(2pl-2p-l+1)}\right) \cdot (n-1+2n^p) \\
&= \left(K^2 n^{(2p-1)(l-1)}\right) \cdot (n-1+2n^p) \qquad (7.6)
\end{aligned}$$

Summing across all levels, we can thus calculate the number of switches per LUT as a function of the size of the network.

$$n_{SW} = \sum_{l=1}^{\log_n(N_{LUT})} \left(K^2 n^{(2p-1)(l-1)}\right) \cdot (n-1+2n^p)$$

Substituting N_{LUT} for n^l, and expanding sum:

$$n_{SW} = \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot \left(\frac{1}{n^{(2p-1)}} + \left(\frac{1}{n^{(2p-1)}}\right)^2 + \cdots + \left(\frac{1}{n^{(2p-1)}}\right)^l\right) \cdot (n-1+2n^p) \qquad (7.7)$$

For $p > 0.5$, this gives us:

$$n_{SW} = \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot \left(\frac{\frac{1}{n^{(2p-1)}} - \left(\frac{1}{n^{(2p-1)}}\right)^{\left(\log_n(N_{LUT})+1\right)}}{\left(1 - \frac{1}{n^{(2p-1)}}\right)} \right) \cdot (n-1+2n^p)$$

$$
\begin{aligned}
&< \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot \left(\frac{\frac{1}{n^{(2p-1)}}}{(1-\frac{1}{n^{(2p-1)}})}\right) \cdot (n-1+2n^p) \\
&< \left(N_{LUT}^{(2p-1)}\right) \cdot K^2 \cdot \left(\frac{1}{n^{(2p-1)}-1}\right) \cdot (n-1+2n^p) \qquad (7.8)
\end{aligned}
$$

For $p = 0.5$, each sum term in Equation 7.7 goes to one:

$$
\begin{aligned}
n_{SW} &= \left(N_{LUT}^0 \log_n (N_{LUT})\right) \cdot K^2 \cdot (n-1+2n^p) \\
n_{SW} &= \log_n (N_{LUT}) \cdot K^2 \cdot (n-1+2n^p) \qquad (7.9)
\end{aligned}
$$

For $p < 0.5$,

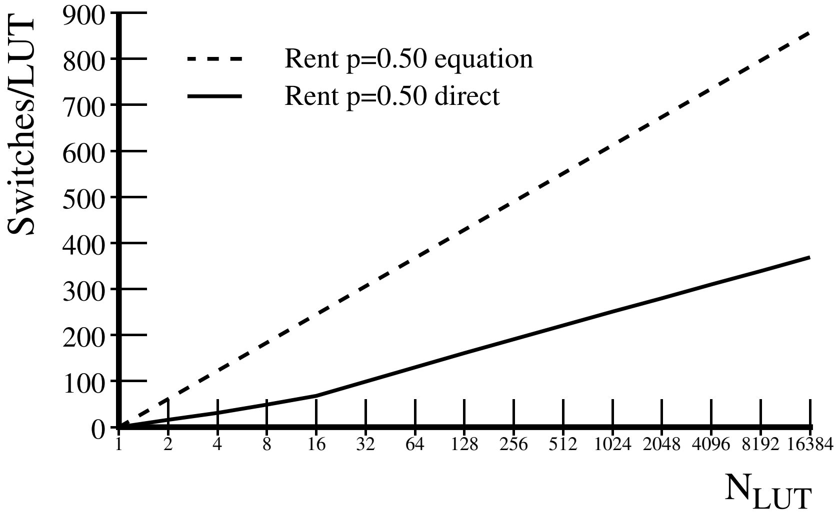
$$
\begin{aligned}
n_{SW} &= \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \left(\left(n^{(1-2p)}\right)^{\log_n(N_{LUT})} + \cdots + \left(n^{(1-2p)}\right)^2 + \left(n^{(1-2p)}\right)\right) \\
n_{SW} &= \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \cdot \left(\left(\left(n^{(1-2p)}\right)^{\log_n(N_{LUT})}\right) \cdot \left(\frac{1-\left(\frac{1}{n^{(1-2p)}}\right)^{\log_n(N_{LUT})}}{1-\frac{1}{n^{(1-2p)}}}\right)\right) \\
n_{SW} &< \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \cdot \left(\left(n^{(1-2p)}\right)^{\log_n(N_{LUT})}\right) \cdot \left(\frac{1}{1-\frac{1}{n^{(1-2p)}}}\right) \\
n_{SW} &< \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \cdot \left(n^{((1-2p)\log_n(N_{LUT}))}\right) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) \\
n_{SW} &< \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \cdot \left(n^{\log_n\left(N_{LUT}^{(1-2p)}\right)}\right) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) \\
n_{SW} &< \left(K^2 N_{LUT}^{(2p-1)}\right) \cdot (n-1+2n^p) \cdot \left(N_{LUT}^{(1-2p)}\right) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) \\
n_{SW} &< \left(N_{LUT}^{(2p-1)}\right) \cdot \left(N_{LUT}^{(1-2p)}\right) \cdot K^2 \cdot (n-1+2n^p) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) \\
n_{SW} &< K^2 \cdot (n-1+2n^p) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) \qquad (7.10)
\end{aligned}
$$

Putting these cases, together:

$$
n_{SW} \leq \begin{cases} K^2 \cdot (n-1+2n^p) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) & p < 0.5 \\ \log_n (N_{LUT}) \cdot K^2 \cdot (n-1+2\sqrt{n}) & p = 0.5 \\ \left(N_{LUT}^{(2p-1)}\right) \cdot K^2 \cdot \left(\frac{1}{n^{(2p-1)}-1}\right) \cdot (n-1+2n^p) & p > 0.5 \end{cases} \qquad (7.11)
$$

Here, we see switching area per LUT grows as $O(1)$, for $p < 0.5$, and $O\left(N_{LUT}^{(2p-1)}\right)$ for $p > 0.5$. Again, this matches our wiring growth expectations (Equation 7.2).

While Equation 7.11 gives the correct growth rates it overestimates the required number of switches on two accounts:



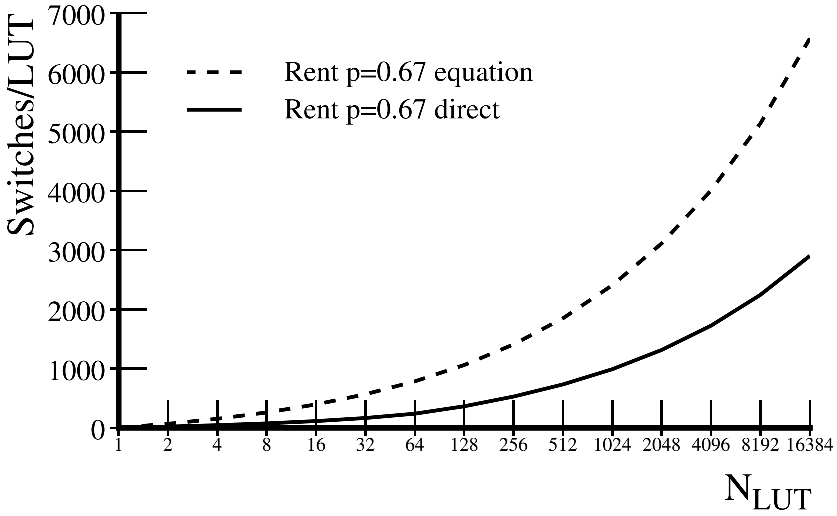
$k = 4, n = 2, p = 0.5$

Figure 7.5: Switches per LUT – Equation versus Direct Calculation

1. It does not take into account the limited number of distinct outputs at the lowest stages of the network – *i.e.* when there are less outputs than the Rent i/o suggests.

2. It approximates each crossbar as requiring $n \times m$ switches. However, since each crossbar is performing an n choose m operation, only $n \times (m - 1)$ crosspoints are actually required to provide full connectivity at each tree interconnect node.

Figures 7.5 and 7.6 show the difference between Equation 7.11 and a direct calculations which includes the above two effects. Asymptotically, the difference is in the constant factor. Note that for $p = 0.5$, the number of switches per LUT computed by the direct calculation in the 256–1024 LUT range is 190-250, which is on par with contemporary interconnects (Section 7.5).



$k = 4$, $n = 2$, $p = 0.67$

Figure 7.6: Switches per LUT – Equation versus Direct Calculation

7.7 Network Utilization Efficiency

In the previous section, we saw that the amount of interconnect we need to provide depends upon the connectivity of the network. This makes it difficult to design a single network which will *efficiently* accommodate arbitrary designs. If the design has limited connectivity, but the network provides a large amount of connectivity, the network is over designed relative to the design and provides less functional density than achievable. If the design has considerable connectivity, but the network provides less, the design must be routed sparsely on the interconnect, leaving many of the device LUTs unusable.

Using the switching models derived in the previous section, we can examine the relative inefficiencies of using a design with Rent exponent p_{design} on a network with Rent exponent p_{net}. We do this by looking at the ratio of the area occupied by a design with N_{LUT} LUTs and on top of a network built using Rent exponent p_{net}. If $p_{net} > p_{design}$, then the ratio is simply the ratio of the area per LUT of a p_{net} interconnect of N_{LUT} LUTs to the area per LUT of a p_{design} interconnect of N_{LUT} LUTs. However, if $p_{design} > p_{net}$, we cannot simply map the design netlist on top of the device LUTs. Here, we have to figure out how much larger the network must be than the number of LUTs in the design in order to accommodate the highly connected design. Let us call this scaling factor C. In order for the network to accommodate the design, it must have enough i/o bandwidth into each subregion. Starting at the top level in the design, this means:

$$n_{io_{design}} \leq n_{io_{net}}$$

The only way to accommodate this requirement with a fixed p_{net} is to scale up the network used. Applying Rent's Rule (Equation 7.1), this means:

$$KN_{LUT}^{p_{design}} \leq K(C \cdot N_{LUT})^{p_{net}}$$

Solving this relation for equality:

$$\begin{aligned} N_{LUT}^{p_{design}} &= (C \cdot N_{LUT})^{p_{net}} \\ N_{LUT}^{p_{design}/p_{net}} &= C \cdot N_{LUT} \\ C &= N_{LUT}^{(p_{design}/p_{net}-1)} \end{aligned} \tag{7.12}$$

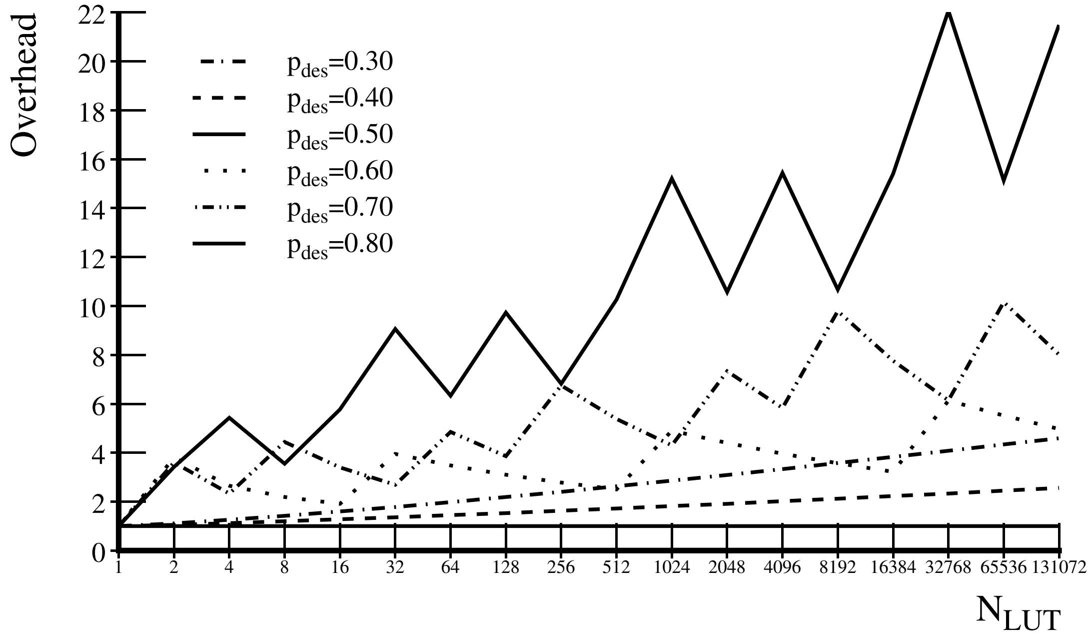
Note that once we accommodate the top level of the design, all other levels are also accommodated as well. That is, once we have chosen C as above, at the top level:

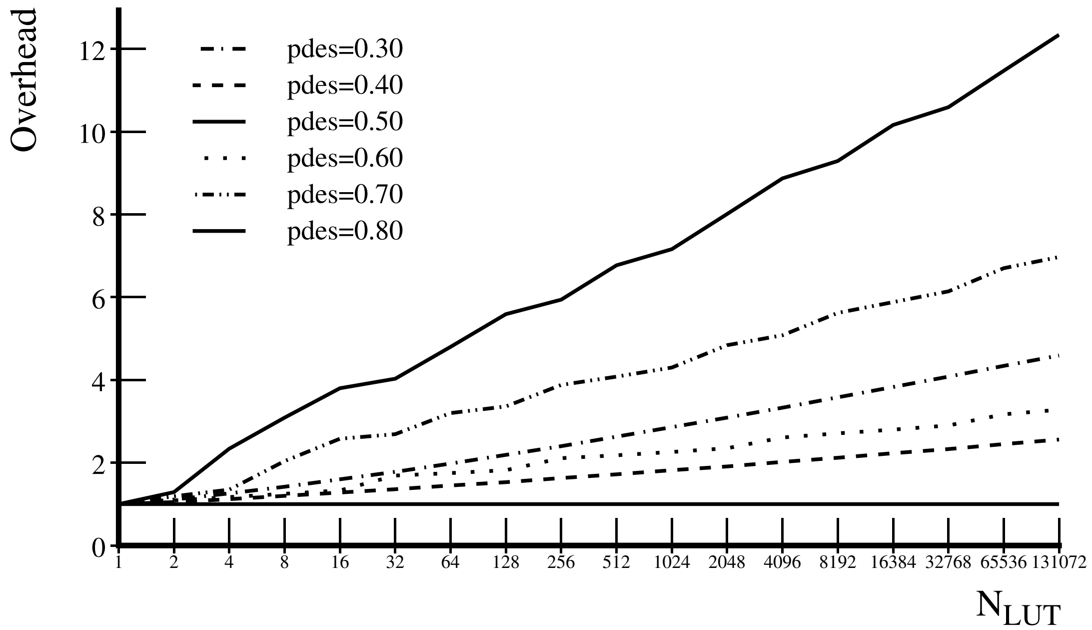
$$Kn^{lp_{design}} \leq K(Cn)^{lp_{net}} \tag{7.13}$$

Since $p_{net} < p_{design}$, at level $l-1$, the connectivity required for the design will shrink faster than the network connectivity, so lower levels are satisfied by the same scale up factor which satisfies the top level in the design. The overhead ratio for the $p_{design} > p_{net}$ case, then, is the ratio of the size of a $C \cdot N_{LUT}$ interconnect with Rent exponent p_{net} compared to the size of an N_{LUT} interconnect with Rent exponent p_{design}.

In making this area comparison, we assume that switching area dominates non-switching area, and we approximate LUT area as proportional to the number of switches. From Section 7.1, we saw that this is true of conventional devices. In the previous section, we saw that switching requirements grow at least as fast as wires, and generally faster than non-switching resources. This suggests that switching area will continue to dominate non-switching area as device capacities grow.

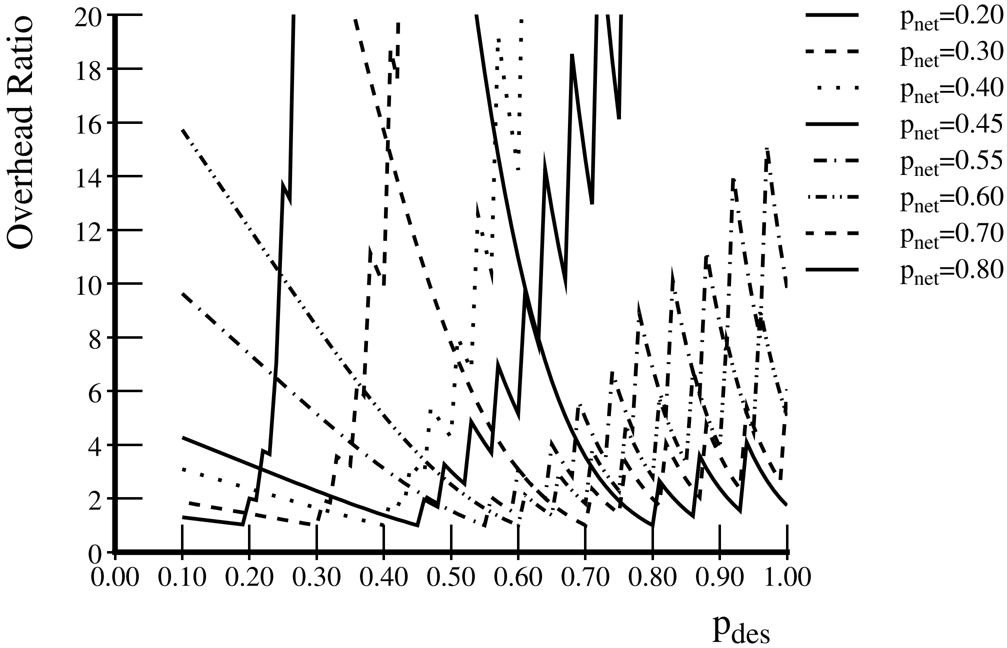
If we solve for C strictly according to Equation 7.13, the ratios are continuous and do not take into account the discretization affects associated with network size and levels. The continuous approximation gives us a smooth way to compare general overhead growth trends. Figure 7.7 shows both the discrete and continuous comparisons for various $p_{designs}$ implemented on a network with $p_{net} = 0.5$ as a function of N_{LUT}. Figure 7.8 similarly shows the relative overheads for implementing p_{des} designs with $N_{LUT} = 4096$ on p_{net} designs. Figure 7.9 plots the same data as the continuous case from Figure 7.8 on three axes. Figure 7.10 plots the continuous efficiency, the inverse of overhead, and Figure 7.11 plots the continuous efficiency on a logarithmic scale.

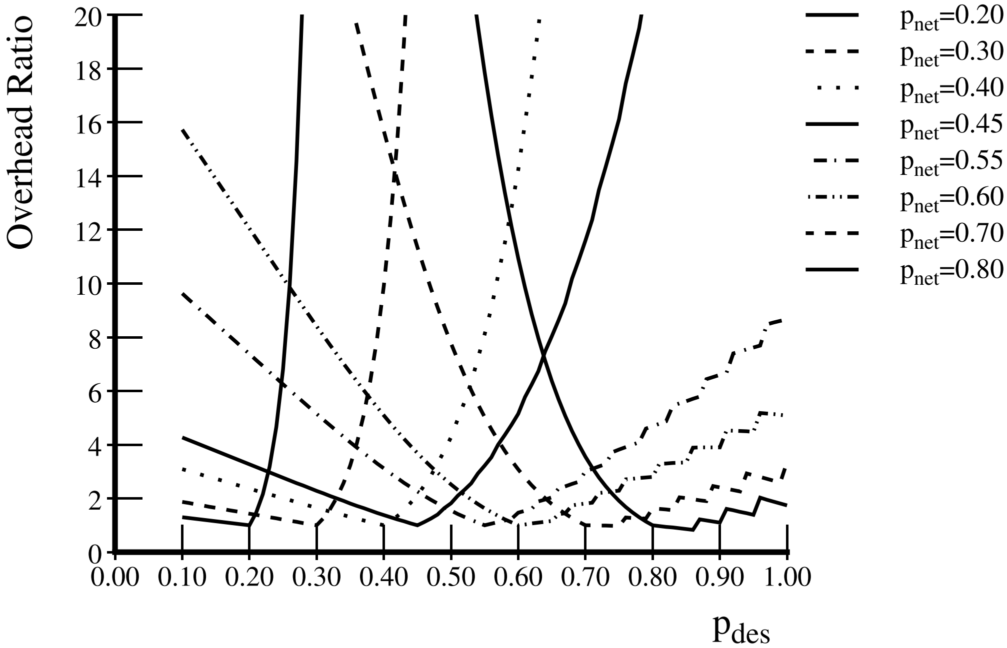




$k = 4$, $n = 2$, $p_{net} = 0.5$
Top - discretized ratios; Bottom - continuous ratios

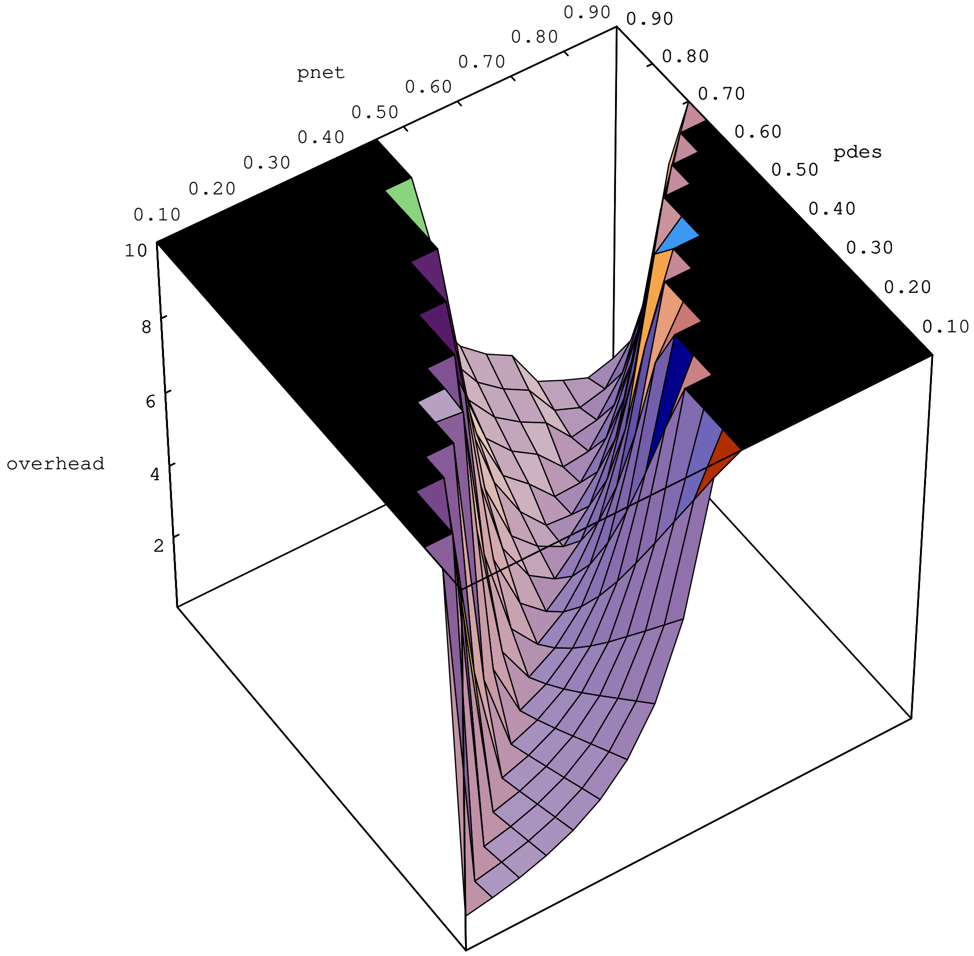
Figure 7.7: Overhead Growth versus N_{LUT} for various p_{net}





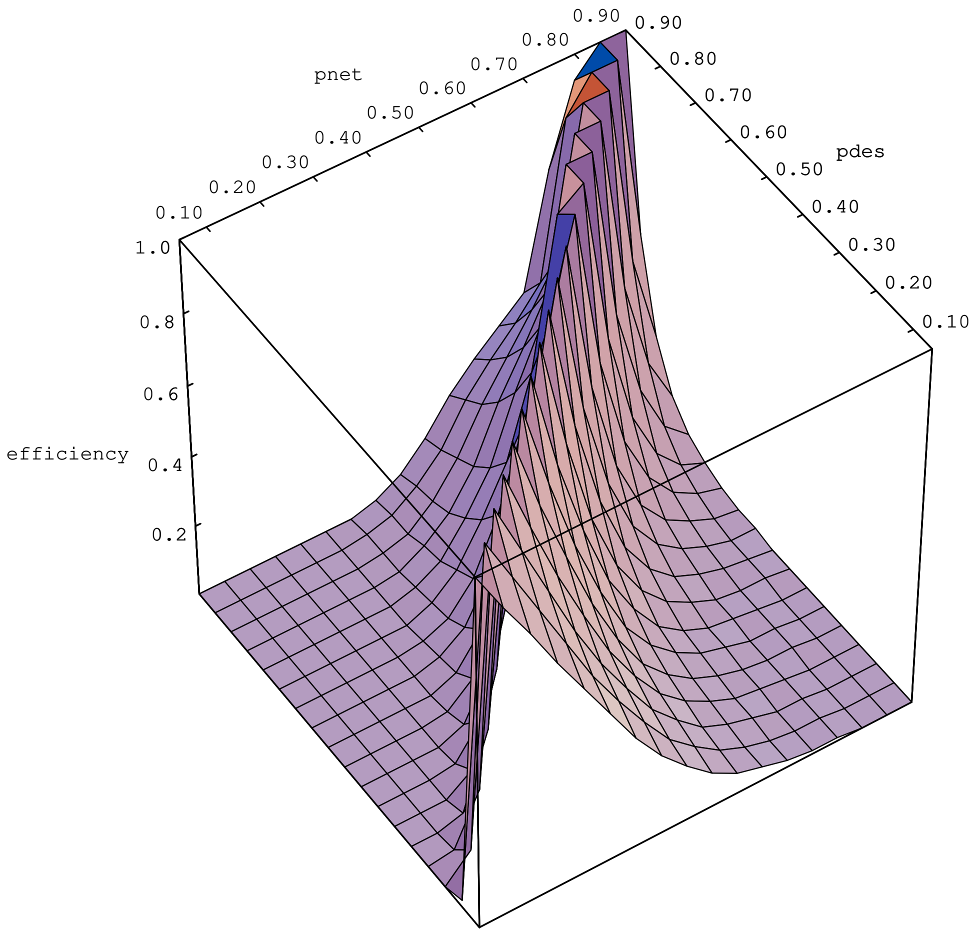
$k = 4$, $n = 2$, $N_{LUT} = 4096$
Top - discretized ratios; Bottom - continuous ratios

Figure 7.8: Overhead for p_{des} versus p_{net}



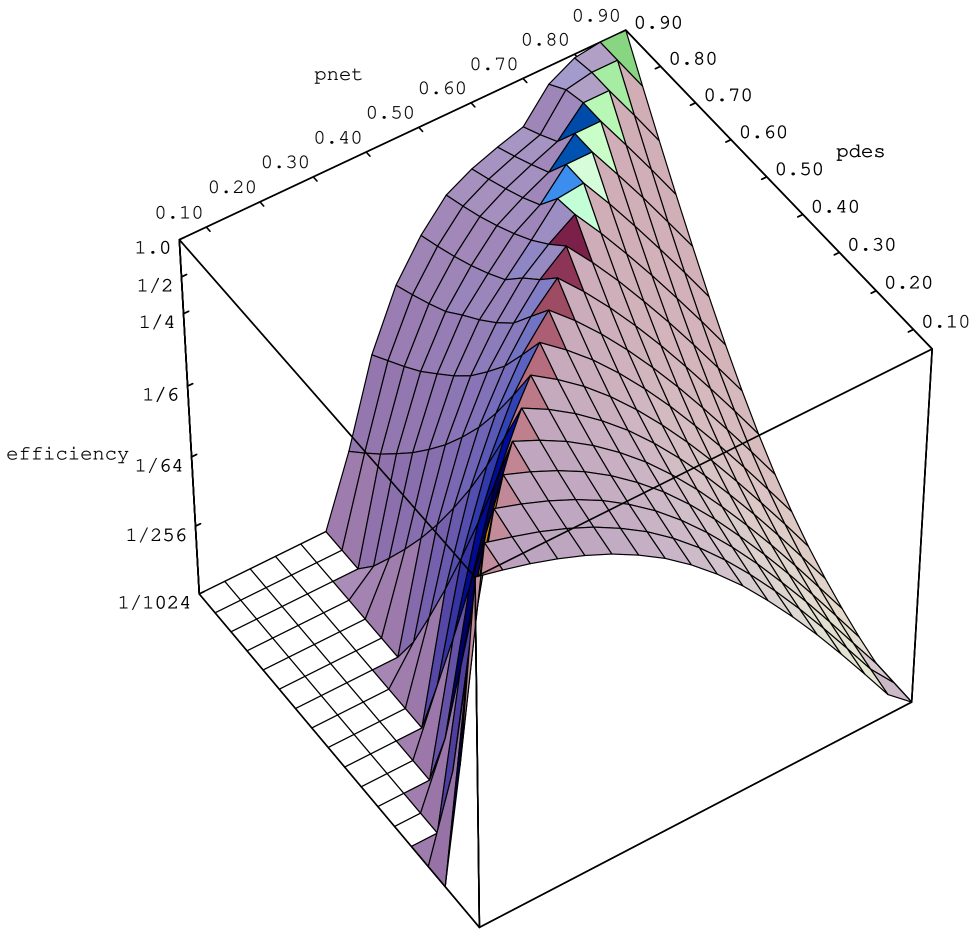
$k = 4$, $n = 2$, $N_{LUT} = 4096$

Figure 7.9: Continuous Overhead for p_{des} versus p_{net}



$k = 4$, $n = 2$, $N_{LUT} = 4096$

Figure 7.10: Continuous Efficiency for p_{des} versus p_{net}



$k = 4$, $n = 2$, $N_{LUT} = 4096$

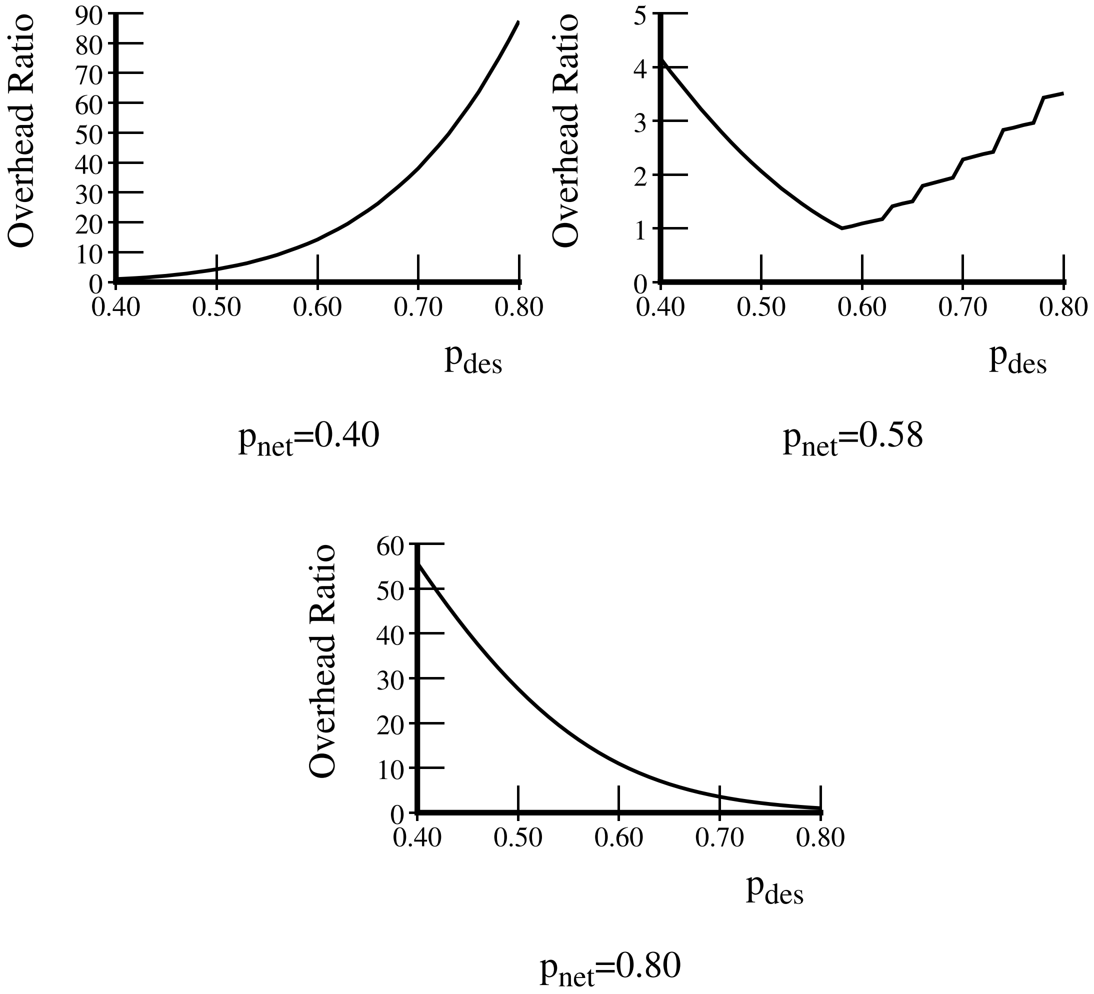
Figure 7.11: Continuous Efficiency for p_{des} versus p_{net} (Log Scale)

Ideally, we would like to match the programmable network connectivity to the design connectivity. Unfortunately, we do not generally get that choice. Figures 7.7 through 7.11 show us that it is just as inefficient to provide too much interconnect for a design as it is to provide too little. *This is important to notice, since there is a tendency to demand rich interconnect that provides high gate utilization across all designs. However, since the non-interconnect area is trivial compared to network area in FPGA devices, optimizing for gate utilization is often short sighted.*

As a final, illustrative example, let us consider the task of picking the network connectivity, p_{net}, assuming that we know typical designs will have a p_{design} between 0.4 and 0.8. Figure 7.12 shows the overheads for p_{net} values of 0.4, 0.58, and 0.8 as a function of p_{design}, respectively. If we further assume that the design Rent exponents are evenly distributed in this range, we can calculate an expected overhead:

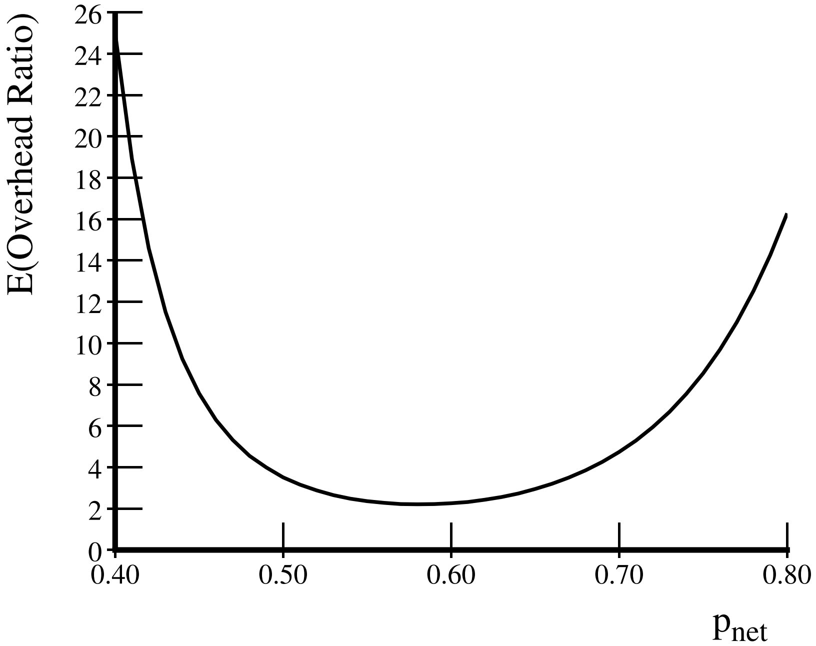
$$E\left(\text{overhead}(p_{net})\right) = \int_{0.4}^{0.8} \text{overhead}\left(p_{design}, p_{net}\right) dp_{design}$$

Figure 7.13 plots this expected overhead for the identified range. We see that the expected overhead is quite flat between p_{net}=0.5 and 0.6 with an expected overhead of just over $2\times$. At the ends of the spectrum, the expected overhead is $8\times$ worse. Note, in particular, if we chose to build $p_{net} = 0.8$ in order to guarantee full utilization of every LUT, we would pay a $16\times$ overhead on average, and a $56\times$ overhead in the worst case. In contrast, choosing $p_{net} = 0.58$ has a worst-case overhead of $4.2\times$ and an average overhead of $2.2\times$.



$k = 4$, $n = 2$, $N_{LUT} = 4096$

Figure 7.12: Sample p_{des} versus p_{net} Overheads



$k = 4$, $n = 2$, $N_{LUT} = 4096$

Figure 7.13: E(overhead) versus p_{net} for Uniform p_{des} Distribution

7.8 Interconnect Description

We can also ask how the requirements for interconnect description will grow. Trivially, we know that it will grow no faster than the number of switches composing the interconnect. However, it can actually grow much slower. We start (Section 7.8.1) by using the full-connectivity model of the crossbar to establish an upper bound on the necessary interconnect description length. We then continue (Section 7.8.2) using the Rent's rule based hierarchical interconnect, as in previous sections, to derive a tighter approximation. By either metric, we see that the instruction sizes for conventional FPGAs are significantly larger than necessary. This observation suggests that context memory area and reload instruction bandwidth can be significantly reduced by judicious coding (Section 7.8.3).

7.8.1 Weak Upper Bound

Assuming that the network may be arbitrarily connected, we can count the number of possible interconnection patterns to get an upper bound on the number of interconnection bits which can be usefully employed describing the input to each LUT. We start by assuming we have a device composed of:

- N_{LUT} k-input lookup tables
- n_{in} inputs to the network (from the chip i/o)
- n_{out} outputs from the network (chip outputs and enables)

Each LUT input can come from any of the other LUT outputs (N_{LUT}) or any of the n_{in} inputs. We can encode the source selection for a single input:

$$n_{LUT\_input\_bits} = \log_2(N_{LUT} + n_{in}) \tag{7.14}$$

Since a LUT has k inputs, the total number of interconnect bits needed is simply:

$$\begin{aligned} n_{LUT\_interconnect\_bits} &= k \times n_{LUT\_input\_bits} \\ n_{LUT\_interconnect\_bits} &= \lceil k \times \log_2(N_{LUT} + n_{in}) \rceil \end{aligned} \tag{7.15}$$

e.g. A 1000 4-LUT device with 200 inputs would require only 41 bits (44 if we encode each input separately) to specify each LUT's interconnect. A 9000 4-LUT device with 600 inputs requires only 53 bits (56 for separate input encodings).

If our functional elements are truly 4-LUTs, then this upper bound can be tightend by noticing that we gain no additional functionality by being able to route a particular source into the LUT multiple times and the assignment of the sources to LUT inputs is inconsequential. With this observation, we really only need to choose k items from $N_{LUT} + n_{in}$ when specifying the LUT interconnect. This gives:

$$n_{LUT\_input\_bits} = \left\lceil \begin{pmatrix} N_{LUT} + n_{in} \\ k \end{pmatrix} \right\rceil \tag{7.16}$$

e.g. our 1000 4-LUT device with 200 inputs requires only 37 bits and our 9000 4-LUT device with 600 inputs requires only 49 bits. Here we save an additional 4 bits per 4-LUT. Asymptotically:

$$\begin{aligned}
\lim_{n\to\infty} n_{bits_saved} &= \lim_{n\to\infty} \log_2\left(\frac{n^k}{\left(\begin{array}{c} n \\ k \end{array}\right)}\right) \\
&= \lim_{n\to\infty} \log_2\left(\frac{n^k}{\left(\frac{n\cdot(n-1)\cdot(n-2)\cdot(n-3)}{k!}\right)}\right) \\
n &\approx n-1 \approx n-2 \approx n-3 \\
n &>> k \\
\lim_{n\to\infty} n_{bits_saved} &\approx \log_2\left(\frac{n^k}{\left(\frac{n^k}{k!}\right)}\right) \\
&\approx \log_2(k!) \qquad (7.17)
\end{aligned}$$

So, we expect that exploiting the equivalence of the k inputs on a k-LUT to save us $\log_2(k!)$ bits from the number of bits required for full interconnect. For $k = 4$, this amounts to a savings of 4-5 bits per LUT.

Commercial devices are not purely composed up of LUTs, but we can draw a box around their basic programming elements and use the above counting arguments to get a loose upper bound on the number of interconnect programming bits they could require. Table 7.4 shows parameters for each of several commercial device families along with a pedagogical reference. Using the parameters given in Table 7.4, we can use the full connectivity assumption to compute an upper bound on the network description length:

$$n_{interconnect\_bits} \leq \lceil n_{b\_ins} \cdot \log_2\left(n_{max\_b} \cdot n_{b\_outs} + n_{max\_io} \cdot n_{io\_outs}\right)\rceil \qquad (7.18)$$

Table 7.5 calculates $N_{interconnect\_bits}$ for each of the device families from Table 7.4 and contrasts these numbers with the number of actual device bits per basic element. The comparison is necessarily crude since vendors do not provide detailed information on their configuration streams. However, we expect the unaccounted control bits in Table 7.5 to not be more than 10% of the total bits per block. With this expectation, we see that the commercial devices exhibit a factor of two to three more interconnect configuration bits than would be required to provide full, placement-independent, interconnect of the logic blocks.

7.8.2 Structure Based-Estimates

The upper bound derived in the previous section assumed full connectivity of the network. However, the network is generally much more restricted. The restrictions imply a smaller class of realizable connection patterns and fewer requisite interconnect bits. In this section we return to our pedagogical, hierarchical interconnect from Section 7.6. For small Rent exponents, p, we can derive tighter bounds.

Family		n_{b_ins}	n_{b_outs}	n_{io_ins}	n_{io_outs}	$n_{b\_logic\_bits}$	n_{max_b}	n_{max_io}
Xilinx 2K	CLB	4	2	2	1	16	100	74
Xilinx 3K	CLB	9	2	2	2	32	484	176
Xilinx 4K	CLB	13	4	4	2	40	1024	256
Xilinx 5K	CLB	16	8	2	1	64	484	244
Altera 8K	LE	4	1	1	1	16	1296	208
Orca 2C	PFU	19	6	3	1	64	900	480
UTFPGA	Tile	11	4	2	1	48	$700^{\dagger}$	$256^{\dagger}$
LEGO	Tile	15	4	$2^{\dagger}$	$1^{\dagger}$	64	$500^{\dagger}$	$256^{\dagger}$
DPGA	4-LUT	4	1	2	1	16	144	48
Reference	4-LUT	4	1	2	1	16	$2000^{\dagger}$	$256^{\dagger}$

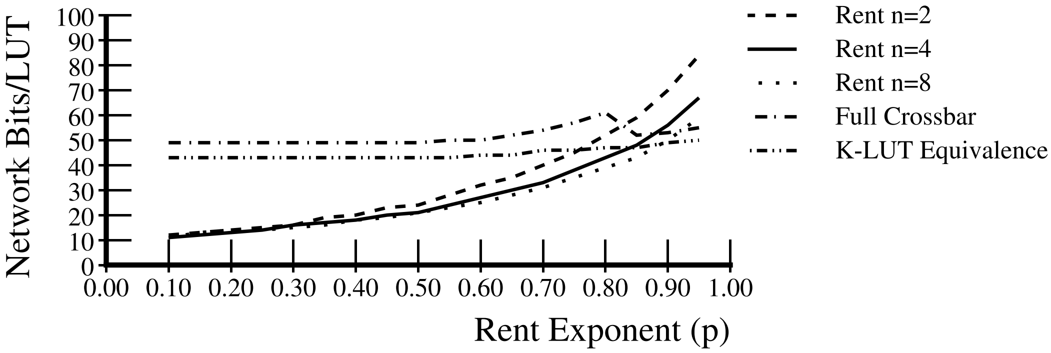
† - assumed value

n_{b_ins}	number of inputs per basic logic element
n_{b_outs}	number of outputs per basic logic element
$n_{b\_logic\_bits}$	number of bits specifying the logic function per basic element
n_{io_ins}	number of inputs from each i/o element
n_{io_outs}	number of outputs to each i/o element
n_{max_b}	number of blocks in largest member of family
n_{max_io}	number of i/o elements in largest member of family

Table 7.4: Parameters for a Sampling of Contemporary Programmable Devices

Part	Logic Bits	Net Bits	Actual Bits (approximate)
Xilinx 2K	16	33	160
Xilinx 3K	16	94	190
Xilinx 4K	40	159	420
Xilinx 5K	64	193	510
Altera 8K	16	43	190
Orca 2C	64	247	480
UTFPGA	48	128	146
LEGO	64	167	492
DPGA	16	31	40
Pedagogical Reference	16	45	–

Table 7.5: Configuration Bits – Requirement Upper Bound v/s Actual



N_{LUT}=4096

Figure 7.14: Network Bits per LUT v/s Rent Exponent for $N_{LUT} = 4096$ (K=4)

Reconsidering, the hierarchical interconnect structure from Figures 7.3 and 7.4, we can calculate the number of bits required per level of the hierarchy.

$$nb_{up_l} = \left(\log_2 \left(\begin{array}{c} ((n-1)\, n_{out_{l-1}} + n_{in_l}) \\ n_{in_{l-1}} \end{array} \right) - \frac{1}{n} \log_2(n!) \right) \cdot \frac{1}{n_{l-1}} \tag{7.19}$$

$$nb_{down_l} = \left(\log_2 \left(\begin{array}{c} n \cdot n_{in_{l-1}} \\ n_{out_l} \end{array} \right) \right) \cdot \frac{1}{n_l} \tag{7.20}$$

Table 7.6 summarizes these values by level for $p = \frac{2}{3}$, along with the number of bits according to the earlier crossbar and K-LUT equivalence calculations. This scheme also gives only an upper bound since the individual treatment of the permutations within each level counts more distinct combinations than actually exist. For moderate values of p, though, this will give a tighter bound than the crossbar bound derived in the previous section.

The number of bits required will vary with the Rent exponent p. Figure 7.14 shows this variation. It also shows the relationship among choices of the arity of the hierarchy, n, and the crossbar and K-LUT equivalence bounds. Figure 7.15 shows the growth rate versus the number of LUTs for several Rent exponents and the two crossbar bounds.

Note that Donath performs a similar calculation in [Don74]. He uses a more restrictive interconnect model. Using 2- to 3-input, single-function gates, he calculates 7-10 bits of memory per $p = 0.5$ to 0.8. In Donath's model, the required description bits does not grow with network size.

7.8.3 Significance and Impact

Resources for instruction storage and distribution, as the next Chapter (Chapter 8) will address, can take up significant area and play a big role in the characteristics of an architecture. Notably, the size of the instruction determines the size of the instruction store on and off chip and the bandwidth required to load new instructions.

N_{LUT}	n_{in}	n_{out}	nb_{xbar}	nb_{klut}	Δnb_{up}	Δnb_{down}	(integer) Δnb_{rent}	(integer) nb_{rent}	nb_{rent}
1	4	1	0	0	0.0	0.0	0	0	0.00
2	7	2	12	6	0.0	5.6	6	6	5.63
4	11	4	15	10	0.0	5.1	6	12	10.75
8	17	8	18	13	0.0	4.5	5	17	15.23
16	26	16	21	16	0.0	3.8	4	21	19.06
32	41	32	24	20	0.0	3.3	4	25	22.37
64	65	64	28	23	0.0	2.9	3	28	25.22
128	102	102	38	27	0.7	2.4	4	32	28.36
256	162	162	40	31	0.6	1.9	3	35	30.87
512	257	257	43	34	0.5	1.6	3	38	32.89
1024	407	407	46	38	0.4	1.2	2	40	34.49
2048	646	646	49	41	0.3	1.0	2	42	35.76
4096	1025	1025	53	44	0.2	0.8	2	44	36.78
8192	1626	1626	56	48	0.2	0.6	1	45	37.58
16384	2581	2581	59	52	0.1	0.5	1	46	38.22

$p = 0.67$ (Rent Parameter); $k = 4$ (K-LUT); $n = 2$ (2-ary hierarchy)

N_{LUT}	Total number of LUTs in level
n_{in}	number of inputs to level
n_{out}	number of outputs from level
nb_{xbar}	number of bits per LUT for full crossbar interconnect
nb_{klut}	number of bits per LUT exploiting K-LUT input equivalence
Δnb_{up}	number of bits per LUT to describe up connections this level
Δnb_{down}	number of bits per LUT to describe down connections this level
Δnb_{rent} (int)	integer number of bits per LUT to describe this level interconnect
nb_{rent} (int)	bits per LUT to describe interconnect to this level with integer rounding
nb_{rent}	total number of bits per LUT to describe interconnect to this level

Table 7.6: 4-LUT in 2-ary Hierarchical Interconnect with $p = \frac{2}{3}$

The bounds we derived in the previous sections show that the instruction sizes in traditional FPGAs are higher than necessary, at least by a factor of 2-4$\times$. For single context devices as we have seen, instruction memory makes up only a small fraction of the area on a conventional FPGA. For this reason, these bloated instructions do not adversely affect FPGA cell area (See Figure 7.16). In fact, in wire limited regimes, they may help by localizing instruction bits to the values they control.

The most significant impact is on reconfiguration time. Smaller instructions mean we can reload instructions in less time, given the same bandwidth for instruction reload. Alternately, it means that correspondingly less resources can be dedicated to instruction distribution in order to achieve the same instruction reload time as the larger instructions.

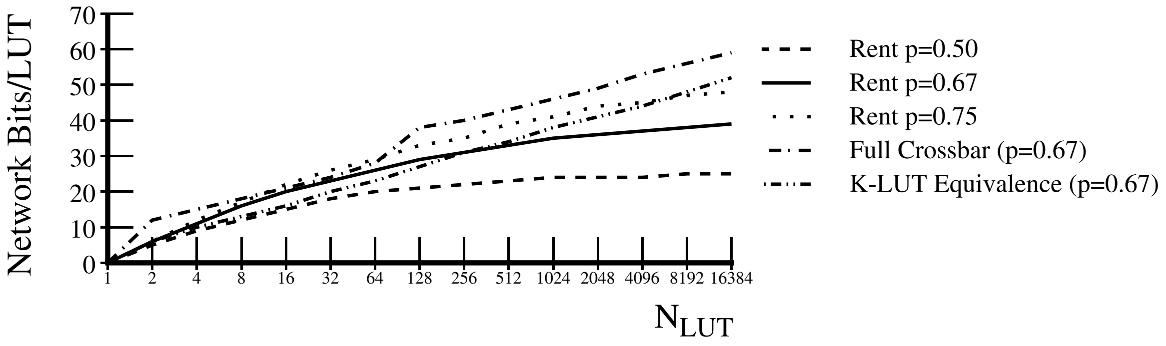


Figure 7.15: Network Bits per LUT v/s Number of LUTs for $n = 2$ (K=4)

As we begin to make heavy use of the reconfigurable aspects of programmable devices, device reconfiguration time becomes an important factor determining the performance provided by the part. In these rapid reuse scenarios, instruction size can play a significant role in determining device area and performance.

1. Off-chip context reloads for single- or multi-context devices are slow because a large amount of configuration data (typically, $> 10^5$ bits) must be transfered across a limited bandwidth i/o path. Reducing the size of the instructions transmitted across the i/o will improve reload performance.

2. One technique for reducing the reconfiguration time is to store multiple, on-chip contexts. When we start replicating the instructions associated with each LUT, the relative area consumed by instruction memory increases, making economy of instruction encoding more important (See Figure 7.17).

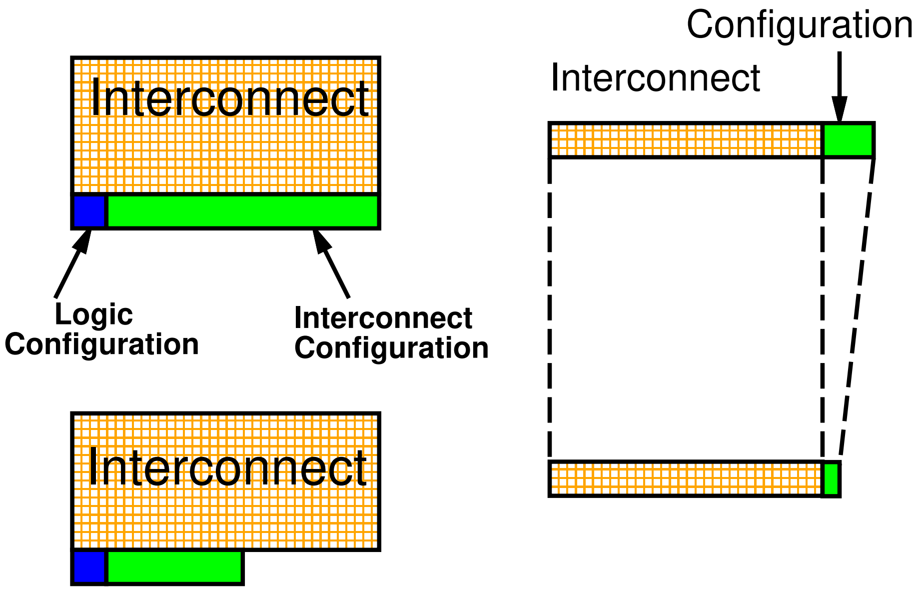
7.8.4 Instruction Growth versus Interconnect Growth

From the previous sections, we have seen that interconnect requirements grow faster than interconnect description requirements. Specifically:

- For $p > 0.5$, the number of switches and the amount of wiring grow as $O(N_{LUT}^{2p-1})$ per LUT.
- The number of interconnect configuration bits grows at most as $O(N_{LUT} \log(N_{LUT}))$.

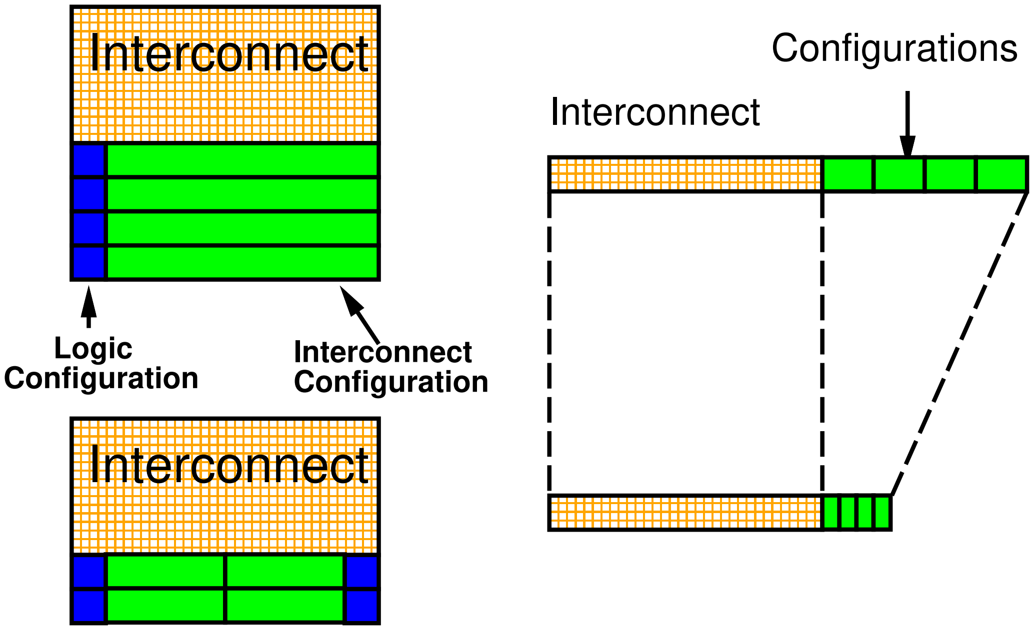
We already see that the switch and wire resources occupy a significantly larger fraction of the are per LUT than the interconnect description (Section 7.1.2). As N_{LUT} gets large, the size disparity will grow.

This is one reason that single context devices can afford to use sparse interconnect encodings. Since the wires and switches are the dominant and limiting resource, additional configuration bits are not costly. In the wire limited case, we may have free area under long routing channels for memory cells. In fact, dense encoding of the configuration space has the negative effect that control signals must be routed from the configuration memory cells to the switching points. The closer



For single context devices, the savings potential from denser interconnect description encodings is small – maybe 5-10%.

Figure 7.16: Single Context FPGA Area



For multicontext devices, the savings potential from denser interconnect description encodings can be large – up to 50-75% as the number of configurations get large.

Figure 7.17: Multicontext FPGA Area

we try to squeeze the bit stream encoding to its minimum, the less locality we have available between configuration bits and controlled switches. These control lines compete with network wiring, exacerbating the routing problems on a wire dominated layout.

7.9 Effects of Interconnect Granularity

So far, we have looked entirely at single-bit level granularity networks and designs. In this section, we look at how the multi-bit designs and networks effect the relations we have already developed.

In general, we will assume a w-bit datapath with N_{LUT} total bit processing elements. Groups of w bit processing elements will act as a single compute node. We thus have $N_{node} = \frac{N_{LUT}}{w}$ such compute nodes.

7.9.1 Wiring

We look at the wiring requirements, as we did before, by looking at the bisection bandwidth implied by the network. Assuming Rent's rule based hierarchical interconnect, at the top level we have $O(N_{node})$ i/o busses of width w. This makes for a total bisection bandwidth:

$$n_{bisect} = CN_{node}^{p} \times w = C\left(\frac{N_{LUT}}{w}\right)^{p} \times w = {N_{LUT}}^{p} w^{(1-p)}$$

This makes the wire dictated area growth go as:

$$A_{net} = O(n_{bisect}^{2}) = O\left(N_{LUT}^{2p} w^{2(1-p)}\right)$$

Per LUT this makes:

$$A_{LUT} = O\left(N_{LUT}^{(2p-1)} w^{2(1-p)}\right) \tag{7.21}$$

Notice that this is actually **larger** than the interconnect wiring area required for single-bit interconnects (Equation 7.4).

This result makes the assumption that the w bits composing a node are tightly interconnected or otherwise coupled such that the minimum bisection occurs between tree levels as before. If the w bits in a node were not interconnected in any way, the network could be decomposed into w single bit networks. In such a case, the size would simply be w times the size of an N_{node} single bit network, making:

$$\begin{aligned} A_{net} &= O\left(\left(\frac{N_{LUT}}{w}\right)^{2p} \times w\right) \\ A_{LUT} &= O\left(N_{LUT}^{(2p-1)} w^{(1-2p)}\right) \end{aligned} \tag{7.22}$$

Equation 7.22 implies the area is actually smaller than the single bit network for $p > 0.5$. In practice the earlier result (Equation 7.21) is most realistic.

One issue this raises is that a w-bit design with Rent exponent p implemented on top of a single bit network will require more interconnect per level than a single bit design with Rent exponent p. Using the same technique as in Section 7.7, we can solve for the required scale up factor:

$$\begin{aligned} KN_{node}^{p} w &= K\left(CN_{LUT}\right)^{p} \\ N_{LUT}^{p} w^{(1-p)} &= \left(CN_{LUT}\right)^{p} \\ w^{(1-p)} &= C^{p} \\ C &= w^{\left(\frac{1}{p}-1\right)} \end{aligned} \tag{7.23}$$

7.9.2 Switches

Switching requirements, in contrast, diminish with increasing w since less flexibility is required of the network with a given number of bit processing elements. We can derive the switching requirements by substituting N_{node} in for N_{LUT} in Equation 7.11 then multiplying by the datapath width:

$$n_{sw} \leq \begin{cases} K^2 \cdot (n - 1 + 2n^p) \cdot \left(\frac{n^{(1-2p)}}{n^{(1-2p)}-1}\right) & p < 0.5 \\ \log_n \left(\frac{N_{LUT}}{w}\right) \cdot K^2 \cdot (n - 1 + 2\sqrt{n}) & p = 0.5 \\ \left(N_{LUT}^{(2p-1)}\right) \cdot w^{(1-2p)} \cdot K^2 \cdot \left(\frac{1}{n^{(2p-1)}-1}\right) \cdot (n - 1 + 2n^p) & p > 0.5 \end{cases} \tag{7.24}$$

For large w, wiring requirements will asymptotically dominate switching requirements in a bussed interconnect scheme.

7.10 Summary

This section focussed on interconnect. We established via an empirical review that interconnect makes up the dominant area and delay in conventional FPGAs. We then went on to look at network design issues. We established basic relations governing the interconnect requirements in terms of network size and wiring complexity. In the processes we showed that it is not always best to build the network with sufficient interconnect to accommodate the most heavily interconnected designs at full gate utilization; rather, since interconnect is the dominant area contributor, more efficient area utilization can be achieved with networks with lower interconnect complexity. We also looked at interconnect description requirements, noting that interconnect descriptions grow more slowly than wire and switching requirements. Here, we pointed out that conventional devices use sparse interconnect description encodings, using, at least a factor of 2-4$\times$ more configuration bits than necessary; this observation suggests that we have an opportunity to reduce the area required to hold descriptions in multicontext devices and the bandwidth required for configuration reload in single or multicontext devices. Finally, we looked at how the interconnect size relationships change with wider word operations and saw that greater word widths increase wiring requirements while decreasing switching requirements.

8. Instructions

The need for instructions to control device operation is one distinguishing feature of general-purpose computing devices. These instructions give general-purpose devices their flexibility to solve a variety of problems. At the same time, instructions require dedicated resources for storage and delivery.

General-purpose computing architectures must address a number of important questions:

1. How are general-purpose processing resources controlled?
2. How much area is dedicated to holding the instructions which control these resources?
3. How many resources are controlled with each instruction?
4. How much bandwidth is provided for instruction distribution?
5. How frequently can instructions change?

There are many, different possible answers to these questions and the answers, in large part, distinguish the various general-purpose architecture categories which we reviewed in Chapter 4 (*e.g.* word-wide uniprocessor, SIMD, MIMD, VLIW, FPGA, reconfigurable ALU). The answers also play a large role in determining the efficiency with which the architecture can handle various applications.

In this chapter, we look at the problem of instruction control and the resources involved. We start by looking at the extreme case where every bit operation is given a unique instruction on every cycle. This example illustrates that instruction distribution resource requirements can be quite large – dominating other areas in a device. To combat these requirements, traditional architectures have placed various, stylized restrictions on instruction distribution in order to contain its resource requirements. Each of these restrictions also limits the realm of efficiency of the architecture. We review these restrictions and their effects on device utilization efficiency in Section 8.3. Of course, the opportunity to compress instruction distribution requirements depends on the inherent *compressibility* of the instruction stream suggesting that some computations will remain description limited while others are compute limited (Section 8.4). We also look at the issue of instruction stream control (Section 8.5). Finally, Section 8.6 organizes the architectural parameters reviewed in this chapter into an expanded taxonomy for multiple data processing architectures.

8.1 General Case Example

Consider that we have N_p bit processing elements. Each of these elements may be a 4-LUT as in the previous section or a one bit ALU. We want to provide a different instruction to each processing element on every cycle of operation. From Section 4, we see that 100 MHz+ operating frequencies are readily achievable today, with many devices already achieving higher frequencies. We will consider a 200 MHz operating frequency as one that will be easily achievable in the very near future. We saw from Section 7.8 that each 4-LUT needs 40-50 bits to describe its network configuration and 16 bits to control the logic function. We will thus assume each LUT requires a 64-bit instruction to control it.

Let us further assume that we distribute the instruction, one per clock cycle, from all four sides of the array of N_p processing elements densely using 2 layers of metal with an 8λ wire pitch. Of course, the assumption that we dedicate two full metal layers to instruction distribution is extreme, but even making this best case assumption, we will see that the resources required for full instruction control can dominate all other concerns. For the sake of easy comparison, we will target a $1024\lambda \times 1024\lambda = 1\text{M}\lambda^2$ array element, which is on par with large, conventional 4-LUT FPGA implementations (Table 7.1).

Across the width of one processing element, we can run instruction distribution busses to control two such elements:

$$2 \times \underbrace{64}_{\text{bits/LUT}} \times \underbrace{8\lambda}_{\text{wire pitch}} = 1024\lambda$$

This means, we can support an array with $2\times$ as many processing elements (N_p) as it has edge widths. That is:

$$2 \times 4 \times \sqrt{N_p} = N_p$$

From which we conclude $N_p = 64$.

At this point, we have fully saturated the i/o bandwidth into the compute array. Any further increase in the number of elements supported in the array must be accompanied by a corresponding increase in LUT size. That is:

$$\begin{aligned} \frac{\sqrt{A_{LUT}}}{64 \times 8\lambda} \times 4 \times \sqrt{N_p} &= N_p \\ \frac{\sqrt{A_{LUT}}}{128\lambda} &= \sqrt{N_p} \\ A_{LUT} &= 16384\lambda^2 N_p \end{aligned} \tag{8.1}$$

Consequently, LUT area increases linearly with the number of processing elements supported in the array. *The instruction distribution bandwidth requirement is the dominante size effect determining the density with which computational elements can be built.* Notice that the LUT area growth rate dictated by instruction bandwidth is larger than the interconnect growth rate for any value of $p < 1$ and, ultimately, both interconnect and instruction distribution compete for the same limited resource – wire bandwidth.

Further, we can calculate the actual instruction bandwidth requirements. At 200 MHz and 64-bit instructions, each LUT requires 1.6GBytes/s of instruction distribution. For the $N_p = 64$ case above, this amounts to over 100GBytes/s.

This kind of bandwidth could not, reasonably, be supported from off chip with any contemporary technology, necessitating on-chip instruction memory. At $1000\lambda^2$ per SRAM cell, 16 64-bit instructions will occupy the same space as each $1\text{M}\lambda^2$ processing element. If more than 16 unique instruction sets are required, instruction memory will occupy more area than the processing elements and interconnect.

8.2 Bits per Instruction

In the previous section we assumed 64 bits per instruction. This seems to be a reasonable, ballpark estimate of the number of bits required to describe a single bit operation, including interconnect.

Processors Modern processors generally employ 32 bits per instruction. However, as we saw in Section 4.1, about half of the instructions issued by a microprocessor are interconnect operations. Particularly, when we looked at gate evaluations, we saw that each processor instruction describes, on average, about 0.5-0.6 gate evaluations.

FPGAs Modern FPGAs use 120-200 bits per 4-LUT. We pointed out in the previous chapter that this was due to non-sparse encoding, and much denser encodings were possible. Simply using the crossbar bound from Section 7.8.1, we see that we can handle a 4000 4-LUT device with 48 bits of interconnect description and 16 bits of logic description. VEGA, a heavily multicontexted FPGA, with 85 bits per instruction, comes closer to this range [JL95].

8.3 Compressing Instruction Stream Requirements

Section 8.1 showed us that we cannot afford to have full, independent, cycle-by-cyle control of every bit operation without instruction storage and distribution requirements dominating all other resource requirements. Consequently, we generally search for application characteristics which allow us to describe the computation more compactly. In this section, we review the most common techniques generally employed to reduce instruction size and bandwidth. We see that every architecture reviewed in Chapter 4 exploits one or more of these compression techniques.

8.3.1 Wide Word Architectures

Processors do not, commonly, operate on single bit data items. Rather, sets of w bit elements ($w = \{8, 16, 32, 64\}$) are grouped together and controlled by a single instruction in SIMD style. This has the effect of reducing instruction bandwidth requirements and instruction storage requirements by a factor of w. This compression scheme takes advantage of the fact that we commonly do want to operate uniformly on multibit quantities. We can, therefore, effectively amortize instruction resources across multiple bit processing elements.

Returning to our opening example, we can support w^2 more processing elements before reaching the same point of wire saturation:

$$\begin{aligned} 2 \times 4 \times \sqrt{N_p} &= \frac{N_p}{w} \\ 8w &= \sqrt{N_p} \\ N_p &= 64w^2 \end{aligned}$$

The utilization efficiency of the resulting architecture depends on the extent to which all operations are w-bit operations, or even multiples thereof. When smaller operations, $w_d < w$, are required, $w - w_d$ bit processing units will sit idle while only w_d units provide useful work.

8.3.2 Broadcast Single Instruction to Multiple Compute Units

SIMD and vector machines take this instruction sharing one step further. They arrange so that multiple functional units operating on nominally different words share the same instruction. This allows them to scale up the number of bit operators without increasing the word granularity or instruction bandwidth. It does, however, increase the operation granularity. To remain efficient now, the application requires n_v w-bit operations, where n_v, is the number of word-wide datapaths controlled by each instruction.

8.3.3 Locally Configure Instruction

Reconfigurable architectures, such as FPGAs, take advantage of the fact that little instruction bandwidth is needed if the instructions do not change on every cycle. Each bit processing element gets its own, unique, instruction which is stored locally. However, this instruction cannot change from cycle to cycle. A limited bandwidth path is used to change array instructions when necessary.

There are two viewpoints from which to approach the efficiency of this restriction:

1. **Frequency of Instruction Change** – Given a lower bandwidth path, I_{BW}, and N_p compute elements with n_{ibits} long instructions, each context reload will take $\frac{n_{ibits} \cdot N_p}{I_{BW}}$. If the inter arrival time between reloads is n_{cycle}, the efficiency of operations is equal to the fraction of time spent computing versus total compute and reload time:

$$\text{Efficiency} = \frac{n_{cycle} \cdot t_{cycle}}{(n_{cycle} \cdot t_{cycle}) + \left(\frac{n_{ibits} \cdot N_p}{I_{BW}}\right)}$$

   This, of course, assumes that every processor is doing useful work on each cycle.

2. **Task Critical Path Length** – Alternately, if we assume the computing array is sufficiently large to perform the task, the efficiency is the fraction of compute elements performing useful work on each cycle. If the device has N_p processors and must perform a task with N_{bit_ops} which has a critical path of length L_{crit}, then the efficiency is the number of useful bit operations divided by the total number of processors and the critical path length:

$$\text{Efficiency} = \frac{N_{bit\_ops}}{L_{crit} \times N_p}$$

For the frequency of change case, *partial reconfiguration* can reduce the reload time by allowing individual processing units to change instructions without requiring an entire reload of all instructions in the array. This can increase reload efficiency if a large fraction of the instructions does not change. Partial reconfiguration can also allow portions of the array to change their instructions while other portions of the array continue to operate. Modern FPGAs from Plessey [Ple90], Atmel [Atm94], and Xilinx [Xil96] support partial reconfiguration for these reasons.

8.3.4 Broadcast Instruction Identifier, Lookup in Local Store

A hybrid form of instruction compression is to broadcast a single instruction identifier and lookup its meaning locally. This allows us to use a moderately short, single "instruction" across the entire array in a manner similar to SIMD instruction broadcast. Each processing element performs a local lookup from the broadcast instruction identifier to generate a full length instruction. DPGAs (Chapter 10), PADDI [CR92], and VLIW machines with an independent cache for each functional unit, exhibit this kind of hybrid control.

This technique is similar to a dictionary compression scheme where the set of entries at each "instruction" address makes up one element in the dictionary. The instruction address is the encoded symbol which can now be transmitted into the array with minimal bandwidth. The key benefit here is that the parallel instruction sets can be tailored to each application in the same way the dictionary can be tailored to a message or message type.

Efficiency in this scheme is similar to the task critical path length case above. The difference being, that each single processor need not be dedicated to a single instruction. With local instruction stores each holding n_{inst} instructions, an array of N_p processing nodes can perform up to $n_{inst} \cdot N_p$ different bit operations. In the single instruction configuration case, a critical path of $L_{crit} > 1$ implied a peak, achievable efficiency of $\frac{1}{L_{crit}}$. In this case, the peak, achievable efficiency is $\min\left(\frac{n_{inst}}{L_{crit}}, 1\right)$.

Viewed in terms of instruction change frequency, the additional local configurations can serve as a cache, diminishing the need to fetch array instruction from outside the array. Like a cache if each instruction set required is unique, it will provide no benefit. However, when an array instruction can be kept in the array and used several times before being replaced, we reduce the required instruction bandwidth. Use of a loaded instruction can occur at the operational cycle rate rather than at the bandwidth limited reload rate.

With multiple configurations it is possible to arrange for instruction reload to occur as a background task operating in parallel with operation. If the reload time, $\frac{n_{ibits} \cdot N_p}{I_{BW}}$, is less than the run time within the balance of the loaded instruction memory, $n_{cycle} \cdot t_{cycle}$, and the next instruction can be predicted sufficiently in advance, reload time can be completely overlapped with computation.

8.3.5 Encode Length by Likelihood

Since it is unlikely that all instructions will be used with equal frequency, one can break instructions into a series of smaller words, giving common instruction short encodings. If we huffman encode our instruction into a series of s-bit words, we can, potentially, reduce the instruction distribution bandwidth by a factor of $\frac{s}{\log_2(|\text{instructions}|)}$; that is, $\frac{s}{64}$ if we assume 64 bit instructions.

The efficiency now depends on the expected number of s-bit words required to construct a single, logical instruction. If the instruction stream entropy is low, this kind of encoding can be very efficient — asymptotically approaching one symbol per instruction. Counterwise, if the instruction stream entropy is high – or even flat, it may take a full $\frac{\log_2(|\text{instructions}|)}{s}$ cycles to built up a single instruction. Worse, if the instruction frequency is substantially different from the instruction frequency for which the encoding was optimized, it can actually take more cycles, on average, to build an instruction. Of course, the huffman encodings could be variable, as well, to avoid this mismatch, but the space required to handle programmable huffman encodings will likely exceed the area of several computational units.

8.3.6 Mode Bits for Early Bound information

All of the bits in an instruction do not always need to change at once — or portions of an instruction may change at different rates. Rather than include the infrequently changing portions of the instruction in the word which is broadcast from cycle to cycle, these portions can be factored out of the broadcast instruction and explicitly loaded with new values only when they need to change. This allows us to describe richer instructions with less bandwidth. These locally configured instructions can be seen as a special case of the previous section on likelihood encoding — but in this case we exploit the low frequency of change rather than simply the low frequency of occurrence of some instructions.

Mode bits such as these are used to define operational modes in several architectures. Floating point coprocessors often use mode bits to define rounding modes. Segmented SIMD architectures such as Abacus [BSV+95] and the dynamic computer groups of [KK79] use mode bits to define segmentation of the SIMD datapaths.

Bandwidth savings depends on the number of bits factored out of the broadcast instruction stream. Efficiency depends on the frequency with which non-broadcast instruction values need to change. Typically, it takes an instruction cycle to load each mode value – which is an instruction cycle which does not serve a purpose towards execution.

8.3.7 Themes

Two major themes emerge from the techniques listed here:

1. **Granularity** – How many resources are controlled by a each instruction? From a resource cost standpoint, this is the motivation behind word-wide datapaths, SIMD, and vector processors. processing

2. **Local Configuration Memory** – How many instructions are stored locally per active computing element? Similarly, this is the motivation behind configurable architectures and local memories.

In the next chapter, we will look effects which these techniques have both on resource requirements and on utilization efficiency.

8.4 Compressibility

Of course, we can only succeed in compressing the instruction bandwidth when there is structure to the task description for us to exploit. If the task descriptive complexity really is as large as implied in Section 8.1, we are instruction bandwidth limited, and instruction distribution does determine achievable, computational density.

This suggests we have two extremes in the characterization of computing tasks:

1. **Descriptive Complexity Limited** – the instruction bandwidth to describe the computation limits the rate of execution.

2. **Compute Limited** – the active computing elements performing the required computation limit the rate of execution.

Regular tasks such as signal and stream processing, systolic computations, and computational inner loops are typically compute limited. Irregular, run once, tasks such as initialization, cleanup, and exception handling are typically descriptive complexity limited. Of course, applications tend to have a mix of both elements. It has long been observed that only a small fraction of the code in a typical application accounts for most of the computational time [Knu71]. The regions composing this small fraction are heavily reused, allowing the computation to be described compactly. The code outside of the heavily used fraction, does not benefit from the heavy reuse amortization and will tend to be more description limited.

As with most compression schemes, the amount of compression achievable, in practice, also depends heavily on the frequency of repetition and storage space available. For example, if a task performs a sequence of one million, unique operations, then restarts the sequence, the stream is very repetitive, and an infinite sequence of such such repetitions contains a constant amount of information. However, unless we have space to hold all one million instructions on chip, we will not be able to take advantage of this regularity and low information content in order to compress instruction bandwidth requirements. Further, holding one million instructions on chip is a large cost to pay for instruction storage, even by today's standards.

8.5 Control Streams

In Sections 8.1 and 8.3, we viewed the set of processing elements as having a single, large, array-wide, instruction. In general, the array-wide instruction context may be decomposed into a number of independent instruction streams. This decomposition does not change the aggregate instruction bandwidth which may be required into the array, but it may change the number of distinct contexts used by the array and hence the requirements for instruction distribution and storage.

Let us assume, as in the case of Section 8.3.4, that each processing element has a local store of n_{inst} instructions. Let us also assume we have a series of m independent tasks, each composed of at most n_{task} instructions. The total number of distinct, array-wide contexts may be as large as $(n_{task})^m$, since the tasks are independent and any combination of instructions is possible. If each of the tasks is controlled separately, we need only $n_{inst} = n_{task}$ instructions to describe and control the tasks. If we must control the m tasks with a single instruction stream, that stream requires all $(n_{task})^m$ contexts and hence a larger number of instructions, $n_{inst} = (n_{task})^m$, are required.

This example demonstrates that there is a control granularity which is a distinct entity from the operation granularity introduced in Sections 8.3.1 and 8.3.2. As with operation granularity, we can compress instruction control requirements by sharing the control among a number of operating units. However, if we control too many units by the same control stream, we are forced to use the device inefficiently. In the worst case, we may pay an efficiency or compaction penalty in proportion to the product of the instruction sets of the independent operations which must be combined into a single control stream.

The separate streams of control are, of course, what distinguishes MIMD architectures (Section 4.9), as well as MSIMD (*e.g.* [Bri90, Nut77]) or MIMD multigauge [Sny85] architectures.

Control Threads (PCs)	Instructions per Control Thread	Instruction Depth	Granularity	Architecture/Examples
0	0	0	n/a	Hardwired Functional Unit (*e.g.* ECC/EDC Unit, FP MPY, Hardware Systolic)
	n	1	1	FPGA, Programmable Cellular Automata
			w	Reconfigurable ALUs Programmable Systolic Datapath Arrays
1	1	c	$n_v \cdot 1$	Bitwise SIMD
			w	Traditional Processors
			$n_v \cdot w$	Vector Processors
	n	c	1	DPGA
		8	16	PADDI
		c	w	VLIW
m	1	c	$n_v \cdot w$	MSIMD
m	1	c	1	VEGA
		8	16	PADDI-2
		c	w	MIMD (traditional)

Table 8.1: Instruction Control Taxonomy

8.6 Instruction Stream Taxonomy

Table 8.1 categorizes the various architectures we have reviewed in Chapter 4 according to the granularity (w,n_v), local instruction storage depth (c), number of distinct instructions per control thread (n), and number of control threads (m) supported. This taxonomy elaborates the multiple data portion of Flynn's classic architecture taxonomy [Fly66] by segregating instructions from control threads and adding granularity.

8.7 Summary

In this section, we have seen that the requirements for instruction distribution and storage can dominate all other resources on general-purpose computing devices, dictating the size and density of computing elements. A major distinguishing feature of modern, general-purpose architectures is the way in which they compress the requirements for instruction control. Traditional microprocessors, SIMD, and vector machines reduce the requirements by sharing a single instruction across many bits or words. FPGAs and programmable systolic arrays reduce requirements by maintaining the same instruction from cycle to cycle. VLIW-like architectures use small, local instruction stores addressed by short addresses so that limited instruction distribution bandwidth can effect cycle-by-cycle changes in non-uniform instructions. Each of the techniques used to reduce instruction control resources comes with its own limitations on achievable efficiency should the needs of the application not meet the stylized form in which the instruction bandwidth reduction is performed. Some instruction sequences are more compressible than others, suggesting we have a continuum of task descriptive complexities such that some tasks are, by nature, instruction bandwidth limited while others are parallel computing resource limited. In this chapter we reviewed both the nature of the resource reductions and the efficiency limits which arise from these techniques. In the following chapter, we will combine these effects with our size and growth observations from Chapters 4 and 7 to model the size and efficiency of reconfigurable computing architectures.

In this chapter, we put together the sizings from Chapter 4 and 7, the growth rates from Chapter 7, and the instruction requirements from Chapter 8 to form a unified area model for *RP*-space, a large class of reconfigurable processing architectures. The area model gives us a first order size estimate for reconfigurable computing devices based on the key parameters identified in the previous chapters. We use this model to estimate peak computational density as a function of granularity and on-chip instruction store sizes. We also use it to characterize the way computational efficiency decreases as application granularity and path lengths differ from the architecture's optimal points.

9.1 Model and Assumptions

We assume an array of homogeneous, general-purpose processing elements. For pedagogical purposes, no special-purpose processing units are included. The area for each bit processing element is taken to include:

- Fixed area for the computational function
- Amortized storage space for instructions
- Storage space for data
- Space for interconnect resources
- Amortized space for control

We compute the area per bit processing element as:

$$\begin{aligned} A_{bit\_elm} &= A_{fixed} + \underbrace{N_{SW}(N_p, w) \cdot A_{SW}}_{\text{interconnect}} + \underbrace{\left(\frac{c}{w}\right) \cdot n_{ibits} \cdot A_{mem\_cell}}_{\text{instruction memory}} + \\ &\quad \underbrace{d \cdot A_{mem\_cell}}_{\text{data memory}} + \underbrace{\left(\frac{N_{ctrl}}{N_p}\right) \cdot A_{ctrl}(c_{total})}_{\text{control area}} \end{aligned} \tag{9.1}$$

Table 9.1 summarizes the parameters used in Equation 9.1.

$A_{mem\_cell} = 1200\lambda^2$ is typical of static memory, which we will assume here. Memory cells packed into large arrays are likely to be denser, on average, than small arrays or isolated memory cells. Dynamic memory cells may be a factor of four smaller in large arrays, where appropriate.

Equation 9.1 assumes that interconnect area is proportional to the number of switches. In Sections 7.6 and 7.9, we saw that switch growth rates match or determine interconnect growth rate. In Section 7.9, we did see that wiring might dominate switch growth for large w, which is not accounted by Equation 9.1. $A_{SW} = 2500\lambda^2$ is a constant of proportionality intended to match the number of switches to the empirical interconnect areas typically seen rather than a model of any particular interconnect geometry. Table 9.2 summarizes the number of switches as a function of

Parameter	Role	Assumed Value
A_{bit_elm}	Area per bit processing element	
A_{fixed}	Fixed area per compute element (LUT mux, output flip-flop, buffers)	20Kλ^2
w	Datapath width – number of bit elements controlled by one instruction	
c	Contexts – number of instructions stored per group of w processing elements	
c_{total}	Total number of instruction or data contexts addressed by controller	
n_{ibits}	Number of bits in each instruction	64
A_{mem_cell}	Area of a configuration or data memory cell	1200λ^2
N_p	Number of bit processing elements in the array	
$N_{SW}(N_p, w)$	Number of switches per bit processing element	[Eq. 7.24]
n	Tree arity in modeled hierarchical interconnect	2
k	Number of LUT inputs	4
p	Rent parameter for network	0.5
A_{SW}	Amortized area of each switch	2500λ^2
d	Number of data bits per bit processing element	
N_{ctrl}	Number of independent stream controllers	
$A_{ctrl}(c_{total})$	Area of instruction stream controller	0.3M$\lambda^2 \cdot \log_2 (c_{total})$

Table 9.1: Summary of Area Model Parameters

N_p and w for $p = 0.5$, as will be used here. This is the same data which was plotted in Figure 7.5; for $p = 0.5$, the only difference is that we use $\frac{N_p}{w}$ as the network size when determining N_{SW} (See Equation 7.24).

For devices with multiple contexts, a controller manages the selection and sequencing of instructions in the array. The area we use for A_{ctrl} is a rough estimate based on a sampling of processor implementations (See Table 9.3). We assume that the area in the controller is proportional to the number of instruction address bits, $\log_2 (c_{total})$. FPGAs traditionally have a single context, making $N_{ctrl} = 0$, while processors have controllers composing the program counter and branching logic.

FPGA Example Traditional FPGAs have $w = 1$ and $c = 1$. Equation 9.1, for $N_p = 4096$, computes $A_{bit\_elm} \approx 870\text{K}\lambda^2$. Comparing with Table 7.1, we see this is in the range of conventional devices.

PADDI-2 Example PADDI-2 is made from 48, 16-bit units. Each has an 8 instruction memory ($c = c_{total} = 8$) and effectively 6 words of data per compute element, $d = 6$. PADDI-2 has

$\frac{N_p}{w}$	N_{SW}	$\frac{N_p}{w}$	N_{SW}	$\frac{N_p}{w}$	N_{SW}
1	0	32	100	1024	252
2	16	64	131	2048	281
4	31	128	162	4096	311
8	49	256	192	8192	340
16	69	512	222	16384	370

Table 9.2: $N_{SW}(N_p, w)$ for $p = 0.5$, $k = 4$, $n = 2$

Design	**Controller Area**
MIPS-X [HHC$^+$87, Cho89]	8Mλ^2
PA-RISC [YFJ$^+$87]	12Mλ^2
VIPER [GNAB93]	12Mλ^2

Table 9.3: Area for Instruction Control Sampling

3-inputs per EXU, $k = 3$, and an initial convergence of $n = 4$. Equation 9.1 predicts 370Kλ^2 per bit operation or 284Mλ^2 for the entire array, which is about half the size of the prototype PADDI-2 die which is 576Mλ^2.

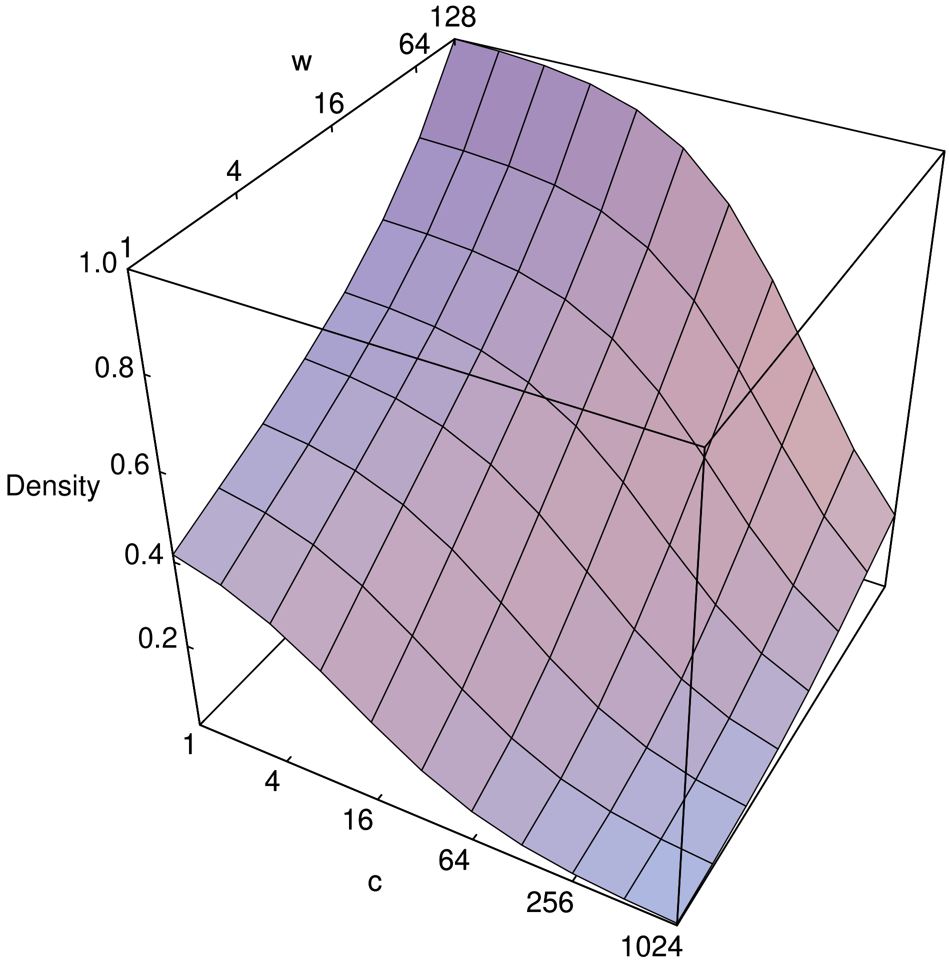
9.2 Peak Performance Density

Using the model, we can examine the peak computational densities from various architectural configurations in *RP*-space. Figure 9.1 plots computational density against datapath width, w, and the number of instructions per function group, c. As w increases there is more sharing of instruction memories and less switches required in the interconnect resulting in smaller bit processing element cell sizes or higher densities. As c increases, there are more instructions per compute element resulting in lower densities. The effect of more instructions is more severe for smaller datapath widths, w, since there are less processing elements against which to amortize instruction overhead.

For single context designs, there is only a factor of $2.5\times$ difference in density between single bit granularity and 128-bit granularity. At this size, network effects dominate instruction effects, and the factor of difference comes almost entirely from the difference in switching requirements. For heavily multicontext devices at the same number of instruction contexts, the difference between fine and coarse granularity is greater since the instruction memory area dominates (See also Figure 9.2). At 1024 contexts, the 128 bit datapath is $36\times$ denser than an array with bit-level granularity.

As the number of contexts, c, increase, the device is supporting more loaded instructions; that is, a larger on chip instruction diversity. Figure 9.2 shows how instruction density increases with increasing numbers of contexts alongside the decrease in peak computational density.

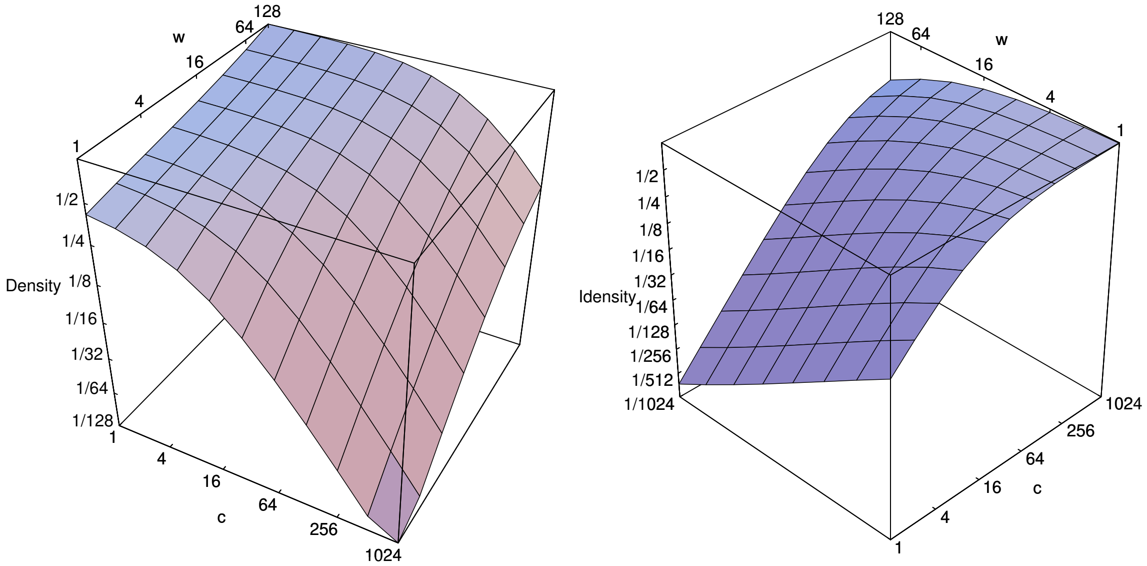
These same density trends hold if we set aside a fixed amount of data memory. The area outside of the data memory will follow the same density curves shown here.



$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Reference Density of 1.0 corresponds to $w = 128$, $c = 1$

Figure 9.1: Peak Computational Density Versus Contexts and Datapath Width



Left – Computational Density; Right – Instruction Density
$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.2: Compute and Instruction Densities Versus Contexts and Datapath Width

9.3 Granularity

As noted in the previous chapter, we can use larger granularity datapaths to reduce instruction overheads. The utility of this optimization depends heavily on the granularity of the data which needs to be processed. As noted in the previous section, the coarser the granularity the higher the peak performance. However, if the architectural granularity is larger than the task data granularity, portions of the device's computational power will go to waste.

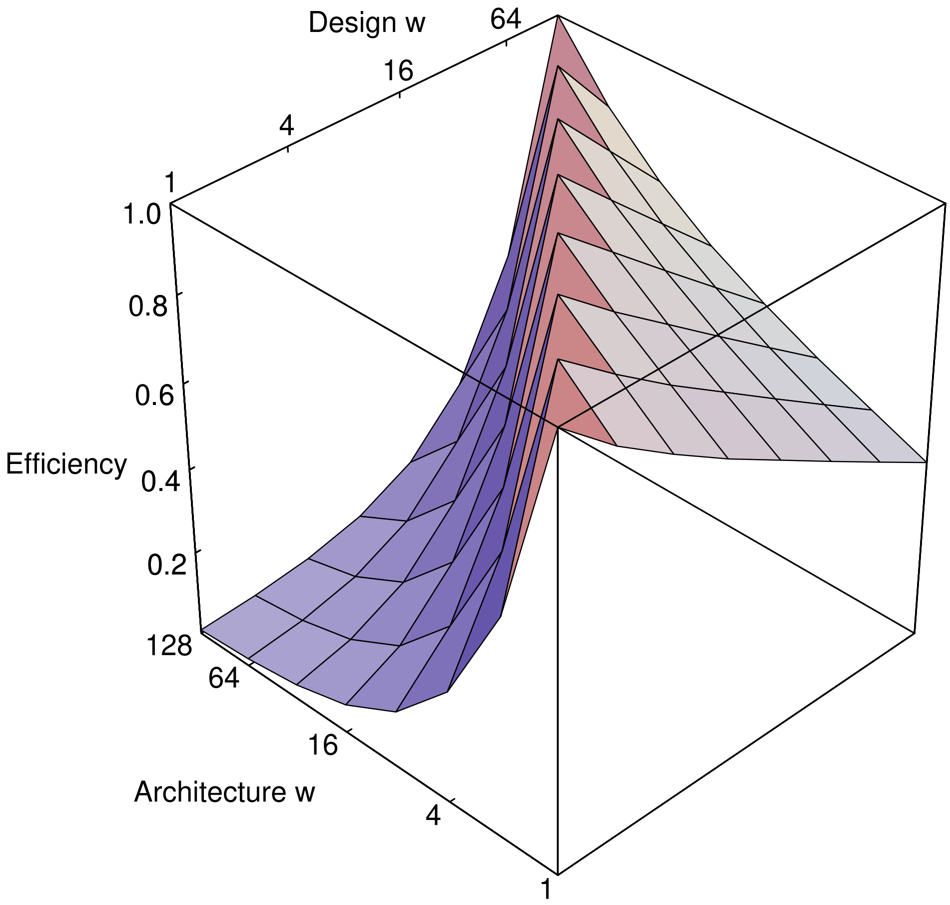
We can model the effects of pure granularity mismatches using the area model developed above. First, we note that the optimal configuration for a given word size will always be the architecture which has the same word size as the task. We can then determine the efficiency associated with running tasks with word size w_{des} on an architecture with word size w_{arch}, by dividing the area required to support the task on a w_{des} architecture by the area required on a w_{arch} architecture. For $w_{des} = C \cdot w_{arch}$, for some integer C, the efficiency is simply the ratio of the bit processing element areas. For $w_{des} > w_{arch}$, the task can run on top of the low w_{des} bit processing elements in the architecture datapath, leaving the remaining processing elements unused. The efficiency here is the ratio of the area of w_{des} bit processing elements from a w_{des} architecture versus w_{arch} bit processing elements from a w_{arch} architecture.

$$\text{Efficiency} = \begin{cases} \frac{A_{bit\_elm}|_{w=w_{des}}}{A_{bit\_elm}|_{w=w_{arch}}} & w_{des} = C \cdot w_{arch} \\ 1 & w_{des} = w_{arch} \\ \frac{w_{des} \cdot A_{bit\_elm}|_{w=w_{des}, N_p=N_{p0}}}{w_{arch} \cdot A_{bit\_elm}|_{w=w_{arch}, N_p=\left(\frac{w_{arch}}{w_{des}}\right) N_{p0}}} & C \cdot w_{des} = w_{arch} \end{cases} \tag{9.2}$$

Note that a single-chip implementation is assumed for comparison so that there are no boundary effects between components.

Figure 9.3 shows the efficiency for various architecture and task granularities. At $c = 1$, the active switching area dominates. The fine granularity ($w = 1$) has the most robust efficiency across task granularities. The efficiency drops off quickly for large grain architectures supporting fine grain tasks.

Figure 9.4 shows that the robustness shifts as the numbers of contexts increases. For $c = 1024$, the instruction memory space dominates the area. Consequently, the redundancy which arises when fine-grained architectures run coarse-grain tasks is quite large, leading to rapidly decreasing efficiency with increasing task grain size. In this regime, the coarse-grain architectures are more robust, since the extra datapath and networking elements are moderately inexpensive compared to the large area dedicated to instruction memory. For $c = 1024$, $w = 32$, is the most robust datapath width as shown extracted in Figure 9.5.



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1$, $N_{ctrl} = 0$, $N_p = 16384$

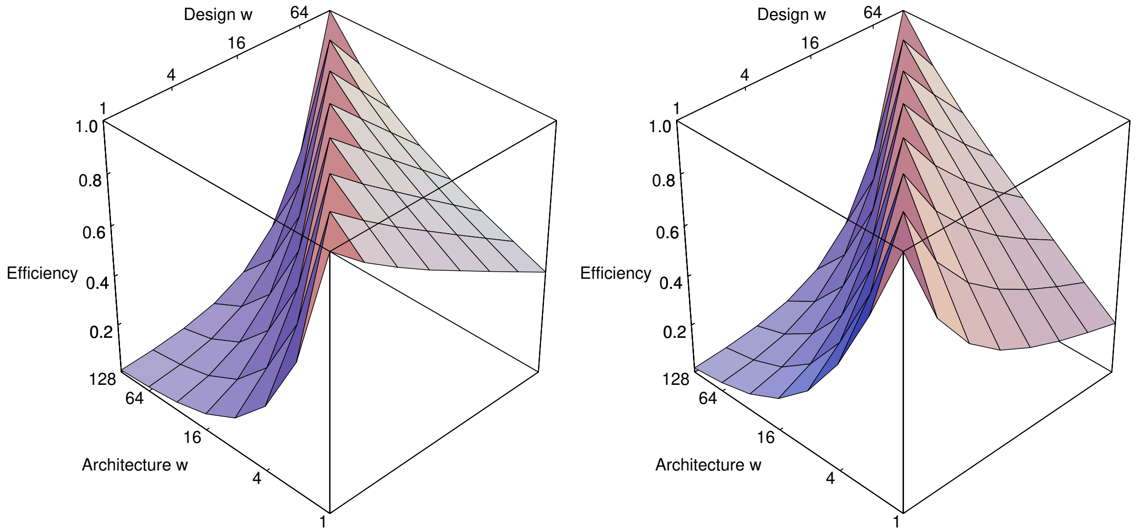
Figure 9.3: Efficiency as a Function of Architectural and Task Granularity for Single Context Architectures

These robust points correspond to the mix where the context memory makes up roughly half the area of the device.

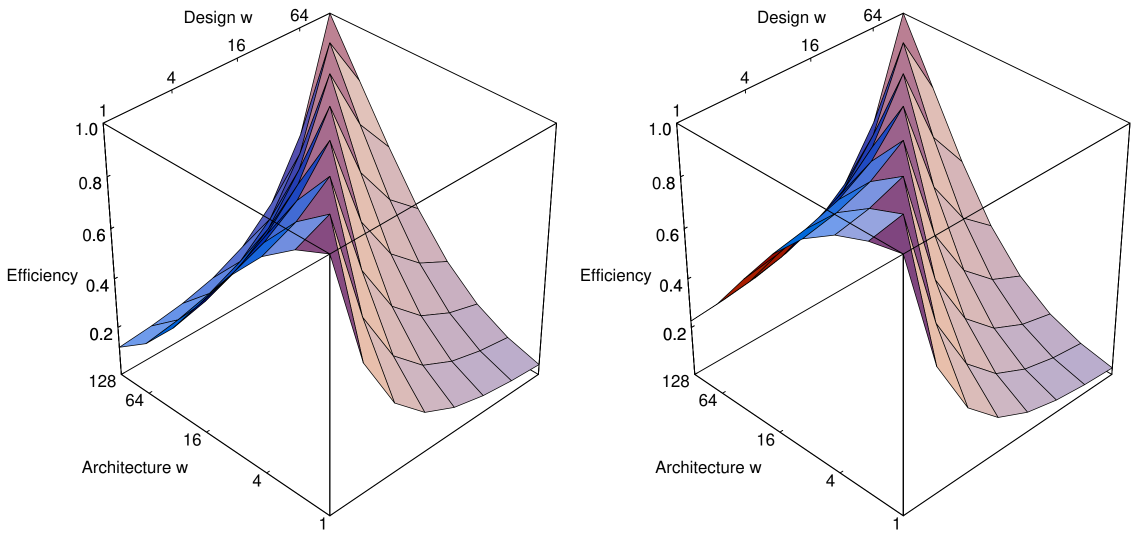
$$A_{bit\_elm}|_{c=0,w=w^*} = \left(\frac{1}{2}\right) A_{bit\_elm}|_{c=c,w=w^*} \tag{9.3}$$

At this point:

- Finer grain devices running coarser granularity tasks waste, at most, a little over half of their area – the memory area plus the switching overhead associated with finer granularity.
- Coarser grain devices running fine-grain tasks waste at most half of their area – the unused datapath area.



Left – $c = 1$; Right – $c = 16$



Left – $c = 256$; Right – $c = 1024$

$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.4: Efficiency as a Function of Architectural and Task Granularity

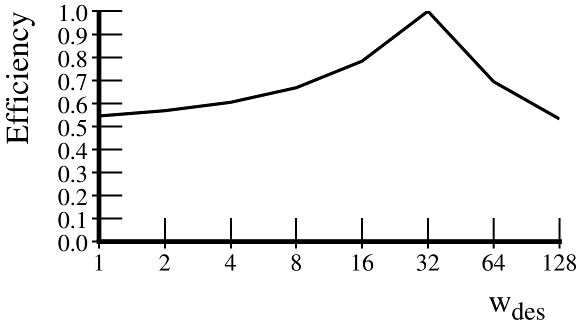


Figure 9.5: Efficiency versus Task Data Width for a 1024-context, 32-bit Granularity Device

9.4 Contexts

We saw in Section 9.2 that the computational density is heavily dependent on the number of instruction contexts supported. Architectures which support substantially more contexts than required by the application, allow a large amount of silicon area dedicated to instruction memory to go unused. Architectures which support too few contexts will leave active computing and switching resources idle waiting for the time when they are needed.

We can model the effects of varying application requirements and architectural support in an ideal setting using the area model. We assume we have a repetitive task requiring N_{op} operations which has a path length l_{path}. In an ideal packing, an architecture with $N_p = \frac{N_{op}}{l_{path}}$ processing units and $c = l_{path}$ instruction contexts can support the task optimally. If $c > l_{path}$, the area per processing element is larger than necessary to support the application. If $c < l_{path}$, it will be necessary to use more processing elements simply to hold the total set of instructions.

$$\text{Efficiency} = \begin{cases} \frac{A_{bit\_elm}|_{c=l_{path}}}{A_{bit\_elm}|_{c=c_{arch}}} & l_{path} < c_{arch} \\ 1 & l_{path} = c_{arch} \\ \frac{c_{arch} \cdot A_{bit\_elm}|_{c=l_{path}, N_p=N_{p0}}}{l_{path} \cdot A_{bit\_elm}|_{c=c_{arch}, N_p=\left(\frac{l_{path}}{c_{arch}}\right) N_{p0}}} & l_{path} > c_{arch} \end{cases} \tag{9.4}$$

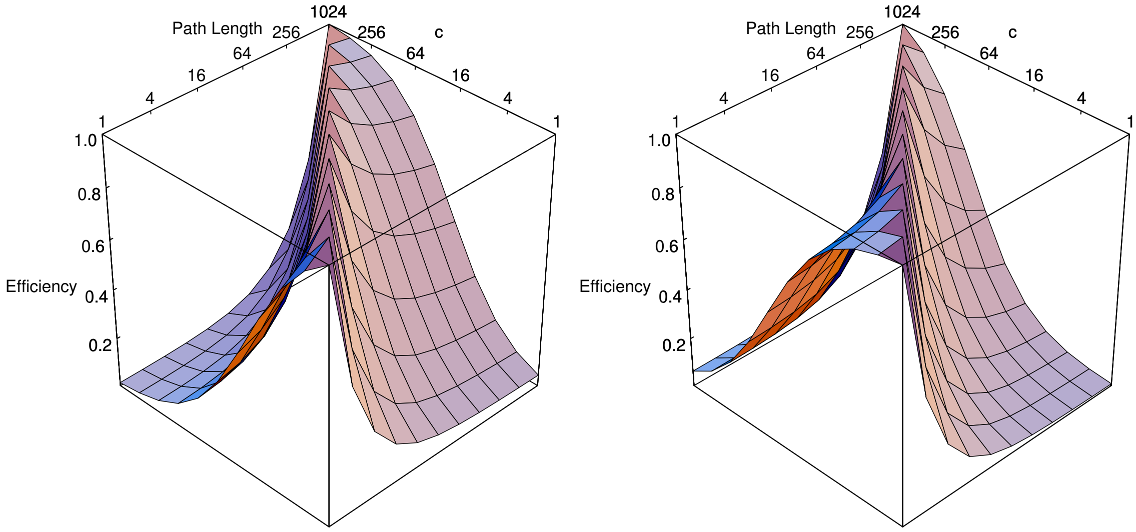
This relation is shown for several datapath widths, w, in Figure 9.6. Again, single chip implementations are assumed for comparison.

The efficiency dropoff for $l_{path} < c$ is less severe for large datapaths, large w, than for small datapaths. Similarly, the dropoff for $c < l_{path}$ is less severe for small datapaths than for large datapaths. This effect is due to the relative area contributed by instructions. In the small w case, the instruction area takes up relatively more area than in the large w case, so costs of extra active area is relatively smaller than in the large w case. In the large datapath case, the instructions make up a lower percentage of the area so the overhead for extra instructions is relatively smaller.

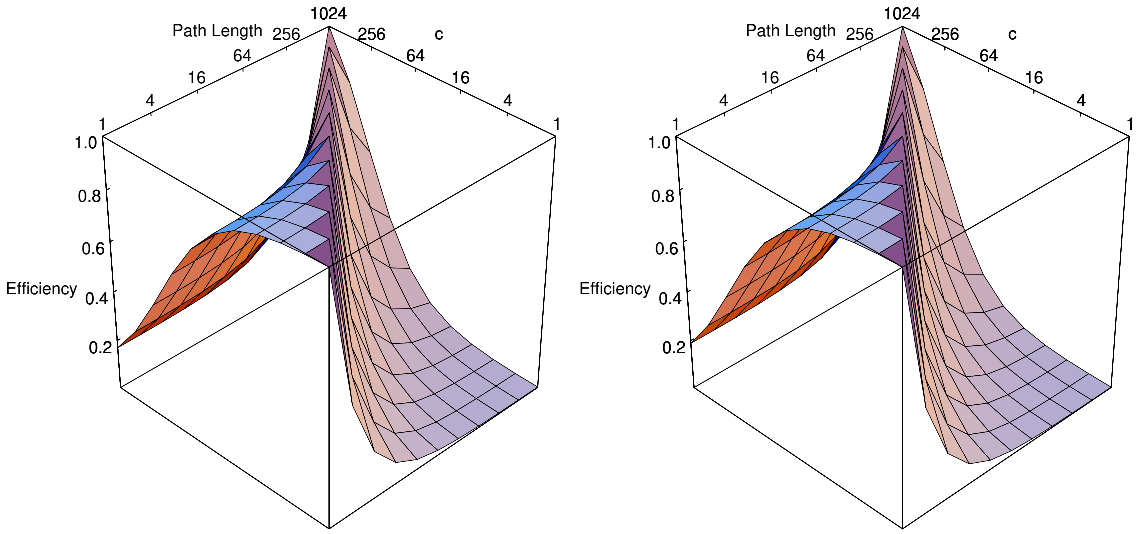
The 16 instruction context case is the most robust across this range for single bit datapaths (See Figure 9.7). Similarly, 256 instruction contexts is the most robust for $w = 128$ (See Figure 9.8). Neither of these cases drops much below 50% efficiency at either the $l_{path} < c$ or $c < l_{path}$ extremes. These "robust" cases correspond to the points where the instruction memory area is roughly equal to the active network and computing area. In either extreme, at most half of the resources are being underutilized. c^*, our robust context selection, can be defined as:

$$A_{bit\_elm}|_{c=0} \approx \left(\frac{1}{2}\right) A_{bit\_elm}|_{c=c^*} \tag{9.5}$$

Remember that the network resource requirements grow with array size. In the $c < l_{path}$ case, where we must deploy more processing elements to handle the task, the total number of processing elements increases causing the switching area per processing element to increase as well. This effects acounts for the fact that the efficiency can drop below 50% and the approximate relation in Equation 9.5.



Left – $w = 1$; Right – $w = 8$



Left – $w = 64$; Right – $w = 128$
$k = 4$, $n = 2$, $p = 0.5$, $c = d$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.6: Efficiency as a Function of Task Path Length and Architectural Contexts

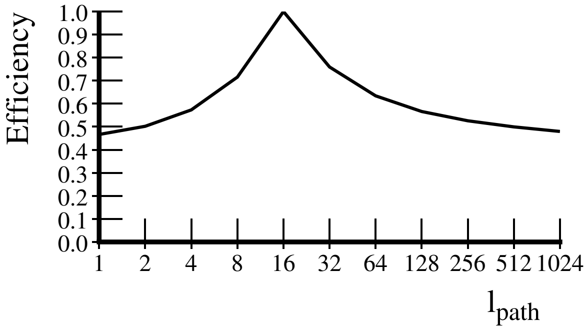


Figure 9.7: Efficiency versus Task Path Length for a 16-context, Single-bit Granularity Device

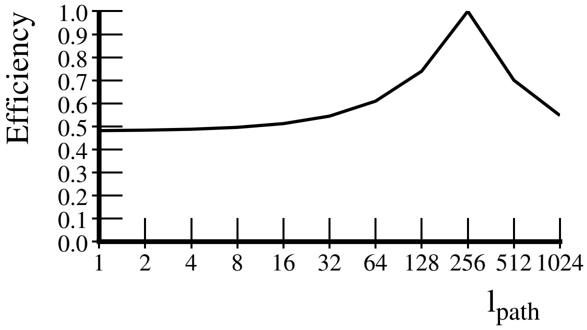


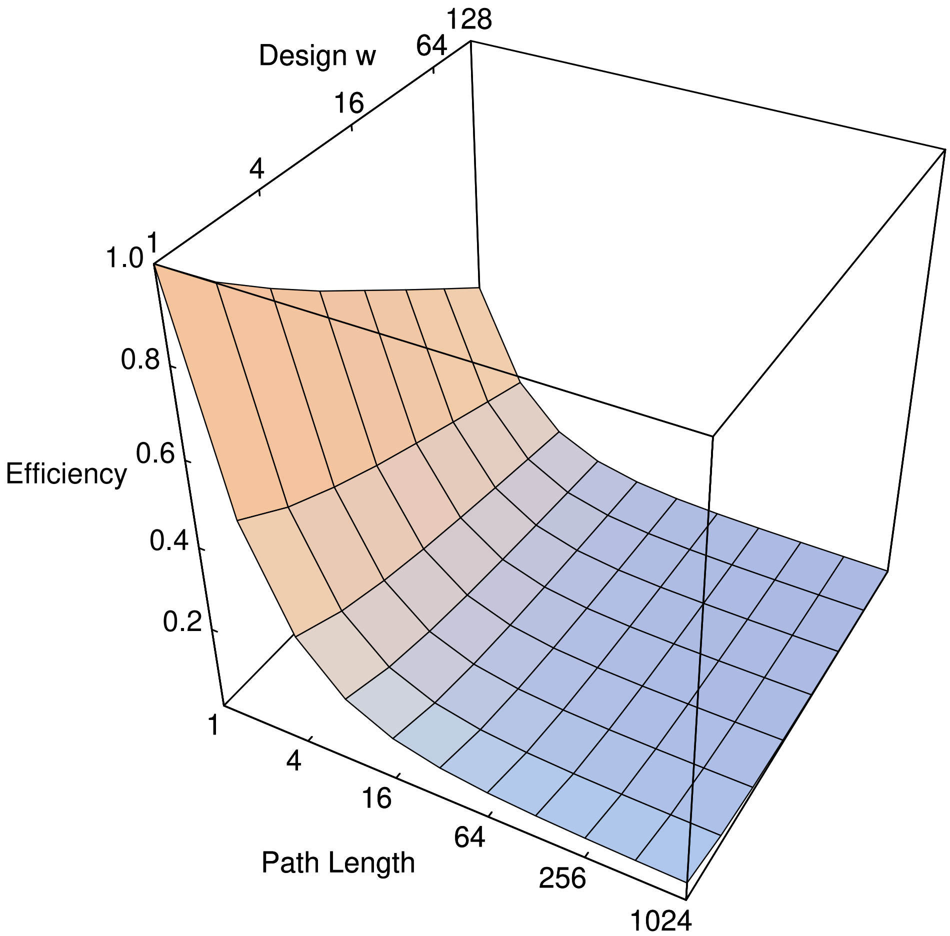
Figure 9.8: Efficiency versus Task Path Length for a 256-context, 128-bit Granularity Device

9.5 Composition

In general, we see cumulative effects of the grain size and context depth mismatches between architecture and task requirements. Figure 9.9 shows the yielded efficiency versus both application path length and grain size for the conventional FPGA design point of a single context and a single bit datapath. The FPGA drops to 1% efficiency for large datapaths with long path lengths. Similarly, Figure 9.10 shows the efficiency of a wide word ($w = 64$), deep memory ($c = 1024$) design point. While this does well for large path lengths and wide data, its efficiency at a path length and data size of one is 0.5%. *Notice here, that the wide, coarse-grain design point is over 100$\times$ less efficient than the FPGA when running tasks whose requirements match the FPGA, and the FPGA is 100$\times$ less efficient than said point when running tasks with coarse-grain data and deep path lengths.*

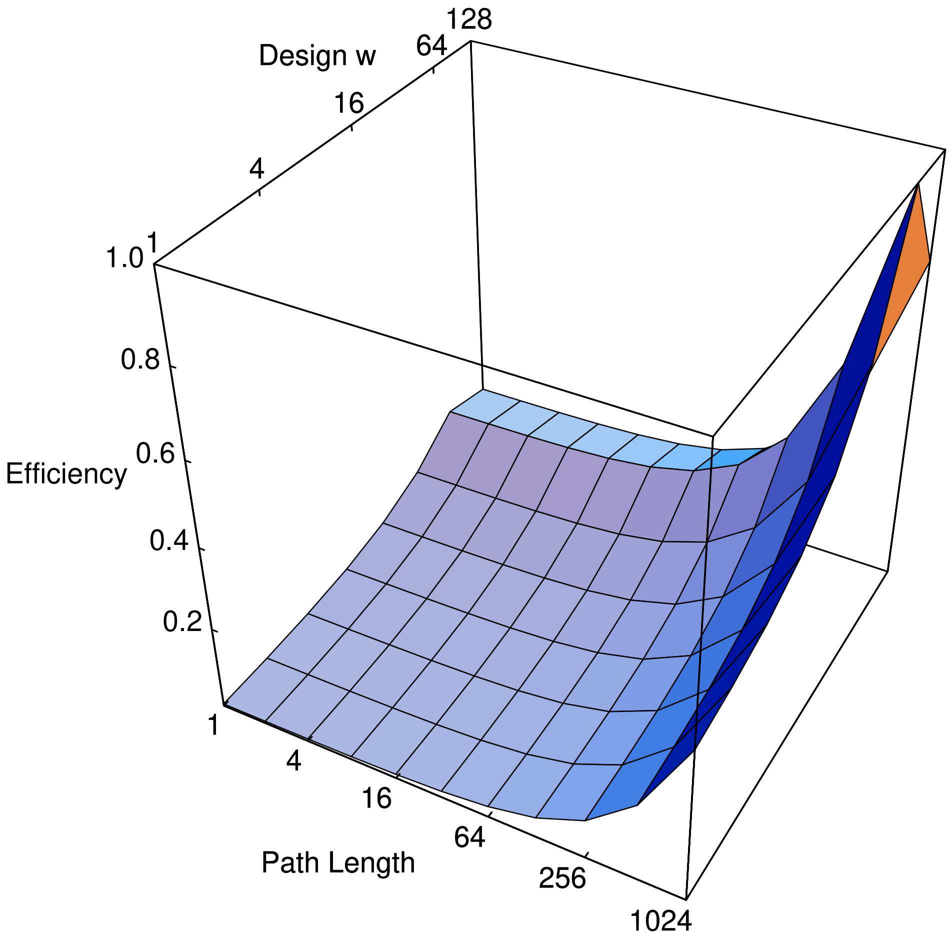
In the previous sections we saw that it was possible to select reasonably robust choices for datapath width or number of instruction contexts given that the other parameter was fixed. We also saw that the robustness criterion followed the same form; that is, the inefficiency overhead can be bounded near 50% if half of the area is dedicated to instruction memory and half to active computing resources. This does not, however, yield a single point optimum since the partitioning of the instructions between more contexts and finer-grain control is handled distinctly in the two cases.

Figure 9.11, for instance, shows the yield for a single design point, $w = 8$, $c = 64$, across varying task path lengths and datapath requirements. While the $w = 8$ and $c = 64$ cross-sections are moderately robust, the efficiencies at the extremas are low. At $l_{path} = 1$, $w = 1$, the efficiency is just under 8%, and at the $l_{path} = 1024$, $w = 128$, the efficiency is just over 8%. This design point is, nonetheless, more robust across the whole space than either of the architectures shown in Figures 9.9 and 9.10.



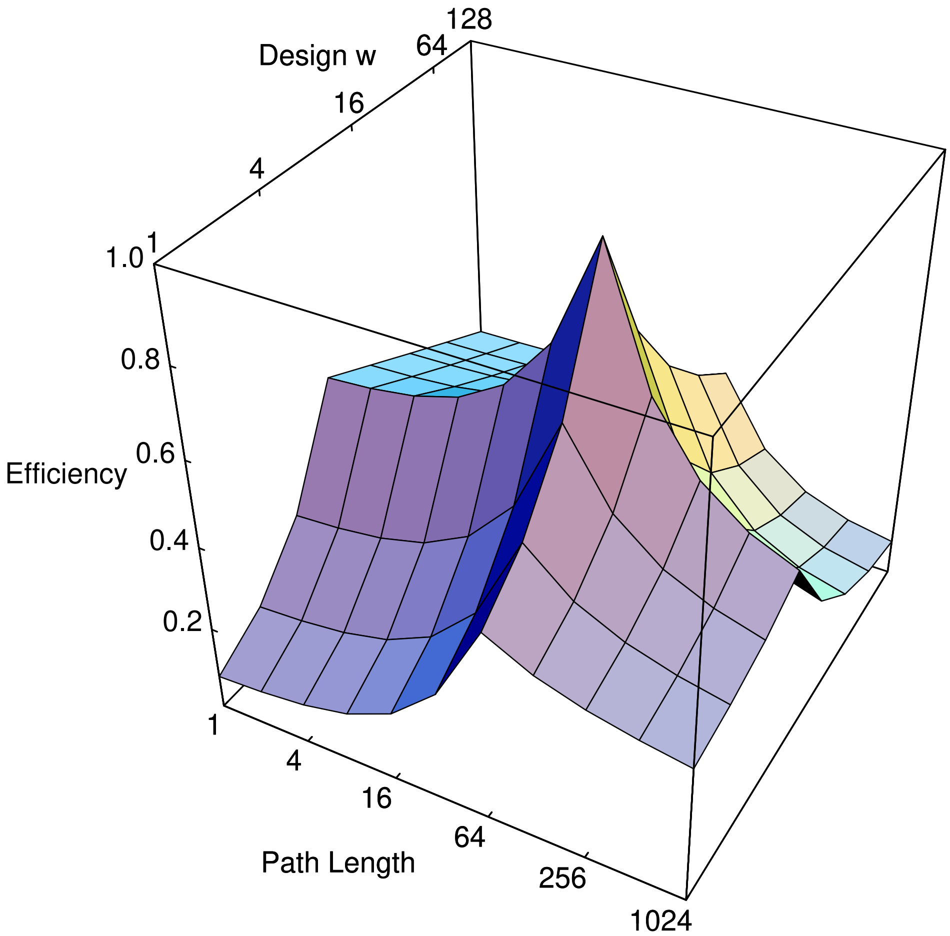
$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1$, $w = 1$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.9: Efficiency for Conventional FPGA Design Point ($w = 1$, $c = 1$)



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 1024$, $w = 64$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.10: Efficiency for Coarse-Grain, Deep Memory Design Point ($w = 64$, $c = 1024$)



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 64$, $w = 8$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 9.11: Efficiency for Fixed $w = 8$, $c = 64$

9.6 Summary

The area model shows us how peak capacity depends on granularity organization and instruction support. We see that the penalty for fine-granularity is moderate, 2.5$\times$ difference between $w = 1$ and $w = 128$, in the configurable domain where there is only instruction memory for a single context. The penalty is large, 36$\times$, in the heavy multicontext domain. We also looked at the effects of application granularity and path length. In both cases, we found that, given *a priori* knowledge of either the task granularity or context requirements, we could set the other parameter such that the efficiency did not drop significantly below 50% for any choice of the unknown parameter. This is significant since the peak performance densities across the range explored differed by roughly a factor of 200$\times$. For both of these cases, the robust selection criterion is to choose the free parameter such that instruction memory accounts for one half of the processing cell area. We saw that the effects of granularity and path length mismatches were cumulative and that FPGAs running tasks suited for deep memory, coarse-grained architectures can be only 1% efficient. If we must select both the datapath granularity and the number of contexts obliviously, we cannot obtain a single design point with as robust a behavior as when we only had one free parameter. A good design point across this region of the *RP*-space suffers a 13$\times$ worst-case overhead.

Part IV

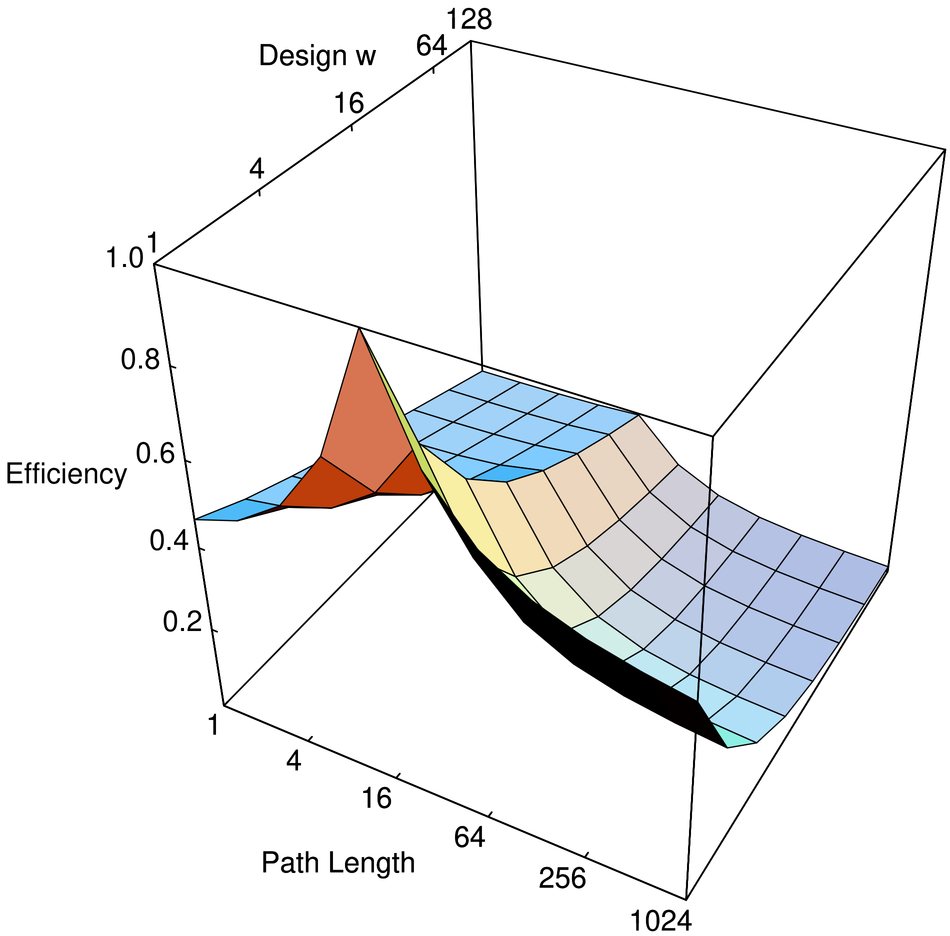
New Architectures

10. Dynamically Programmable Gate Arrays

In Chapter 9 we demonstrated that if we settle on a single word width, w, we can select a robust context depth, c^*, such that the area required to implement any task on the architecture with fixed $c = c^*$ is at most $2\times$ the area of using an architecture with optimal c. Further, for single bit granularities, $w = 1$, the model predicted a robust context depth $c^* = 16$. In contrast, the primary, conventional, general-purpose devices with independent, bit-level control over each bit-processing unit are Field-Programmable Gate Arrays (FPGAs), which have $c = 1$. Our analysis from Chapter 9 suggests that we can often realize more compact designs with multicontext devices. Figure 10.1 shows the yielded efficiency of a 16-context, single-bit granularity device for comparison with Figure 9.9, emphasizing the broader range of efficiency for these multicontext devices.

In this chapter, we introduce Dynamically Programmable Gate Arrays (DPGAs), fine-grained, multicontext devices which are often more area efficient than FPGAs. The chapter features:

- a characterization of where DPGAs are most area efficient and why
- a detailed prototype DPGA implementation
- design automation for two realms of DPGA application: (1) levelized circuit evaluation and (2) Finite-State Machine mapping
- an identification of major, pragmatic limitations to achieving the full benefits which look possible in theory



$k = 4$, $n = 2$, $p = 0.5$, $c = d = 16$, $w = 1$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 10.1: Efficiency for DPGA Design Point ($w = 1$, $c = 16$)

10.1 DPGA Introduction

The DPGA is a multicontext ($c > 1$), fine-grained ($w = 1$), computing device. Initially, we assume a single control stream ($N_{ctrl} = 1$). Each compute and interconnect resource has its own, small, memory for describing its behavior (See Figure 10.2). These instruction memories are read in parallel whenever a context (instruction) switch is indicated.

The DPGA exploits two facts:

1. The description of an operation is much smaller than the active area necessary to perform the operation.

2. It is seldom necessary to evaluate every gate or bit computation in a design simultaneously in order to achieve the desired task latency or throughput.

To illustrate the issue, consider the task of converting an ASCII Hex digit into binary. Figure 10.3 describes the basic computation required. Assuming we care about the latency of this operation, a mapping which minimizes the critical path length using SIS [SSL+92] and Chortle [Fra92] has a

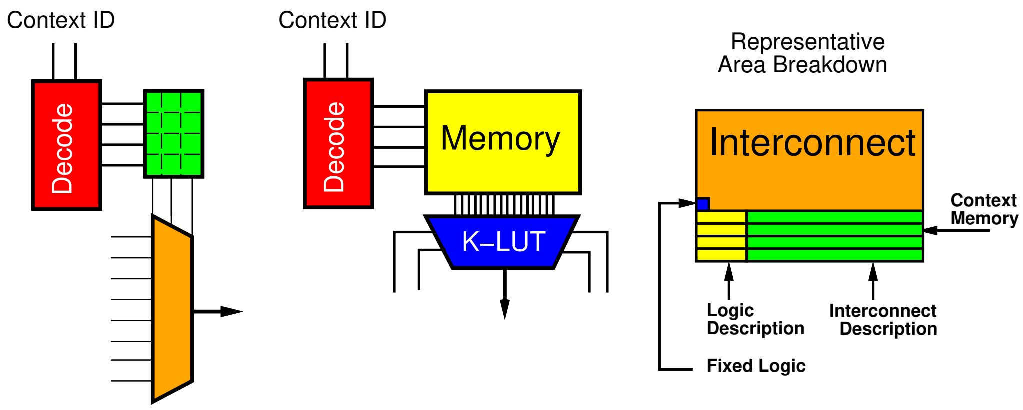


Figure 10.2: LUT and Interconnect Primitives for Multicontext FPGA

```
if (c >= 0x30 && c <= 0x39)
     res = c-0x30;
else if (c >= 0x40 && c <= 0x46)
     res = c - 0x40 + 10;
else if (c >= 0x60 && c <= 0x66)
     res = c - 0x60 + 10;
else
     res = 0;
```

Figure 10.3: ASCII Hex→Binary Task Description

```
INORDER = C[7] C[6] C[5] C[4] C[3] C[2] C[1] C[0] ;
OUTORDER = O[3] O[2] O[1] O[0] ;
# stage 1 – 8 LUTs [C[3:0] pass through]
i0 = !C[1] * !C[2] ;
i1 = C[4] * C[5] * !C[6] * !C[7] ;
i3 = C[0] * C[1] * !C[2] ;
i4 = !C[3] * !C[4] * C[6] * !C[7] ;
i6 = !C[0] * C[2] ;
i7 = !C[0] * C[1] ;
i8 = C[0] * !C[1] ;
i11 = !C[7] * C[6] * !C[4] * !C[3] ;
# stage 2 – 9 LUTs [i1,C[3],C[1] pass through]
i5 = i0 * i1  + i3 * i4 ;
i9 = i6 * i4 + i7 * i4 + i8 * i4 ;
i10 = C[3] + i3 * i4 ;
i12 = i3 * i4 + i6 * i4;
i13 =  i1 * !C[3]  * C[2] ;
i14 = C[2] * !C[1] * i11 ;
i15 = i8 * i4 + i7 * i4 ;
i16 = i7 * i4 + i6 * i4 ;
i17 = i1 * !C[3] * C[0] + C[0] * i0 * i1 ;
# stage 3 – 4 LUTs
O[3] = (i10+i9)*(i5+i9);
O[2] = i12 + i13 + i14 ;
O[1] = i1 * !C[3] * C[1]  + i15 ;
O[0] = i16 + i17 ;
```

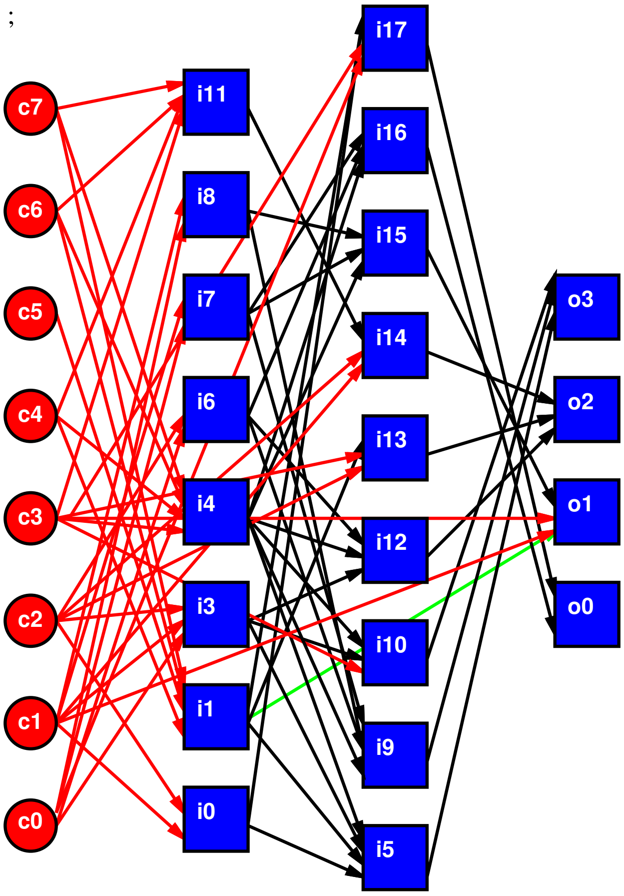


Figure 10.4: 4-LUT Mapping of ASCII Hex→Binary

path length of 3 and requires 21 4-LUTs. Figure 10.4 shows the LUT mapping both in equations and circuit topology.

Traditional Pipelining for Throughput If we cared only about achieving the highest throughput, we would fully pipeline this implementation such that it took in a new character on each cycle and output its encoding three cycles later. This pipelining would require an additional 7 LUTs to pipeline data which is needed more than one pipeline stage after being generated (*i.e.* 4 to retime `c<3:0>` for presentation to the second stage and 3 to retime `c<3>`, `c<1>` and `i1` for presentation to the final stage – See Figure 10.5). Consequently, we effectively evaluate a design with $N_{LUT_{design}} = 21$ 4-LUTs with $N_p = 28$ physical 4-LUTs. Typical LUT delay, including a moderate amount of local interconnect traversal, is 7 ns (See Table 4.13). Assuming this is the only limit to cycle time, the implementation could achieve 140 MHz operation. Notice that the only reason we had to have any more LUTs or LUT descriptions than strictly required by the task description was in order to perform signal retiming based on the dependency structure of the computation. Using our FPGA area based on the model in the previous chapter, an FPGA LUT in a large array occupies $A_{bit\_elm} \approx 880\text{K}\lambda^2$. Consequently, this implementation requires:

$$A_{pipe} = 28 \cdot 880\text{K}\lambda^2 = 24.6\text{M}\lambda^2$$

Multicontext Implementation – Temporal Pipelining If, instead, we cared about the latency, but did not need 140 MHz operation, we could use a multicontext device with 3 LUT descriptions per active element ($c = 3$). To achieve the target latency of 3 LUT delays, we need to have enough active LUTs to implement the largest stage – the middle one. If the inputs are arriving from some other circuit which is also operating in multicontext fashion, we must retime them as before (Figure 10.5). Consequently, we require 3 extra LUTs in the largest stage, making for a total $N_p = 12$. Note that the 4 retiming LUTs added to stage 1 also bring its total LUT usage up to 12 LUTs. We end up implementing $N_{LUT\_design} = 21$, with $N_p = 12$ and $c = 3$. If `c<7:0>` were inputs which did not change during these three cycles, we would only need one extra retiming LUT in stage 2 for `i1`, allowing us to use $N_p = 10$.

The multicontext LUT is slightly larger due to the extract contexts. Two additional contexts add $160\text{K}\lambda^2$ to the LUT area, making for $A_{bit\_elm} \approx 1.04\text{M}\lambda^2$. The multicontext implementation requires:

$$A_{time-pipe} = 12 \cdot 1\text{M}\lambda^2 = 12.5\text{M}\lambda^2$$

In contrast, a non-pipelined, single-context implementation would require $N_p = 21$ LUTs, for an area of:

$$A_{nonpipe} = 21 \cdot 880\text{K}\lambda^2 = 18.5\text{M}\lambda^2$$

If we assume that we can pipeline the configuration read, the multicontext device can achieve comparable delay per LUT evaluation to the single context device. The total latency then is 21 ns, as before. The throughput at the 7 ns clock rate is 48 MHz. If we do not pipeline the configuration read, as was the case for the DPGA prototype (Section 10.4), the configuration read adds another 2.5 ns to the LUT delay, making for a total latency of 28.5 ns and a throughput of 35 MHz.

General Observations We were able to realize this area savings because the single context device had to deploy active compute and interconnect area for each portion of the task even though the task only required a smaller number of active elements at any point in time. In general, we have two components which combine to define the requisite area for a computational device:

1. N_d – the total number of 4-LUTs in the design – the descriptive complexity
2. N_a – the total number of 4-LUTs which must be evaluated simultaneously in order to achieve the desired task time or computational throughput – the parallelism required to achieve the temporal requirements

In an *ideal* packing, a computation requiring N_a active compute elements and N_d total 4-LUTs, can be implemented in area:

$$A_{compute} = N_a \cdot A_{LUT} + N_d \cdot A_{LUT\_config\_mem} \tag{10.1}$$

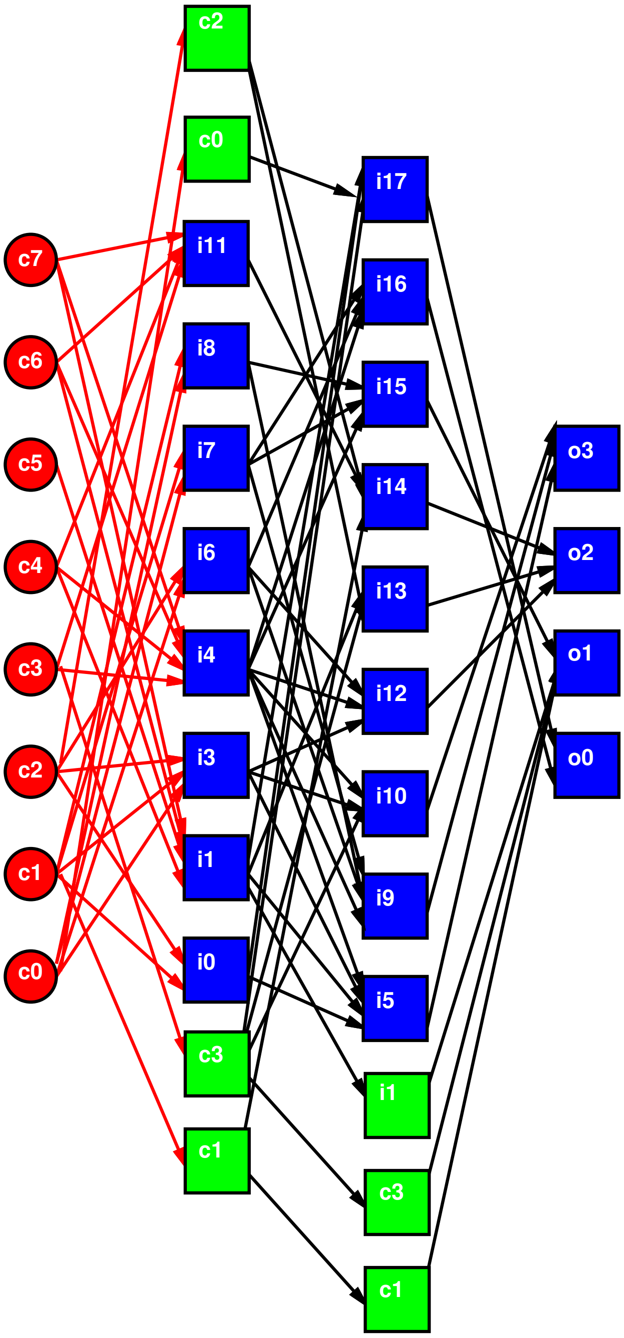


Figure 10.5: ASCII→Hex Binary Circuit Retimed for Full Pipelining

Equation 10.1 is a simplification of our area model (Equation 9.1). Using the typical values suggested in the previous chapter:

$$A_{LUT} \approx 800\text{K}\lambda^2 \tag{10.2}$$

$$A_{LUT\_config\_mem} \approx 78\text{K}\lambda^2 \tag{10.3}$$

In practice, a perfect packing is difficult to achieve due to connectivity and dependency requirements such that $N'_d > N_d$ configuration memories are required. In the previous example, we saw $N'_d = 3 \cdot 12 = 36$ for $N_d = 21$ due to retiming and packing constraints. In fact, with the model described so far, retiming requirements prevent us from implementing this task on any fewer than 12 active LUTs. Retiming requirements are one of the main obstacles to realizing the full, ideal benefits. We will see retiming effects more clearly when we look at circuit benchmarks in Section 10.5.

10.2 Related Architectures

Several hardware logic simulator have been built which share a similar execution model to the DPGA. These designs were generally motivated to reduce the area required to emulate complex designs and, consequently, took advantage of the fact that task descriptions are small compared to to their physical realizations in order to increase logic density.

The Logic Simulation Machine [BLMR83], and later, the Yorktown Simulation Engine (YSE) [Den82] were the earliest such hardware emulators. The YSE was built out of discrete TTL and MOS memories, requiring hundreds of components for each logic processor. Processors had an 8K deep instruction memory ($c = 8192$), 128 bit instructions ($n_{ibits} = 128$, $n_{ibits} = 136$ once processor-to-processor interconnect is included) and produced two results per cycle ($w = 2$). The YSE design supported arrays of up to 256 processors ($N_p = 256$), with a single controller ($N_{ctrl} = 1$) running the logic processors in lock step, and a full 256×256, 2-bit wide crossbar ($p = 1$).

The Hydra processor which Arkos Design's developed for their Pegasus hardware emulator is a closer cousin to the DPGA [Mal94]. They integrate 32, 16-context, bit processors on each Hydra chip ($N_p = 32$, $c = 16$, $w = 1$). The logic function is an 8-input NAND with programmable input inversions.

VEGA uses 1K-2K context memories to achieved a 7× logic description density improvement over single context FPGAs. At $c > 1024$, VEGA is optimized to be efficient for very large ratios, $N_d : N_a$, and can be quite inefficient for regular, high-throughput tasks. With $n_{ibits} \approx 86$, and a separate controller per processor ($N_p = N_{ctrl}$), Equation 9.1 predicts a $c = c_{max} = 2048$ VEGA processing element will have $A_{bit\_elm} \approx 218\text{M}\lambda^2$, which is about 8.5× smaller than the 2048 single context processing elements which it emulates – so the area savings realized by VEGA is quite consistent with our area model developed in Chapter 9.

Hydra and VEGA were developed independently and concurrently to the DPGA, which was first described in [BDK94].

Dharma [BCK93, Bha93] was designed to solve the FPGA routing problem. Logic is evaluated in strict levels similar to the scheme used for circuit evaluation in Section 10.5 with one gate-delay evaluation per cycle. Dharma is based on a few, monolithic crossbars ($p = 1$) which are reused at each level of logic. Once gates have been assigned to evaluation levels, the full crossbar makes placement and routing trivial. While this arrangement is quite beneficial for small arrays, the scaling rate of the full crossbar makes this scheme less attractive for large arrays, N_p, as we saw in Section 7.2.1.

10.3 Realm of Application

DPGAs, as with any general-purpose computing device supporting the rapid selection among instructions, are beneficial in cases where only a limited amount of functionality is needed at any point in time, and where it is necessary to rapidly switch among the possible functions needed. That is, if we need all the throughput we can possibly get out of a single function, as in the fully-pipelined ASCII Hex$\rightarrow$Binary converter in Section 10.1, then an FPGA, or other purely spatial reconfigurable architecture will handle the task efficiently. However, when the throughput requirements from a function are limited or the function is needed only intermittently, a multicontext device can provide a more efficient implementation. In this section, we look at several, commonly arising situations where multicontext devices are preferable to single-context devices, including:

- Tasks with limited throughput requirements
- Latency limited tasks
- Time or data varying logical functions

We also briefly revisit instruction bandwidth to see why partial reconfiguration, alone, is not an adequate substitute for many of these tasks.

10.3.1 Limited Throughput Requirements

Often the system environment places limits on the useful throughput for a subtask. As we saw in the introduction to this chapter, when the raw device supports a higher throughput than that required from the task, we can share the active resources in time among tasks or among different portions of the same task.

Relative Processing Speeds Most designs are composed of several sub-components or sub-tasks, each performing a task necessary to complete the entire application (See Figure 10.6). The overall performance of the design is limited by the processing throughput of the slowest device. If the performance of the slowest device is fixed, there is no need for the other devices in the system to process at substantially higher throughputs.

In these situations, reuse of the active silicon area on the non-bottleneck components can improve performance or lower costs. If we are getting sufficient performance out of the bottleneck resource, then we may be able to reduce cost by sharing the gates on the non-bottleneck resources between multiple "components" of the original design (See Figure 10.7). If we are not getting sufficient performance on the bottleneck resource and its task is parallelizable, we may be able to employ underused resources on the non-bottleneck components to improve system performance without increasing system cost (See Figure 10.8).

Fixed Functional Requirements Many applications have fixed functional requirements. Input processing on sensor data, display processing, or video processing all have task defined processing rates which are fixed. In many applications, processing faster than the sample or display rate is not necessary or useful. Once we achieve the desired rate, the rest of the "capacity" of the device is

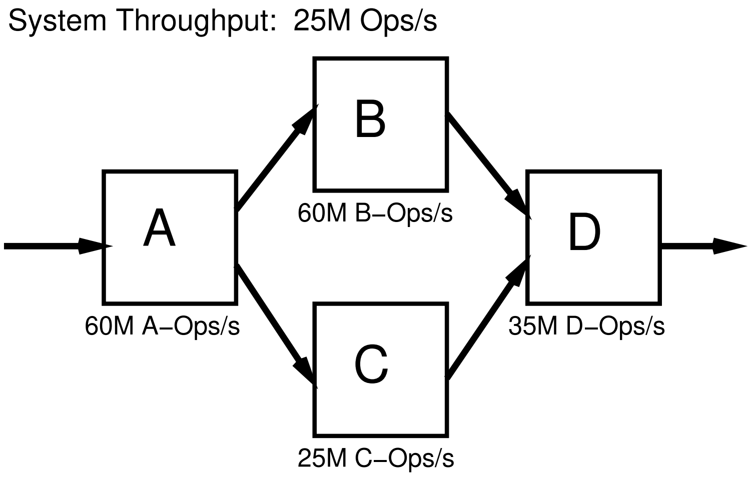


Figure 10.6: Typical Multicomponent System

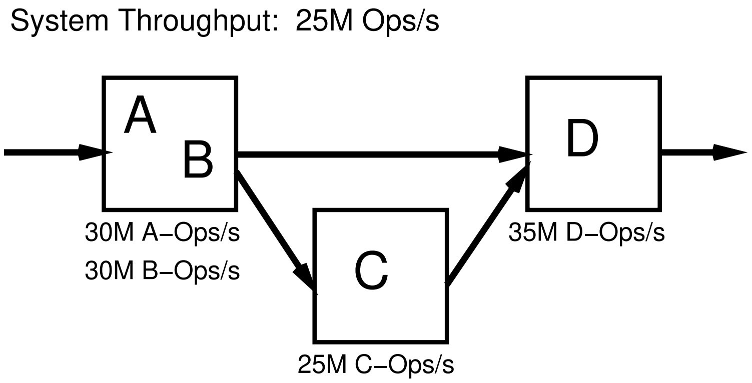


Figure 10.7: Multifunction Component in System

not required for the function. With reuse of active silicon, the residual processing capacity can be employed on other computations.

I/O Latency and Bandwidth Device I/O bandwidth often acts as a system bottleneck, limiting the rate at which data can be delivered to a part. This, in turn, limits the useful throughput we can extract from the internal logic. Even when the I/O pins are heavily reused (*e.g.* [BTA93]), components often have less I/O throughput than they have computational throughput. Reviewing technology costs, we expect this bottleneck to only get worse over time.

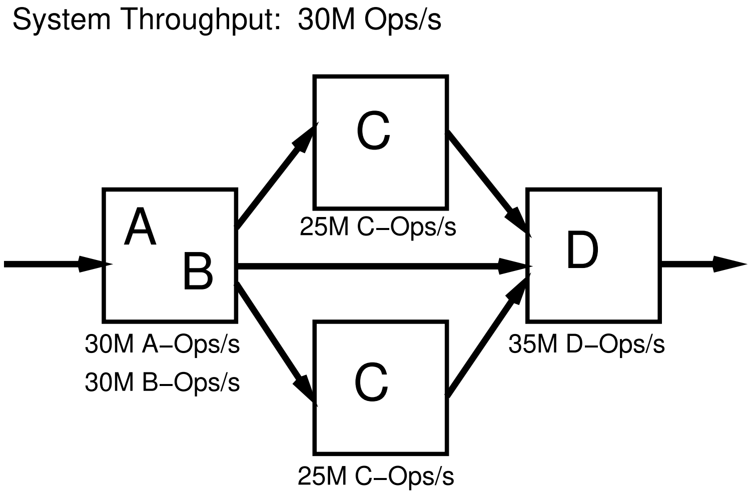


Figure 10.8: Function Distribution in System

- Since I/O's must drive off-chip capacitances, the inherent bandwidth through each pin is often lower than the logic cycle time. With on-chip logic speeds scaling faster than I/O speeds, this bandwidth gap will only increase as technology advances.
- Handling signals above $\approx$ 30 MHz becomes difficult at the PCB level, requiring more expensive packaging and more complex design. On-chip handling of high speed clocks is much more manageable.
- With conventional perimeter I/O pads, the number of I/O's scales as the square root of the internal logic area. As device capacity continues to increase, the disparity between internal logic real estate and I/O's provided grows larger.

When data throughput is limited by I/O bandwidth, we can reuse the internal resources to provide a larger, effective, internal gate capacity. This reuse decrease the total number of devices required in the system. It may also help lower the I/O bandwidth requirements by grouping larger sets of interacting functions on each IC.

10.3.2 Latency Limited Designs

Some designs are limited by latency not throughput. Here, high throughput may be unimportant. Often it is irrelevant how quickly we can begin processing the next datum if that time is shorter than the latency through the design. This is particularly true of applications which must be serialized for correctness (*e.g.* atomic actions, database updates, resource allocation/deallocation, adaptive feedback control).

By reusing gates and wires, we can use device capacity to implement these latency limited operations with less resources than would be required without reuse. This will allow us to use smaller devices to implement a function or to place more functionality onto each device.

Cyclic dependencies Some computations have cyclic dependencies such that they cannot continue until the result of the previous computation is known. For example, we cannot reuse a multiplier when performing exponentiation until the previous multiply result is known. Finite state machines (FSMs) also have the requirement that they cannot begin to calculate their behavior in the next state, until that state is known. In a purely spatial implementation, each gate or wire performs its function during one gate delay time and sits idle the rest of the cycle. Active resource reuse is the most beneficial way to increase utilization in cases such as these.

10.3.3 Temporally Varying or Data Dependent Functional Requirements

Another characteristic of finite state machines is that the computational task varies over time and as a function of the input data. At any single point in time, only a small subset of the total computational graph is needed. In a spatial implementation, all of the functionality must be implemented simultaneously, even though only small subsets are ever used at once. This is a general property held by many computational tasks.

Many tasks may perform quite different computations based on the kind of data they receive. A network interface may handle packets differently based on packet type. A computational function may handle new data differently based on its range. Data objects of different types may require widely different handling. Rather than providing separate, active resources for each of these mutually exclusive cases, a multicontext device can use a minimum amount of active resources, selecting the proper operational behavior as needed.

10.3.4 Multicontext versus Monolithic and Partial Reconfiguration

Multicontext devices are specifically tailored to the cases where we need a limited amount of active functionality at any point in time, but we need to be able to select or change that functionality rapidly. This rapid switching is necessary to obtain reasonable performance for the kinds of applications described in this section. This requirement makes reconfigurations from a central memory pool, on or off chip, inadequate.

In this section, we draw out this point, reviewing the application domains identified in the previous section. We also look at cases where one can get away without multicontext devices. At the end of this section, we articulate a reconfiguration rate taxonomy which allows us to categorize both device architectures and applications.

Tasks with limited throughput requirements As we discussed in Section 10.3.1, tasks with limited throughput requirements can be implemented in less area using multicontext devices. If, we placed the configuration contexts off-chip, the context-switch rate would be paced by the limited bandwidth into configuration memory. Returning to our ASCII Hex→Binary converter, in the three context case, we would have to reload 12 LUT instructions between contexts 1 and 2, 4 between contexts 2 and 3, and 12 between contexts 3 and 1. If we assume a 500MB/s RAMBUS I/O port [Ram93] operating at peak burst performance, we can load one byte/2 ns. The evaluation time would be:

$$t_{off\_context} = \frac{12 \cdot n_{ibits}}{\text{8b/2 ns}} + t_{LUT\_delay} + \frac{12 \cdot n_{ibits}}{\text{8b/2 ns}} + t_{LUT\_delay} + \frac{4 \cdot n_{ibits}}{\text{8b/2 ns}} + t_{LUT\_delay}$$

Assuming $n_{ibits} = 64$, as in Section 8.2 and Chapter 9, and $t_{LUT\_delay} = 7$ ns:

$$
\begin{aligned}
&= \frac{12 \cdot 64}{8\text{b/2 ns}} + 7\text{ ns} + \frac{12 \cdot 64}{8\text{b/2 ns}} + 7\text{ ns} + \frac{4 \cdot 64}{8\text{b/2 ns}} + 7\text{ ns} \\
&= \underbrace{(192\text{ ns} + 192\text{ ns} + 64\text{ ns})}_{\text{instruction load time}} + \underbrace{(21\text{ ns})}_{\text{operation time}} \\
&= 448\text{ ns} + 21\text{ ns} \\
&= 469\text{ ns}
\end{aligned}
$$

Such a solution is simultaneously: (1) over an order of magnitude slower than the multicontext implementation, which operated at 21-28 ns, and (2) over two order of magnitude larger when you consider the 500-700Mλ^2 occupied by a 4Mb RAMBUS DRAM. Arguably, the DRAM could be smaller than 4Mb, but it is not economical to build, package, and sell such small memories. Further, notice that this is a tiny subtask with 12 active LUTs, while reasonably sized FPGAs contain hundreds to thousands of LUTs, making the reconfiguration time orders of magnitude slower. As noted in Chapter 8, reconfiguration bandwidth limitations will dictate the rate of operation rather than the circuit path length.

Latency limited tasks The same effect described above occurs in latency limited designs. If we want to save real-estate by reusing active area, the time to load in the next instruction may pace operation. Off-chip memory, or an on-chip central memory pool, will suffer from the memory bandwidth bottleneck just noted.

Data varying logical functions In finite-state machines, or other tasks which may change the function they perform at each point in time based on the data arriving, this reconfiguration latency also determines cycle time. Many tasks will exhibit the characteristics identified here – in response to a new data item, hundreds of LUT instructions must be loaded before the actual task, which may take only a few LUT delays to evaluate, can be performed.

Infrequent temporal change Of course, if the distinct pieces of functionality required change only infrequently, and can operationally tolerate long reload latencies, then off-chip reconfigurations may be acceptable and efficient. For example, the UCLA configurable computing system for automatic target recognition [VSCZ96] takes advantage of the fact that a loaded correlation configuration can be used against an entire image segment before a new correlation is required. With 128$\times$128 pixel images, a complete filter match of a 16$\times$16 correlation template across the full image requires roughly 128^2 correlations amounting to 16K clock cycles on the correlator. Operating at a 60 ns clock rate, this full correlation takes roughly 1 ms. The conventional FPGA actually used for the UCLA implementation, a Xilinx XC4010, takes 10 ms to reload its configuration [Xil94b]. However, as we noted in Section 7.8, the sparse encoding used by conventional devices makes them excessively slow at reconfiguration. Assuming a RAMBUS reconfiguration port and 64-bits/4-LUT, the 1600 4-LUTs on the XC4010 can be reloaded in roughly:

$$t_{reload} = \frac{1600 \cdot 64}{8\text{b/2 ns}} = 25.6\mu\text{s}$$

Here, the reload time is small compared to the loaded context operating time ($t_{reload} << t_{operate}$), such that reload has a small effect on the rate of operation. In fact, as the UCLA paper notes, when the next context is predictable in advance and $t_{reload} < t_{operate}$, a two context FPGA would be able to completely overlap the loading of the next instruction with the operation in the current configuration.

Large-grain, data-dependent blocks Similarly, when performing data dependent computations and the type of data changes slowly compared to the processing rate, long reconfiguration times might be acceptable. For example, a video display which can handle different video data formats (*e.g.* PAL, NTSC, MPEG-1, MPEG-2, HDTV), will only have to process and display one kind of video stream at a time. For human consumption, it will typically display the same data stream for a long time and the 10's of milliseconds of latency it may take to load the configuration with the appropriate display engine would not be noticeable to the human observer.

Minor configuration edits Sometimes configurations need only minor edits in order to evolve over time or be properly configured for different data types. For example, an n-character text matching filter may only require the configuration of a n/4 4-LUTs to change to handle a different n-character search target. If these only represent a small portion of the entire configuration, the reconfiguration can be described as an edit on the existing configuration with less bandwidth than a full context reload. In cases like this where the edits are small, *partial reconfiguration* – the ability to efficiently change small portions of the configuration while leaving the rest in place – may be adequate to reduce context switch bandwidths sufficiently to keep reload latency low. We see partial reconfiguration support in modern devices from Plessey [Ple90], Atmel [Atm94], and Xilinx [Xil96] to support configuration edits such as this.

Reconfiguration Rate Taxonomy From the above, we see three cases for configuration management based on the rate at which the task requires distinct pieces of functionality and the rate at which it is efficient to change the configuration applied to the active processing elements:

1. **Static** – the configuration does not change within an operational epoch
    - **Usage Scenario:** Traditional ASIC and FPGA applications where all the functionality is needed all the time. Particularly appropriate for throughput limited cases where one wants all the throughput one can get out of a device for every function it provides.
    - **Architectures:** single-context FPGAs
2. **Quasistatic** – the configuration changes slowly compared to the rate of operation upon data
    - **Usage Scenario:** Context load time is amortized across long periods of processing with the loaded context (*e.g.* UCLA wireless video [JOSV95], UCLA ATR [VSCZ96], BYU DISC [WH95], BYU run-time reconfigurable neural networks [EH94]).
    - **Architectures:** FPGAs with rapid reconfiguration (*e.g.* Atmel [Atm94], Xilinx 6200 [Xil96]) along with traditional, in-circuit reprogrammable FPGAs for very coarse-grained tasks

3. **Dynamic** – configuration changes at the same rate as data, potentially on a cycle-by-cycle basis

   - **Usage Scenario:** Limited active resources are shared among multiple operations to extract full usage of the active resources when the task throughput requirements are low compared to the potential device throughput. (*e.g.* multicontext circuit evaluation introduced above and detailed in Section 10.5, finite-state machine evaluation (Section 10.6), interleaved, multifunction components (Section 10.7.1)).
   - **Architectures:** DPGAs, traditional processor architectures including DSPs and VLIW processors, SIMD and Vector array processors

We can further subdivide configuration management capabilities of architecture and application requirements based on whether they can take advantage of limited bandwidth configuration edits:

1. **Atomic** – the vector of instructions across the array must change all at once

   - **Architectures:** Traditional FPGAs (*e.g.* Xilinx 2K, 3K, 4K, 5K [Xil94b], Altera FLEX 8K [Alt94]), VLIW processors

2. **Non-atomic** – small subsets of the array instructions can be changed independently

   - **Architectures:** FPGAs supporting partial reconfiguration (*e.g.* Xilinx 6200 [Xil96], Atmel [Atm94])

Strictly speaking, the atomicity of configuration changes is orthogonal to the rate of reconfiguration. For statically configured applications, the atomicity of reload is irrelevant since the context does not change. The atomicity is most relevant for quasistatic configuration changes since those are the cases which benefit from reduced bandwidth requirements. Dynamic architectures can change their active instruction on a cycle-by-cycle basis so non-atomic changes do not allow an array-wide context switch to occur any faster. However, edits to the non-active contexts on dynamic architectures may still benefit from the bandwidth reduction enabled by non-atomic updates.

10.4 A Prototype DPGA

Jeremy Brown, Derrick Chen, Ian Eslick, and Edward Tau started a first-generation prototype DPGA prototype while they were taking MIT's introductory VLSI course (6.371) during the Fall of 1994. The chip was completed during the Spring of 1995 with additional help from Ethan Mirsky. André DeHon helped the group hash out the microarchitecture and oversaw the project. The prototype was first presented publicly in [TEC+95]. A project report containing lower level details is available as [BCE+94].

In this section, we describe this prototype DPGA implementation. The design represents a first generation effort and contains considerable room for optimization. Nonetheless, the design demonstrates the viability of DPGAs, underscores the costs and benefits of DPGAs as compared to traditional FPGAs, and highlights many of the important issues in the design of programmable arrays. The fabricated prototype did have one timing problem which prevented it from functioning fully, but our *post mortem* analysis suggests that the problem is easily avoidable.

Our DPGA prototype features:

- 4 on-chip configuration contexts
- DRAM configuration cells
- non-intrusive background loading
- automatic refresh of dynamic memory elements
- wide bus architecture for high-speed context loading
- two-level routing architecture

We begin by detailing our basic DPGA architecture in Section 10.4.1. Section 10.4.2 provides highlights from our implementation including key details on our prototype DPGA IC. In Section 10.4.3, we describe several aspects of the prototype's operation. Section 10.4.4 extracts a DPGA area model based on the prototype implementation. Section 10.4.5 closes out this section on the DPGA prototype by summarizing the major lessons from the effort.

10.4.1 Architecture

Figure 10.9 depicts the basic architecture for this DPGA. Each array element is a conventional 4-input lookup table (4-LUT). Small collections of array elements, in this case 4×4 arrays, are grouped together into *subarrays*. These subarrays are then tiled to compose the entire array. Crossbars between subarrays serve to route inter-subarray connections. A single, 2-bit, global context identifier is distributed throughout the array to select the configuration for use. Additionally, programming lines are distributed to read and write configuration memories.

DRAM Memory The basic memory primitive is a 4×32 bit DRAM array which provides four context configurations for both the LUT and interconnection network (See Figure 10.10). The memory cell is a standard three transistor DRAM cell. Notably, the context memory cells are built entirely out of N-well devices, allowing the memory array to be packed densely, avoiding the large cost for N-well to P-well separation. The active context data is read onto a row of standard, complementary CMOS inverters which drive LUT programming and selection logic.

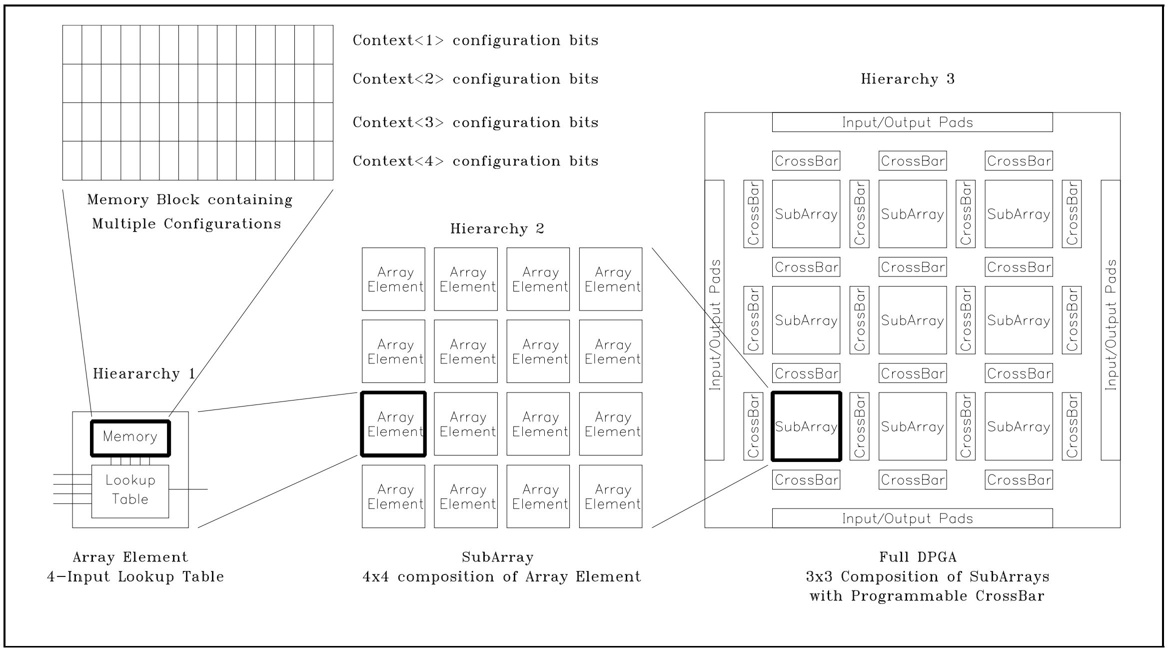


Figure 10.9: Architecture and Composition of DPGA

Array Element The array element is a 4-LUT which includes an optional flip-flop on its output (Figure 10.11). Each array element contains a context memory array. For our prototype, this is the 4×32 bit memory described above. 16 bits provide the LUT programming, 12 configure the four 8-input multiplexors which select each input to the 4-LUT, and one selects the optional flip-flop. The remaining three memory bits are presently unused.

Subarrays The subarray organizes the lowest level of the interconnect hierarchy. Each array element output is run vertically and horizontally across the entire span of the subarray (Figure 10.12). Each array element can, in turn, select as an input the output of any array element in its subarray which shares the same row or column. This topology allows a reasonably high degree of local connectivity.

This leaf topology is limited to moderately small subarrays since it ultimately does not scale. The row and column widths remains fixed regardless of array size so the horizontal and vertical interconnect would eventually saturate the row and column channel capacity if the topology were scaled up. Additionally, the delay on the local interconnect increases with each additional element in a row or column. For small subarrays, there is adequate channel capacity to route all outputs across a row and column without increasing array element size, so the topology is feasible and desirable. Further, the additional delay for the few elements in the row or column of a small subarray is moderately small compared to the fixed delays in the array element and routing network. In general, the subarray size should be carefully chosen with these properties in mind.

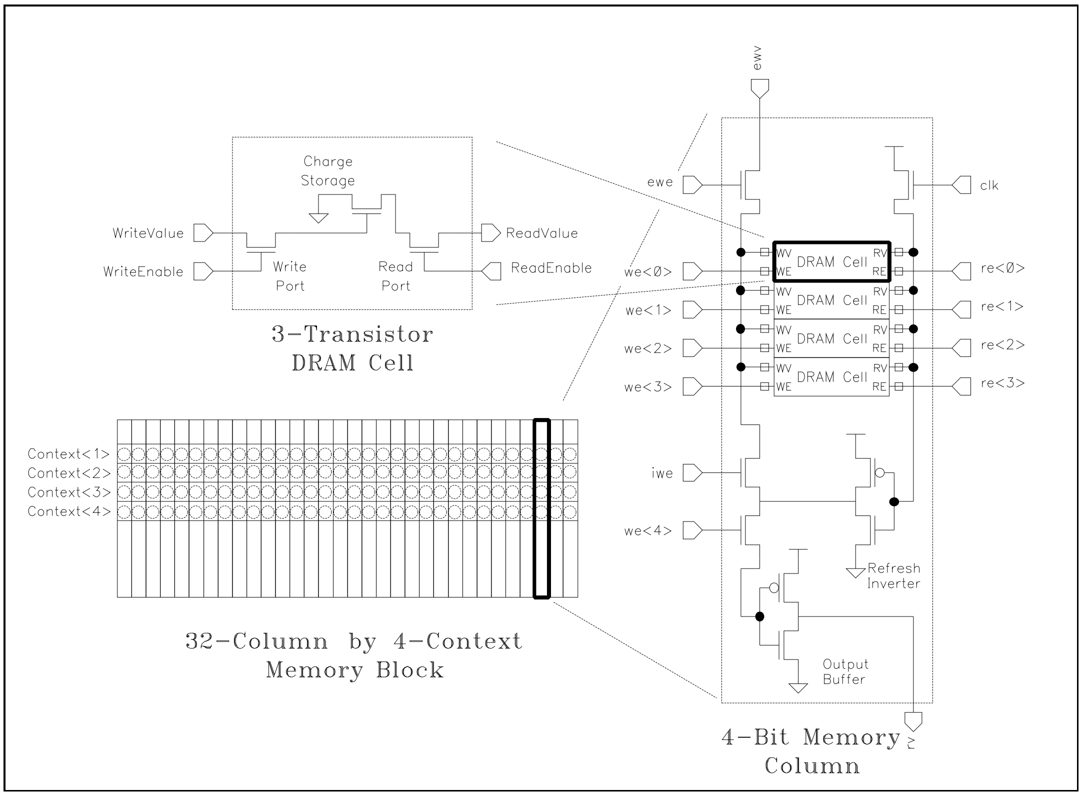


Figure 10.10: DRAM Memory Primitive

Non-Local Interconnect In addition to the local outputs which run across each row and column, a number of non-local lines are also allocated to each row and column. The non-local lines are driven by the global interconnect (Figure 10.12). Each LUT can then pick inputs from among the lines which cross its array element. In the prototype, each row and column supports four non-local lines. Each array element could thus pick its inputs from eight global lines, six row and column neighbor outputs, and its own output. Each input is configured with an 8:1 selector as noted above (Figure 10.11). Of course, not all combinations of 15 inputs taken 4 at a time are available with this scheme. The inputs are arranged so any combination of local signals can be selected along with many subsets of global signals. Freedom available at the crossbar in assigning global lines to tracks reduces the impact of this restriction, but complicates placement.

Local Decode Row select lines for the context memories are decoded and buffered locally from the 2-bit context identifier. A single decoder services each row of array elements in a subarray. One decoder also services the crossbar memories for four of the adjacent crossbars. In our prototype, this placed five decoders in each subarray, each servicing four array element or crossbar memory blocks for a total of 128 memory columns. Each local decoder also contains circuitry to refresh the DRAM memory on contexts which are not being actively read or written.

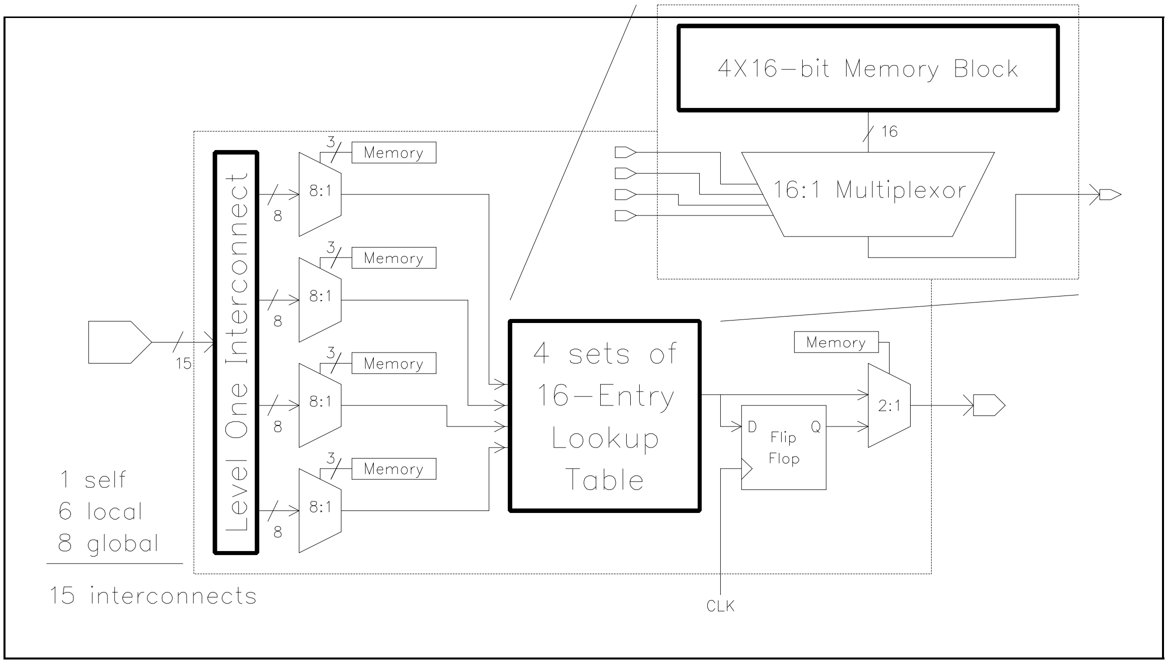


Figure 10.11: Array Element

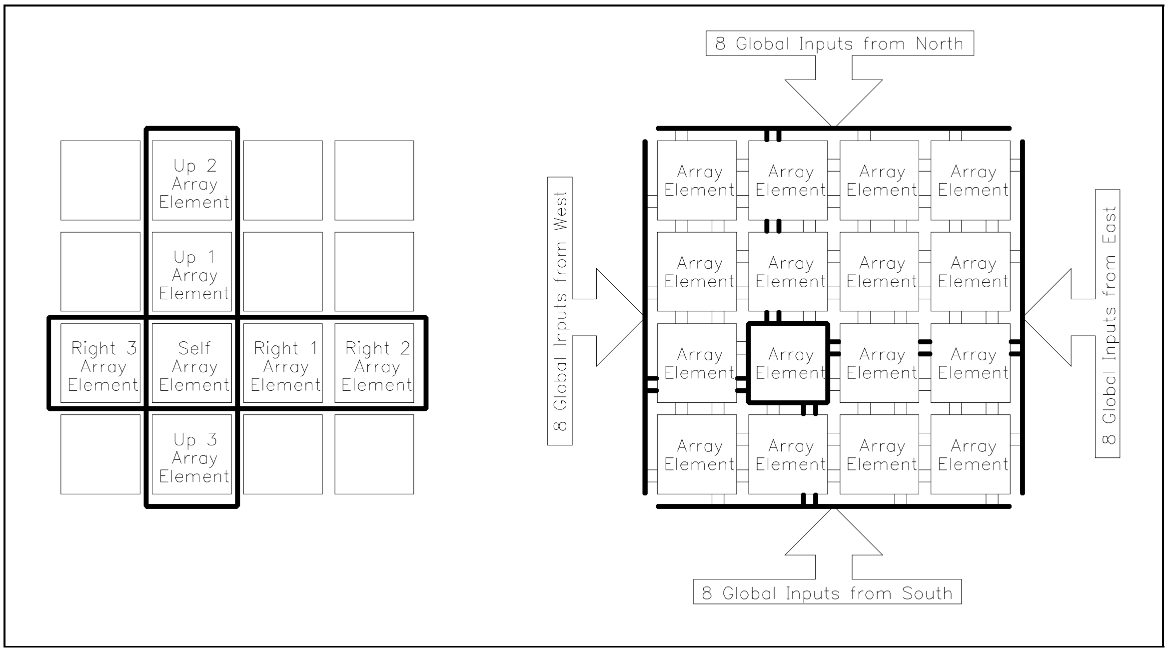


Figure 10.12: Subarray Local Interconnect

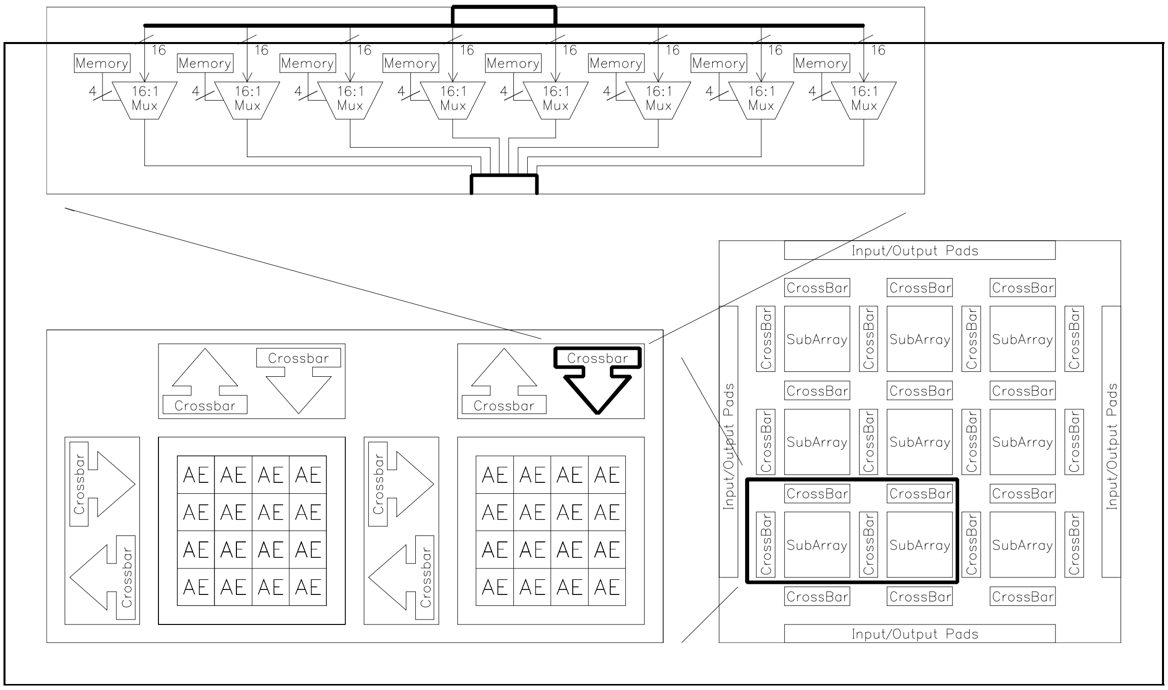


Figure 10.13: Inter Subarray Interconnect

Global Interconnect Between each subarray a pair of crossbars route the subarray outputs from one subarray into the non-local inputs of the adjacent subarray. Note that all array element outputs are available on all four sides of the subarray. In our prototype, this means that each crossbar is a 16×8 crossbar which routes 8 of the 16 outputs to the neighboring subarray's 8 inputs on that side (Figure 10.13). Each 16×8 crossbar is backed by a 4×32 DRAM array to provide the 4 context configurations. Each crossbar output is configured by decoding 4 configuration bits to select among the 16 crossbar input signals.

While the nearest neighbor interconnect is sufficient for the 3×3 array in the prototype, a larger array should include a richer interconnection scheme among subarrays. At present, we anticipate that a mesh with bypass structure with hierarchically distributed interconnect lines will be appropriate for larger arrays.

Programming The programming port makes the entire array look like one large, 32-bit wide, synchronous memory. The programming interface was designed to support high-bandwidth data transfer from an attached processor and is suitable for applications where the array is integrated on the processor die. Any non-active context may be written during operation. Read back is provided in the prototype primarily for verification.

Technology	1μ CMOS, 3 metal
Subarray Area	$1750\mu\times1460\mu$=2.6Mμ^2 (10.2Mλ^2)
LUTs/subarray	16
LUT inputs	4
Array Element Area	640Kλ^2
Contexts	4
Configuration Bits/LUT	40
Context Memory Area/LUT	24Kλ^2
Subarrays	9
Typical Cycle	9.5 ns

Table 10.1: DPGA Prototype Implementation Characteristics

Unit	Size	Composition
Die	6.8mm$\times$6.8mm	Core with pads
Core	5.6mm$\times$4.7mm	All internal logic except pads
Array Core	5.25mm$\times$4.4mm	3$\times$3 subarrays including crossbars (no pads)
Subarray+crossbar tile	$1460\mu\times1750\mu$	Subarray + 4 adjacent crossbars and memory
Crossbar (Xbar)	$495\mu\times270\mu$	16$\times$8 Crossbar including memory
Local Decode (LD)	$253\mu\times167\mu$	
Array Element (AE)	$275\mu\times240\mu$	Includes local routing channels

Table 10.2: Basic Component Sizes for Prototype

10.4.2 Implementation

The DPGA prototype is targeted for a 1μ drawn, 0.85μ effective gate length CMOS process with 3 metal layers and silicided polysilicon and diffusion. Table 10.1 highlights the prototype's major characteristics. Figure 10.14 shows the fabricated die, and Figure 10.15 shows a closeup of the basic subarray tile containing a 4$\times$4 array of LUTs and four inter-subarray crossbars. Table 10.2 summarizes the areas for the constituent parts.

Table 10.3 breaks down the chip area by consumers. In Table 10.3, configuration memory is divided between those supporting the LUT programming and that supporting interconnect. All together, the configuration memory accounts for 33% of the total die area or 40% of the area used on the die. The network area, including local interconnect, wiring, switching, and network configuration area accounts for 66% of the die area or 80% of the area actually used on the die. Leaving out the configuration memory, the fixed portion of the interconnect area is 45% of the total area or over half of the active die area.

Layout Inefficiencies The prototype could be packed more tightly since it has large blank areas and large areas dedicated to wire routing. A more careful co-design of the interconnect and subarray resources would eliminate much or all of the unused space between functional elements. Most of

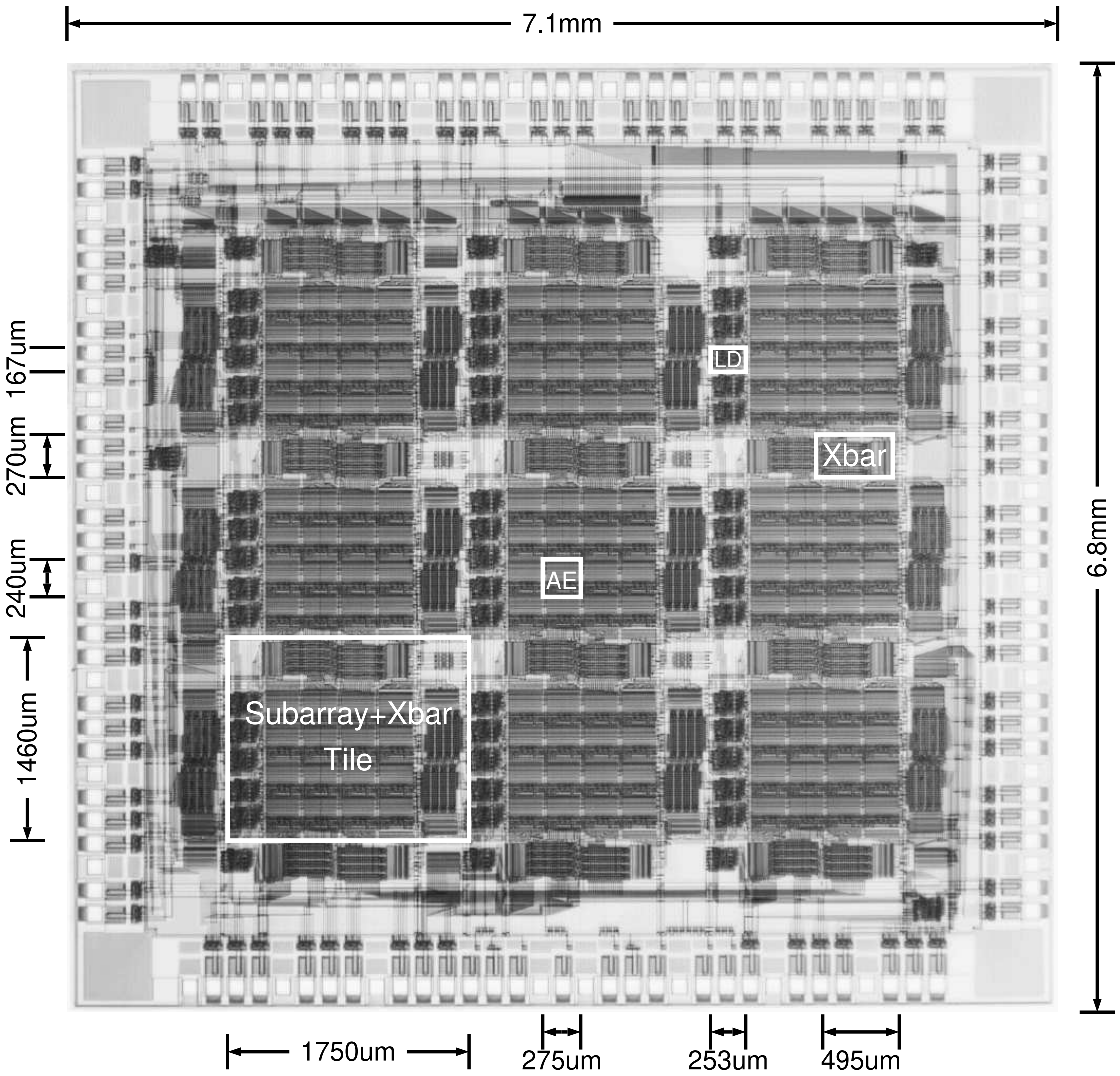


Figure 10.14: Annotated Die Photo of DPGA Prototype

the dedicated wiring channels are associated with the local interconnect within a subarray. With careful planning, it should be possible to route all of these wires over the subarray cells in metal 2 and 3. As a result, a careful design might be 40-50% smaller than our first generation prototype.

Memory

Area From the start, we suspected that memory density would be a large determinant of array size. Table 10.3 demonstrates this to be true. In order to reduce the size of the memory, we employed a 3 transistor DRAM cell design as shown in Figure 10.10. To keep the aspect ratio on the 4$\times$32 memory small, we targeted a very narrow DRAM column (See Figure 10.16).

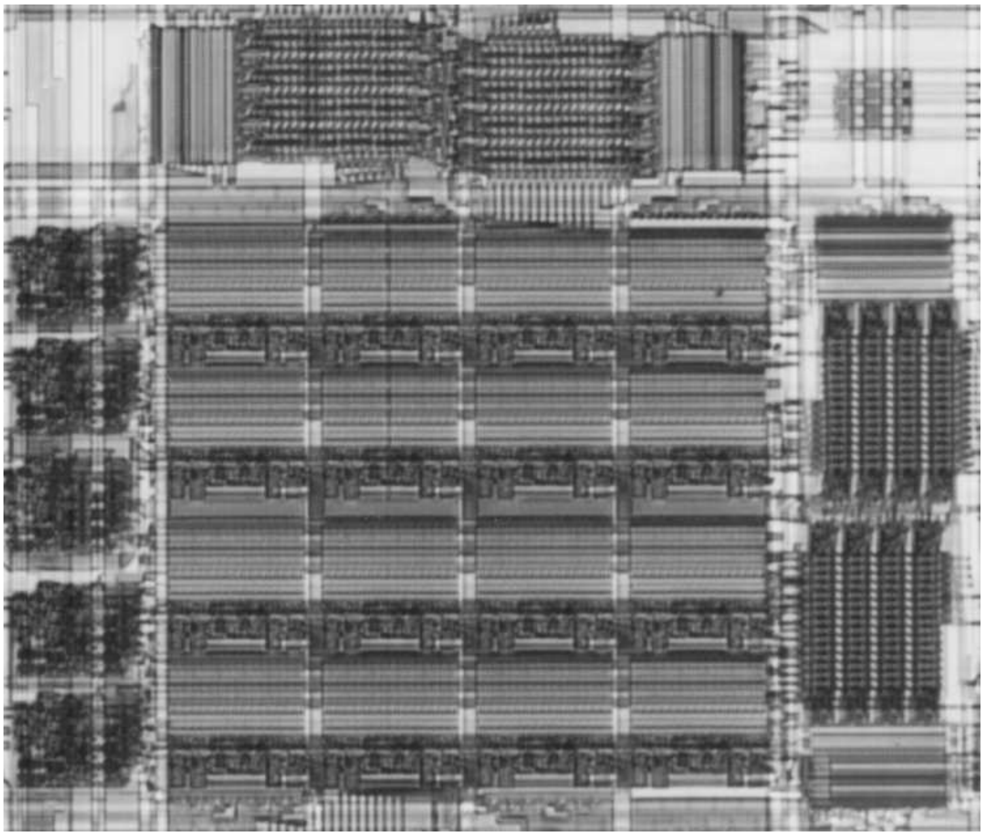


Figure 10.15: Photo of DPGA Subarray and Crossbar Tile

Function	Elements	Percent
Logic	Total	16
	Memory array	10
	Memory decode	3
	Fixed Logic	3
Network	Total	66
	Memory array	15
	Memory decode	5
	Switching	19
	Wiring	27
Blank		18
Total		100

Table 10.3: Array Core Area Breakdown by Programmable Function

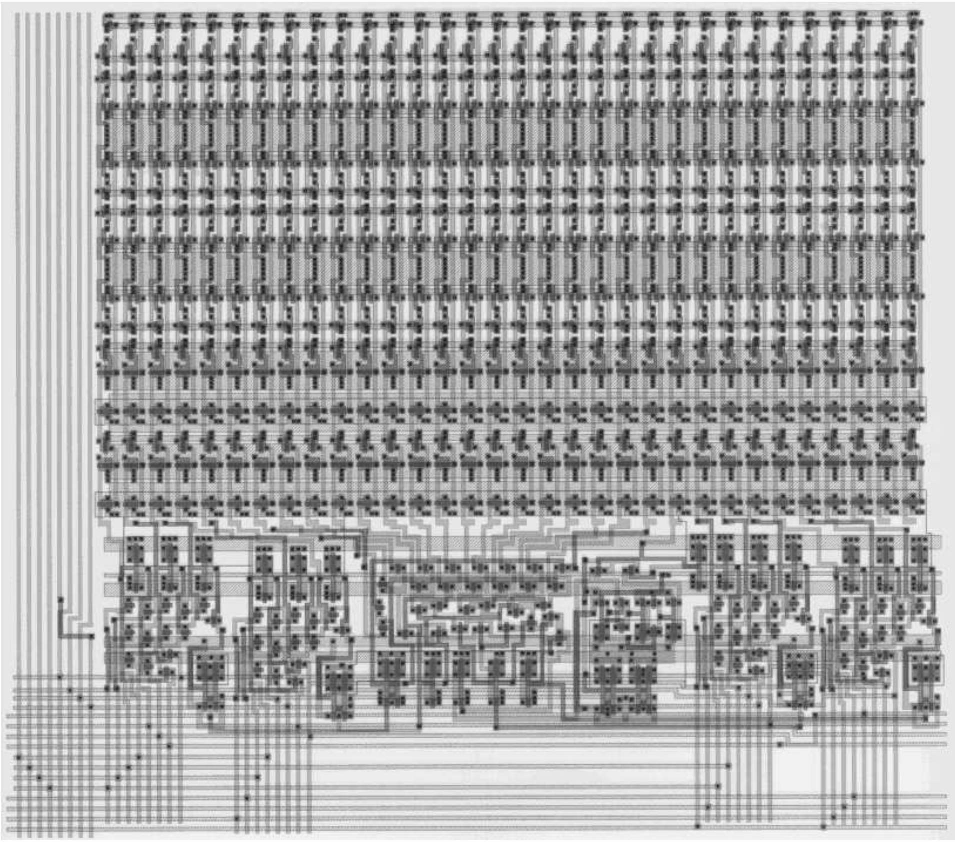


Figure 10.16: Plot of Array Element with Configuration Memory

Unfortunately, this emphasis on aspect ratio did not allow us to realize the most area efficient DRAM implementation (See Table 10.4). In particular, our DRAM cell was $7.6\mu \times 19.2\mu$, or almost $600\lambda^2$. A tight DRAM should have been 75-80μ^2, or about $300\lambda^2$. Our tall and thin DRAM was via and wire limited and hence could not be packed as area efficiently as a more square DRAM cell.

One key reason for targeting a low aspect ratio was to balance the number of interconnect channels available in each array element row and column. However, with 8 interconnect signals currently crossing each side of the array element, we are far from being limited by saturated interconnect area. Instead, array element cell size is largely limited by memory area. Further, we route programming lines vertically into each array element memory. This creates an asymmetric need for interconnect channel capacity since the vertical dimension needs to support 32 signals while the horizontal dimension need only support a dozen memory select and control lines.

For future array elements we should optimize memory cell area with less concern about aspect ratio. In fact, the array element memory can easily be split in half with 16 bits above the fixed logic in the array element and 16 below. This rearrangement will also allow us to distribute only 16 programming lines to each array element if we load the top and bottom 16 bits separately. This revision does not sacrifice total programming bandwidth if we load the top or bottom half of a pair

Element	#	Size
DRAM Cell	4	$7.6\mu \times 19.2\mu$
Output Buffer	1	$7.6\mu \times 28.0\mu$
Pass Gates	1	$7.6\mu \times 26.4\mu$
Column		$7.6\mu \times 131.2\mu$

Table 10.4: DRAM Column Breakdown

Function	Percent	
	of Total	of Memory
Memory decode	8	25
Memory cells	15	44
Buffer and gate	10	31
Total	33	100

Table 10.5: Memory Area Breakdown

of adjacent array elements simultaneously.

Table 10.5 further decomposes memory area percentages by function. We have already noted that a tight DRAM cell would be half the area of the prototype DRAM cell and an SRAM cell would be twice as large. Using these breakdowns and assuming commensurate savings in proportion to memory cell area, the tight DRAM implementation would save about 7% total area over the current design. An SRAM implementation would be, at most, 15% larger. In practice, the SRAM implementation would probably be only 5-10% larger for a 4-context design since the refresh control circuitry would no longer be needed. Of course, as one goes to greater numbers of contexts, the relative area differences for the memory cells will provide a larger contribution to overall die size.

Memory Timing The memory in the fabricated prototype suffered from a timing problem due to the skew between the read precharge enable and the internal write enable. As shown in Figure 10.10, the read bus is precharged directly on the high edge of the clock signal `clk`. The internal write enable, `iwe`, controls write-back during refresh. `iwe` and the write enable signals, `we<4:0>`, are generated by the local decoder and driven across an entire row of four array elements in a subarray, which makes for a 128-bit wide memory. Both `iwe` and `we<4:0>` are pipelined signals which transition on the rising edge of `clk`. On the rising edge of `clk`, we have a race between the turn on of the precharge transistor and the turn off of `iwe` and `we<4:0>`. Since `clk` directly controls the precharge transistor, precharge begins immediately. However, since `iwe` and `we<4:0>` are registered, it takes a clock-to-q delay before they can begin to change. Further, since there are 128 consumers spread across 1100μ, the signal propagation time across the subarray is non-trivial. Consequently, it is possible for the precharge to race through write enables left on at the end of

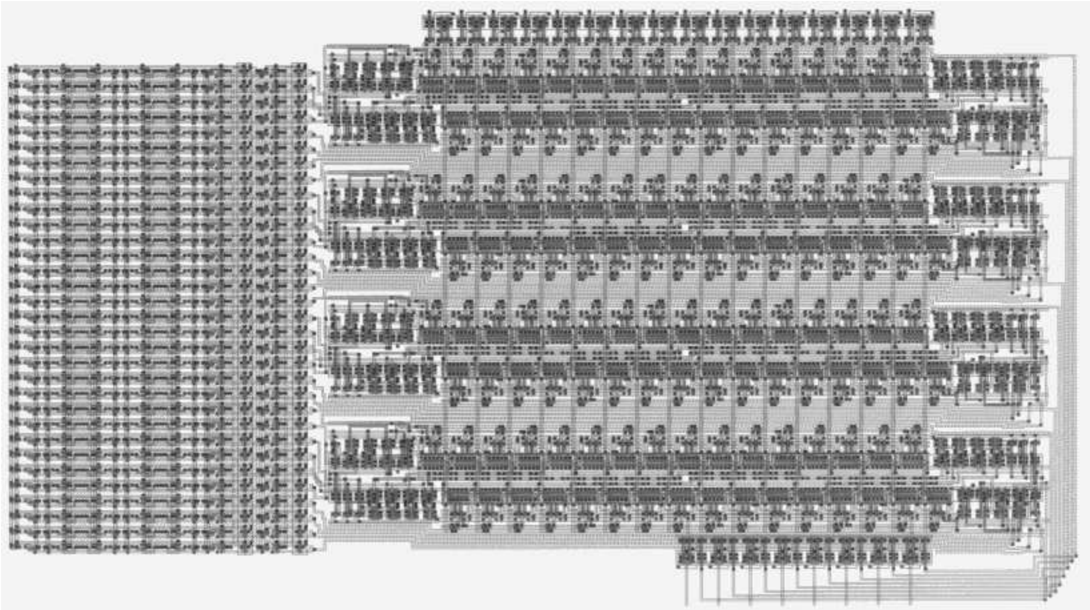


Figure 10.17: Plot of Crossbar with Configuration Memory

a previous cycle and overwrite memory. This problem is most acute for the memories which are farthest from the local decoders.

Empirically, we noticed that the memories farthest from the local decoder lost their values after short time periods. In the extreme cases of the input and output pads, which were often very far from their configuration memories, the programmed values were overwritten almost immediately. The memories closer to the local decoder were more stable. The array elements adjacent to the decoders were generally quite reliable. After identifying this potential failure mode, we simulated explicit skew between `clk` and the write enables in SPICE. In simulation, the circuit could tolerate about 1.5 ns of skew between `clk` and the write enables before the memory values began to degrade.

We were able to verify that refresh was basically operational. By continually writing to single context, we can starve other contexts from ever refreshing. When we forced the chip into this mode, data disappeared from the non-refreshed memories very quickly. The time constant on this decay was significantly different from the time constants observed due to the timing decay giving us confidence that the basic refresh scheme worked aside from the timing skew problem.

Obviously, the circuit should have been designed to tolerate this kind of skew. A simple and robust solution would have been to disable the refresh inverter or the writeback path directly on $\overline{\texttt{clk}}$ to avoid simultaneously enabling both the precharge and writeback transistors. Alternately, the precharge could have been gated and distributed consistently with the write enables.

Crossbar Implementation

To keep configuration memory small, the crossbar enables were stored encoded in configuration memory then decoded for crossbar control. The same 4×32 memory used for the array element

Path	Delay symbol	slow-speed	nominal
CLK→configuration memory stable	t_{mem}	4 ns	2.5 ns
CLK→XBAR out	t_{xbar1}	8.5 ns	5 ns
XBAR in→XBAR out	t_{xbar}	4.5 ns	2.5 ns
LUT in→LUT output (1 level)	t_{lut}	9 ns	3.5 ns
CLK→CLK (maximum, DRAM leakage)	$t_{clk_{max}}$		200 ns

Table 10.6: Estimated Timings

was used to control each 16×8 crossbar. Note that the entire memory is 128Kλ^2. The crossbar itself is 535Kλ^2, making the pair 660Kλ^2. Had we not encoded the crossbar controls, the crossbar memory alone would have occupied 512Kλ^2 before we consider the crossbar itself. This suggests that the encoding was marginally beneficial for our four context case, and would be of even greater benefit for greater numbers of contexts. For fewer contexts, the encoding would not necessarily be beneficial.

Timing

Table 10.6 summarizes the key timing estimates for the DPGA prototype at the slow-speed and nominal process points. As shown, context switches can occur on a cycle-by-cycle basis and contribute only a few nanoseconds to the operational cycle time. Equation 10.4 relates minimum achievable cycle time to the number of LUT delays, n_l, and crossbar crossings, n_x in the critical path of a design.

$$t_{cycle} = t_{mem} + n_l \cdot t_{lut} + n_x \cdot t_{xbar} \tag{10.4}$$

These estimates suggest a heavily pipelined design which placed only one level of lookup table logic ($n_l = 1$) and one crossbar traversal ($n_x = 1$) in each pipeline stage could achieve 60-100MHz operation allowing for a context switch on every cycle. Our prototype, however, does not have a suitably aggressive clocking, packaging, or i/o design to actually sustain such a high clock rate. DRAM refresh requirements force a minimum operating frequency of 5MHz.

Pipelining Two areas for pipelining are worth considering. Currently, the context memory read time happens at the beginning of each cycle. In many applications, the next context is predictable and the next context read can be pipelined in parallel with operation in the current context. This pipelining can hide the additional latency, t_{mem}. Also, notice that the inter-subarray crossbar delay is comparable to the LUT plus local interconnect delay. For aggressive implementations, allowing the non-local interconnect to be pipelined will facilitate small microcycles and very high throughput operation. Pipelining both the crossbar routing and the context reads could potentially allow a 3-4 ns operational cycle.

10.4.3 Component Operation

Inter-context Communication The only method of inter-context communication for the prototype is through the array element output register. That is, when a succeeding context wishes to use a value produced by the immediately preceding context, we enable the register output on the associated array element in the succeeding context (See Figure 10.11). When the clock edge occurs signaling the end of the preceding cycle, the signal value is latched into the output register and the new context programming is read. In the new context, the designated array element output now provides the value stored from the previous context rather than the value produced combinationally by the associated LUT. This, of course, makes the associated LUT a logical choice to use to produce values for the new context's succeeding context since it cannot be used combinationally in the new context, itself.

In the prototype, the array element output register is also the only means of state storage. Consequently, it is not possible to perform orthogonal operations in each context and preserve context-dependent state.

Note that a single context which acts as a shift register can be used to snapshot, offload, and reload the entire state of a context. In an input/output minimal case, all the array elements in the array can be linked into a single shift register. Changing to the shift register context will allow the shift register to read all the values produced by the preceding context. Clocking the device in this context will shift data to the output pin and shift data in from the input pin. Changing from this context to an operating context which registers the needed inputs will insert loaded values for operation. Such a scheme may be useful to take snapshots during debugging or to support context switches where it is important to save state. If only a subset of the array elements in the array produce meaningful state values, the shift register can be built out of only those elements. If more input/output signals can be assigned to data onload and offload, a parallel shift register can be built, limiting the depth and hence onload/offload time.

Context Switching Context switches are signaled by a context strobe. If context strobe is asserted at a clock edge, a context read occurs. If context strobe is not asserted, the component remains in the same context.

DRAM Refresh DRAM memory is refreshed under one of two conditions:

1. **Context Read** – Whenever a context is read, that context will be refreshed.

2. **Clocked in Same Context** – Whenever a clock cycle occurs but the context strobe is not asserted and there is no read or write to any of the memories serviced by a particular local decoder, the "next" context is refreshed. Each local decoder maintains a modulo four counter which it increments each time it is able to perform a context refresh in this manner. If the array stays in the same context for more that four cycles, every fourth cycle, the active context value is refreshed through `we<4>` (See Figure 10.10).

This refresh scheme does place some restrictions on the context sequencing, but it allows most common patterns. In particular, proper refresh occurs if we:

- continually cycle through all contexts, switching on each clock cycle

- stay in each context for several clock cycles

If one continually changes contexts on every clock cycle and only walks through a small subset of the entire set of contexts, the non-visited contexts will be starved from refresh. For example, switching continually between context zero and context one would prevent contexts two and three from ever getting a refresh.

The context memory typically gets very stylized usage. For any single memory, writes are infrequent. Common usage patterns are to read through all the contexts or to remain in one context for a number of cycles. As such the usage pattern is complementary to DRAM refresh requirements.

Background Load Notice from Figure 10.10 that the write path is completely separate from the read path. This allows background writes to occur orthogonally to normal operation. Data can be read through the refresh inverter and `we<4>` with `iwe` disabled to prevent refresh or writeback. At the same time, new data arriving on `ewv` may be loaded through `ewe` and written into memory using `we<3:0>`.

10.4.4 Prototype Context Area Model

Using the prototype areas, we can formulate a simplified model for the area of an n-context DPGA array element.

$$A_{ae} = A_{base} + c \cdot A_{context}$$

From the prototype:

$$\begin{aligned} A_{base} &= 544\text{K}\lambda^2 \\ A_{context_{tight\_DRAM}} &= 12\text{K}\lambda^2 \\ A_{context_{proto\_DRAM}} &= 24\text{K}\lambda^2 \\ A_{context_{SRAM}} &= 48\text{K}\lambda^2 \end{aligned}$$

Based on this area model, our robust context point, c^*, is 45, 23, and 11, respectively for each of the various memory implementations.

10.4.5 Prototype Conclusions

The prototype demonstrates that efficient, dynamically programmable gate arrays can be implemented which support a single cycle, array-wide context switch. As noted in Chapter 9 and the introduction to this chapter, when the instruction description area is small compared to the active compute and network area, multiple context implementations are more efficient than single context implementations for a large range of application characteristics. The prototype bears out this relationship with the context memory for each array element occupying at least an order of magnitude less area than the fixed logic and interconnect area. The prototype further shows that the context memory read overhead can be small, only a couple of nanoseconds.

The prototype has room for improvement in many areas:

- **Tighter layout** – Both the memory cells and the fixed portions of the array elements are larger than necessary for the function provided and can be improved with more careful layout and a better understanding of the relative areas of constituent components.
- **Pipeline interconnect** – Over half of the cycle time on a minimum, typical cycle is in the non-local interconnect, suggesting it may be worthwhile to optionally pipeline the non-local interconnect to increase the achievable computational density.
- **Overly limited routing** – The routing in the prototype is limited and probably inadequate for automated mapping.
- **Amortize refresh logic** – Separate refresh control is provided for every four memory blocks. This function can likely be moved to higher levels and the associated area amortized over a larger number of memory blocks.

Additional, architectural, areas for improvement over the prototype are identified in the following sections and in the next chapter.

10.5 Circuit Evaluation

One large class of workloads for traditional FPGAs is conventional circuit evaluation. In this section, we look at circuit *levelization* where traditional circuits are automatically mapped into multicontext implementations. In latency limited designs (Section 10.5.2), the DPGA can achieve comparable delays in less area. In applications requiring limited task throughput (Section 10.5.3), DPGAs can often achieve the required throughput in less area.

10.5.1 Levelization

Levelized logic is a CAD technique for automatic temporal pipelining of existing circuit netlists. Bhat refers to this technique as *temporal partitioning* in the context of the Dharma architecture [Bha93]. The basic idea is to assign an evaluation context to each gate so that:

- with a total ordering on contexts, all the inputs to context c_i are computed in that context or one of its predecessors (*i.e.* in a context c_j such that $j \leq i$)
- we minimize total capacity required for the calculation by minimizing the maximum resource usage per context

With latency constraints, we may further require that the levelized network not take any more t_{min_clk} steps than necessary. With this assignment, the series of contexts c_0, c_1, $\ldots$, c_{max-1} evaluates the logic netlist in sequence over max microcycles. With a full levelization scheme, the number of contexts used to evaluate a netlist is equal to the critical path in the netlist.

For sake of illustration, Figure 10.18 shows a fraction of the ASCII→Hex binary circuit extracted from Figure 10.4. The critical paths (*e.g.* c<1>→i8→i15→o<1>) is three elements long. Spatially implemented, this netlist evaluates a 6 gate function in 3 cycles using 6 physical gates. In three cycles, these three gates could have provided $6 \times 3 = 18$ gate evaluations, so we underutilize all the gates in the circuit. The circuit can be fully levelized as shown in Figure 10.18 (c_0 = {i1,i4,i7,i8}, c_1 = {i15}, c_2 = {o<1>}) or partially levelized combining the final two stages (c_0 = {i1,i4,i7,i8}, c_1 = {i15,o<1>}). If the inputs are held constant during evaluation, we need only four LUTs to evaluate either case. In this case, if the inputs are not held constant, we will need two additional LUTs in c_0 for retiming so there is no benefit to multicontext evaluation. However, as we saw in Section 10.1, for the whole circuit, even with retiming, the total number of LUTs needed for levelized evaluation is smaller than the number needed in a fully spatial implementation.

Recall also from Section 10.1, that grouping is one of the limits to even levelization. When there is *slack* in the circuit network, the slack gives us some freedom in the context placement for components outside of the critical path. In general, this slack should be used to equalize context size, minimizing the number of active LUTs required to achieve the desired task latency or throughput. As we see in both this subcircuit and the full circuit, signal retiming requirements also serve to increase the number of active elements we need in each evaluations level.

10.5.2 Latency Limited Designs

As noted in Section 10.3.2, many tasks are latency limited. This could be due to data dependencies, such that the output from the previous evaluation must be available before subsequent

```
INORDER = C[7] C[6] C[5] C[4] C[3] C[1] C[0] ;
OUTORDER =  O[1] ;
# stage 1 – 8 LUTs [C[3], C[1] pass through]
i1 = C[4] * C[5] * !C[6] * !C[7] ;
i4 = !C[3] * !C[4] * C[6] * !C[7] ;
i7 = !C[0] * C[1] ;
i8 = C[0] * !C[1] ;
# stage 2 – 9 LUTs [i1,C[3],C[1] pass through]
i15 = i8 * i4 + i7 * i4 ;
# stage 3 – 4 LUTs
O[1] = i1 * !C[3] * C[1]  + i15 ;
```

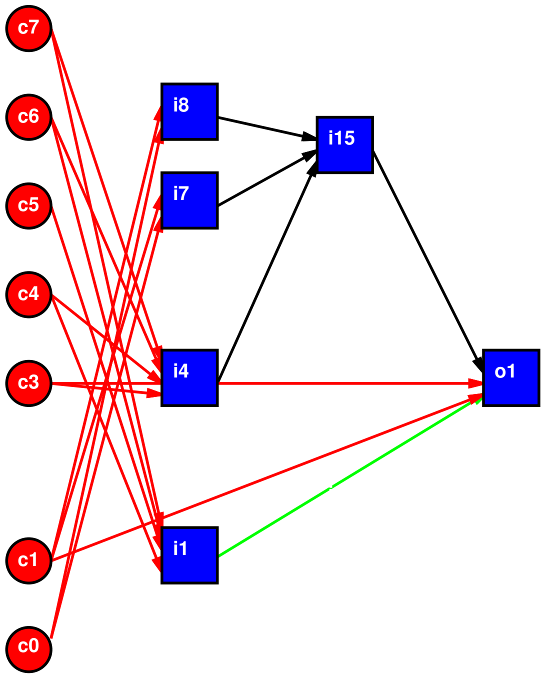


Figure 10.18: ASCII Hex→Binary Subcircuit

evaluation may begin. Alternately, this subtask may be the latency limiting portion of some larger computational task. Further, the task may be one where the repetition rate is not hight, but the response time is critical. In these cases, multicontext evaluation will allow implementations with fewer active LUTs and, consequently, less implementation area.

We use the MCNC circuit benchmark suite to characterize the benefits of multicontext evaluation. Each benchmark circuit is mapped to a netlist of 4-LUTs using `sis` [SSL+92] for technology independent optimization and `Chortle` [Fra92] [BFRV92] for LUT mapping. Since we are assuming latency is critical in this case, both `sis` and `Chortle` were run in delay mode. No modifications to the mapping and netlist generation were made for levelized computation.

LUTs are initially assigned to evaluation contexts randomly without violating the circuit dataflow requirements. A simple, annealing-based swapping schedule is then used to minimize total evaluation costs. Evaluation cost is taken as the number of 4-LUTs in the final mapping including the LUTs added to perform retiming. Table 10.7 shows the circuits mapped to a two-context DPGA, and Table 10.8, a four-context DPGA. Table 10.9 shows the full levelization case – that is circuits are mapped to an n-context DPGA, where n is equal to the number of LUT delays in the circuit's critical path. The tables break out the number of active LUTs in the multicontext implementations to show the effects of signal retiming requirements.

Circuit	Single Context 4-LUTs	Single Context levels	Single Context Model Area (Mλ^2)	2-Context 4-LUTs	2-Context 4-LUTs w/retime	2-Context Model Area (Mλ^2)	Area Ratio $\frac{\text{2-ctx}}{\text{1-ctxt}}$
5xp1	55	6	48.3	35	42	40.2	0.831
9sym	155	5	136.1	140	144	137.7	1.012
9symml	130	5	114.1	115	119	113.8	0.997
C499	406	7	356.5	219	219	209.4	0.587
C880	289	9	253.7	153	161	153.9	0.607
alu2	323	10	283.6	214	224	214.1	0.755
apex6	454	5	398.6	277	284	271.5	0.681
apex7	158	5	138.7	91	96	91.8	0.662
b9	55	3	48.3	33	43	41.1	0.851
clip	162	6	142.2	127	136	130.0	0.914
cordic	529	8	464.5	426	435	415.9	0.895
count	128	4	112.4	83	101	96.6	0.859
des	2749	8	2413.6	1397	1653	1580.3	0.655
e64	385	4	338.0	229	271	259.1	0.766
f51m	152	7	133.5	104	112	107.1	0.802
misex1	24	3	21.1	14	15	14.3	0.681
misex2	58	4	50.9	31	40	38.2	0.751
rd73	157	5	137.8	117	124	118.5	0.860
rd84	381	5	334.5	317	322	307.8	0.920
rot	398	8	349.4	217	263	251.4	0.720
sao2	98	5	86.0	61	71	67.9	0.789
vg2	92	5	80.8	51	65	62.1	0.769
z4ml	13	4	11.4	8	10	9.6	0.838
Averages	**Active LUT Ratio**			0.65			
	LUT+Retime/FPGA-LUT Ratio				0.73		
	Area Ratio						0.79

$$
\begin{aligned}
A_{circuit} &= N_{active\_LUTs} \times A_{LUT} \\
A_{LUT} &= A_{base} + N_{context} \times A_{context} \\
A_{base} &= 800\text{K}\lambda^2 \\
A_{context} &= 78\text{K}\lambda^2
\end{aligned}
$$

Table 10.7: MCNC Circuit Benchmarks – Latency Limited – Two-Context DPGA Implemenation

Circuit	Single Context 4-LUTs	Single Context levels	Single Context Model Area ($M\lambda^2$)	4-Context 4-LUTs	4-Context 4-LUTs w/retime	4-Context Model Area ($M\lambda^2$)	Area Ratio $\frac{\text{4-ctx}}{\text{1-ctxt}}$
5xp1	55	6	48.3	19	24	26.7	0.553
9sym	155	5	136.1	66	75	83.4	0.613
9symml	130	5	114.1	53	63	70.1	0.614
C499	406	7	356.5	119	150	166.8	0.468
C880	289	9	253.7	91	121	134.6	0.530
alu2	323	10	283.6	107	125	139.0	0.490
apex6	454	5	398.6	127	246	273.6	0.686
apex7	158	5	138.7	49	87	96.7	0.697
b9	55	3	48.3	24	40	44.5	0.921
clip	162	6	142.2	56	65	72.3	0.508
cordic	529	8	464.5	226	243	270.2	0.582
count	128	4	112.4	44	70	77.8	0.693
des	2749	8	2413.6	854	1110	1234.3	0.511
e64	385	4	338.0	128	186	206.8	0.612
f51m	152	7	133.5	58	66	73.4	0.550
misex1	24	3	21.1	9	14	15.6	0.739
misex2	58	4	50.9	17	32	35.6	0.699
rd73	157	5	137.8	57	64	71.2	0.516
rd84	381	5	334.5	152	161	179.0	0.535
rot	398	8	349.4	119	214	238.0	0.681
sao2	98	5	86.0	33	43	47.8	0.556
vg2	92	5	80.8	34	50	55.6	0.688
z4ml	13	4	11.4	6	9	10.0	0.877
Averages	**Active LUT Ratio**			0.36			
	LUT+Retime/FPGA-LUT Ratio				0.49		
	Area Ratio						0.62

$$
\begin{aligned}
A_{circuit} &= N_{active\_LUTs} \times A_{LUT} \\
A_{LUT} &= A_{base} + N_{context} \times A_{context} \\
A_{base} &= 800\text{K}\lambda^2 \\
A_{context} &= 78\text{K}\lambda^2
\end{aligned}
$$

Table 10.8: MCNC Circuit Benchmarks – Latency Limited – Four-Context DPGA Implemenation

Circuit	Single Context 4-LUTs	levels	Model Area ($M\lambda^2$)	Context/Level 4-LUTs	4-LUTs w/retime	Model Area ($M\lambda^2$)	Area Ratio $\frac{\text{level-ctx}}{\text{1-ctxt}}$
5xp1	55	6	48.3	13	23	29.2	0.604
9sym	155	5	136.1	102	111	132.1	0.971
9symml	130	5	114.1	85	93	110.7	0.970
C499	406	7	356.5	93	144	193.8	0.544
C880	289	9	253.7	55	106	159.2	0.627
alu2	323	10	283.6	55	92	145.4	0.513
apex6	454	5	398.6	128	256	304.6	0.764
apex7	158	5	138.7	40	92	109.5	0.789
b9	55	3	48.3	22	45	46.5	0.964
clip	162	6	142.2	54	63	79.9	0.562
cordic	529	8	464.5	136	184	262.0	0.564
count	128	4	112.4	48	69	76.7	0.683
des	2749	8	2413.6	456	915	1303.0	0.540
e64	385	4	338.0	132	186	206.8	0.612
f51m	152	7	133.5	45	55	74.0	0.555
misex1	24	3	21.1	12	15	15.5	0.736
misex2	58	4	50.9	19	33	36.7	0.721
rd73	157	5	137.8	60	67	79.7	0.578
rd84	381	5	334.5	187	195	232.1	0.694
rot	398	8	349.4	66	204	290.5	0.831
sao2	98	5	86.0	33	43	51.2	0.595
vg2	92	5	80.8	34	51	60.7	0.751
z4ml	13	4	11.4	6	9	10.0	0.877
Averages	**Active LUT Ratio**			0.34			
	LUT+Retime/FPGA-LUT Ratio				0.50		
	Area Ratio						0.70

$$
\begin{aligned}
A_{circuit} &= N_{active\_LUTs} \times A_{LUT} \\
A_{LUT} &= A_{base} + N_{context} \times A_{context} \\
A_{base} &= 800\text{K}\lambda^2 \\
A_{context} &= 78\text{K}\lambda^2
\end{aligned}
$$

Table 10.9: MCNC Circuit Benchmarks – Latency Limited – Context per Level DPGA Implemenation

From the mapped results, we see a 30-40% overall area reduction using multicontext FPGAs, with some designs achieving almost 50%. For this collection of benchmark circuits, which has an average critical path length of 5-6 4-LUT delays, a four context DPGA gives the best, overall, results. Note that retiming requirements for these circuits dictates that each context contain, on average, 50% of the LUTs in the original design. Without the retiming requirements, 60-70% area savings look possible without increasing evaluation path length.

In addition to task delay requirements, three effects are working together here to limit the number of contexts which are actually beneficial for for these circuits:

1. packing limitations
2. retiming requirements
3. non-trivial, finite instruction area

The annealing step explicitly minimized total LUT throughput including retiming. Nonetheless, looking at the total number of LUTs actually used, we see the number of active LUTs actually used for computation continues to decrease as the number of contexts increase, while the total number of LUTs tends to level out due to retiming saturation. Figure 10.19 shows the area breakdown of these effects in terms of the number of LUTs and total area required as a function of the number of contexts used for the `des` benchmark. Figures 10.20 and 10.21 show similar data for `C880` and `alu2`.

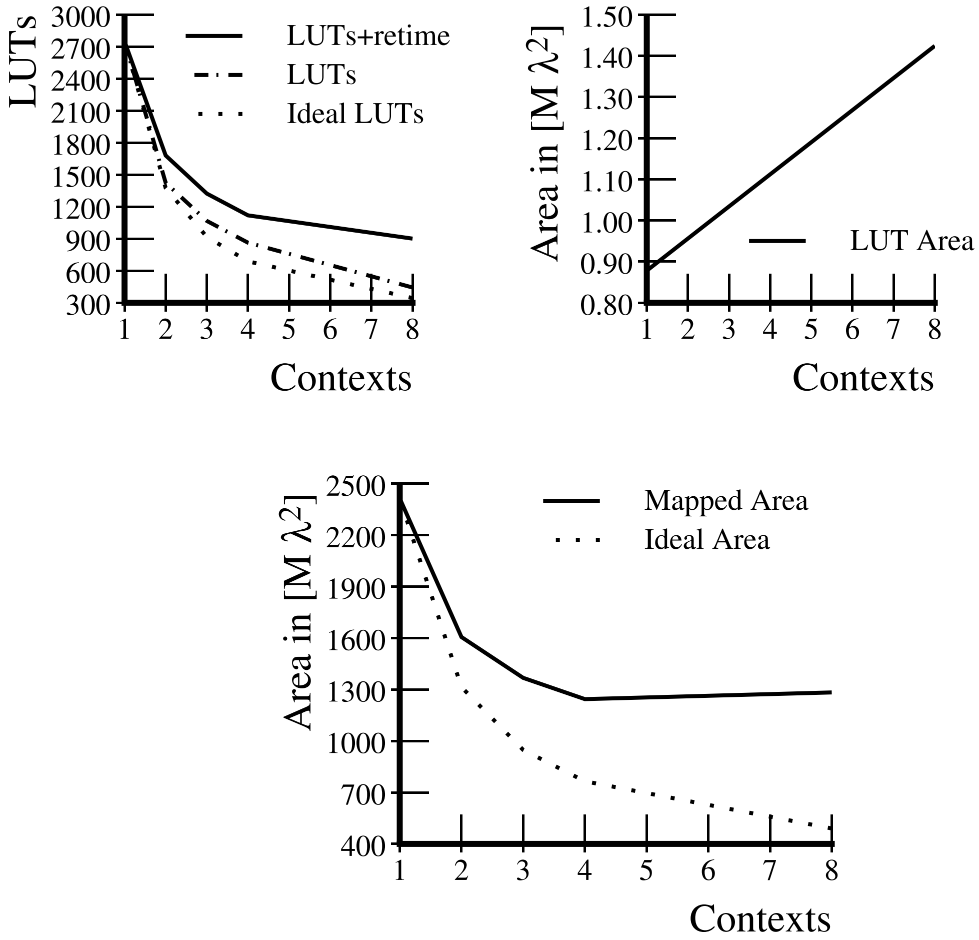
Timing There are two potential sources of additional latency for the multicontext cases versus the single context cases.

1. stage balancing time
2. context-switch time

When the number of LUT delays in the path is not an even multiple of the number of contexts, it is not possible to allocate an even number of LUT delays to each context. For example, since the `des` circuit takes eight LUT delays to evaluate, a three context implementation will place three LUT delays in two of the three contexts and two LUT delays in the third. In a simple clocking scheme, each context would get the same amount of time. In the `des` case, that would be three LUT delays, making the total evaluation time nine LUT delays.

Changing contexts will add some latency overhead at least for registering values during context switches. In the circuit evaluation case, the next context is always deterministic and could effectively be pipelined in parallel with evaluation of the previous context. From the DPGA prototype, we saw that LUT-to-LUT delay was roughly 6 ns and the context read was 2.5 ns. Register clocking overhead is likely to be on the order of 1 ns. This gives:

$$\begin{aligned}
t_{lut} &= 6 \text{ ns} \\
\text{(Pipelined Read) } t_{ctx-switch} &= 1 \text{ ns} \\
\text{(Non-Pipelined Read) } t_{ctx-switch} &= 2.5 \text{ ns} \\
t_{one-ctx} &= l_{crit-path} \times t_{lut}
\end{aligned}$$



Ideal Case $\equiv$ Perfect packing and no retiming overhead

Figure 10.19: Area Breakdown versus Number of Contexts for `des` Benchmark

$$t_{multi-ctx} = l_{crit-path} \times t_{lut} + n_{ctxt} \times t_{ctx-switch}$$

In the pipelined read case, we add 1 ns per context switch or at most $\frac{1}{6} \approx 17\%$ delay to the critical path. In the non-pipelined read case, we add 2.5 ns per context switch, or at most $\frac{2.5}{6} \approx 42\%$ delay to the critical path.

The area for the multicontext implementation is smaller and the number of LUTs involved is smaller. As a result, the interconnect traversed in each context may be more physically and logically local, thus contributing less to the LUT-to-LUT delay.

Area In the model used, we assume that the basic interconnect area per LUT is the same in the single and multiple context case. Since the total number of LUTs needed for the multicontext implementation is smaller, the multicontext implementation can use an array with fewer LUTs than the single context implementation. We saw in Section 7.6 that interconnect area grows with array size, so the area going into interconnect will be less for the multicontext array assuming the Rent parameter remains the same.

Area for Improvement The results presented in this section are based on:

1. LUT area model numbers
2. DPGA architecture resembling the DPGA prototype
3. conventional circuit netlist mapping

It may be possible to achieve better results by improving each of these areas.

1. **component area** – The model assumes an instruction storage cost based on 64 instruction bits/LUT and conventional SRAM memory cells. Smaller context area can be achieved by tighter instruction encoding (*e.g.* Section 7.8) or smaller memory cells (*e.g.* DRAM used in the prototype DPGA described in Section 10.4).
2. **architecture** – The largest gap between the ideal case and practice is in retiming. The architecture can be modified to better handle retiming (See Chapter 11).
3. **mapping** – LUT mapping which is sensitive to the retiming costs may be capable of generating netlists with lower retiming requirements.

10.5.3 Limited Task Throughput

In Section 10.3.1, we saw that system and application requirements often limit the throughput required out of each individual subtask or circuit. When throughput requirements are limited, we can often meet the throughput requirement with fewer active LUTs than design LUTs, realizing a smaller and more economical implementation.

To characterize this opportunity we again use the MCNC circuit benchmarks. `sis` and `Chortle` are used for mapping, as before. Since we are assuming here that the target criteria is throughput, both `sis` and `Chortle` are run in delay mode. As before, no modifications to the mapping and netlist generation are made.

For baseline comparison in the single-context FPGA case, we insert retiming registers in the mapped design to achieve the required throughput. That is, if we wish to produce a new result every n LUT delays, we add pipelining registers every n LUTs in the critical path. For example, if the critical path on a circuit is 8 LUT delays long and the desired throughput is one result ever 2 LUT delays, we break the circuit into four pipeline stages, adding registers every 2 LUT delays in the original circuit. We use a simple annealing algorithm to assign non-critical path LUTs in order to minimize the number of retiming registers which must be added to the design.

Similarly, we divide the multicontext case into separate spatial pipeline stages such that the path length between pipeline registers is equal to the acceptable period between results. The LUTs within a phase are then evaluated in multicontext fashion using the available contexts. Again, if the critical path on a circuit is 8 LUT delays long and the desired throughput is one result every 2 LUT delays, we break the circuit into four spatial pipeline stages, adding registers ever 2 LUT delays in the original circuit. The spatial pipeline stage is further subdivided into two temporal pipeline stages which are evaluated using two contexts. This multicontext implementation switches contexts on a one LUT delay period. Similarly, if the desired throughput was only one result every 4 LUT delays, the design would be divided into 2 spatial pipeline stages and up to 4 temporal pipeline stages, depending on the number of contexts available on the target device. The same annealing algorithm is used to assign spatial and temporal pipeline stages to non-critical path LUTs in a manner which minimizes the number of total design and retiming LUTs required in the levelized circuit.

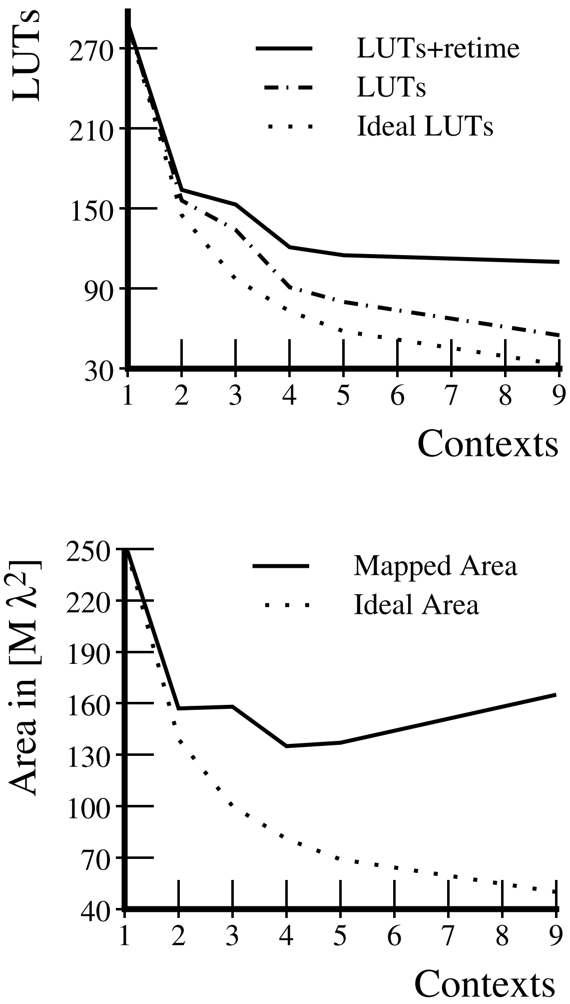
As the throughput requirements diminish, we can generally achieve smaller implementations. Unfortunately, as noted in the previous section retiming requirements prevent us from effectively using a large number of contexts to decrease implementation area. For the `alu2` benchmark, Table 10.10 shows how LUT requirements vary with throughput and Table 10.11 translates the LUT requirements into areas based on the model parameters used in the previous section. Figure 10.22 plots the areas from Table 10.11. Table 10.12 recasts the areas from Table 10.11 as ratios to the the best implementation area at a given throughput. For this circuit, the four or five context implementation is $\approx$45% smaller than the single context implementation for low throughput requirements.

Tables 10.14 through 10.16 highlight area ratios at three throughput points for the entire benchmark set. For reference, Table 10.13 summarizes the number of mapped design LUTs and path lengths for the netlists used for these experiments. We see that the 2-4 context implementations are 20-30% smaller than the single context implementations for low throughput requirements.

clocks per result	LUTs including Retiming							
	Contexts							
	1	2	3	4	5	6	7	8
1	585	585	585	585	585	585	585	585
2	353	347	347	347	347	347	347	347
3	286	252	252	252	252	252	252	252
4	240	207	161	161	161	161	161	161
5	216	188	161	139	139	139	139	139
6	212	185	156	139	126	126	126	126
7	189	145	143	139	124	118	118	118
8	189	145	143	137	124	118	110	110
9	189	145	134	125	124	118	110	110
10	178	138	129	122	120	106	96	86
11	178	138	128	120	99	99	96	86
12	178	138	128	120	99	99	96	86
13	178	128	128	120	99	99	96	86
14	178	128	127	116	99	99	96	86
15	178	127	124	116	99	99	86	86
16	178	126	116	116	99	99	86	86
17	178	125	116	116	99	99	86	86
18	178	125	116	116	99	99	86	86
19	169	91	86	73	70	69	68	68
20	169	91	86	73	68	67	67	66

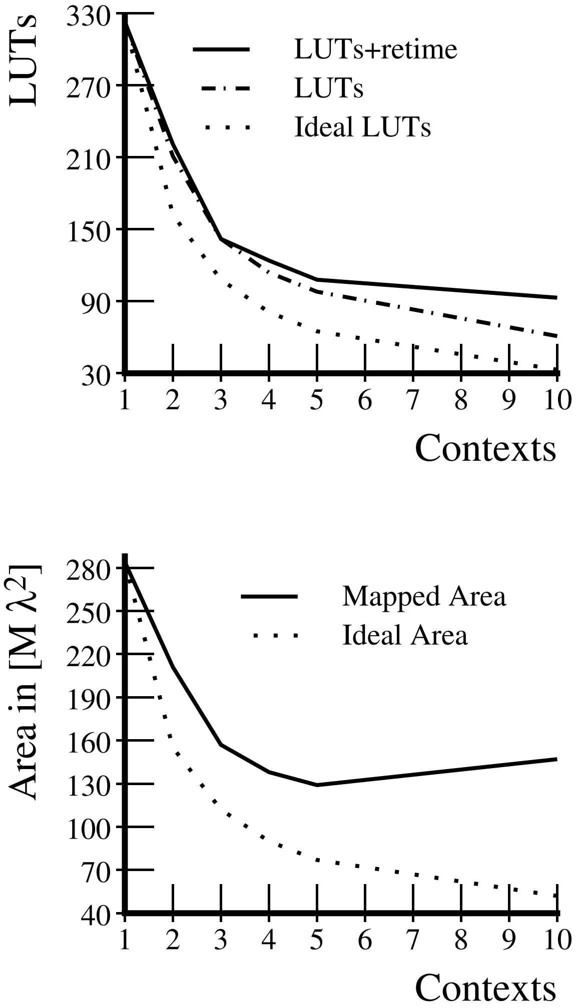
Design Luts	169
Critical Path	19

Table 10.10: Multicontext Implementations of `alu2` versus Throughput (LUTs)



Ideal Case ≡ Perfect packing and no retiming overhead

Figure 10.20: Area Breakdown versus Number of Contexts for `C880` Benchmark



Ideal Case ≡ Perfect packing and no retiming overhead

Figure 10.21: Area Breakdown versus Number of Contexts for `alu2` Benchmark

clocks per result	Model Area in Mλ^2							
	Contexts							
	1	2	3	4	5	6	7	8
1	513.6	589.7	635.3	680.9	726.6	772.2	817.8	863.5
2	309.9	349.8	376.8	403.9	431.0	458.0	485.1	512.2
3	251.1	254.0	273.7	293.3	313.0	332.6	352.3	372.0
4	210.7	208.7	174.8	187.4	200.0	212.5	225.1	237.6
5	189.6	189.5	174.8	161.8	172.6	183.5	194.3	205.2
6	186.1	186.5	169.4	161.8	156.5	166.3	176.1	186.0
7	165.9	146.2	155.3	161.8	154.0	155.8	165.0	174.2
8	165.9	146.2	155.3	159.5	154.0	155.8	153.8	162.4
9	165.9	146.2	145.5	145.5	154.0	155.8	153.8	162.4
10	156.3	139.1	140.1	142.0	149.0	139.9	134.2	126.9
11	156.3	139.1	139.0	139.7	123.0	130.7	134.2	126.9
12	156.3	139.1	139.0	139.7	123.0	130.7	134.2	126.9
13	156.3	129.0	139.0	139.7	123.0	130.7	134.2	126.9
14	156.3	129.0	137.9	135.0	123.0	130.7	134.2	126.9
15	156.3	128.0	134.7	135.0	123.0	130.7	120.2	126.9
16	156.3	127.0	126.0	135.0	123.0	130.7	120.2	126.9
17	156.3	126.0	126.0	135.0	123.0	130.7	120.2	126.9
18	156.3	126.0	126.0	135.0	123.0	130.7	120.2	126.9
19	148.4	91.7	93.4	85.0	86.9	91.1	95.1	100.4
20	148.4	91.7	93.4	85.0	84.5	88.4	93.7	97.4

$$
\begin{aligned}
A_{circuit} &= N_{active\_LUTs} \times A_{LUT} \\
A_{LUT} &= A_{base} + N_{context} \times A_{context} \\
A_{base} &= 800\text{K}\lambda^2 \\
A_{context} &= 78\text{K}\lambda^2
\end{aligned}
$$

Table 10.11: Multicontext Implementations of `alu2` versus Throughput (Area)

clocks per result	Area/Best Area Contexts 1	2	3	4	5	6	7	8
1	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
2	1.00	1.13	1.22	1.30	1.39	1.48	1.57	1.65
3	1.00	1.01	1.09	1.17	1.25	1.32	1.40	1.48
4	1.21	1.19	1.00	1.07	1.14	1.22	1.29	1.36
5	1.17	1.17	1.08	1.00	1.07	1.13	1.20	1.27
6	1.19	1.19	1.08	1.03	1.00	1.06	1.13	1.19
7	1.14	1.00	1.06	1.11	1.05	1.07	1.13	1.19
8	1.14	1.00	1.06	1.09	1.05	1.07	1.05	1.11
9	1.14	1.00	1.00	1.00	1.06	1.07	1.06	1.12
10	1.23	1.10	1.10	1.12	1.17	1.10	1.06	1.00
11	1.27	1.13	1.13	1.14	1.00	1.06	1.09	1.03
12	1.27	1.13	1.13	1.14	1.00	1.06	1.09	1.03
13	1.27	1.05	1.13	1.14	1.00	1.06	1.09	1.03
14	1.27	1.05	1.12	1.10	1.00	1.06	1.09	1.03
15	1.30	1.06	1.12	1.12	1.02	1.09	1.00	1.06
16	1.30	1.06	1.05	1.12	1.02	1.09	1.00	1.06
17	1.30	1.05	1.05	1.12	1.02	1.09	1.00	1.06
18	1.30	1.05	1.05	1.12	1.02	1.09	1.00	1.06
19	1.75	1.08	1.10	1.00	1.02	1.07	1.12	1.18
20	1.76	1.09	1.11	1.01	1.00	1.05	1.11	1.15

Table 10.12: Multicontext Implementations of `alu2` versus Throughput (Area Ratios)

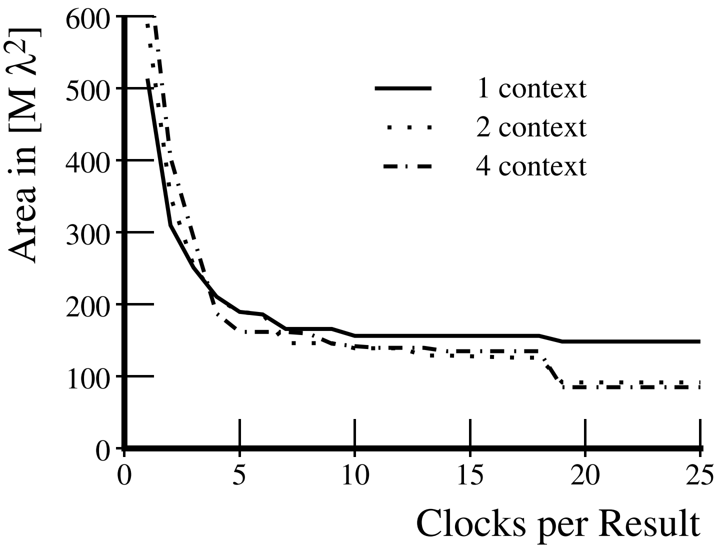


Figure 10.22: Area versus Throughput for Multicontext Implemenations of `alu2` Benchmark

Circuit	Mapped Design LUTs	Path Length
5xp1	46	10
9sym	123	7
9symml	108	8
C499	85	10
C880	176	21
alu2	169	19
apex6	248	9
apex7	77	7
b9	46	7
clip	121	9
cordic	367	13
count	46	16
des	1267	13
e64	230	9
f51m	45	17
misex1	20	6
misex2	38	8
rd73	105	10
rd84	150	9
rot	293	16
sao2	73	9
vg2	60	9
z4ml	8	7

Table 10.13: Benchmark Set Area – Mapped Characteristics

Circuit	Area/Best Area							
	Contexts							
	1	2	3	4	5	6	7	8
5xp1	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
9sym	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
9symml	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
C499	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
C880	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
alu2	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
apex6	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
apex7	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
b9	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
clip	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
cordic	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
count	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
des	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
e64	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
f51m	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
misex1	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
misex2	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
rd73	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
rd84	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
rot	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
sao2	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
vg2	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
z4ml	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68

Clocks per Result = 1

Table 10.14: Selected Area/Throughput Points for Benchmark Set (1 Clock/Result)

Circuit	Area/Best Area							
	Contexts							
	1	2	3	4	5	6	7	8
5xp1	1.30	1.17	1.05	1.01	1.00	1.06	1.13	1.14
9sym	1.69	1.04	1.00	1.05	1.07	1.13	1.20	1.27
9symml	1.59	1.00	1.00	1.06	1.07	1.13	1.20	1.19
C499	1.13	1.06	1.06	1.04	1.00	1.06	1.13	1.19
C880	1.24	1.15	1.16	1.15	1.19	1.07	1.00	1.06
alu2	1.23	1.10	1.10	1.12	1.17	1.10	1.06	1.00
apex6	1.10	1.00	1.04	1.12	1.14	1.20	1.27	1.34
apex7	1.10	1.00	1.02	1.08	1.13	1.20	1.27	1.32
b9	1.03	1.00	1.05	1.13	1.20	1.28	1.35	1.43
clip	1.64	1.20	1.04	1.05	1.00	1.00	1.06	1.12
cordic	1.58	1.21	1.27	1.00	1.01	1.02	1.00	1.06
count	1.00	1.04	1.12	1.10	1.18	1.25	1.17	1.24
des	1.19	1.00	1.05	1.06	1.12	1.14	1.16	1.22
e64	1.42	1.00	1.07	1.15	1.22	1.30	1.38	1.45
f51m	1.17	1.00	1.05	1.04	1.11	1.15	1.15	1.07
misex1	1.24	1.00	1.08	1.15	1.23	1.31	1.39	1.36
misex2	1.23	1.00	1.08	1.15	1.19	1.21	1.28	1.36
rd73	1.63	1.00	1.00	1.07	1.10	1.12	1.19	1.20
rd84	1.78	1.02	1.00	1.07	1.14	1.22	1.29	1.34
rot	1.00	1.01	1.03	1.03	1.10	1.13	1.15	1.20
sao2	1.55	1.00	1.02	1.10	1.11	1.18	1.25	1.32
vg2	1.34	1.00	1.02	1.04	1.11	1.14	1.21	1.28
z4ml	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
average	1.31	1.05	1.07	1.09	1.13	1.17	1.21	1.25

Clocks per Result = 10

Table 10.15: Selected Area/Throughput Points for Benchmark Set (10 Clock/Result)

Circuit	**Area/Best Area**							
	Contexts							
	1	2	3	4	5	6	7	8
5xp1	1.48	1.03	1.03	1.02	1.00	1.06	1.13	1.19
9sym	1.69	1.04	1.00	1.04	1.07	1.13	1.20	1.27
9symml	1.77	1.09	1.10	1.00	1.07	1.13	1.20	1.27
C499	1.21	1.04	1.00	1.02	1.06	1.11	1.15	1.22
C880	1.16	1.02	1.00	1.06	1.05	1.02	1.04	1.10
alu2	1.76	1.09	1.11	1.01	1.00	1.05	1.11	1.15
apex6	1.16	1.00	1.08	1.08	1.15	1.23	1.30	1.37
apex7	1.12	1.00	1.04	1.10	1.15	1.22	1.29	1.34
b9	1.05	1.00	1.05	1.12	1.20	1.28	1.35	1.43
clip	1.79	1.10	1.06	1.00	1.07	1.09	1.15	1.17
cordic	1.96	1.15	1.05	1.00	1.07	1.10	1.11	1.17
count	1.00	1.05	1.05	1.10	1.14	1.21	1.25	1.32
des	1.47	1.00	1.01	1.02	1.09	1.13	1.17	1.23
e64	1.43	1.00	1.08	1.15	1.23	1.31	1.39	1.46
f51m	1.52	1.08	1.00	1.03	1.05	1.06	1.13	1.13
misex1	1.34	1.00	1.08	1.15	1.23	1.31	1.39	1.46
misex2	1.23	1.00	1.08	1.11	1.19	1.21	1.28	1.36
rd73	1.72	1.05	1.03	1.00	1.07	1.13	1.20	1.27
rd84	1.78	1.02	1.00	1.06	1.09	1.16	1.23	1.30
rot	1.27	1.01	1.00	1.05	1.09	1.15	1.22	1.27
sao2	1.55	1.00	1.00	1.04	1.08	1.15	1.15	1.22
vg2	1.39	1.03	1.00	1.04	1.11	1.15	1.21	1.28
z4ml	1.00	1.15	1.24	1.33	1.41	1.50	1.59	1.68
average	1.43	1.04	1.05	1.07	1.12	1.17	1.23	1.29

Clocks per Result = 20

Table 10.16: Selected Area/Throughput Points for Benchmark Set (20 Clock/Result)

Area for Improvement The results shown here are moderately disappointing. Retiming requirements prevent us from collapsing the number of active LUTs substantially as we go to deeper multicontext implementations. As with the previous section, the results presented in this section are based on our area model, the prototype DPGA architecture, and conventional circuit netlist mapping. More than in the previous section, the results here also depend upon the experimental temporal partitioning CAD software. Groupings into temporal and spatial pipelining stages are more rigid than necessary, so better packing may be possible with more flexible stage assignment. The retiming limitations identified here also motivate architectural modifications which we will see in the next chapter.

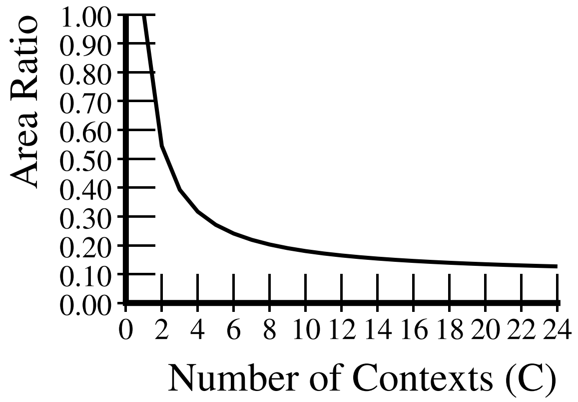


Figure 10.23: $\frac{A_{design\_LUT}}{A_{FPGA\_LUT}}$ versus N_{ctxt} for Coarse-grain Interleaved Contexts

Time-Sliced Interleaving The retiming limitation we are encountering here arises largely from packing the circuit into a limited number of LUTs and serializing the communication of intermediate results. An alternate strategy would be to share a larger group of LUTs more coarsely between multiple subcircuits in a time-sliced fashion. That is, rather than trying to sequentialize the evaluation, we retain the full circuit, or a partially sequentialized version, and only invoke it periodically.

Considering again our `alu2` example, for moderately low throughput tasks, one context may hold the 169 mapped design LUTs, while other contexts hold other, independent, tasks. A two context DPGA could alternate switch between evaluating the `alu2` example and some other circuit or task. In this two context case, the amortized area would be:

$$A_{2interleave} = \frac{1}{2} \cdot 169 \cdot A_{2ctx\_LUT} = 81\text{M}\lambda^2$$

Note that $81\text{M}\lambda^2$ is smaller than the $91\text{M}\lambda^2$ area which the two context, non-interleaved implementation achieved and smaller than the 84-85Mλ^2 for the four and five context implementation (Table 10.11). Further interleaving can yield even lower amortized costs. *e.g.*

$$A_{4interleave} = \frac{1}{4} \cdot 169 \cdot A_{4ctx\_LUT} = 47\text{M}\lambda^2$$

This coarse-grain interleaving achieves a more ideal reduction in area:

$$A_{design\_LUT} = \frac{1}{N_{ctx}} A_{base} + A_{context} \qquad (10.5)$$

Figure 10.23 plots the area ratio $\frac{A_{design\_LUT}}{A_{FPGA\_LUT}}$ versus N_{ctxt}. Note that the ratio is ultimately bounded by the $\frac{A_{context}}{A_{FPGA\_LUT}}$ ratio, which is roughly 10% for the model parameters assumed throughout this section. On the negative side,

- Coarse-grained interleaving is only suitable for very low throughput or when the tasks themselves have a moderately short evaluation path to begin with.
- Each task cannot be given its own, independent set of LUTs, but must share a larger number of LUTs with separate tasks.

10.6 Temporally Varying Logic – Finite State Machines

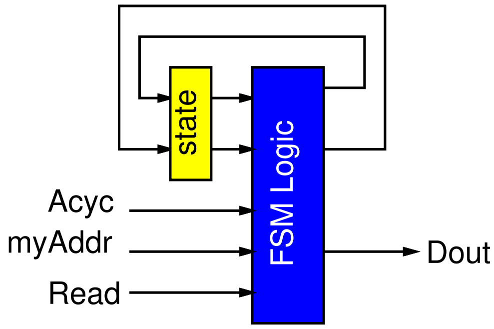
As we noted in Sections 10.3.2 and 10.3.3, the performance of a finite state machine is dictated by its latency rather than its throughput. Since the next state calculation must complete and be fed back to the input of the FSM before the next state behavior can begin, there is no benefit to be gained from spatial pipelining within the FSM logic. Temporal pipelining can be used within the FSM logic to increase gate and wire utilization as seen in Section 10.5.2. Finite state machines, however, happen to have additional structure over random logic which can be exploited. In particular, one never needs the full FSM logic at any point in time. During any cycle, the logic from only one state is active. In a traditional FPGA, we have to implement all of this logic at once in order to get the full FSM functionality. With multiple contexts, each context need only contain a portion of the state graph. When the machine transitions to a state whose logic resides in another context, we can switch contexts making a different portion of the FSM active. National Semiconductor, for example, exploits this feature in their multicontext programmable logic array (PLA), MAPL [Haw91].

10.6.1 Example

Figure 10.24 shows a simple, four-state FSM for illustrative purposes. The conventional, single-context implementation requires four 4-LUTs, one to implement each of `Dout` and `NS1` and two to calculate `NS0`. Figure 10.25 shows a two-context DPGA implementation of this same FSM. The design is partitioned into two separate circuits based on the original state variable `S1`. The two circuits are placed in separate contexts and `NS1` is used to select the circuit to execute as appropriate. Each circuit only requires three 4-LUTs, making the overall design smaller than the flat, single context implementation.

FSM Description

```
Idle  (00):
      if (Acyc & myAddr & Read)
          goto Wait1
      else
          goto Idle
Wait1 (01):
      goto Data
Data  (10):
      Assert Dout
      goto Wait2
Wait2 (11):
      goto Idle
```



FSM Logic

```
Dout = S1*/S0
NS0 =  /S1*/S0*Acyc*myAddr*Read + S1*/S0
NS1 = /S1*S0 + S1*/S0
```

Figure 10.24: Simple FSM Example

Context 0 `(S1=0)`

```
Dout  = 0
NS0  = /S0*Acyc*myAddr*Read
NS1  = S0
```

Context 1 `(S1=1)`

```
Dout  = 0
NS0  = /S0
NS1  = /S0
```

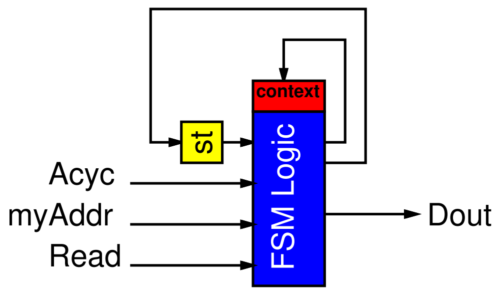


Figure 10.25: Two Context Implementation of Simple FSM Example

10.6.2 Full Temporal Partitioning

In the most extreme case, each FSM state is assigned its own context. The next state computation simply selects the appropriate next context in which to operate. Tables 10.17 and 10.18 show the reduction in area and path delay which results from state-per-context multiple context implementation of the MCNC FSM benchmarks. FSMs were mapped using `mustang` [DMNSV88]. Logic minimization and LUT mapping were done with `espresso`, `sis`, and `Chortle`. For single context FSM implementations, both one-hot and dense encodings were synthesized and the best mapping was selected. The multicontext FSM implementations use dense encodings so the state specification can directly serve as the context select. For multicontext implementations, delay and capacity are dictated by the logic required for the largest and slowest state. On average, the fully partitioned, multicontext implementation is 35-45% smaller that the single context implementation. Many FSMs are 3-5$\times$ smaller.

Timing From Tables 10.17 and 10.18, the multicontext FSM implementations generally have one or two fewer logic levels in their critical path than the single context implementations when mapped for minimum latency. The multicontext implementations have an even greater reduction in path length when mapped for minimum area. The multicontext FSMs, however, require additional time to distribute the context select and perform the multi-context read. *i.e.*

$$T_{single-context} = (\text{Levels}) \cdot t_{lut} \tag{10.6}$$

$$T_{multi-context} = (\text{Levels}) \cdot t_{lut} + t_{distribute} + t_{ctx-switch} \tag{10.7}$$

Recall $t_{ctx-switch} = 2.5$ ns from the prototype and $t_{lut} = 6$ ns with a typical amount of switching. Properly engineered, context distribution should take a few nanoseconds, which means the multicontext and single-context implementations run at comparable speeds when the multicontext implementation has one fewer LUT delays in its critical path than the single-context implementation.

10.6.3 Partial Temporal Partitioning

The capacity utilization and delay are often dictated by a few of the more complex states. It is often possible to reduce the number of contexts required without increasing the capacity required or increasing the delay. Tables 10.19 and 10.20 show the `cse` benchmark partitioned into various numbers of contexts and optimized for area or path delay, respectively. These partitions were obtained by partitioning along `mustang` assigned state bits starting with a four bit state encoding. Figures 10.26 and 10.27 plot the LUT count, area, and delay data from the tables versus the number of contexts employed.

One thing we note from both the introductory example (Figures 10.24 and 10.25) and the `cse` example is that the full state-per-context case is not always the most area efficient mapping. In the introductory example, once we had partitioned to two contexts, no further LUT reduction could be realized by going to four contexts. Consequently, the four context implementation would be larger than the two context implementation owing to the deeper context memories. In the `cse` example, the reduction in LUTs associated with going to going from 8 to 11 or 11 to 16 contexts saved less area than the cost of the additional context memories.

FSM	States	Single Context			Context per State			**Ratio**	**Delta**
		Levels	N_{4LUT}	**Area** [Mλ^2]	Levels	N_{4LUT}	**Area** [Mλ^2]	$\frac{A_{multi}}{A_{single}}$	**Levels**
bbara	10	6	25	22.0	1	6	9.5	0.43	5
bbsse	16	4	50	43.9	3	12	24.6	0.56	1
bbtas	6	3	7	6.1	1	5	6.34	1.0	2
beecount	7	4	14	12.3	1	7	9.4	0.77	3
cse	16	6	83	72.9	2	15	30.7	0.42	4
dk14	7	4	58	50.9	1	8	10.8	0.21	3
dk15	4	12	25	22.0	1	7	7.8	0.35	11
dk16	27	5	80	70.2	1	8	23.2	0.33	4
dk17	8	6	19	16.7	1	6	8.5	0.51	5
dk512	15	2	20	17.6	1	7	13.8	0.79	1
donfile	24	2	46	40.4	1	6	16.0	0.40	1
ex1	20	7	120	105.4	2	26	61.4	0.58	5
ex4	14	7	21	18.4	1	13	24.6	1.33	6
ex6	8	5	57	50.0	1	11	15.7	0.31	4
keyb	19	7	112	98.3	4	14	32.0	0.32	3
mc	4	2	8	7.0	1	7	7.8	1.10	1
modulo12	12	6	12	10.5	1	5	8.7	0.82	5
planet	48	6	150	131.7	1	25	113.6	0.86	5
pma	24	6	82	72.0	2	15	40.1	0.56	4
s1	20	5	137	120.3	5	25	59.0	0.49	0
s1488	48	6	152	133.5	3	27	122.7	0.92	3
s1a	20	5	72	63.2	7	21	49.6	0.78	-2
s208	18	4	38	33.4	1	7	15.4	0.46	3
s27	6	2	5	4.4	1	4	5.1	1.20	1
s386	13	5	42	36.9	2	12	21.8	0.59	3
s420	18	3	40	35.1	1	7	15.4	0.44	2
s510	47	5	54	47.4	1	13	58.1	1.22	4
s8	5	4	12	10.5	1	4	4.7	0.45	3
s820	25	6	92	80.8	3	30	82.5	1.02	3
sand	32	7	178	156.3	5	30	98.9	0.63	2
sse	16	4	50	43.9	3	12	24.6	0.56	1
styr	30	7	186	163.3	4	21	65.9	0.40	3
tbk	32	8	340	298.5	6	33	108.8	0.36	2
Average								0.64	3

Table 10.17: Full Partitioning of MCNC FSM Benchmarks (Area Target)

FSM	States	Single Context			Context per State			**Ratio**	**Delta**
		Levels	N_{4LUT}	**Area** [Mλ^2]	Levels	N_{4LUT}	**Area** [Mλ^2]	$\frac{A_{multi}}{A_{single}}$	**Levels**
bbara	10	3	40	35.1	1	6	9.5	0.27	2
bbsse	16	3	60	52.7	2	14	28.7	0.54	1
bbtas	6	2	9	7.9	1	5	6.3	0.80	1
beecount	7	2	19	16.7	1	7	9.4	0.57	1
cse	16	4	97	85.2	2	15	30.7	0.36	2
dk14	7	3	67	58.8	1	8	10.8	0.18	2
dk15	4	3	37	32.5	1	7	7.8	0.24	2
dk16	27	3	83	72.9	1	8	23.2	0.32	2
dk17	8	2	26	22.8	1	6	8.5	0.37	1
dk512	15	2	20	17.6	1	7	13.8	0.79	1
donfile	24	2	46	40.4	1	6	16.0	0.40	1
ex1	20	4	151	132.6	2	26	61.4	0.46	2
ex4	14	2	25	22.0	1	13	24.6	1.12	1
ex6	8	3	62	54.4	1	11	15.7	0.29	2
keyb	19	4	150	131.7	3	26	59.3	0.45	1
mc	4	2	8	7.0	1	7	7.8	1.10	1
modulo12	12	1	13	11.4	1	5	8.7	0.76	0
planet	48	4	172	151.0	1	25	113.6	0.75	3
pma	24	4	139	122.0	2	15	40.1	0.33	2
s1	20	4	195	171.2	3	30	70.8	0.41	1
s1488	48	4	183	160.7	2	28	127.2	0.79	2
s1a	20	3	107	93.9	4	30	70.8	0.75	-1
s208	18	3	40	35.1	1	7	15.4	0.44	2
s27	6	2	5	4.4	1	4	5.0	1.16	1
s386	13	4	54	47.4	2	12	21.8	0.46	2
s420	18	3	40	35.1	1	7	15.4	0.44	2
s510	47	3	76	66.7	1	13	58.1	0.87	2
s8	5	2	13	11.4	1	4	4.8	0.42	1
s820	25	3	137	120.3	3	30	82.5	0.69	0
sand	32	4	224	196.7	3	43	141.7	0.72	1
sse	16	3	60	52.7	2	14	28.7	0.54	1
styr	30	5	285	250.2	3	23	72.2	0.29	2
tbk	32	5	510	447.8	4	42	138.4	0.31	1
Average								0.56	1.36

Table 10.18: Full Partitioning of MCNC FSM Benchmarks (Delay Target)

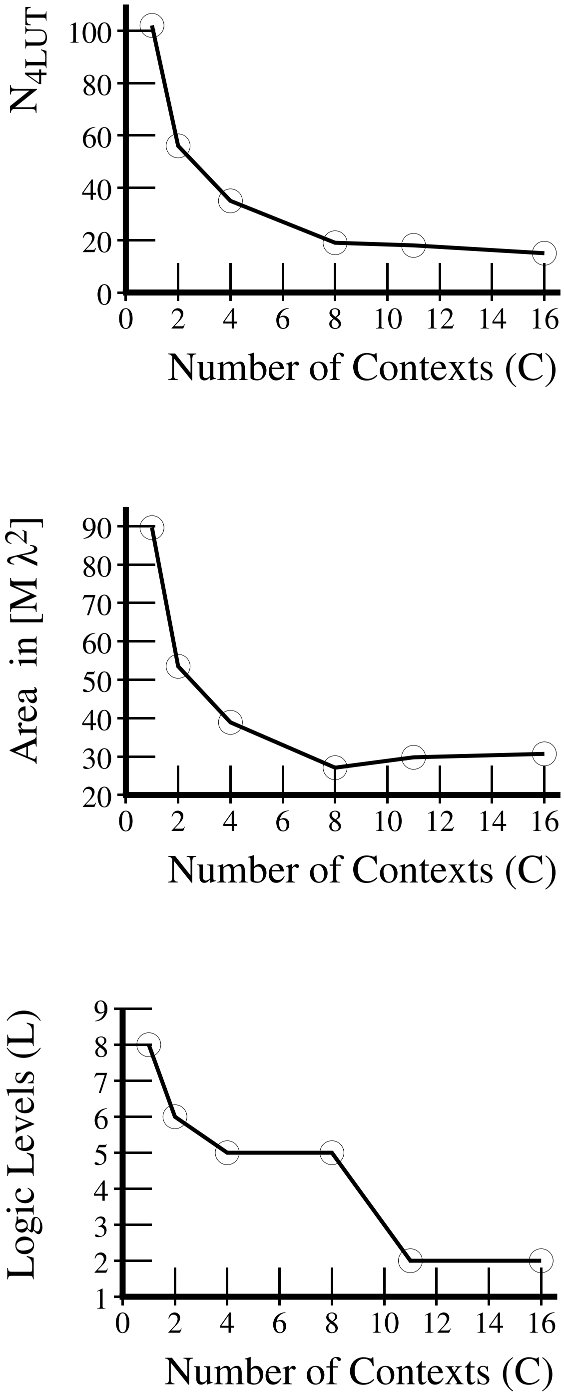


Figure 10.26: Area and Delay versus Number of Contexts for `cse` FSM Benchmark (Area Target)

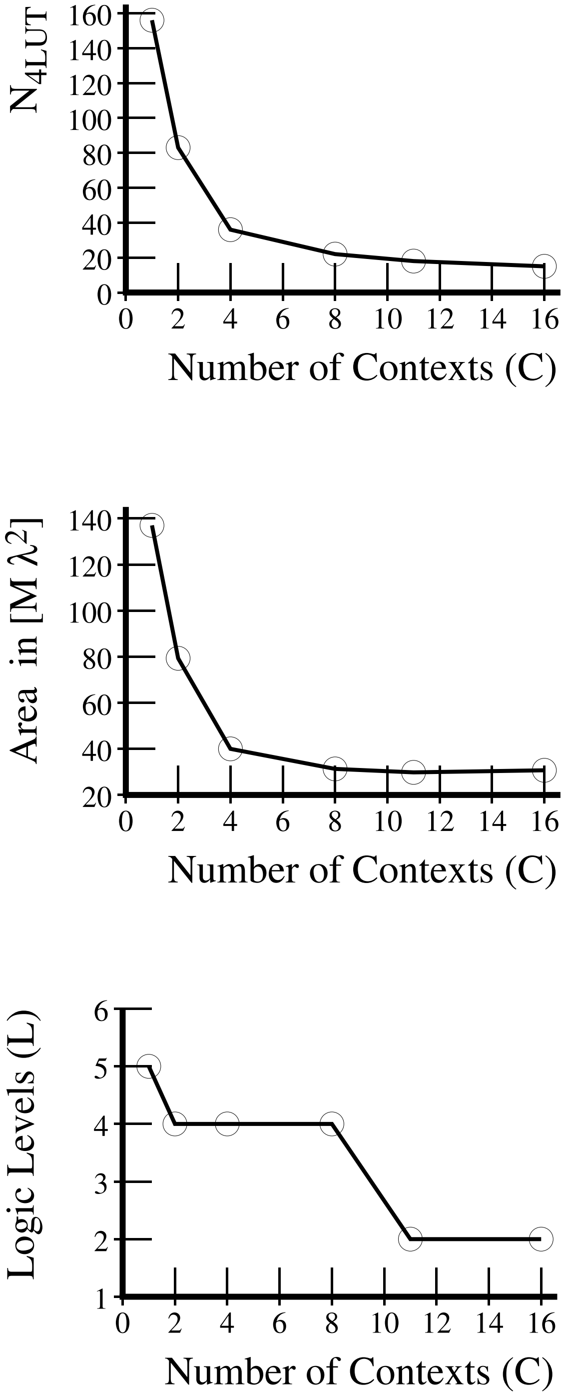


Figure 10.27: Area and Delay versus Number of Contexts for `cse` FSM Benchmark (Delay Target)

Multicontext Implementations for CSE FSM					
Contexts	Levels	N_{4LUT}	**Area**	$\frac{A_{mc}}{A_{single}}$	**Delta**
(one-hot)					
1	6	83	72.9	1.00	0
(dense)					
1	8	102	89.6	1.23	-2
2	6	56	53.5	0.73	0
4	5	35	38.9	0.53	1
8	5	19	27.1	0.37	1
11	2	18	29.8	0.41	4
16	2	15	30.7	0.42	4

Table 10.19: Area and Delay versus Number of Contexts for `cse` FSM Benchmark (Area Target)

Multicontext Implementations for CSE FSM					
Contexts	Levels	N_{4LUT}	**Area**	$\frac{A_{mc}}{A_{single}}$	**Delta**
(one-hot)					
1	4	97	85.2	1.00	0
(dense)					
1	5	156	137.0	1.60	-1
2	4	83	79.3	0.93	0
4	4	36	40.0	0.47	0
8	4	22	31.3	0.37	0
11	2	18	29.8	0.35	2
16	2	15	30.7	0.36	2

Table 10.20: Area and Delay versus Number of Contexts for `cse` FSM Benchmark (Delay Target)

Tables 10.21 through 10.25 show the benchmark set mapped to various multicontext implementations for minimum area. All partitioning is performed along `mustang` state bits. For these results, we examined all possible state bits along which to split and chose the best set. On average across the benchmark set, the 8-context mapping saves over 40% in area versus the best single-context case. The best multicontext mapping is often 3-5$\times$ smaller than the best single context mapping.

FSM	States	**Best Single Context**	**LUTs by Number of Context** Dense Encodings						
			1	2	4	8	16	32	64
bbara	10	25	25	19	12	8	6	6	6
bbsse	16	50	68	39	20	15	12	12	12
bbtas	6	7	7	5	5	5	5	5	5
beecount	7	14	14	11	7	7	7	7	7
cse	16	83	102	56	35	19	15	15	15
dk14	7	58	58	22	8	8	8	8	8
dk15	4	25	25	7	7	7	7	7	7
dk16	27	80	162	57	27	8	8	8	8
dk17	8	19	19	6	6	6	6	6	6
dk512	15	20	21	7	7	7	7	7	7
donfile	24	46	162	57	31	6	6	6	6
ex1	20	120	193	85	59	39	31	26	26
ex4	14	21	21	20	14	13	13	13	13
ex6	8	57	83	31	17	11	11	11	11
keyb	19	112	173	65	37	22	17	14	14
mc	4	8	8	7	7	7	7	7	7
modulo12	12	12	12	5	5	5	5	5	5
planet	48	150	346	122	80	38	29	27	25
pma	24	82	82	78	45	24	18	15	15
s1	20	137	196	144	57	44	28	25	25
s1488	48	152	305	153	129	52	34	28	27
s1a	20	72	136	73	51	38	24	21	21
s208	18	38	55	35	14	9	8	7	7
s27	6	5	5	5	4	4	4	4	4
s386	13	42	64	35	19	13	12	12	12
s420	18	40	54	33	15	10	8	7	7
s510	47	54	133	95	35	18	13	13	13
s8	5	12	12	13	9	4	4	4	4
s820	25	92	245	92	62	45	32	30	30
sand	32	178	358	139	95	44	33	30	30
sse	16	50	68	39	20	15	12	12	12
styr	30	186	387	133	67	40	28	21	21
tbk	32	340	513	137	75	48	35	33	33

Table 10.21: MCNC FSM Benchmarks LUTs v/s Number of Contexts (Area Target)

FSM	States	**Best Single Context**	**Area [Mλ^2] by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	21.9	21.9	18.2	13.3	11.4	12.3	19.8	34.8
bbsse	16	43.9	59.7	37.3	22.2	21.4	24.6	39.6	69.5
bbtas	6	6.1	6.1	4.8	5.6	7.1	10.2	16.5	29.0
beecount	7	12.3	12.3	10.5	7.8	10.0	14.3	23.1	40.5
cse	16	72.9	89.6	53.5	38.9	27.1	30.7	49.4	86.9
dk14	7	50.9	50.9	21.0	8.9	11.4	16.4	26.4	46.3
dk15	4	21.9	21.9	6.7	7.8	10.0	14.3	23.1	40.5
dk16	27	70.2	142.2	54.5	30.0	11.4	16.4	26.4	46.3
dk17	8	16.7	16.7	5.7	6.7	8.5	12.3	19.8	34.8
dk512	15	17.6	18.4	6.7	7.8	10.0	14.3	23.1	40.5
donfile	24	40.4	142.2	54.5	34.5	8.5	12.3	19.8	34.8
ex1	20	105.4	169.5	81.3	65.6	55.5	63.5	85.7	150.6
ex4	14	18.4	18.4	19.1	15.6	18.5	26.6	42.8	75.3
ex6	8	50.0	72.9	29.6	18.9	15.7	22.5	36.3	63.7
keyb	19	98.3	151.9	62.1	41.1	31.3	34.8	46.1	81.1
mc	4	7.0	7.0	6.7	7.8	10.0	14.3	23.1	40.5
modulo12	12	10.5	10.5	4.8	5.6	7.1	10.2	16.5	29.0
planet	48	131.7	303.8	116.6	89.0	54.1	59.4	89.0	144.8
pma	24	72.0	72.0	74.6	50.0	34.2	36.9	49.4	86.9
s1	20	120.3	172.1	137.7	63.4	62.7	57.3	82.4	144.8
s1488	48	133.5	267.8	146.3	143.4	74.0	69.6	92.3	156.4
s1a	20	63.2	119.4	69.8	56.7	54.1	49.2	69.2	121.6
s208	18	33.4	48.3	33.5	15.6	12.8	16.4	23.1	40.5
s27	6	4.4	4.4	4.8	4.4	5.7	8.2	13.2	23.2
s386	13	36.9	56.2	33.5	21.1	18.5	24.6	39.6	69.5
s420	18	35.1	47.4	31.5	16.7	14.2	16.4	23.1	40.5
s510	47	47.4	116.8	90.8	38.9	25.6	26.6	42.8	75.3
s8	5	10.5	10.5	12.4	10.0	5.7	8.2	13.2	23.2
s820	25	80.8	215.1	88.0	68.9	64.1	65.5	98.9	173.8
sand	32	156.3	314.3	132.9	105.6	62.7	67.6	98.9	173.8
sse	16	43.9	59.7	37.3	22.2	21.4	24.6	39.6	69.5
styr	30	163.3	339.8	127.1	74.5	57.0	57.3	69.2	121.6
tbk	32	298.5	450.4	131.0	83.4	68.4	71.7	108.8	191.1

Table 10.22: MCNC FSM Benchmarks Area v/s Number of Contexts (Area Target)

FSM	States	**Best Single Context**	**Delay by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	6	6	4	3	3	1	1	1
bbsse	16	4	6	6	4	3	3	3	3
bbtas	6	3	3	1	1	1	1	1	1
beecount	7	4	4	4	1	1	1	1	1
cse	16	6	8	6	5	5	2	2	2
dk14	7	4	4	4	1	1	1	1	1
dk15	4	12	12	1	1	1	1	1	1
dk16	27	5	4	5	8	1	1	1	1
dk17	8	6	6	1	1	1	1	1	1
dk512	15	2	10	1	1	1	1	1	1
donfile	24	2	4	6	5	1	1	1	1
ex1	20	7	4	7	7	5	4	2	2
ex4	14	7	7	3	2	1	1	1	1
ex6	8	5	8	6	4	1	1	1	1
keyb	19	7	4	7	5	4	5	4	4
mc	4	2	2	1	1	1	1	1	1
modulo12	12	6	6	1	1	1	1	1	1
planet	48	6	4	6	5	5	4	2	1
pma	24	6	6	8	5	5	3	2	2
s1	20	5	5	5	6	5	4	5	5
s1488	48	6	9	7	5	5	5	3	3
s1a	20	5	5	6	5	5	5	7	7
s208	18	4	4	6	3	3	2	1	1
s27	6	2	2	2	1	1	1	1	1
s386	13	5	5	5	4	3	2	2	2
s420	18	3	5	5	4	3	2	1	1
s510	47	5	6	5	3	3	1	1	1
s8	5	4	4	3	3	1	1	1	1
s820	25	6	5	7	7	6	4	3	3
sand	32	7	5	7	6	6	5	5	5
sse	16	4	6	6	4	3	3	3	3
styr	30	7	5	7	7	6	6	4	4
tbk	32	8	6	11	10	7	7	6	6

Table 10.23: MCNC FSM Benchmarks Delay v/s Number of Contexts (Area Target)

FSM	States	**Best Single Context**	**Area Ratio by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	1.00	1.00	0.83	0.61	0.52	0.56	0.90	1.58
bbsse	16	1.00	1.36	0.85	0.51	0.49	0.56	0.90	1.58
bbtas	6	1.00	1.00	0.78	0.90	1.16	1.67	2.68	4.71
beecount	7	1.00	1.00	0.86	0.63	0.81	1.17	1.88	3.30
cse	16	1.00	1.23	0.73	0.53	0.37	0.42	0.68	1.19
dk14	7	1.00	1.00	0.41	0.17	0.22	0.32	0.52	0.91
dk15	4	1.00	1.00	0.30	0.35	0.45	0.65	1.05	1.85
dk16	27	1.00	2.02	0.78	0.43	0.16	0.23	0.38	0.66
dk17	8	1.00	1.00	0.34	0.40	0.51	0.74	1.19	2.08
dk512	15	1.00	1.05	0.38	0.44	0.57	0.82	1.31	2.31
donfile	24	1.00	3.52	1.35	0.85	0.21	0.30	0.49	0.86
ex1	20	1.00	1.61	0.77	0.62	0.53	0.60	0.81	1.43
ex4	14	1.00	1.00	1.04	0.84	1.00	1.44	2.32	4.08
ex6	8	1.00	1.46	0.59	0.38	0.31	0.45	0.72	1.27
keyb	19	1.00	1.54	0.63	0.42	0.32	0.35	0.47	0.82
mc	4	1.00	1.00	0.95	1.11	1.42	2.04	3.28	5.77
modulo12	12	1.00	1.00	0.45	0.53	0.68	0.97	1.56	2.75
planet	48	1.00	2.31	0.89	0.68	0.41	0.45	0.68	1.10
pma	24	1.00	1.00	1.04	0.70	0.47	0.51	0.69	1.21
s1	20	1.00	1.43	1.14	0.53	0.52	0.48	0.69	1.20
s1488	48	1.00	2.01	1.10	1.07	0.55	0.52	0.69	1.17
s1a	20	1.00	1.89	1.10	0.90	0.86	0.78	1.09	1.92
s208	18	1.00	1.45	1.00	0.47	0.38	0.49	0.69	1.22
s27	6	1.00	1.00	1.09	1.01	1.30	1.87	3.00	5.28
s386	13	1.00	1.52	0.91	0.57	0.50	0.67	1.07	1.88
s420	18	1.00	1.35	0.90	0.47	0.41	0.47	0.66	1.15
s510	47	1.00	2.46	1.92	0.82	0.54	0.56	0.90	1.59
s8	5	1.00	1.00	1.18	0.95	0.54	0.78	1.25	2.20
s820	25	1.00	2.66	1.09	0.85	0.79	0.81	1.22	2.15
sand	32	1.00	2.01	0.85	0.68	0.40	0.43	0.63	1.11
sse	16	1.00	1.36	0.85	0.51	0.49	0.56	0.90	1.58
styr	30	1.00	2.08	0.78	0.46	0.35	0.35	0.42	0.74
tbk	32	1.00	1.51	0.44	0.28	0.23	0.24	0.36	0.64
average		1.00	1.51	0.86	0.63	0.56	0.70	1.09	1.92

Table 10.24: MCNC FSM Benchmarks Area Ratio v/s Number of Contexts (Area Target)

FSM	States	**Best Single Context**	**Delay Reduction by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	0	0	2	3	3	5	5	5
bbsse	16	0	-2	-2	0	1	1	1	1
bbtas	6	0	0	2	2	2	2	2	2
beecount	7	0	0	0	3	3	3	3	3
cse	16	0	-2	0	1	1	4	4	4
dk14	7	0	0	0	3	3	3	3	3
dk15	4	0	0	11	11	11	11	11	11
dk16	27	0	1	0	-3	4	4	4	4
dk17	8	0	0	5	5	5	5	5	5
dk512	15	0	-8	1	1	1	1	1	1
donfile	24	0	-2	-4	-3	1	1	1	1
ex1	20	0	3	0	0	2	3	5	5
ex4	14	0	0	4	5	6	6	6	6
ex6	8	0	-3	-1	1	4	4	4	4
keyb	19	0	3	0	2	3	2	3	3
mc	4	0	0	1	1	1	1	1	1
modulo12	12	0	0	5	5	5	5	5	5
planet	48	0	2	0	1	1	2	4	5
pma	24	0	0	-2	1	1	3	4	4
s1	20	0	0	0	-1	0	1	0	0
s1488	48	0	-3	-1	1	1	1	3	3
s1a	20	0	0	-1	0	0	0	-2	-2
s208	18	0	0	-2	1	1	2	3	3
s27	6	0	0	0	1	1	1	1	1
s386	13	0	0	0	1	2	3	3	3
s420	18	0	-2	-2	-1	0	1	2	2
s510	47	0	-1	0	2	2	4	4	4
s8	5	0	0	1	1	3	3	3	3
s820	25	0	1	-1	-1	0	2	3	3
sand	32	0	2	0	1	1	2	2	2
sse	16	0	-2	-2	0	1	1	1	1
styr	30	0	2	0	0	1	1	3	3
tbk	32	0	2	-3	-2	1	1	2	2
average		0.00	-0.27	0.33	1.27	2.18	2.70	3.03	3.06

Table 10.25: MCNC FSM Benchmarks Delta Delay v/s Number of Contexts (Area Target)

Tables 10.21 through 10.25 show the benchmark set mapped to various numbers of context implementations with delay minimization as the target. All partitioning is performed along `mustang` state bits. For these results, we examined all possible state bits along which to split and chose the best set. On average across the benchmark set, the 8 context mapping are half the area of the best single-context case while achieving comparable delay.

FSM	States	**Best Single Context**	**Delay by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	3	4	3	3	3	1	1	1
bbsse	16	3	4	4	3	3	2	2	2
bbtas	6	2	3	1	1	1	1	1	1
beecount	7	2	3	2	1	1	1	1	1
cse	16	4	5	4	4	4	2	2	2
dk14	7	3	4	3	1	1	1	1	1
dk15	4	3	5	1	1	1	1	1	1
dk16	27	3	4	4	5	1	1	1	1
dk17	8	2	4	1	1	1	1	1	1
dk512	15	2	5	1	1	1	1	1	1
donfile	24	2	4	5	5	1	1	1	1
ex1	20	4	4	4	4	3	3	2	2
ex4	14	2	3	3	2	1	1	1	1
ex6	8	3	4	4	3	1	1	1	1
keyb	19	4	4	6	5	4	4	3	3
mc	4	2	2	1	1	1	1	1	1
modulo12	12	1	3	1	1	1	1	1	1
planet	48	4	4	5	3	3	3	2	1
pma	24	4	4	5	4	3	3	2	2
s1	20	4	5	5	4	4	3	3	3
s1488	48	4	5	4	7	4	3	3	2
s1a	20	3	5	6	5	5	4	4	4
s208	18	3	4	5	3	2	2	1	1
s27	6	2	2	2	1	1	1	1	1
s386	13	4	4	4	3	2	2	2	2
s420	18	3	5	5	3	2	2	1	1
s510	47	3	4	4	3	2	1	1	1
s8	5	2	2	3	3	1	1	1	1
s820	25	3	5	4	4	4	3	3	3
sand	32	4	5	5	4	4	4	3	3
sse	16	3	4	4	3	3	2	2	2
styr	30	5	5	4	4	4	4	3	3
tbk	32	5	6	7	6	6	5	4	4

Table 10.26: MCNC FSM Benchmarks Delay v/s Number of Contexts (Delay Target)

FSM	States	**Best Single Context**	**LUTs by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	40	33	24	12	8	6	6	6
bbsse	16	60	85	53	24	15	14	14	14
bbtas	6	9	7	5	5	5	5	5	5
beecount	7	19	18	12	7	7	7	7	7
cse	16	97	156	83	36	22	15	15	15
dk14	7	67	58	26	8	8	8	8	8
dk15	4	37	38	7	7	7	7	7	7
dk16	27	83	162	82	35	8	8	8	8
dk17	8	26	31	6	6	6	6	6	6
dk512	15	20	52	7	7	7	7	7	7
donfile	24	46	162	199	31	6	6	6	6
ex1	20	151	193	136	80	47	32	26	26
ex4	14	25	28	20	14	13	13	13	13
ex6	8	62	97	42	18	11	11	11	11
keyb	19	150	173	202	37	22	22	26	26
mc	4	8	8	7	7	7	7	7	7
modulo12	12	13	21	5	5	5	5	5	5
planet	48	172	346	202	93	38	30	27	25
pma	24	139	97	148	67	33	18	15	15
s1	20	195	196	144	80	56	40	30	30
s1488	48	183	455	264	99	57	35	28	28
s1a	20	107	136	73	51	38	33	30	30
s208	18	40	55	64	14	11	8	7	7
s27	6	5	5	5	4	4	4	4	4
s386	13	54	84	46	20	15	12	12	12
s420	18	40	54	33	20	11	8	7	7
s510	47	76	185	97	35	20	13	13	13
s8	5	13	13	13	9	4	4	4	4
s820	25	137	245	154	90	60	39	30	30
sand	32	224	358	219	130	67	61	43	43
sse	16	60	85	53	24	15	14	14	14
styr	30	285	387	211	129	60	34	23	23
tbk	32	510	513	676	353	71	52	42	42

Table 10.27: MCNC FSM Benchmarks LUTs v/s Number of Contexts (Delay Target)

FSM	States	**Best Single Context**	**Area [Mλ^2] by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	35.1	29.0	22.9	13.3	11.4	12.3	19.8	34.8
bbsse	16	52.7	74.6	50.7	26.7	21.4	28.7	46.1	81.1
bbtas	6	7.9	6.1	4.8	5.6	7.1	10.2	16.5	29.0
beecount	7	16.7	15.8	11.5	7.8	10.0	14.3	23.1	40.5
cse	16	85.2	137.0	79.3	40.0	31.3	30.7	49.4	86.9
dk14	7	58.8	50.9	24.9	8.9	11.4	16.4	26.4	46.3
dk15	4	32.5	33.4	6.7	7.8	10.0	14.3	23.1	40.5
dk16	27	72.9	142.2	78.4	38.9	11.4	16.4	26.4	46.3
dk17	8	22.8	27.2	5.7	6.7	8.5	12.3	19.8	34.8
dk512	15	17.6	45.7	6.7	7.8	10.0	14.3	23.1	40.5
donfile	24	40.4	142.2	190.2	34.5	8.5	12.3	19.8	34.8
ex1	20	132.6	169.5	130.0	89.0	66.9	65.5	85.7	150.6
ex4	14	21.9	24.6	19.1	15.6	18.5	26.6	42.8	75.3
ex6	8	54.4	85.2	40.2	20.0	15.7	22.5	36.3	63.7
keyb	19	131.7	151.9	193.1	41.1	31.3	45.1	85.7	150.6
mc	4	7.0	7.0	6.7	7.8	10.0	14.3	23.1	40.5
modulo12	12	11.4	18.4	4.8	5.6	7.1	10.2	16.5	29.0
planet	48	151.0	303.8	193.1	103.4	54.1	61.4	89.0	144.8
pma	24	122.0	85.2	141.5	74.5	47.0	36.9	49.4	86.9
s1	20	171.2	172.1	137.7	89.0	79.7	81.9	98.9	173.8
s1488	48	160.7	399.5	252.4	110.1	81.2	71.7	92.3	162.2
s1a	20	93.9	119.4	69.8	56.7	54.1	67.6	98.9	173.8
s208	18	35.1	48.3	61.2	15.6	15.7	16.4	23.1	40.5
s27	6	4.4	4.4	4.8	4.4	5.7	8.2	13.2	23.2
s386	13	47.4	73.8	44.0	22.2	21.4	24.6	39.6	69.5
s420	18	35.1	47.4	31.5	22.2	15.7	16.4	23.1	40.5
s510	47	66.7	162.4	92.7	38.9	28.5	26.6	42.8	75.3
s8	5	11.4	11.4	12.4	10.0	5.7	8.2	13.2	23.2
s820	25	120.3	215.1	147.2	100.1	85.4	79.9	98.9	173.8
sand	32	196.7	314.3	209.4	144.6	95.4	124.9	141.7	249.1
sse	16	52.7	74.6	50.7	26.7	21.4	28.7	46.1	81.1
styr	30	250.2	339.8	201.7	143.4	85.4	69.6	75.8	133.2
tbk	32	447.8	450.4	646.3	392.5	101.1	106.5	138.4	243.3

Table 10.28: MCNC FSM Benchmarks Area v/s Number of Contexts (Time Target)

FSM	States	**Best Single Context**	**Delay Reduction by Number of Context** Dense Encodings 1	2	4	8	16	32	64
bbara	10	0	0	0	0	0	0	0	0
bbsse	16	0	-1	-1	0	0	1	1	1
bbtas	6	0	-1	1	1	1	1	1	1
beecount	7	0	-1	0	1	1	1	1	1
cse	16	0	-1	0	0	0	2	2	2
dk14	7	0	-1	0	2	2	2	2	2
dk15	4	0	-2	2	2	2	2	2	2
dk16	27	0	0	0	0	0	0	0	0
dk17	8	0	-2	1	1	1	1	1	1
dk512	15	0	-3	1	1	1	1	1	1
donfile	24	0	-2	-3	-3	1	1	1	1
ex1	20	0	0	0	0	1	1	2	2
ex4	14	0	-1	-1	0	1	1	1	1
ex6	8	0	-1	-1	0	2	2	2	2
keyb	19	0	0	-2	-1	0	0	1	1
mc	4	0	0	1	1	1	1	1	1
modulo12	12	0	-2	0	0	0	0	0	0
planet	48	0	0	-1	1	1	1	2	3
pma	24	0	0	-1	0	1	1	2	2
s1	20	0	-1	-1	0	0	1	1	1
s1488	48	0	-1	0	-3	0	1	1	2
s1a	20	0	-2	-3	-2	-2	-1	-1	-1
s208	18	0	-1	-2	0	1	1	2	2
s27	6	0	0	0	1	1	1	1	1
s386	13	0	0	0	1	2	2	2	2
s420	18	0	-2	-2	0	1	1	2	2
s510	47	0	-1	-1	0	1	2	2	2
s8	5	0	0	-1	-1	1	1	1	1
s820	25	0	-2	-1	-1	-1	0	0	0
sand	32	0	-1	-1	0	0	0	1	1
sse	16	0	-1	-1	0	0	1	1	1
styr	30	0	0	1	1	1	1	2	2
average		0.00	-0.91	-0.48	0.06	0.64	0.91	1.15	1.21

Table 10.29: MCNC FSM Benchmarks Delta Delay v/s Number of Contexts (Delay Target)

FSM	States	**Best Single Context**	**Area Ratio by Number of Context** Dense Encodings						
			1	2	4	8	16	32	64
bbara	10	1.00	0.83	0.65	0.38	0.32	0.35	0.56	0.99
bbsse	16	1.00	1.42	0.96	0.51	0.41	0.54	0.88	1.54
bbtas	6	1.00	0.78	0.60	0.70	0.90	1.30	2.09	3.66
beecount	7	1.00	0.95	0.69	0.47	0.60	0.86	1.38	2.43
cse	16	1.00	1.61	0.93	0.47	0.37	0.36	0.58	1.02
dk14	7	1.00	0.87	0.42	0.15	0.19	0.28	0.45	0.79
dk15	4	1.00	1.03	0.21	0.24	0.31	0.44	0.71	1.25
dk16	27	1.00	1.95	1.08	0.53	0.16	0.22	0.36	0.64
dk17	8	1.00	1.19	0.25	0.29	0.37	0.54	0.87	1.52
dk512	15	1.00	2.60	0.38	0.44	0.57	0.82	1.31	2.31
donfile	24	1.00	3.52	4.71	0.85	0.21	0.30	0.49	0.86
ex1	20	1.00	1.28	0.98	0.67	0.50	0.49	0.65	1.14
ex4	14	1.00	1.12	0.87	0.71	0.84	1.21	1.95	3.43
ex6	8	1.00	1.56	0.74	0.37	0.29	0.41	0.67	1.17
keyb	19	1.00	1.15	1.47	0.31	0.24	0.34	0.65	1.14
mc	4	1.00	1.00	0.95	1.11	1.42	2.04	3.28	5.77
modulo12	12	1.00	1.62	0.42	0.49	0.62	0.90	1.44	2.54
planet	48	1.00	2.01	1.28	0.68	0.36	0.41	0.59	0.96
pma	24	1.00	0.70	1.16	0.61	0.39	0.30	0.41	0.71
s1	20	1.00	1.01	0.80	0.52	0.47	0.48	0.58	1.01
s1488	48	1.00	2.49	1.57	0.69	0.51	0.45	0.57	1.01
s1a	20	1.00	1.27	0.74	0.60	0.58	0.72	1.05	1.85
s208	18	1.00	1.38	1.74	0.44	0.45	0.47	0.66	1.15
s27	6	1.00	1.00	1.09	1.01	1.30	1.87	3.00	5.28
s386	13	1.00	1.56	0.93	0.47	0.45	0.52	0.83	1.47
s420	18	1.00	1.35	0.90	0.63	0.45	0.47	0.66	1.15
s510	47	1.00	2.43	1.39	0.58	0.43	0.40	0.64	1.13
s8	5	1.00	1.00	1.09	0.88	0.50	0.72	1.16	2.03
s820	25	1.00	1.79	1.22	0.83	0.71	0.66	0.82	1.44
sand	32	1.00	1.60	1.06	0.74	0.49	0.64	0.72	1.27
sse	16	1.00	1.42	0.96	0.51	0.41	0.54	0.88	1.54
styr	30	1.00	1.36	0.81	0.57	0.34	0.28	0.30	0.53
tbk	32	1.00	1.01	1.44	0.88	0.23	0.24	0.31	0.54
average		1.00	1.45	1.05	0.59	0.50	0.62	0.95	1.67

Table 10.30: MCNC FSM Benchmarks Area Ratio v/s Number of Contexts (Delay Target)

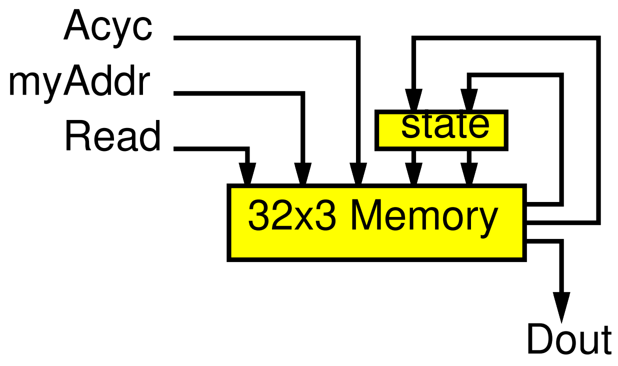


Figure 10.28: Memory-based Implementation for Simple FSM Example

10.6.4 Comparison with Memory-based FSM Implementations

A memory and a state register can also be used to implement finite-state machines. The data inputs and current state are packed together and used as addresses into the memory, and the memory outputs serve as machine outputs and next state outputs. Figure 10.28 shows a memory implementation of our simple FSM example from Figures 10.24 and 10.25.

Used for finite-state machines, the DPGA is a hybrid between a purely gate (FPGA) implementation and a purely memory implementation. The DPGA takes advantage of the memory to realize smaller, state-specific logic than an FPGA which must implement all logic simultaneously. The DPGA uses the restructurable interconnect in the array to implement next-state and output computations out of gates. As we noticed in Section 4.5, the gate implementation allow the DPGA to exploit regularities in the computational task. In this case, we avoid the necessarily exponential area increase associated with additional inputs in a memory implementation, the linear-log increase associated with additional states, and the linear increase associated with additional outputs.

Assuming we could build just the right sized memory for a given FSM, the area would be:

$$A_{memory-fsm} = \left(N_{state} \cdot 2^{N_{inputs}}\right) \times \left(N_{outputs} + \lceil \log_2 (N_{states}) \rceil\right) \times A_{mem\_cell}$$

Table 10.31 summarizes the areas of the best memory-based FSM implementations along with the areas for FPGA and 8-context DPGA implementations. The "Min area" column indicates the area assuming a memory of exactly the right size is used, while the "Memory Area" field shows the area for the smallest memory with an integral number of address bits as shown in the "organization" column. When the total number of state bits and input bits is less than 11, the optimal memory implementations can be much smaller than the FPGA or DPGA implementation. Above 11 input and state bits, the DPGA implementation is smaller. Since the DPGA implementation size increases with task complexity rather than number of inputs, while the memory implementation is exponential in the number of inputs and state bits, the disparity grows substantially as the number of inputs and state bits increase.

FSM	states	ins	outs	**Min area** [Mλ^2]	Integral Addr. & Data Organization	**Memory area** [Mλ^2]	**FPGA area** [Mλ^2]	**8-ctx DPGA area** [Mλ^2]
bbtas	6	2	2	0.1	$2^5\times5$	0.2	6.1	7.1
dk15	4	3	5	0.3	$2^5\times7$	0.3	21.9	10.0
dk17	8	2	3	0.2	$2^5\times6$	0.2	16.7	8.5
dk512	15	1	3	0.3	$2^5\times7$	0.3	17.6	10.0
mc	4	3	5	0.3	$2^5\times7$	0.3	7.0	10.0
modulo12	12	1	1	0.1	$2^5\times5$	0.2	10.5	7.1
beecount	7	3	4	0.5	$2^6\times7$	0.5	12.3	10.0
dk14	7	3	5	0.5	$2^6\times8$	0.6	50.9	11.4
dk16	27	2	3	1.0	$2^7\times8$	1.3	70.2	11.4
donfile	24	2	1	0.7	$2^7\times6$	0.9	40.4	8.5
s27	6	4	1	0.5	$2^7\times4$	0.6	4.4	5.7
s8	5	4	1	0.4	$2^7\times4$	0.6	10.5	5.7
bbara	10	4	2	1.2	$2^8\times6$	1.8	21.9	11.4
ex6	8	5	8	3.4	$2^8\times11$	3.4	50.0	15.7
ex4	14	6	9	14.0	$2^{10}\times13$	16.0	18.4	18.5
bbsse	16	7	7	27.0	$2^{11}\times11$	27.0	43.9	21.4
cse	16	7	7	27.0	$2^{11}\times11$	27.0	72.9	27.1
tbk	32	6	3	19.7	$2^{11}\times8$	19.7	298.5	68.4
sse	16	7	7	27.0	$2^{11}\times11$	27.0	43.9	21.4
s386	13	7	7	22.9	$2^{11}\times11$	27.0	36.9	18.5
keyb	19	7	2	20.4	$2^{12}\times7$	34.4	98.3	31.3
planet	48	7	19	184.3	$2^{13}\times25$	245.8	131.7	54.1
pma	24	8	8	95.8	$2^{13}\times13$	127.8	72.0	34.2
s1	20	8	6	67.6	$2^{13}\times11$	108.1	120.3	62.7
s1a	20	8	6	67.6	$2^{13}\times11$	108.1	63.2	54.1
ex1	20	9	19	294.9	$2^{14}\times24$	471.9	105.4	55.5
s1488	48	8	19	368.6	$2^{14}\times25$	491.5	133.5	74.0
styr	30	9	10	276.5	$2^{14}\times15$	294.9	163.3	57.0
s208	18	11	2	309.7	$2^{16}\times7$	550.5	33.4	12.8
sand	32	11	9	1101.0	$2^{16}\times14$	1101.0	156.3	62.7
s820	25	18	19	188743.7	$2^{23}\times24$	241591.9	80.8	64.1
s420	18	19	2	79272.3	$2^{24}\times7$	140928.6	35.1	14.2
s510	47	19	7	384408.9	$2^{25}\times13$	523449.1	47.4	25.6

N.b. – benchmarks reordered by the sum of the number of inputs and densely encoded state bits

Table 10.31: Memory Implementations for MCNC FSM Benchmarks

10.6.5 Areas for Improvement

Timing In this section, we assumed that the context read occurred in series with execution within the target context and state. It is possible to overlap context reads with execution by using a more sophisticated FSM mapping model. On state transition, instead of reading a context with the target state logic, we read a context with all the logic for any state which may *follow* the target state. This can be viewed as speculatively fetching just the set of logic which may be needed by the time it has been read a cycle later. Using this scheme, we can reduce the time to the actual delay in the context rather than the context delay plus the read time. For heavily branching FSMs, the target logic will often have to include more state logic per context than with this style of mapping than it was with the simple division described here. As we see here, including more state logic increases the delay so it is not immediately obvious which case will generally have superior performance.

Partitioning For the partial temporal partitioning above, we partitioned strictly along `mustang` state bits. This is likely to give less than optimal partitions since `mustang`'s cost model is aimed at multi-level logic implementations. It assumes all logic must be available at once and is not trying to maximize the independence among state groups. A more sophisticated mapping would go back to the original state-transition graph and partition states explicitly to minimize the logic required in each partition. Informally, the goal would be to group states with similar logic together and separate states performing disparate logical functions.

10.6.6 General Technique

While demonstrated in the contexts of FSMs, the basic technique used here is fairly general. When we can predict which portions of a netlist, circuit, or computational task are needed at a given point in time, we can generate a more specialized design which only includes the required logic. The specialized design is often smaller and faster than the fully general design. With a multicontext component, we can use the contexts to hold many specialized variants of a design, selecting them as needed.

In the synthesis of computations, it is common to synthesize a controller along with datapaths or computational elements. The computations required are generally different from state to state. Traditional, spatial implementations must have hardware to handle all the computations or generalize a common datapath to the point where it can be used for all the computations. Multicontext devices can segregate different computational elements into different contexts and select them only as needed.

For example, in both dbC [GM93] and PRISM-II [AWG94] a general datapath capable of handling all of the computational subtasks in computation is synthesized alongside the controller. At any point in time, however, only a small portion of the functionality contained in the datapath is actually needed and enabled by the controller. The multicontext implementation would be smaller by folding the disparate logic into memory and reusing the same active logic and interconnect to implement them as needed.

10.7 Additional Application Styles

10.7.1 Multifunction Components

With multiple, on-chip contexts, a device may be loaded with several different functions, any of which is immediately accessible with minimal overhead. A DPGA can thus act as a "multifunction peripheral," performing distinct tasks without idling for long reconfiguration intervals. In a system such as the one shown in Figure 10.7, a single device may perform several tasks. When used as a reconfigurable accelerator for a processor (*e.g.* [AS93] [DeH94] [RS94]) or to implement a dynamic processor (*e.g.* [WH95]), the DPGA can support multiple loaded acceleration functions simultaneously. The DPGA is more efficient in these allocations than single-context FPGAs because it allows rapid reuse of resources without paying the large idle-time overheads associated with reconfiguration from off-chip memory.

In a data storage or transmission application, for instance, one may be limited by the network or disk bandwidth. A single device may be loaded with functions to perform:

- (De)compression
- Cryptographic (*e.g.* DES) (de)encoding
- ECC Calculation, error detection, and correction

The device would be then called upon to perform the required tasks as needed.

Within a CAD application, such as `espresso` [RSV87], one needs to perform several distinct operations at different times, each of which could be accelerated with reconfigurable logic. We could load the DPGA with assist functions, such as:

- ASCII decoding (*e.g.* [Raz94])
- Bitvector manipulation
- Find first one (*e.g.* [AS93])
- Hamming distance calculation (*e.g.* [AS93])

Since these tasks are needed at distinct times, they can easily be stacked in separate contexts. Contexts are selected as the program needs these functions. To the extent that function usage is interleaved, the on-chip context configurations reduce the reload idle time which would be required to share a conventional device among as diverse a set of functions.

10.7.2 Utility Functions

Some classes of functionality are needed, occasionally but not continuously. In conventional systems, to get the functionality at all, we have to dedicate wire or gate capacity to such functions, even though they may be used very infrequently. A variety of data loading and unloading tasks fit into this "infrequent use" category, including:

- Data offload
  - Debugging snapshot
  - Testing observability
  - Fault recovery snapshot
  - Context data offload
- Data onload

- Configuration setting
- Value initialization
- Debugging value injection
- Testing accessibility
- Fault recovery
- Context data reload (after coarse-grain context switch)

- Operation idle/enable
  - Conditional operation
  - Exception handling
  - Stall

In a multicontext DPGA, the resources to handle these infrequent cases can be relegated to a separate context, or contexts, from the "normal" case code. The wires and control required to shift in (out) data and load it are allocated for use only when the respective utility context is selected. The operative circuitry then, does not contend with the utility circuitry for wiring channels or switches, and the utility functions do not complicate the operative logic. In this manner, the utility functions can exist without increasing critical path delay during operation.

A relaxation algorithm, for instance, might operate as follows:

1. Load in starting point and boundary conditions
2. Calculate relaxation updates
3. Check for convergence, return to 2 if not converged
4. Offload result

Each of these operations may be separate contexts. The relaxation computation may even be spread over several contexts. This general operation style, where inputs and outputs are distinct and infrequent phases of operation, is common for many kinds of operations (*e.g.* multi-round encryption, hashing, searching, and many optimization problems).

10.7.3 Temporally Systolic Computations

Figure 10.29 shows a typical video coding pipeline (*e.g.* [JOSV95]). In a conventional FPGA implementation, we would lay this pipeline out spatially, streaming data through the pipeline. If we needed the throughput capacity offered by the most heavily pipelined spatial implementation, that would be the design of choice. However, if we needed less throughput, the spatially pipelined version would require the same space while underutilizing the silicon. In this case, a DPGA implementation could stack the pipeline stages in time. The DPGA can execute a number of cycles on one pipeline function then switch to another context and execute a few cycles on the next pipeline function (See Figure 10.30). In this manner, the lower throughput requirement could be translated directly into lower device requirements.

This is the same basic organizational scheme used for levelized logic evaluation (Section 10.5.1). The primary difference being that evaluation levels are divided according to application subtasks.

This is a general schema with broad application. The pipeline design style is quite familiar and can be readily adapted for multicontext implementation. The amount of temporal pipelining can be varied as throughput requirements change or technology advances. As silicon feature sizes

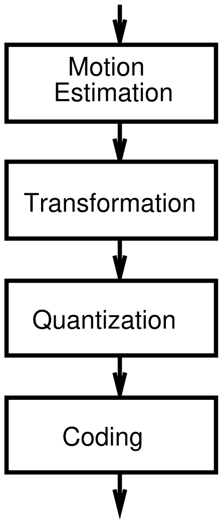


Figure 10.29: Canonical Video Coding Pipeline

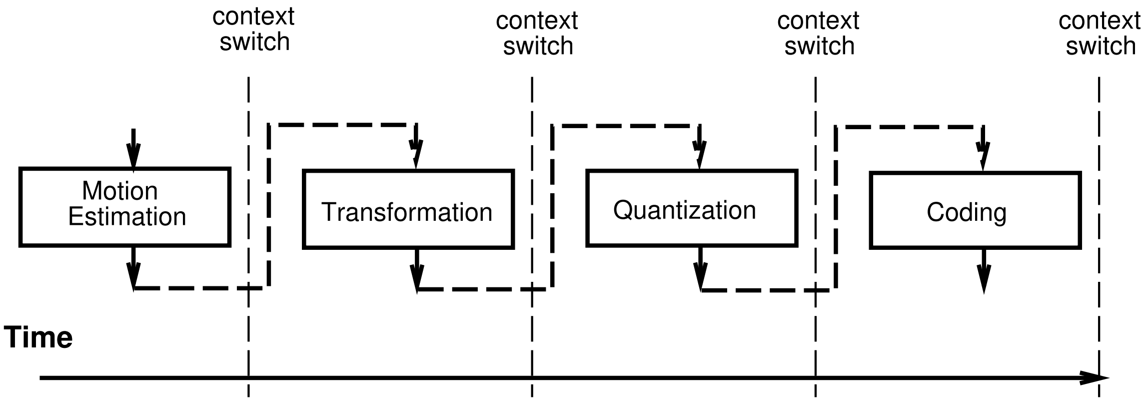


Figure 10.30: Temporally Systolic Video Coding Pipeline

shrink, primitive device bandwidth increases. Operations with fixed bandwidth requirements can increasingly be compressed into more temporal and less spatial evaluation.

When temporally pipelining functions, the data flowing between functional blocks can be transmitted in time rather than space. This saves routing resources by bringing the functionality to the data rather than routing the data to the required functional block. By packing more functions onto a chip, temporal pipelining can also help avoid component I/O bandwidth bottlenecks between function blocks.

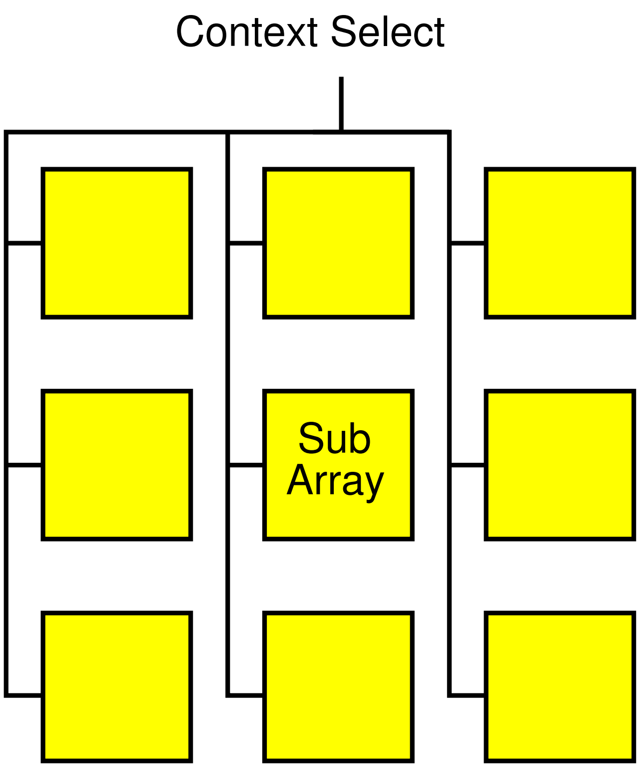


Figure 10.31: Control Distribution on DPGA Prototype

10.8 Control

In the prototype DPGA (Section 10.4), we had a single, array-wide control thread, the context select line, which was driven from off-chip. In general, as we noted in Chapter 8, the array may be segregated into regions controlled by a number of distinct control threads. Further, in many applications it will be beneficial to control execution on-chip – perhaps even from portions of the array, itself.

10.8.1 Segregation

In the prototype *subarrays* were used to organize local interconnect. The subarray level of array decomposition can also be used to segregate independent control domains. As shown in Figure 10.31, the context select lines were simply buffered and routed to each subarray. The actual decoding to control memories in the prototype occured in the **local decode** block. We can control the subarrays independently by providing distinct sets of control lines for each subarrays, or groups of subarrays.

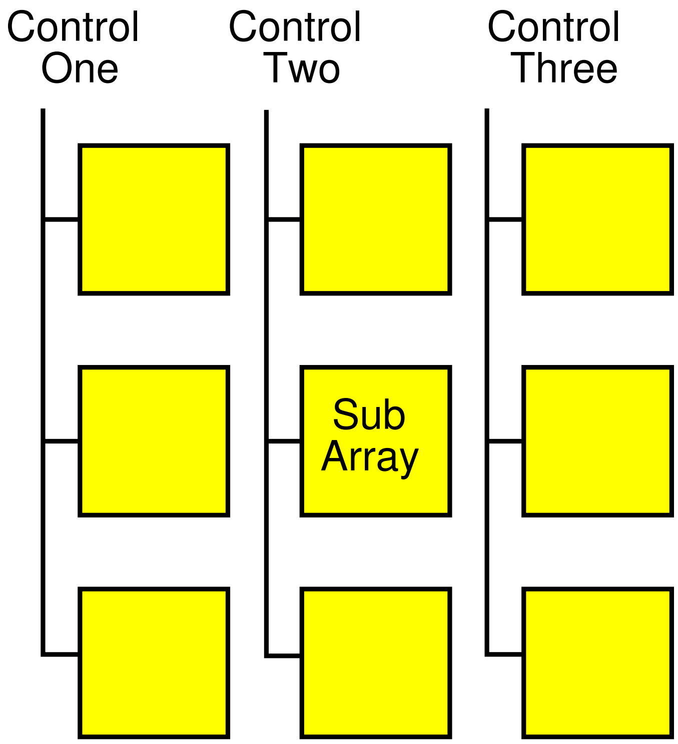


Figure 10.32: Multiple Controllers – Hardwired Control

10.8.2 Distribution

Hardwired Control In the simplest case, separate control streams can be physically assigned to each subarray or subarray group. For example, Figure 10.32 shows a 3×3 subarray design with a separate control stream for each column.

Configurable Control Alternately, multiple control streams can be physically routed to each subarray with local configuration used to select the appropriate one for use. For example, Figure 10.33 shows a 3×3 subarray design with three controllers where each subarray can be configured to select any of the three control streams. The configurable control may even be integrated with the configurable interconnection network.

Metaconfiguration In scenarios where array control can be configured it will often be necessary to have a separate level of configuration from the array itself. This *meta*-level configuration is used to define the sources for control data and perhaps control distribution paths. It does not change from cycle-to-cycle as does regular array configuration data. The MATRIX design described in Chapter 13 deals explicitly with this kind of a multi-level configuration scheme.

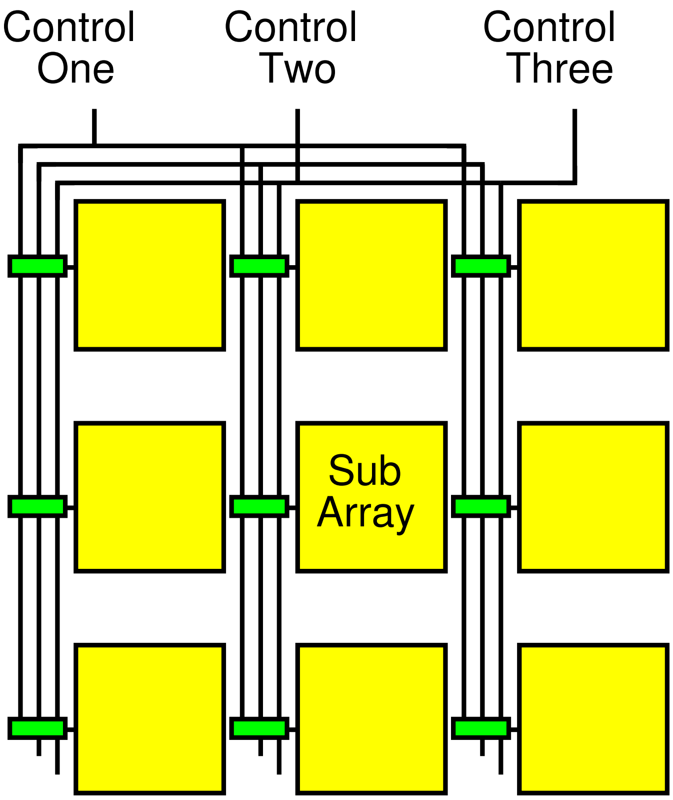


Figure 10.33: Multiple Controllers – Configurable Control

10.8.3 Source

Off-Chip The control stream can be sourced from off-chip, as in the prototype, providing considerable flexibility to the application or system. Off-chip control, however, implies additional latency in the control path and the additional cost of a separate controller component. It also requires precious i/o pins be dedicated to control rather than data. Tasks which benefit from rapid feedback between data in the computation stream and the control stream are hindered by the data→control path which most cross first off-chip then back on-chip.

Local Dedicated Controller A dedicated, programmable controller can be integrated on-chip to manage the control stream. The controller could come in the form of a simple counter, a programmable PLA, a basic microcontroller, or a core microprocessor. Integrated on-chip, it has low latency and high bandwidth to the array and avoids consuming i/o pins. In order to integrate such controllers on chip, we must decide how much space to dedicate to them, how many separate controllers to provide, and what form the controller will take. Recall from Section 8.5, that we would like to match the number of control streams with the needs of the application, but we cannot

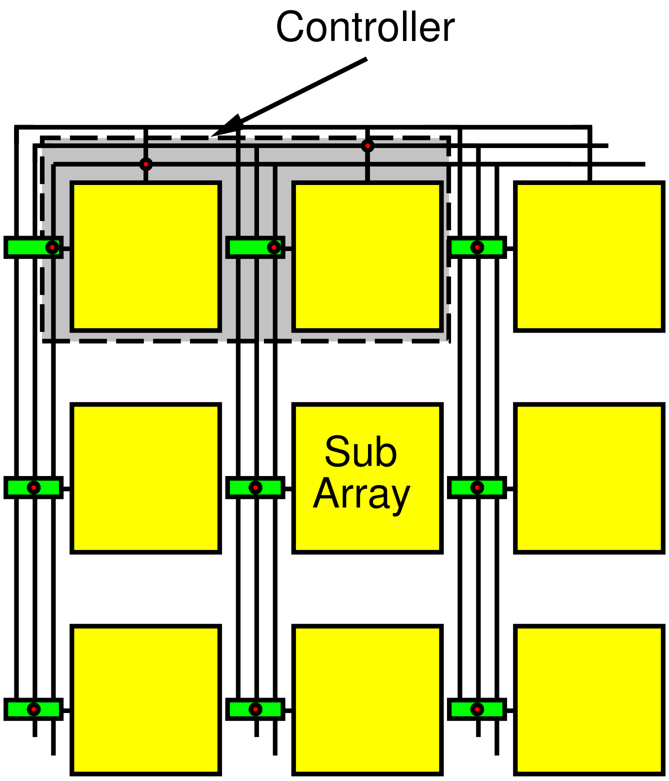


Figure 10.34: Array Self Control Example

do that if the controllers must be allocated prior to fabrication.

Feedback Self Control For mapped FSMs (Section 10.6), we saw that it was beneficial to route some of the design outputs back into the control port (*e.g.* Figure 10.24). As noted above, this entails some integration of the reconfigurable network and the control distribution path.

Self Control We can also build the controller out of FPGA/DPGA logic. The controller generally implements an FSM. It would be plausible, then to allocate one or more subarrays to build a controller which is, in turn, used to control the other subarrays on the component. With this scheme, we can partition the array and build just as many controllers as are required for the task at hand. Figure 10.34 shows a case where two subarrays are used to build the controller which is then responsible for controlling the rest of the array.

10.9 Conclusions

Conventional FPGAs underutilize silicon in most situations. When throughput requirements are limited, task latency is important, or when the computation required varies with time or the data being processed, conventional designs leave much of the active LUTs and interconnect idle for most of the time. Since the area to hold a configuration is small compared to the active LUT and interconnect area, we can generally meet task throughput or latency requirements with less implementation area using a multicontext component.

In this section we introduced the DPGA, a bit-level computational array which stores several instructions or configurations along with each active computing element. We described a complete prototype of the architecture and some elementary design automation schemes for mapping traditional circuits and FSMs onto these multicontext architectures.

We showed how to automatically map conventional tasks onto these multicontext devices. For latency limited and low throughput tasks, a 4-context DPGA implementation is, on average, 20-40% smaller than an FPGA implementation. For FSMs, the 4-context DPGA implementation is generally 30-40% smaller than the FPGA implementation, while the 8-context DPGA implementation is 40-50% smaller. Signal retiming requirements are the primary limitation which prevents the DPGA from realizing greater savings on circuit benchmarks, so it is worthwhile to consider architectural changes to support retiming in a less expensive manner. We will look at one such modification in the next chapter. All of these results are based on a context-memory area to active compute and interconnect area ratio of 1:10. The smaller the context memories can be implemented relative to the fixed logic, the greater the reduction in implementation area we can achieve with multicontext devices.

For hand-mapped designs or coarse-grain interleaving, the potential area savings is much greater. With the 1:10 context memory to active ratio, the 4-context DPGA can be one-third the size of an FPGA supporting the same number of LUTs. An 8-context DPGA can be 20% of the size of the FPGA. Several of the automatically mapped FSM examples come close to achieving these area reductions.

11. Dynamically Programmable Gate Arrays with Input Registers

In Chapter 10 we noticed that retiming requirements often prevented us from realizing as significant a reduction in active LUTs as should be possible. As a result of retiming, we often had to dedicate active LUTs simply to pass data through intermediate contexts. Retiming requirments also created a saturation level below which no further reduction in active LUTs was possible even if we were willing to take more time or add more context memories.

In this chapter we introduce input registers to the simple DPGA model used in the previous chapter. These input registers allow us to store values which need to traverse LUT evaluation levels in memories rather than having them consume active resources during the period of time which they are being retimed. This addition reduces the retiming limit we encountered in the previous chapter.

We introduce input registers to the base DPGA architecture (Section 11.1) and expand our computing device model accordingly (Section 11.2). Section 11.3 provides a basic example of the benefits of adding input registers. We expand our experimental, multicontext mapping software from the previous chapter to handle input registers (Section 11.4) and examine the aggregate results of mapping circuit benchmarks to these devices. In Section 11.5, we briefly relate the input register model used in this chapter to potential alternatives. At the end of this chapter (Section 11.7) we review the key points about multicontext devices as developed over the last several chapters.

11.1 Input Registers

We established in Chapter 7 that most of the active area in conventional FPGAs goes into interconnect. When a signal must cross multiple succeeding contexts between the producer and the final consumer, in the existing model, we must dedicate precious, active routing resources to the signal for all intervening contexts. Note that this property is essentially true of single context FPGAs, as well. If a value is produced early in some critical path, but not consumed until several LUT delays later, the wires and switches between the producer and consumer are tied up holding the value for the entire time. Tying up switches and wires to transport a value in time is a poor use of a scarce resource.

The conventional model results from storing values in registers on the *output* of each computational element (See Figures 11.1 and 11.2). With this arrangement, we must hold the value on the output and *tie up switches and wires between the producer and the consumer* until such time as the final consumer has used the value. Since values are produced at different times, and several values from different sources must converge at a consuming LUT in order for it to produce its output value, this gives rise to the situation where switches and wires are forced to sit idle holding values for much longer than the time it takes for them to transport the values from their sources to their destinations.

The alternative is to move the value registers to the *inputs* of the computational elements (See Figure 11.3). In the simplest case, this means having four flip-flops on the input of each 4-LUT

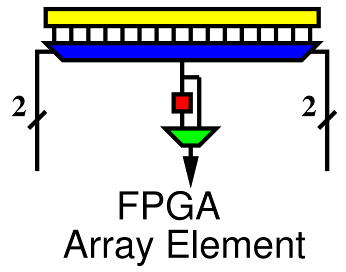


Figure 11.1: FPGA Array Element

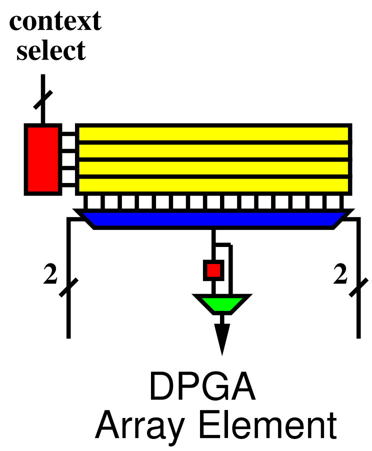


Figure 11.2: DPGA Array Element

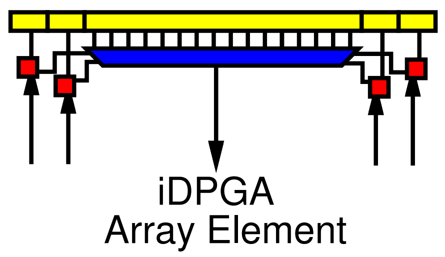


Figure 11.3: DPGA Array Element with Input Registers

rather than one flip-flop on the output. This modification allows us to move the data from the producer to consumer in the minimum transit time – a time independent of when the consumer will actually use the data. We now tie up space in a register to perform the retiming function rather than tying up all the wires and interconnect required to route the value from producers to consumers. Since the register can be much smaller than the intervening interconnect, this results in a tighter implementation.

Conceptually, the key idea here is that signal transport and retiming are two different functions:

1. **Spatial Transport** – moves data in space – route data from source to destination
2. **Temporal Transport** (Retiming) – moves data in time – make data available at some later time when it is actually required

By segregating the mechanisms we use for these two functions, we can optimize them independently and achieve a tighter implementation.

We can view this multicontext progression as successively relaxing the strict interconnect requirements for this class of devices:

- In a traditional, single-context FPGA we must have enough wires and switches to simultaneously route all the connections in the entire task description graph.
- In a prototype-style DPGA as described in the previous section, we must have enough LUT outputs, switches, and wires to carry one temporal slice through the computation.
- In a DPGA with input registers, in the extreme, we need only a single wire. More wires facilitate more parallelism in transport and hence higher throughput and lower latency implementations, but are not required for functionality.

11.2 iDPGA Model

A DPGA with input registers (iDPGA) associates an i-bit long shift register with each LUT input in addition to the c instructions per active LUT. The LUT instruction tells the LUT which of the i values on the shift register to actually select on each cycle. Each LUT input can thus retime a value by up to i cycles. That is, values may arrive at the destination LUT up to i clock cycles before they are consumed. Figure 11.4 shows a possible iDPGA array element with 4 contexts and an input register with depth 3.

The input registers do place a restriction on the grouping of logical LUTs into physical LUTs which was not present in the original DPGA. Multiple LUTs cannot have inputs arriving at the same input position on the same cycle. Fortunately, LUT input permutability often allows us to rearrange the inputs to avoid such potential conflicts. Nonetheless, the restriction does complicate LUT placement.

The additional resources required for this model are i-additional register cells for each input and one $i \times 1$ multiplexor for each input. For a k-LUT, the area then is:

$$\begin{aligned} A_{iDPGA\_LUT} &= A_{base} + c \cdot A_{context} + i \cdot k \cdot A_{shiftreg} + k \cdot A_{mux} \qquad (11.1) \\ A_{base} &= 800\text{K}\lambda^2 \end{aligned}$$

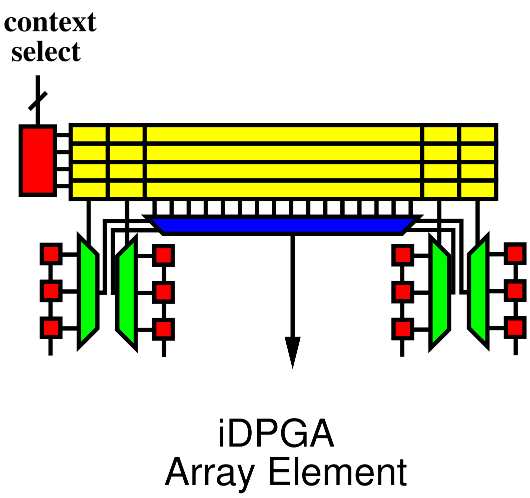


Figure 11.4: iDPGA Array Element $c = 4$, $i = 3$

$$\begin{aligned} A_{context} &= 78\text{K}\lambda^2 \\ A_{shiftreg} &= 4\text{K}\lambda^2 \\ A_{mux} &= i \cdot A_{SW} \\ A_{SW} &= 2.5\text{K}\lambda^2 \end{aligned}$$

Composing areas for a 4-LUT, we have:

$$\begin{aligned} A_{iDPGA\_4LUT} &= A_{base} + c \cdot A_{context} + i \cdot A_{input} \\ A_{base} &= 800\text{K}\lambda^2 \\ A_{context} &= 78\text{K}\lambda^2 \\ A_{input} &= 26\text{K}\lambda^2 \end{aligned} \tag{11.2}$$

Note here that we assume the total number of context description bits does not change. Rather, the bits that indicate which of the i inputs to select are bits which have been shuffled from spatial routing to temporal routing. That is, this scheme reduces the spatial interconnect requirements by performing temporal retiming in these registers. We are assuming that the bits are shuffled from one task to another without any significant change in the overall number of bits required.

11.3 Example

Recall from Section 10.1, that our ASCII$\rightarrow$hex binary circuit could be mapped to three contexts, but could not, viably, be mapped to fewer contexts. By adding the i-input register as suggested above, the active LUT requirements continue to decline with throughput reductions. Figure 11.5 shows this same circuit mapped with varying input register depth. As the number of input registers increases from 1 to 4, the saturation point reduces from 7 active LUTs to 4. Using our area model

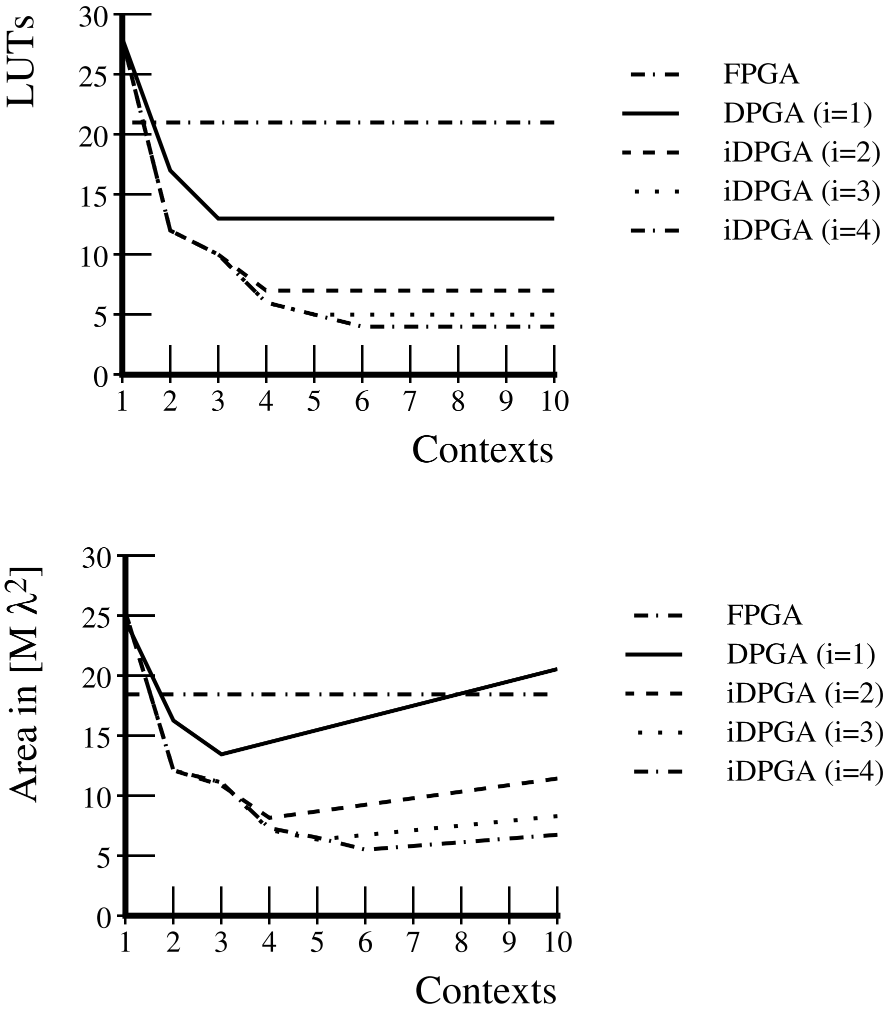


Figure 11.5: ASCII→Hex Binary Implementation versus Contexts and Input Register Depth

from the previous section, the $i = 4$, $c = 6$ iDPGA is 5.5Mλ^2, or over 3$\times$ smaller than the single context FPGA implementation at 18.4Mλ^2 and over 2$\times$ smaller than the smallest DPGA without input registers at 12.5Mλ^2.

11.4 Circuit Benchmarks: Input Depth

To examine the merits of input registers, we return to our throughput optimized circuit benchmarks as we originally visited in Section 10.5.3 for DPGAs. We use the same MCNC circuit benchmark set and the same input netlists as synthesized and mapped by `sis` and `Chortle`. Again, since we are assuming here that the target criteria is throughput, both `sis` and `Chortle` netlists were synthesized in area mode. As before, no modifications to the mapping and netlist generation are made.

11.4.1 Mapping

As before, we divide the multi-context case into separate spatial pipeline stages such that the path length between pipeline registers is equal to the acceptable period between results. The LUTs within a phase are then evaluated in multicontext fashion using the available contexts. The main difference from Section 10.5.3 is the cost metric for retiming. Since each LUT can retime up to i cycles, we only charge for retiming registers every i temporal stages between the original source and the final destination.

When we do need to place retiming registers, they are placed in a stylized fashion. Starting from the final consumer, we walk back through the circuit toward the primary inputs, placing a retiming repeater LUT every ith stage. In practice, we often have much more freedom in the placement of retiming registers, but this freedom was not exploited in our experimental mapping tools. During the annealing step, whenever the final consumer for a particular value is moved, the retiming chain is stripped out and replaced based on the consumer's new location.

After all levelization has been done, a grouping pass is performed. The grouping pass attempts to group together c logical LUTs within a spatial partition to reside on one physical LUT. For a group of LUTs to be compatible, it must be possible to permute the LUTs' inputs such that no two LUTs require a different value to arrive on the same input on the same clock cycle. Rather than trying all $(4!)^{(c-1)}$ permutations, we use a randomized, greedy placement scheme. We randomly pick which input in a LUT to place first, then greedily place it in a non-conflicting location. Other inputs within a LUT are placed sequentially after the initial random selection. The compatibility routine will make several attempts to find a satisfying assignment before declaring the grouping incompatible.

Grouping is performed independently on each spatial partition. The grouping routine starts by packing all the logical LUTs in a spatial partition into the minimum number of physical LUTs – *i.e.* the number of physical LUTs required to implement the largest temporal stage. The attempt is made by first randomly assigning logical LUTs to physical LUTs, then randomly selecting logical LUTs to swap in order to reduce incompatibility conflicts. Swaps which do not increase the incompatibilities in the grouping are greedily accepted. Swapping continues until a satisfying set of groupings is found or the swapping runs longer than a predetermined time limit which is proportional to the number of logical LUTs which can be described in the spatial partition. When packing fails, we increment the number of target physical LUTs and retry packing.

In review, circuit mapping proceeds through the following steps:

1. Technology Independent Optimization (`sis`)

2. LUT Mapping (`Chortle`)
3. Spatial and Temporal Levelization (simulated annealing)
4. Physical LUT Grouping (greedy swapping with heuristic compatibility verification)

`alu2` **at 4 clocks/result throughput**								
	LUTs by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
(1)	240	207	161	161	161	161	161	161
2		149	104	104	104	104	104	104
3			81	81	81	81	81	81
4				81	81	81	81	81
5					79	79	79	79
6						79	79	79
7							78	78
8								78

Table 11.1: Total Physial LUTs Required to Implement `alu2` Benchmark

`alu2` **at 4 clocks/result throughput**								
	Area in Mλ^2 by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
(1)	210.7	208.7	174.8	187.4	200.0	212.5	225.1	237.6
2		150.2	112.9	121.1	129.2	137.3	145.4	153.5
3			90.1	96.4	102.7	109.0	115.3	121.7
4				98.5	104.8	111.1	117.4	123.8
5					104.3	110.4	116.6	122.8
6						112.5	118.7	124.8
7							119.2	125.3
8								127.3

Table 11.2: Total Area Required to Implement `alu2` Benchmark

11.4.2 Detailed Example: `alu2`

Table 11.1 shows the total LUTs required after retiming and packing for the `alu2` benchmark mapped to provide a throughput of one result every four LUT delays. The table shows mappings for various values of i and c. We constrain $i \leq c$ in the current mapping software, so there are no configurations with $i > c$. Up to $i = 3$, we see that each additional input register allows us to further reduce the total number of physical LUTs required in the implementation. Table 11.2 uses the area model from Section 11.2 to translate the LUT counts into areas, and Table 11.3 shows the area savings versus a traditional FPGA implementation ($c = i = 1$). The $i = 3$, $c = 3$, iDPGA implementation is smallest at 43% of the area of the FPGA implementation.

Figure 11.6 shows the area of the family of `alu2` implementations as a function of context (c) and input (i) depth. Figure 11.7 plots the areas as ratios versus the FPGA implementation. The first

`alu2` at 4 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	0.991	0.830	0.889	0.949	1.009	1.068	1.128
2		0.713	0.536	0.575	0.613	0.652	0.690	0.729
3			0.428	0.458	0.487	0.517	0.547	0.578
4				0.467	0.497	0.527	0.557	0.588
5					0.495	0.524	0.553	0.583
6						0.534	0.563	0.592
7							0.566	0.595
8								0.604

Table 11.3: Area Ratios for `alu2` Benchmark Implementation

couple of input registers (i goes from $1 \rightarrow 2$ and $2 \rightarrow 3$) show significant gains for this benchmark. Gains diminish for greater input register depth. The best implementations are one-third the size of the FPGA implementation.

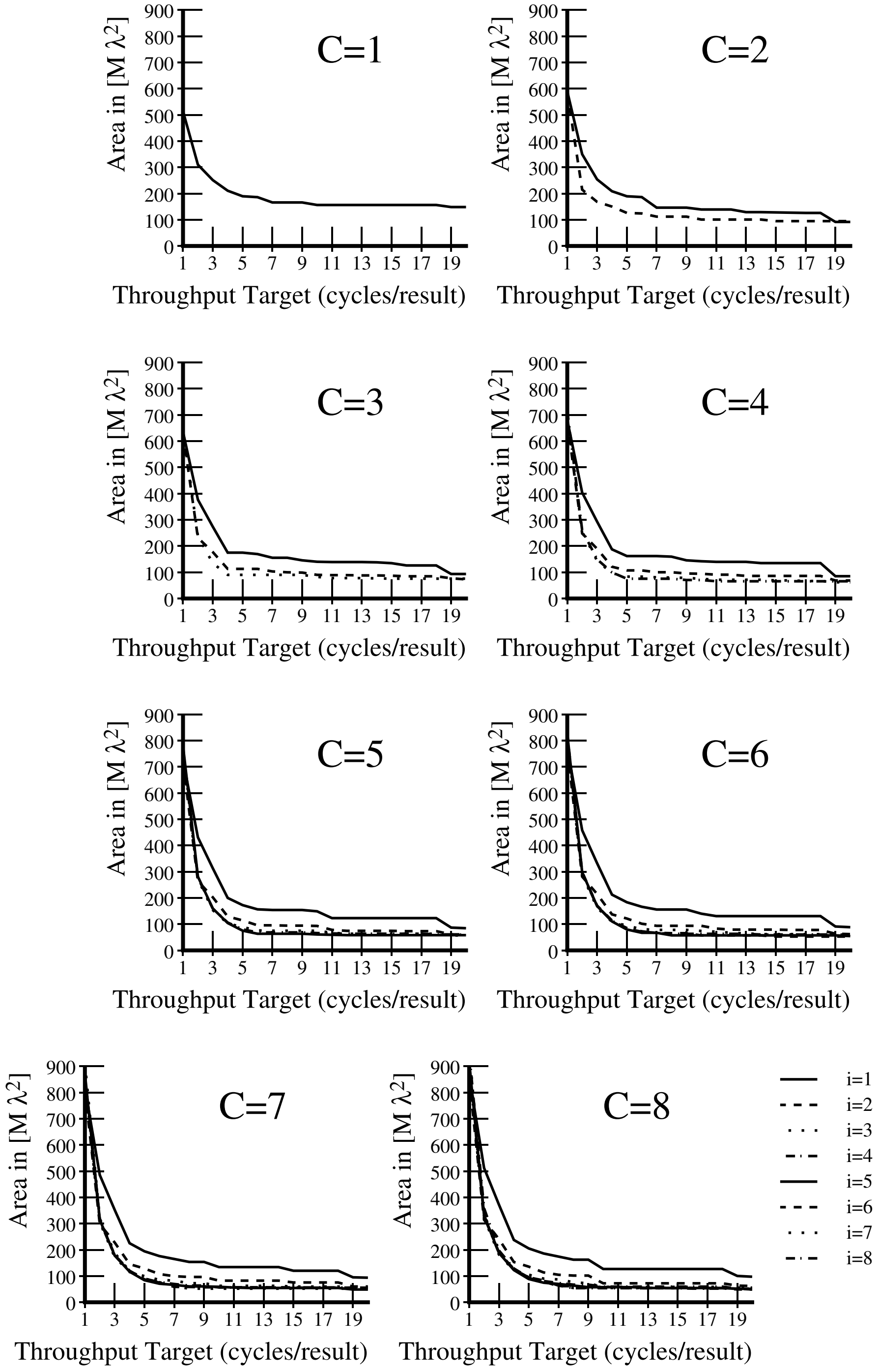


Figure 11.6: `alu2` Implementation Area versus Throughput

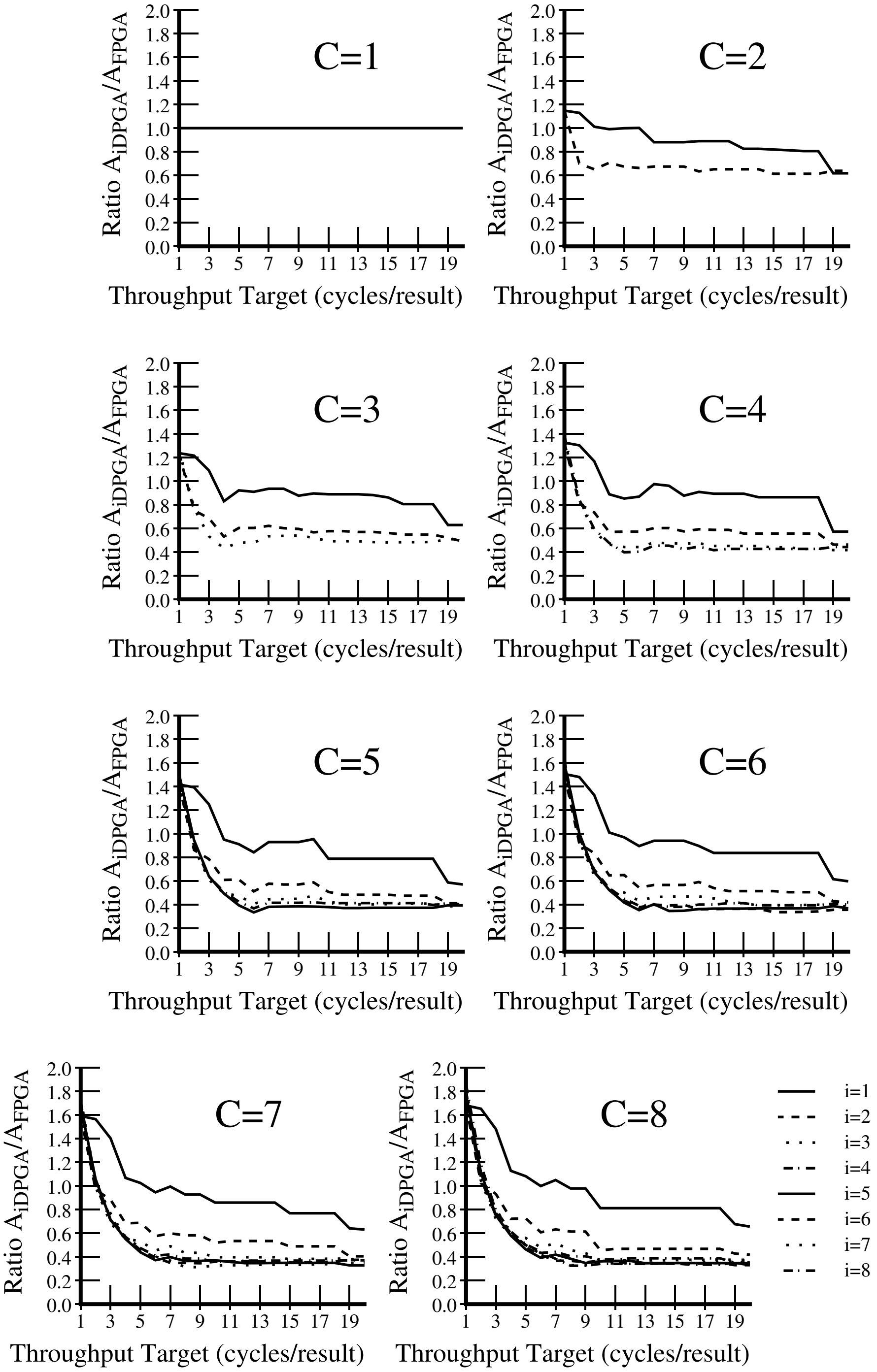


Figure 11.7: `alu2` Area Ratios versus Throughput

Average Ratio at 1 clock/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	1.148	1.237	1.326	1.415	1.503	1.592	1.681
2		1.148	1.237	1.326	1.415	1.503	1.592	1.681
3			1.267	1.355	1.444	1.533	1.622	1.711
4				1.385	1.474	1.563	1.651	1.740
5					1.503	1.592	1.681	1.770
6						1.622	1.711	1.800
7							1.740	1.829
8								1.859

Table 11.4: Average Ratios for Benchmark Set

11.4.3 Average Characteristics

Figure 11.8 shows the average area ratios across the entire benchmark set (See Table 10.13) analogously to Figure 11.7. We see here that an input register depth of four provides almost all of the benefits of input registers, with most of the benefit realized by a depth of three, as we saw with the `alu2` case in the previous section.

Figure 11.9 plots area versus throughput for various context depths (c), at a single values for input depth (i). Here, i was chosen to give the best results for low throughputs. For lower throughput values, the 5-8 context cases differ by only 10%. At the extreme of 20 clocks per result, the $c = 8$,$i = 6$ case is 33.7% the size of the single context case, versus the $c = 5$, $i = 4$ case which is 37.6%.

Tables 11.4 through 11.11 record implementation area ratio for all values of i and c. Each table reports implemenation areas for a different fixed throughput target in analog with Table 11.3. For the maximum throughput of one result per LUT delay, the traditional, single-context FPGA provides the best implementation. For all other cases, the multicontext implementations are always smaller than the single-context implementation. With a LUT-cycle delay in the 7-9.5 ns range, even today's "high" throughput implementations in the 30-50 MHz range are producing new results only once ever 3-5 LUT delays. At these speeds 3-4 context devices are 40-50% smaller than the single context implementation. At lower throughputs, the multiple context implementations are almost one-third the size of the single-context implementation on average.

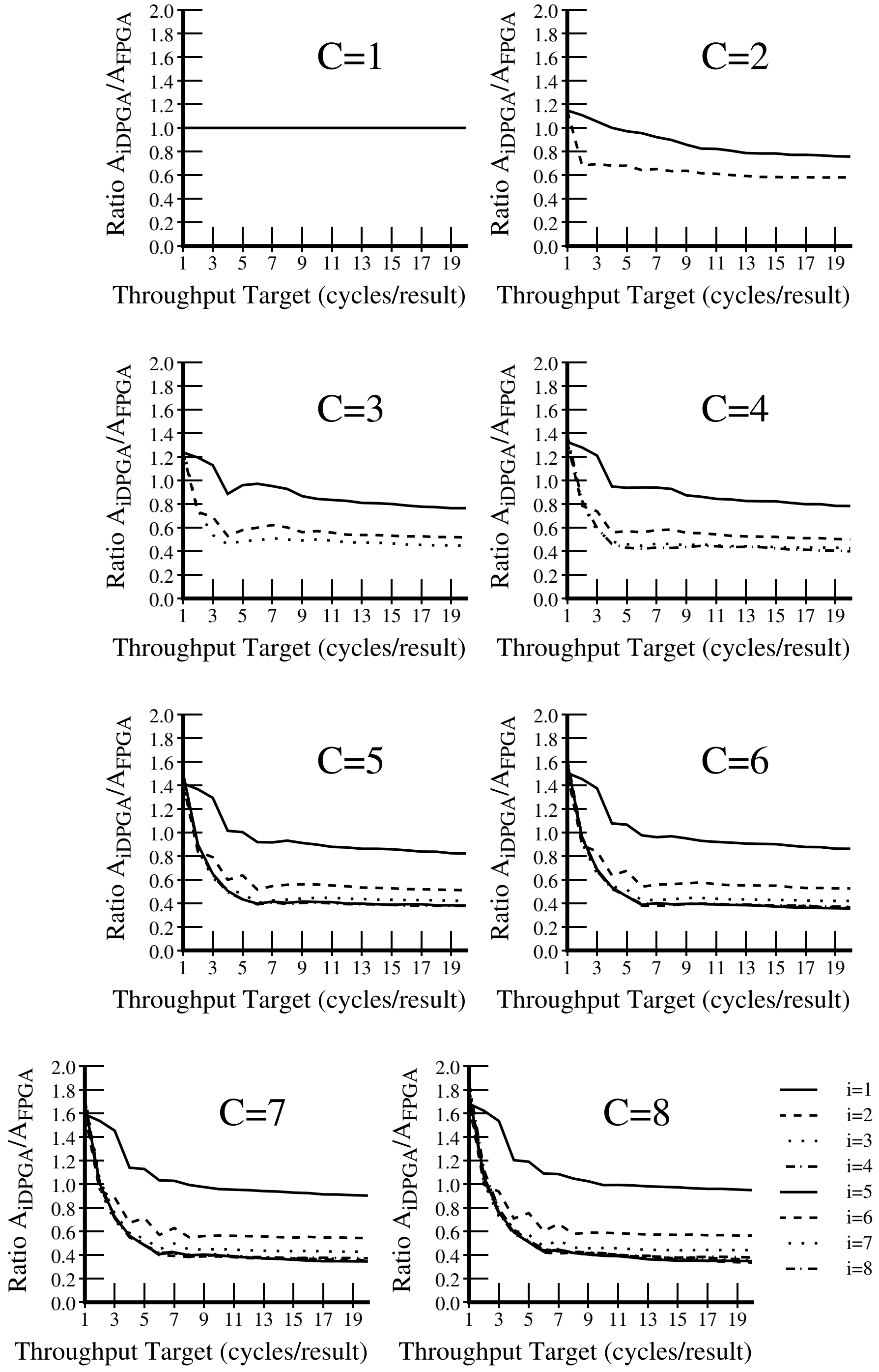


Figure 11.8: Average Area Ratios versus Throughput

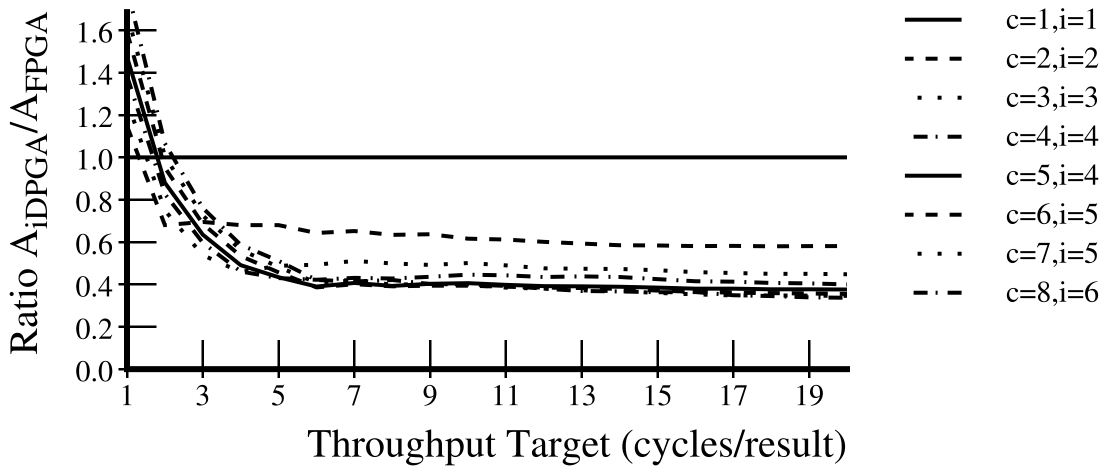


Figure 11.9: Average Area Ratios versus Contexts and Throughput

Average Ratio at 2 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	1.108	1.194	1.279	1.365	1.451	1.536	1.622
2		0.680	0.733	0.785	0.838	0.890	0.943	0.996
3			0.749	0.801	0.854	0.907	0.959	1.012
4				0.827	0.880	0.933	0.986	1.039
5					0.897	0.951	1.004	1.057
6						0.960	1.013	1.066
7							1.036	1.089
8								1.117

Table 11.5: Average Ratios for Benchmark Set

Average Ratio at 3 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	1.054	1.129	1.211	1.292	1.373	1.454	1.535
2		0.695	0.690	0.739	0.789	0.838	0.888	0.937
3			0.538	0.576	0.613	0.651	0.689	0.727
4				0.597	0.635	0.674	0.712	0.750
5					0.648	0.686	0.725	0.763
6						0.686	0.723	0.761
7							0.751	0.789
8								0.790

Table 11.6: Average Ratios for Benchmark Set

Average Ratio at 4 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	1.001	0.886	0.949	1.013	1.076	1.140	1.204
2		0.680	0.530	0.560	0.598	0.635	0.673	0.710
3			0.459	0.481	0.513	0.544	0.576	0.607
4				0.461	0.491	0.520	0.550	0.579
5					0.504	0.534	0.564	0.594
6						0.529	0.558	0.587
7							0.586	0.616
8								0.616

Table 11.7: Average Ratios for Benchmark Set

Average Ratio at 5 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	0.972	0.960	0.938	1.002	1.065	1.129	1.191
2		0.680	0.580	0.570	0.635	0.675	0.714	0.755
3			0.488	0.460	0.484	0.514	0.543	0.573
4				0.430	0.434	0.460	0.486	0.513
5					0.432	0.457	0.482	0.508
6						0.461	0.485	0.511
7							0.487	0.512
8								0.523

Table 11.8: Average Ratios for Benchmark Set

Average Ratio at 6 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	0.956	0.972	0.941	0.917	0.975	1.033	1.091
2		0.643	0.600	0.561	0.514	0.539	0.571	0.603
3			0.493	0.447	0.401	0.424	0.449	0.473
4				0.422	0.390	0.394	0.416	0.439
5					0.396	0.386	0.408	0.429
6						0.379	0.400	0.421
7							0.422	0.444
8								0.451

Table 11.9: Average Ratios for Benchmark Set

Average Ratio at 10 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	0.825	0.844	0.862	0.896	0.928	0.959	0.993
2		0.616	0.571	0.553	0.558	0.576	0.565	0.589
3			0.501	0.458	0.445	0.444	0.450	0.460
4				0.446	0.406	0.397	0.398	0.413
5					0.410	0.394	0.398	0.394
6						0.392	0.388	0.400
7							0.408	0.413
8								0.416

Table 11.10: Average Ratios for Benchmark Set

Average Ratio at 20 clocks/result throughput								
	Ratio $\left(\frac{A_{iDPGA(i,c)}}{A_{FPGA}}\right)$ by Number of Contexts (c)							
i	1	2	3	4	5	6	7	8
1	1.000	0.758	0.765	0.784	0.821	0.861	0.904	0.950
2		0.581	0.518	0.500	0.510	0.525	0.544	0.566
3			0.448	0.425	0.418	0.419	0.427	0.442
4				0.400	0.376	0.369	0.372	0.380
5					0.380	0.355	0.346	0.349
6						0.364	0.356	0.337
7							0.358	0.343
8								0.355

Table 11.11: Average Ratios for Benchmark Set

11.4.4 Area for Improvement

As noted previously (Sections 10.5.2 and 10.5.3), netlist mapping is oblivious of the final temporal implementations. The allocation of temporal and spatial pipeline stages is more rigid than strictly necessary. As we noted above (Section 11.4.1), retiming LUTs are inserted in a stylized fashion which is not likely to be optimal. Compatibility testing is stochastic and may declare many compatible LUT groups incompatible. Consequently, tighter packing of LUTs is likely with more sophisticated mapping tools.

11.5 Other Input Retiming Models

Register File LSM, YSE, and VEGA all use a *register file* to hold data values between production and consumption like most processors. These machines are also targeted at significantly more sequentialization (1K→8K contexts). Consequently, they manage to use only a single port into the register file. The register file organization has a more general access pattern since any value can be written to any memory location and read from any location to any output. The generality avoids packing compatibility restrictions, allowing data to be packed more tightly into memories. However, the more general access is also significantly more expense to support; LSM and YSE replicate the entire memory bank, storing four copies of every data value, in order to achieve four read ports. The restriction to a single write port is a simplification which these machines use in order to make the register file implementation viable.

Time Matching Instead of shifting data through a continually advancing shift register, we can make each of the input registers take its value from the input line and load it at a specified time. In this scheme, the input registers hold a finite number of values (i), but are not be limited to only the last i values. Such a scheme would require a unit to match input times, making each input larger than the iDPGA, but the increased range and packing density relaxes timing constraints on data arrival which are useful for simplifying the task of physical mapping. This is the scheme used by TSFPGA and it will be explored more fully in the following chapter.

11.6 Summary

Typical tasks require two, different kinds of data transport – *spatial transport* to move data from the processing element that generated it to the ones which will consume it and *temporal transport* to take data from the time when it is generated to the times when it is consumed. It is inefficient to tie up expensive, spatial transport resources such as wires and switches, to perform a temporal transport task. Tasks such as circuit evaluation have sufficient requirements for temporal transport that input retiming registers are clearly a worthwhile architectural feature to include in a multicontext device. Implementations with multiple retiming registers are more compact than implementations with no additional retiming resources.

As with multiple contexts, the extent to which we can save area with deep input registers depends on the area ratio between the active interconnect and the retiming registers. Here, we assumed the ratio between active area and instruction area was 10:1 (800Kλ^2:78Kλ^2), as in the previous chapter. We assumed, the ratio between the active area and context area including both instruction and retiming was roughly 8:1 (800Kλ^2:104Kλ^2). At these ratios, 4-5 context iDPGA implementations were, on average, half to one-third the size of the single context alternative.

The best implementation varies with target throughput. At these size ratios, the $i = 4$, $c = 4$ case is moderately good across throughput ranges. It is only worse than the single context implementation at the highest throughput, and is within 20% of the best implementation at the lowest throughput measured here.

11.7 Review

In the development since Chapter 7, we have seen that the area required to implement a general-purpose computational task is composed of four parts:

1. Active interconnect area
2. Active computational processing element area
3. Task description (instruction storage) area
4. Intermediate value storage for temporal retiming

While traditional FPGA architectures have a one-to-one mapping between these components, this resource ratio is neither necessary or efficient. We further saw that active interconnect area is, by far, the largest single component of this area, while task description and value storage areas are small in comparison.

For a given computational task, we saw that the requirements for each of these four parts arise from different sources. The number of instructions required to describe the task and number of intermediates held during computation arise from the basic computational task, itself. The size of the active interconnect and processing are dictated by the task's target throughput. For the highest possible throughput, the conventional FPGA strategy of allocating a single instruction to each piece of active interconnect and processing is an efficient allocation of resources. However, as throughput requirements drop below this extreme, multicontext implementations compress the implementation into less space by sharing and reusing a smaller number of active resources. This sharing increases the ratio of instructions and intermediates to active resources. DPGAs are the practical implementation of such a sharing scheme, assigning multiple instructions and multiple intermediate values to each active resource.

Note that the amount of compressibility we achieve with DPGAs is critically dependent upon how small we can make the non-active residue. That is, when we remove active interconnect and processing elements, we are left with the instruction and the intermediate values. The amount of area savings we can realize depends on how much smaller the space to hold instructions and intermediates is than the space for the active area necessary to actually process the instruction and its data. It is this effect which motivates our interests in reducing the number of bits used to describe each instruction (Section 7.8) and in reducing the area required to store those bits (*e.g.* DRAM context implementations in the DPGA prototype – Section 10.4).

It is also worthwhile to note that the style of compression used in the last two chapters (Chapters 10 and 11), makes instructions and data readily accessible and is largely independent of task structure. While densely encoded instructions need some decoding, each instruction is encoded separately so that it can be stored locally and used immediately upon being read. If we are willing to pay additional access latency and work with variable size encodings, block and structure-based encoding schemes can be used, making it is possible to compress the instruction requirements further. Ultimately, the minimum task description area will depend on the descriptive complexity of the task (See Section 8.4). Exploiting structure, such as, data widths, operation commonality, and task recurrence requires more general instruction distribution datapaths and more sequential decoding of task instructions. Nonetheless, variants on these techniques may be valuable in further compressing instruction and data residues and hence reducing task implementation size.

12. Time-Switched Field Programmable Gate Arrays

We established in Chapter 7 that active interconnect area consumed most of the space on traditional, single-context FPGAs. In Chapter 10, we saw that adding small, local, context memories allowed us to reuse active area and achieve smaller task implementations. Even in these multicontext devices, we saw that interconnect consumed most of the area (Section 10.4.2). In Chapter 11, we added architectural registers for retiming and saw more clearly the way in which multiple context evaluation saves area primarily by reducing the need for active interconnect. In this chapter, we describe the Time-Switched Field Programmable Gate Array (TSFPGA), a multicontext device designed explicitly around the idea of time-switching the internal interconnect in order to implement more effective connectivity with less physical interconnect.

One issue which we have not addressed in the previous sections is the complexity of physical mapping and, consequently, the time it takes to perform said mapping. Because of the computational complexity, physical mapping time can often be the primary performance bottleneck in the edit-compile-debug cycle. It can also be the primary obstacle to achieving acceptable mapping time for large arrays and multi-chip systems.

In particular, when the physical routing network provides limited interconnectivity between LUTs, it is necessary to carefully map logical LUTs to physical LUTs in accordance with both netlist connectivity and interconnect connectivity. The number of ways we can map a set of logical LUTs to a set of physical LUTs is exponential in the the number of mapped LUTs, making the search for an acceptable mapping which simultaneously satisfies the netlist connectivity constraints and the limited physical interconnect constrains – *i.e.* physical place and route – computationally difficult. Finding an optimal mapping is generally an NP-complete problem. Consequently, in traditional FPGAs, this mapping time can be quite large. It often take hours to place and route designs with a couple of thousand LUTs. The computational complexity arises from two features of the mapping problem:

1. As noted in Section 11.1, traditional FPGAs **must** have enough routing resources to physically route all task connections simultaneously.
2. Since interconnect is the dominant area in FPGAs (Chapter 7), conventional FPGAs try to use as little interconnect as feasible to provide high computational density.

The result is a large set of simultaneous constraints which **must** be satisfied during mapping, making the task of physical mapping computationally intensive. TSFPGA virtually eliminates the simultaneous constraint satisfaction required to successfully route a component, making it possible to rapidly map tasks to the array. Simultaneous constraint satisfaction is still necessary to achieve the highest performance mappings on TSFPGA, but is not necessary to achieve any mapping. This gives the device user control over mapping time and quality.

This chapter details a complete TSFPGA design including:

1. Time-switched input register

2. Techniques used by TSFPGA to avoid constraints
3. Sample interconnect model for time-switched routing
4. Complete gate-array architecture built around:
   (a) time-switched input register
   (b) switched interconnect
   (c) pipelined interconnect
5. Area and time estimates for TSFPGA building blocks
6. Experimental, quick mapping software
7. Mapped benchmark results using experimental software and a sample design point

TSFPGA was developed jointly by Derrick Chen and André DeHon. Derrick worked out VLSI implementation and layout issues, while André developed the architecture and mapping tools.

12.1 Time-Switched Input Registers

As noted in Section 11.1, if all retiming can be done in input registers, only a single wire is strictly needed to successfully route the task. The simple input register model used for the previous chapter had limited temporal range and hence did not quite provide this generality. In this section, we introduce an alternative input strategy which extends the temporal range on the inputs without the linear increase in input retiming size which we saw with the shift-register based input microarchitecture in the previous chapter.

The trick we employ here is to have each logical input load its value from the active interconnect at just the right time. As we have seen, multicontext evaluation typically involves execution of a series of microcycles. A subset of the task is evaluated on each microcycle, and only that subset requires active resources in each microcycle. We call each microcycle a **timestep** and, conceptually at least, number them from zero up to the total number of microcycles required to complete the task. If we broadcast the current timestep, each input can simply load its value when its programmed load time matches the current timestep.

Figure 12.1 shows a 4-LUT with this input arrangement which we will call the *Time-Switched Input Register*. Each LUT input can load any value which appears on its input line in any of the last i cycles. The timestep value is $\lceil \log_2(i) \rceil$ bits wide, as is the comparator. With this scheme, if the entire computation completes in i timesteps, all retiming is accomplished by simply loading the LUT inputs at the appropriate time – *i.e.* loading each input just when its source value has been produced and spatially routed to the destination input. Since the hardware resources required for this scheme are only logarithmic in the total number of timesteps, it may be reasonable to make i large enough to support most all desirable computations.

With this input structure, logical LUT evaluation time is now decoupled from input arrival time. This decoupling was not true in FPGAs, DPGAs, or even iDPGAs. With FPGAs, the LUT is evaluated only while the inputs are held stable. With DPGAs, the LUT is evaluated only on the microcycle when the inputs are delivered. With the iDPGA, the LUT must be evaluated on the

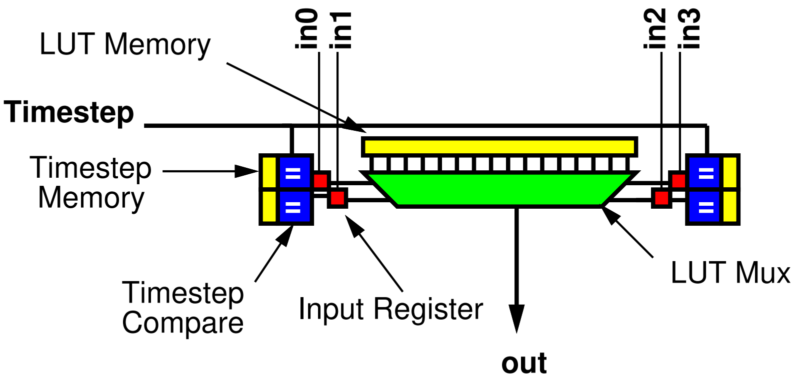


Figure 12.1: 4-LUT with Time-Switched Input Register

correct cycle relative to the arrival of the input, and the range of feasible cycles was limited by i. Further, with the time-switched input register, the inputs are stored, allowing the LUT result to be produced on any microcycle, or microcycles, following the arrival of the final input. In the extreme case of a single wire for interconnect, each LUT output would be produced and routed on a separate microcycle. Strictly speaking, of course, with a single wire there need be only one physical LUT, as well.

This decoupling of the input arrival time and LUT evaluation time allows us to remove the simultaneous constraints which, when coupled with limited interconnectivity, made traditional programmable gate array mapping difficult. We are left with a single constraint: schedule the entire task within i timesteps.

12.2 Switched Interconnect – Folding

Now that we no longer need to involve the physical interconnect in temporal transport, we are free to reuse physical interconnect resources at their minimum operating time. This reuse allows us to employ less physical interconnect than traditional FPGAs, while simultaneously providing more connectivity.

12.2.1 Subarray Structure

Conceptually, let us consider array interconnect as composed of a series of fully interconnected subarrays. That is, we arrange groups of LUTs in subarrays, as in the DPGA prototype (See Section 10.4). Within a subarray, LUTs are fully interconnected with a monolithic crossbar. Also feeding into and out of this subarray crossbar are connections to the other subarrays.

The subarray contains a number of LUTs, N_{sa_LUT}, where we consider $N_{sa\_LUT} = 64$ as typical. Connecting into the subarray are N_{sa_in} inputs from outside. Similarly, N_{sa_out} connect out. N_{sa_in}, and N_{sa_out} are typically governed by Rent's Rule. With $N_{sa\_LUT} = 64$, $k = 4$, and $0.5 \leq p \leq 0.7$, we might expect $32 \leq N_{sa\_in} = N_{sa\_out} \leq 74$, and consider $N_{sa\_in} = N_{sa\_out} =$

64 typical and convenient.

Together, this suggests a $(N_{sa\_LUT} + N_{sa\_in}) \times (k \times N_{sa\_LUT} + N_{sa\_out})$ crossbar, which is 128×320 for the typical values listed above. This amounts to 640 switches per 4-LUT, which is about 2-3$\times$ the values used in conventional FPGA architectures as we reviewed in Section 7.5. Conventional architectures, effectively, only populate 30-50% of the switches in such a block relying on placement freedom to make up for the missing switches. It is, of course, the complexity of the placement problem in light of this depopulation which is largely responsible for the difficulty of place and route on conventional architectures.

We also need to interconnect these subarrays. For small arrays it may be possible to simple interwire the connections between subarrays without substantial, additional switching. This is likely the case for the 100-1000 LUT cases reviewed in Section 7.5. For larger arrays, more, inter-array switching will be required to provide reasonable connectivity. As we derived in Section 7.6, the interconnect requirements will grow with the array size.

12.2.2 Interconnect Folding

With switched interconnect, we can realize a given level of connectivity without placing all of the physical switches that such connectivity implies. Rather, with the ability to reuse the switches, we effect the complete connectivity over a series of microcycles.

We can view this reuse as a folding of the interconnect in time. For instance, we could map pairs of LUTs together such that they share input sets. This, coupled with cutting the number of array outputs (N_{sa_out}) in half, will cut the number of crossbar outputs in half and hence halve the subarray crossbar size. For full connectivity, it may now takes us two microcycles to route the connections, delivering the inputs to half the LUTs and half the array outputs in each cycle. In this particular case we have performed **output folding** by sharing crossbar outputs (See Figure 12.2). Notice that the time-switched input register allows us to get away with this folding by latching and holding values on the correct microcycle. The input register also allows the non-local subarray outputs to be transmitted over two cycles. In the most trivial case, the array outputs will be connected directly to array inputs in some other array and, through the destination array's crossbar, they will, in turn be connected to LUT inputs where they can be latched on the appropriate microcycle as they arrive.

There is one additional feature worth noting about output folding. When two or more folded LUTs share input values all the LUTs can load the input when it arrives. For heavily output folded scenarios, these shared inputs can be exploited by appropriate grouping to allow the task to be routed in less microcycles than the total network sharing.

We can also perform **input folding**. With input folding, we pair LUTs so that they share a single LUT output. Here we cut the number of array inputs (N_{sa_in}) in half, as well. The array crossbar now requires only half as many inputs as before and is, consequently, also half as large in this case. Again, the latched inputs allow us to load each LUT input value only on the microcycle on which the associated value is actually being routed through the crossbar. For input folding, we must add an effective pre-crossbar multiplexor so that we can select among the sources which share a single crossbar input (See Figure 12.3).

It is also possible to fold together distinct functions. For example, we could perform an input fold such that the 64 LUT outputs each shared a connection into the crossbar with the 64 array inputs. Alternately, we could perform an output fold such that LUT inputs shared their connections

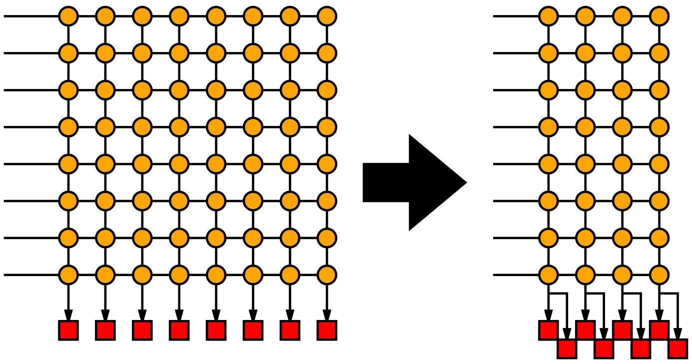


Figure 12.2: Output Folding

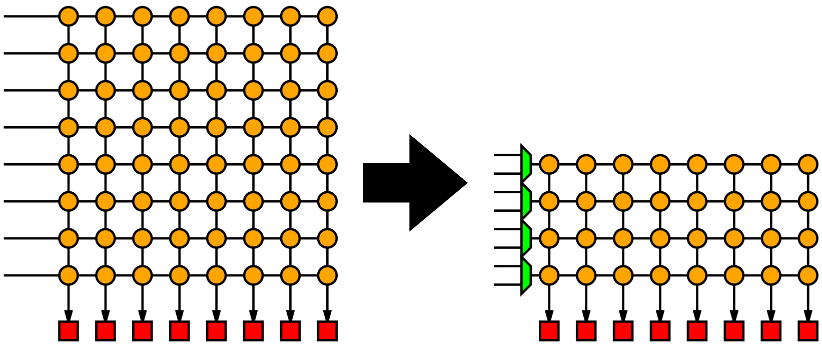


Figure 12.3: Input Folding

with array outputs.

Finally, note that we can perform input folding and output folding simultaneously (See Figure 12.4). We can think of the DPGAs introduced in Chapter 10 as folded interconnect where we folded both the network input and output c times. Each DPGA array element (See Figure 11.2) shared c logical LUT inputs on one set of physical LUT inputs and shared c logical LUT outputs on a single LUT output. Figure 12.5 shows how a two context DPGA results from a single input and output fold. In the DPGA, we had only c routing contexts for this c^2 total folding. To get away with this factor of c reduction in interconnect description, we had to restrict routing to temporally adjacent contexts. As we saw, in Chapter 10 this sometimes meant we had to allocate LUTs for through routing when connections were needed between contexts.

Routing on these folded networks naturally proceeds in both space and time. This gives the networks the familiar time-space-time routing characteristics pioneered in telephone switching systems.

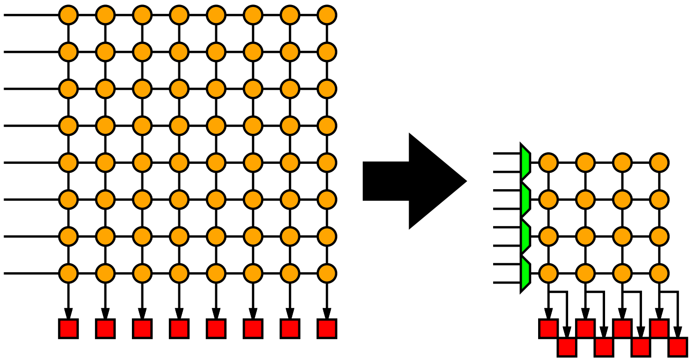


Figure 12.4: Input and Output Folding

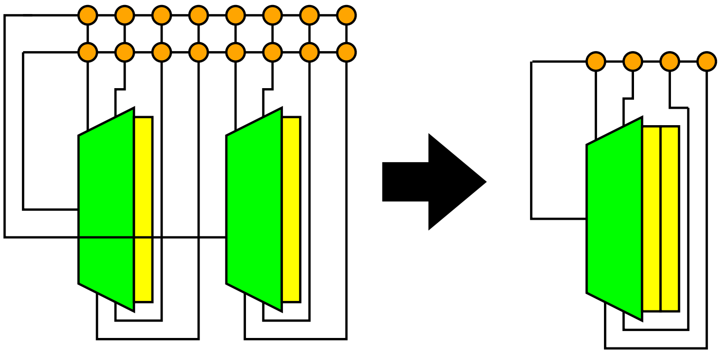


Figure 12.5: Two-Context DPGA as Input and Output Fold

12.3 Architecture

In this section, we detail a complete TSFPGA architecture. The basic TSFPGA building block is the subarray tile (See Figure 12.6) which contains a collection of LUTs and a central switching crossbar. LUTs share output connections to the crossbar and input connections from the crossbar in the folded manner described in the previous section. Communication within the subarray can occur in one TSFPGA clock cycle. Non-local input and output lines to other subarrays also share crossbar I/O's to route signals anywhere in the device. Routes over long wires are pipelined to maintain a high basic clock rate.

Array Element The TSFPGA array element is made up of a number of LUTs which share the same crossbar outputs and input (See Figure 12.7). The LUT output into the crossbar is selected based on the routing context programming. As shown, each array element shares its crossbar input with several network inputs.

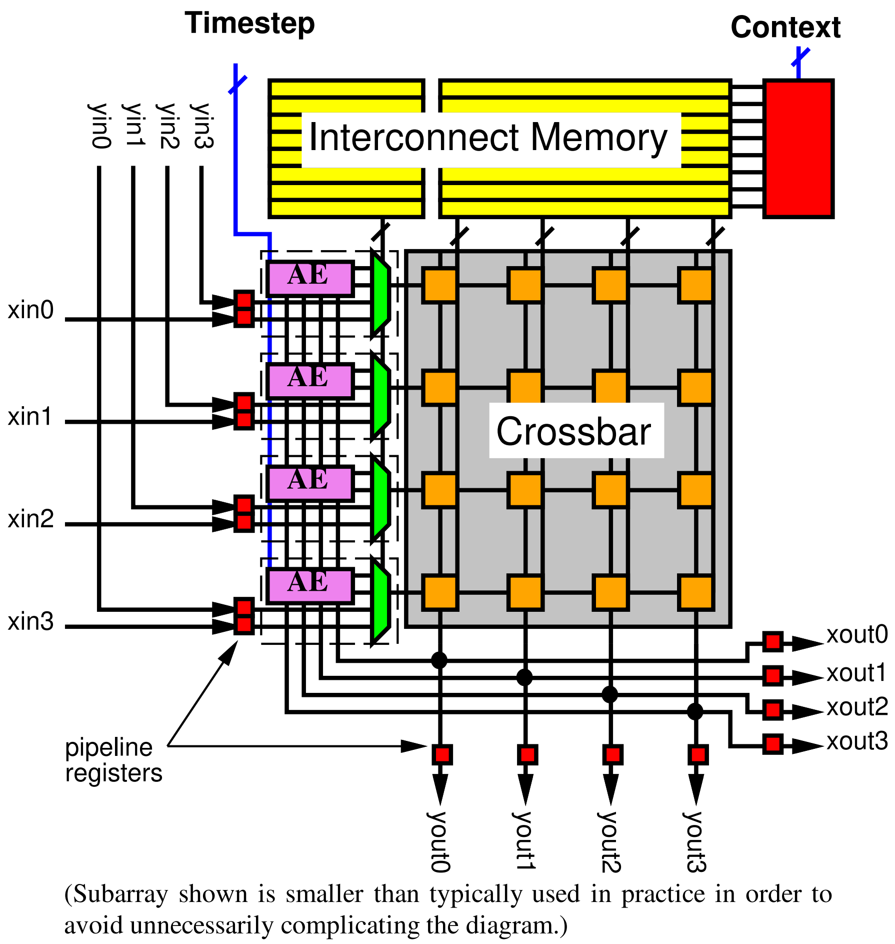


Figure 12.6: TSFPGA Subarray Composition

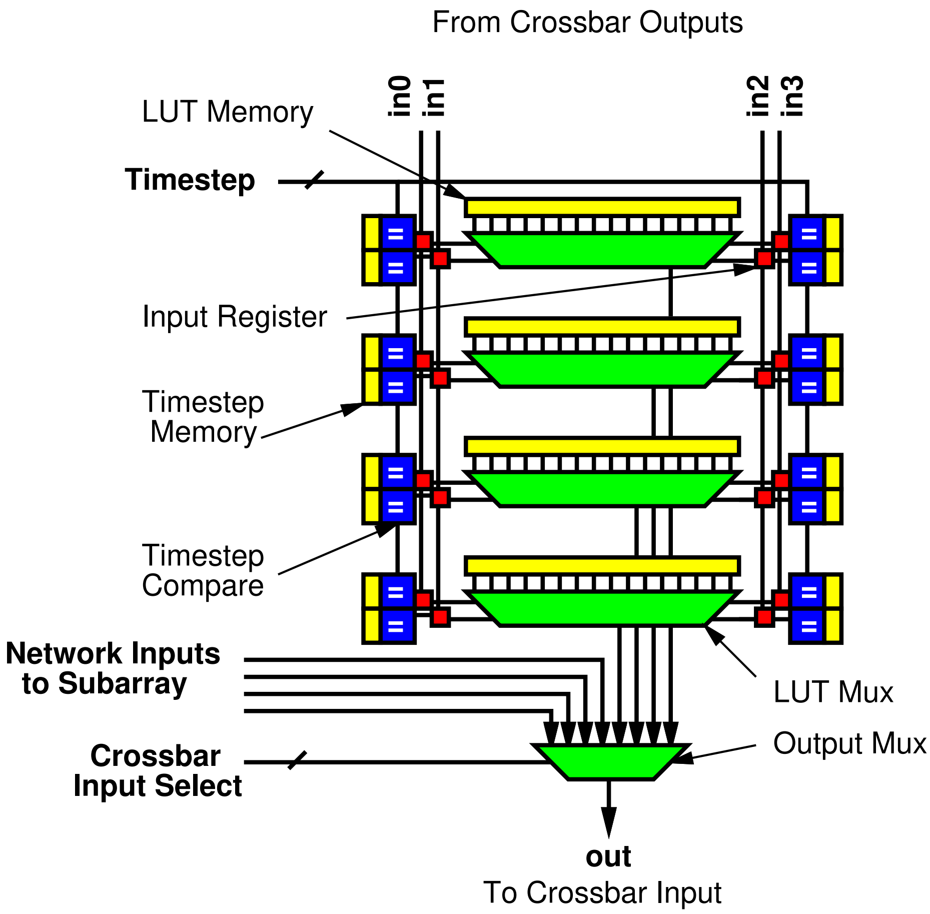


Figure 12.7: TSFPGA Array Element Composition

The LUT input values are stored in time-switched input registers. The inputs to the array element are run to all LUT input registers. When the current timestep matches the programmed load time, the input register is enabled to load the value on the array-element input. When multiple LUTs in an array element take the same signal as input, they may be loaded simultaneously.

Unlike the DPGA architectures detailed in Chapters 10 and 11, the LUT multiplexor is replicated for each logical LUT. As we saw in Section 10.4.2, the LUT mux is only a tiny portion of the area in an array. Replicating it along with context memory avoids the need for final LUT input multiplexors which would otherwise be in the critical path. When considering both the additional input multiplexors and the requirements for selecting among the LUT programming memory, the benefit of resource sharing at this level have been minimal in most of the implementations we have examined.

Crossbar The primary switching element is the subarray crossbar. As shown in Figures 12.6 and 12.7, each crossbar input is selected from a collection of subarray network inputs and subarray LUT outputs via by a pre-crossbar multiplexor. Subarray inputs are registered prior to the pre-crossbar multiplexor and outputs are registered immediately after the crossbar, either on the LUT inputs or before traversing network wires. This pipelining makes the LUT evaluations and crossbar traversal a single pipeline stage. Each registered, crossbar output is routed in several directions to provide connections to other subarrays or chip I/O.

Inter-subarray wire traversals are isolated into a separate pipeline stage between crossbars. As we saw both in Section 7.1.3 and the DPGA prototype implementation (Section 10.4.2), wire and switch traversals are responsible for most of the delay in programmable gate arrays. By pipelining routes at the subarray level, we can achieve a smaller microcycle time and effectively extract higher capacity from our interconnect.

Notice that the network in TSFPGA is folded such that the single subarray crossbar performs all major switching roles:

1. **output crossbar** – routing data from LUT outputs to destinations or intermediate switching crossbars
2. **routing crossbar** – routing data through the network between source and destination subarrays
3. **input crossbar** – receiving data from the network and routing it to the appropriate destination LUT input

This sharing avoids dedicating specialized routing resources to any single function so that the available resources can be deployed as needed by the task. Connections on TSFPGA are statically routed in a distributed, multistage switching fashion.

Intra-Subarray Switching Communication within the subarray is simple and takes one clock cycle per LUT evaluation and interconnect. Once a LUT has all of its inputs loaded, the LUT output can be selected as an input to the crossbar, and the LUT's consumers within the subarray may be selected as crossbar outputs. At the end of the cycle, the LUT's value is loaded into the consumers' input registers, making the value available for use on the next cycle.

Inter-Subarray Switching Figure 12.8 shows the way a subarray may be connected to other subarrays on a component. A number of subarray outputs are run to each subarray in the same row and column. For large designs, hierarchical connections may be used to keep the bandwidth between subarrays reasonable for while maintaining a limited crossbar size and allowing distant connections. The hierarchical connections can give the logical effect of a three or four dimensional network.

Routing data within the same row or column involves:

1. Route LUT output through crossbar to the outputs headed for the destination subarray.
2. Traverse the wire between subarrays.
3. Select network input with source value as a crossbar source and route through the crossbar to the destination LUT input.

When data needs to traverse both row and column:

1. Route LUT output to first dimension destination (row, column).
2. Traverse first dimension interconnect.

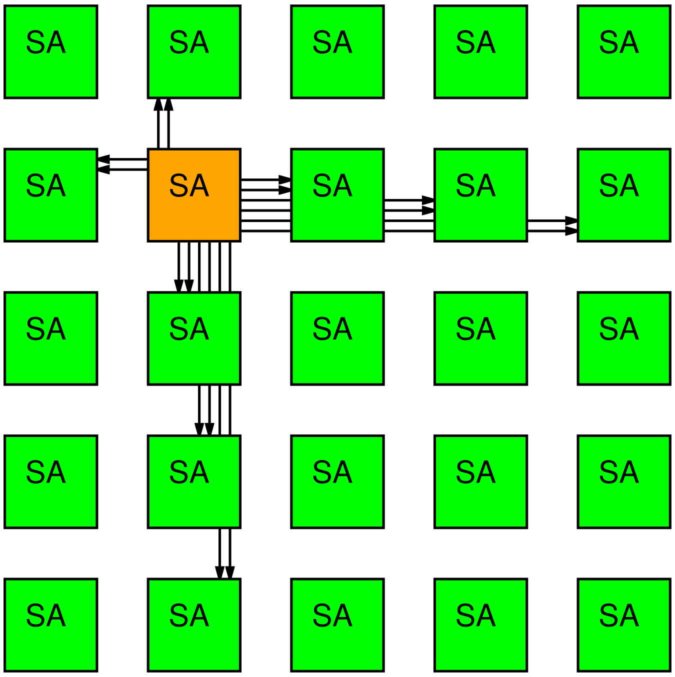


Figure 12.8: Sample Inter-Subarray Network Connections

3. Switch output in second dimension (column, row).
4. Traverse second dimension interconnect.
5. Switch to LUT input and load.

Pipelining places each of these operations in a different clock cycle. Long wire connections may merit multiple clock cycles for wire traversal – this is likely to be true for long, hierarchical connections. Short wires, particularly the nearest neighbor connections, may not always merit a separate pipeline stage for wire traversal.

I/O Connections I/O connections are treated like hierarchical network lines and are routed into and out of the subarrays in a similar manner. Each input has an associated subarray through which it may enter the switched network. Similarly, each output is associated with the crossbar output of some subarray. Device outputs are composed of time-switched input registers and load values from the network at designated timesteps like LUT inputs. Alternately, an output may look like inter-subarray pipeline register for routing in multichip systems.

Array Control Two "instruction" values are used to control the operation of each subarray on a per clock cycle basis, *timestep* and routing *context* (shown in Figure 12.6). The routing context serves as an instruction pointer into the subarray's routing memory. It selects the configuration of the crossbar and pre-crossbar multiplexors on each cycle. *timestep* denotes time events and

k	Number of LUT inputs
i	Maximum retiming depth
N_{sa_LUT}	LUTs in subarray
N_{sa_in}	External inputs to subarray
N_{sa_out}	External outputs to subarray
N_{xbar_in}	Subarray crossbar inputs
N_{xbar_out}	Subarray crossbar outputs
c_r	Number of routing contexts

Table 12.1: TSFPGA Subarray Parameters

indicates when values should be loaded from shared lines.

These two values are distinct in order to allow designs which take more microcycles to complete than they actually require contexts. The evaluation of a function will take a certain number of TSFPGA clock cycles. This time is dictated by the routed delay. For designs with large serial paths, long critical paths, or poor locality of communications, the routed delay may be large without consuming all the routing resources. For this reason, it is worthwhile to segregate the notion of timestep from the notion of routing contexts. Each routing interconnect pattern, or context, may be invoked multiple times at different timesteps during an evaluation. This allows us to have a small number of routing contexts even when the design topology necessitates a large number of timesteps.

As a trivial example, consider the case of an unfolded subarray. With full subarray interconnect there may be enough physical interconnect in a single context to actually route the complete design. However, since the design has a critical path composed of multiple LUT delays, it will take multiple microcycles to evaluate. In this case, it is not necessary to allocate separate routing contexts for each timestep as long as we segregate these two specifications into separate entities.

12.4 Architecture Parameters

The subarray composition effectively determines the makeup of a TSFPGA component. Table 12.1 summarizes the base parameters characterizing a TSFPGA subarray implementation. From these, we can calculate resource size and sharing:

$$N_{pre\_xbar\_in} = \frac{N_{sa\_in} + N_{sa\_LUT}}{N_{xbar\_in}} \tag{12.1}$$

$$N_{share\_xbar\_out} = \frac{N_{sa\_out} + k \cdot N_{sa\_LUT}}{N_{xbar\_out}} \tag{12.2}$$

$$N_{sw/LUT} = \underbrace{\frac{N_{xbar\_in} \cdot N_{xbar\_out}}{N_{sa\_LUT}}}_{\text{crossbar}} + \underbrace{\frac{N_{sa\_in} + N_{sa\_LUT}}{N_{sa\_LUT}}}_{\text{pre-crossbar mux}} \tag{12.3}$$

Assuming we need to route through one intermediate subarray, on average, the number of routing contexts, c_r, needed for full connectivity is:

$$c_r \leq N_{pre\_xbar\_in} \cdot N_{share\_xbar\_out} \quad (12.4)$$

That is, we need one context to drive each crossbar source for each crossbar sink. When we have the same number of contexts as we have total network sharing, we can guarantee to route anything. Relation 12.4 assumes that N_{sa_in} and N_{sa_out} are chosen reasonably large to support traffic to, from, and through the array. If not, the sharing of inter-subarray network lines will dominate strict crossbar sharing, and should be used instead.

In practice, we can generally get away with less contexts than implied by Relation 12.4 by selectively routing outputs and inputs as needed. When LUT inputs sharing a crossbar output also share input values or when the mapped design requires limited connectivity, less switching is needed, and the routing tasks can be completed with fewer contexts. The freedom to specify which output drives each crossbar input on a given cycle, as provided by the TSFPGA subarray, is not strictly necessary. We could have a running counter which enables each source in turn. However, with a fixed counter it would always take $N_{pre\_xbar\_in}$ cycles to pull out any particular source, despite the fact that only a few are typically needed at any time. The effect is particularly acute when we think about levelized evaluation where we may be able to simultaneously route all the LUTs whose results are currently ready and needed in a single cycle. For this reason, TSFPGA provides independent control over the pre-crossbar input mux.

In total, the number of routing bits per LUT, then, is:

$$N_{route\_bits/LUT} = c_r \cdot \left(\frac{\log_2\left(N_{pre\_xbar\_in}\right) \cdot N_{xbar\_in} + \log_2\left(N_{xbar\_in}\right) \cdot N_{xbar\_out}}{N_{sa\_LUT}} \right) \quad (12.5)$$

Additionally, each LUT has its own programmable function and matching inputs:

$$\begin{aligned} N_{bits/LUT} &= N_{logic\_bits/LUT} + N_{ir\_bits/LUT} + N_{route\_bits/LUT} && (12.6) \\ N_{logic\_bits/LUT} &= 2^k && (12.7) \\ N_{ir\_bits/LUT} &= k \cdot \lceil \log_2(i) \rceil && (12.8) \end{aligned}$$

12.5 TSFPGA Implementation Estimates

12.5.1 Area

The time-switched input register is the most novel building block in the TSFPGA design. A prototype layout by Derrick Chen was:

Time-Switched Input register with comparators (i = 256)	32Kλ^2
LUT multiplexor with SRAM function memory	32Kλ^2
Complete base LUT (A_{base})	160Kλ^2

Note that A_{base} contains the LUT multiplexor, LUT function memory, all k = 4 input registers, their associated comparators, and the comparator programming.

The amortized area per LUT then is:

$$A_{LUT} = A_{base} + N_{sw/LUT} \cdot A_{SW} + N_{route\_bits/LUT} \cdot A_{mem\_cell} \quad (12.9)$$

Using the $N_{sa\_LUT} = N_{sa\_in} = N_{sa\_out}$ = 64 subarray as a reference, a version with no folding has, $N_{route\_bits/LUT} \approx$ 35, $N_{sw/LUT}$ = 640, making A_{LUT} = 1800Kλ^2, which is about 2-3$\times$ the size of typical 4-LUTs. But, as we noted above, unfolded, we have 2-3$\times$ as many switches as a conventional FPGA implementation. Also, unfolded, the expensive, matching input register is not needed.

If we fold the input and output each once, N_{xbar_in} = 64, N_{xbar_out} = 160, the number of switches drops to $N_{sw/LUT}$ = 162. With four routing contexts (c_r = 4), routing bits rise to $N_{route\_bits/LUT}$ = 64. The total area is $A_{LUT} \approx$ 640Kλ^2, which is comparable in size with modern FPGA implementations, while providing 2-3$\times$ the total connectivity.

Focus Design Point For the sake of evaluation, we settled on a single, highly folded, design point for close inspection. From our experience with the DPGA and other VLSI efforts, we chose to use a 16$\times$16 crossbar as the base interconnect ($N_{xbar_in} = N_{xbar_out}$ = 16), balancing the desire to keep the crossbar compact and fast with the desire to perform as high radix switching as feasible. Per LUT switches drops to almost a trivial level, $N_{sw/LUT}$ = 6. With 64 routing contexts, switching bits rises to $N_{route\_bits/LUT}$ = 112. Along with the 16 bits for LUT function programming, and 32 bits for input match programming, this brings the total number of programming bits per LUT up to 160, which is comparable to conventional FPGAs (See Table 7.2). The LUT area is $A_{LUT} \approx$ 310Kλ^2, or about half the size of a conventional FPGA 4-LUT.

At this size, each TSFPGA 4-LUT is effectively larger than the logical 4-LUT area in the iDPGAs of the previous chapter. The added complexity and range (i = 256) of the time-switched input registers is largely responsible for the greater size. The time-switched input register features, in turn, are what allow us map designs without satisfying a large number of simultaneously constraints.

12.5.2 Timing

Within the subarray, the critical path for the operating cycle of this design point contains:

1. clock to Q delay on the context address

2. Context memory read from 64-word deep memory
3. 8:1 pre-crossbar input mux
4. 16×16 crossbar traversal
5. setup time for the crossbar output flip-flops

For higher performance, the context read could be placed in its own pipeline stage. As noted, wire traversal already operate as a separate pipeline stage of its own. When wire delays begin to exceed the intra-subarray cycle delay, we can add additional pipelining to wire traversal. From simulations, it looks feasible to run with a 200 MHz microcycle. This is roughly twice the microcycle frequency for the DPGA design. The speedup here comes primarily from separating intra-subarray routing and inter-subarray routing into separate pipeline stages.

12.6 TSFPGA Fast Circuit Mapping

Traditional logic and state-element netlists can be mapped to TSFPGA for levelized logic evaluation similar to the DPGA mapping in the previous two chapters. Using this model, only the final place-and-route operation must be specialized to handle TSFPGA's time-switched operation. Of course, front-end netlist mapping which takes the TSFPGA architecture into account may be able to better exploit the architecture, producing higher performance designs.

`tspr`, our first-pass place-and-route tool for TSFPGA, performs placement by min-cut partitioning and routing by a greedy, list-scheduling heuristic. Both techniques are employed for their simplicity and mapping-time efficiency rather than their quality or optimality. The availability of adequate switching resource, expandable by allocating more configuration contexts, allows us to obtain reasonable performance with these simple mapping heuristics. For the most part, the penalty for poor quality placement and routing in TSFPGA is a slower design, not an unroutable design.

Timesteps and Contexts It is again worth noting that the number of *timesteps* and routing *contexts* are dictated by different properties of the mapped network.

The topology of the circuit will determine the critical path length, or the number of logical LUT delays between the inputs and outputs of the circuit. This critical path length is one lower bound on the number of timesteps required to evaluate a circuit. However, once placed onto subarrays, there is another, potentially longer, bound, the *distance delay* through the network. The distance delay is the length of the longest path through the circuit including the cycles required for inter-subarray routing. If all the LUTs directly along every critical path can be mapped to a single subarray, it is possible that the distance delay is equal to the critical path length. However, in general, the placed critical path crosses subarrays resulting in a longer distance delay. The quality of the distance delay is determined entirely during the placement phase.

The actual routed delay is generally larger than the distance delay because of contention. That is, if the architecture does not provide enough physical resources to route all the connections in the placed critical path simultaneously, or the if the greedy routing algorithms allocates those resources suboptimally, signals may take additional microcycles to actually be routed.

Placement Partitioning is based on the Fiduccia-Mattheyses min-cut heuristic [FM82]. Netlists are recursively partitioned along TSFPGA dimension boundaries. That is, for a simple, two-dimensional network topology, as shown in Figure 12.8, the design is first partitioned for columns, then columns are partitioned into subarrays. For larger networks, top-level row and column partitioning would precede low-level row and column partitioning. The Fiduccia-Mattheyses heuristic aims to minimize the size of the cut net, but does not, directly, minimize the effect of cuts on circuit delay. As a consequence partitioning is useful in reducing the routing congestion contribution to routed delay, but does not explicitly try to minimize the distance delay.

For the fastest mapping times, no sophisticated placement is done. Circuit netlists are packed directly into subarrays as they are parsed from the netlists. Such oblivious placement may create unnecessarily long paths by separating logically adjacent LUTs and may create unnecessary congestion by not grouping tightly connected subgraphs. However, with enough routing contexts the TSFPGA architecture allows us to succeed at routing with such poor placement.

Routing Routing is directed by the circuit netlist topology using a greedy, list-scheduling heuristic. At the start, a ready list is initialized with all inputs and flip-flop outputs. Routing proceeds by picking the output in the ready list which is farthest from the end set of primary outputs and flip-flop inputs. Routing a signal implies reserving switch capacity in each context and timestep involved in the route. If a route cannot be made starting at the current evaluation time, the starting timestep for the route is incremented and the route search is repeated. Currently, only minimum distance routes are considered. Assuming adequate context memory, every route will eventual succeed. Once a route succeeds, any LUT outputs which are then ready for routing are added to the ready list. In this manner, the routing algorithm works through the design netlist from inputs to outputs, placing and routing each LUT as it is encountered.

Modulo Context Routing The total number of contexts is dictated by the amount of contention for shared resources. Since some timesteps may route only a few connections, a routing context may be used at multiple timesteps. In the simplest case, switches in a routing context not used during one timestep may be allocated and used during another. In more complicated cases, a switch allocated in one context can be reused with the same setting in another routing context. This is particularly useful for the inter-subarray routing of patterns, but may be computationally difficult to exploit.

Our experimental mapping software can share contexts among routing timesteps by modulo context assignment. That is context n mod max_ctx is used to route on timestep n. As we will see in the next section, this generally allows us to reduce the number of required contexts. Further context reduction is possible when we are willing to increase the number of timesteps required for evaluation. More sophisticated sharing schemes are likely to be capable of producing better results.

Design	Netlist Size		Target Array			Quick Map				Performance Map			Best
	LUTs	IOs	Tiles	LUTs	IOs	Time	Delays			Time	Delays		Map
			(SA)			(sec.)	LUT	Dist	Rte	(sec.)	Dist	Rte	Rte
5xp1	46	17	2×1	128	32	0.05	11	14	19	0.67	14	19	18
9sym	123	10	2×1	128	32	0.18	8	15	29	4.02	15	25	23
9symml	108	10	2×1	128	32	0.15	9	17	27	9.48	13	24	21
C499	85	73	3×2	384	96	0.15	11	22	34	3.06	23	33	25
C880	176	86	3×2	384	96	0.34	22	44	48	9.67	31	36	32
alu2	169	16	2×2	256	64	0.28	20	43	45	10.00	43	47	34
apex6	248	234	4×4	1024	256	0.69	10	27	37	34.06	19	23	23
apex7	77	86	3×2	384	96	0.16	8	19	24	3.15	16	19	19
b9	46	62	2×2	256	64	0.08	8	15	21	0.74	12	14	14
clip	121	14	2×1	128	32	0.19	10	23	29	5.08	18	26	23
cordic	367	25	3×2	384	96	0.98	14	47	60	26.59	40	43	39
count	46	51	2×2	256	64	0.10	17	26	27	1.22	24	25	21
des	1267	501	6×6	2304	576	6.30	14	51	66	626.40	37	43	35
e64	230	130	3×3	576	144	0.63	10	29	40	18.90	32	33	26
f51m	45	16	2×1	128	32	0.07	18	21	22	0.05	21	22	22
misex1	20	15	1×1	64	16	0.03	7	10	16	0.02	10	13	13
misex2	38	43	2×2	256	64	0.07	9	15	18	0.95	15	16	15
rd73	105	10	2×1	128	32	0.14	11	18	27	4.35	14	22	21
rd84	150	12	2×2	256	64	0.24	10	30	35	4.88	26	30	24
rot	293	242	4×4	1024	256	0.76	17	44	45	21.14	28	31	31
sao2	73	14	2×1	128	32	0.11	10	14	22	1.79	13	20	18
vg2	60	33	2×2	256	64	0.11	10	17	23	1.07	14	19	19
z4ml	8	11	1×1	64	16	0.03	8	11	12	0.02	11	12	12

Run times given are in seconds on a SparcStation 20 Model 71 (rated at 125 SPECint92).

Table 12.2: TSFPGA Mappings for MCNC Circuit Benchmarks

12.7 Circuit Mapping

In this section we show the results of mapping the same MCNC benchmark circuit suite used for the DPGA in the previous two chapters to TSFPGA. These benchmarks are mapped viewing TSFPGA simply as an FPGA with time-switched interconnect, ignoring the way one might tailor tasks to take full advantage of the architecture.

Table 12.2 shows the results of mapping the benchmark circuits to TSFPGA. The same area mapped circuits from `sis` and `Chortle` used in Sections 10.5.3 and 11.4 were used for this mapping. Each design was mapped to the smallest rectangular collection of subarray tiles which supported both the design's I/O and LUT requirements. Quick mapping does oblivious placement while the performance mapping takes time to do partitioning. Both the quick and performance map

Design	Target Ratios LUTs	IOs	Quick Map Delay Ratios Dist	Route	Performance Map Delay Ratios Dist	Route	Best Route Ratio
	% used						
5xp1	0.36	0.53	1.27	1.73	1.27	1.73	1.64
9sym	0.96	0.31	1.88	3.62	1.88	3.12	2.88
9symml	0.84	0.31	1.89	3.00	1.44	2.67	2.33
C499	0.22	0.76	2.00	3.09	2.09	3.00	2.27
C880	0.46	0.90	2.00	2.18	1.41	1.64	1.45
alu2	0.66	0.25	2.15	2.25	2.15	2.35	1.70
apex6	0.24	0.91	2.70	3.70	1.90	2.30	2.30
apex7	0.20	0.90	2.38	3.00	2.00	2.38	2.38
b9	0.18	0.97	1.88	2.62	1.50	1.75	1.75
clip	0.95	0.44	2.30	2.90	1.80	2.60	2.30
cordic	0.96	0.26	3.36	4.29	2.86	3.07	2.79
count	0.18	0.80	1.53	1.59	1.41	1.47	1.24
des	0.55	0.87	3.64	4.71	2.64	3.07	2.50
e64	0.40	0.90	2.90	4.00	3.20	3.30	2.60
f51m	0.35	0.50	1.17	1.22	1.17	1.22	1.22
misex1	0.31	0.94	1.43	2.29	1.43	1.86	1.86
misex2	0.15	0.67	1.67	2.00	1.67	1.78	1.67
rd73	0.82	0.31	1.64	2.45	1.27	2.00	1.91
rd84	0.59	0.19	3.00	3.50	2.60	3.00	2.40
rot	0.29	0.95	2.59	2.65	1.65	1.82	1.82
sao2	0.57	0.44	1.40	2.20	1.30	2.00	1.80
vg2	0.23	0.52	1.70	2.30	1.40	1.90	1.90
z4ml	0.12	0.69	1.38	1.50	1.38	1.50	1.50
Average	0.46	0.62	2.08	2.73	1.80	2.24	2.01

All delay ratios are $\frac{\{\text{Dist,Route}\}\text{ Delay}}{\text{LUT Delay}}$

Table 12.3: TSFPGA Mappings for MCNC Circuit Benchmarks (Ratios)

use the same, greedy routing algorithm. As noted in Section 12.6, fairly simple placement and routing techniques are employed, so higher quality routing results are likely with more sophisticated algorithms. Quick mapping can route designs in the order of seconds, while performance mapping runs in minutes. The experimental mapping software implementation has not been optimized for performance, so the times shown here are, at best, a loose upper bound on the potential mapping time. The "Best map" results in Table 12.2 summarize the best results seen over several runs of the "performance" map.

Table 12.3 shows usage and time ratios derived from Table 12.2. All of the mapped delay ratios are normalized to the number of LUT delays in the critical path. We see that the quick mapped

Design	Min Delay				Min Contexts			
	delay	**# ctx**	**Δ ctx**	**Δ %**	**delay**	**# ctx**	**Δ ctx**	**Δ %**
5xp1	19	15	4	0.21	28	12	7	0.37
9sym	25	21	4	0.16	38	17	8	0.32
9symml	24	20	4	0.17	41	19	5	0.21
C499	33	19	14	0.42	48	16	17	0.52
C880	36	29	7	0.19	54	19	17	0.47
alu2	47	43	4	0.09	107	28	19	0.40
apex6	23	19	4	0.17	34	16	7	0.30
apex7	19	14	5	0.26	25	11	8	0.42
b9	14	10	4	0.29	18	8	6	0.43
clip	26	22	4	0.15	42	19	7	0.27
cordic	43	39	4	0.09	106	34	9	0.21
count	25	21	4	0.16	31	14	11	0.44
des	43	39	4	0.09	67	35	8	0.19
e64	33	30	3	0.09	58	28	5	0.15
f51m	22	18	4	0.18	40	13	9	0.41
misex1	13	10	3	0.23	16	8	5	0.38
misex2	16	12	4	0.25	20	9	7	0.44
rd73	22	17	5	0.23	32	14	8	0.36
rd84	30	26	4	0.13	52	25	5	0.17
rot	31	27	4	0.13	48	21	10	0.32
sao2	20	16	4	0.20	33	15	5	0.25
vg2	19	15	4	0.21	29	14	5	0.26
z4ml	12	4	8	0.67	16	3	9	0.75

Table 12.4: Modulo Context Sharing for MCNC Benchmarks

delays are almost 3× the critical path LUT delay, while the performance mapped delays are closer to 2×. As we noted in Section 12.5.2, the basic microcycle on TSFPGA is half that on the DPGA, suggesting that the performance mapped designs achieve roughly the same average latency as full, levelized evaluation on the DPGA. We can see from the distance delay averages that placement dictated delay is responsible for a larger percentage of the difference between critical path delay and routed delay. However, since the routed delay is larger than the distance delay, network resource contention and suboptimal routing are partially responsible for the overall routed delay time.

Context Compression As noted in Section 12.6 we can use modulo context assignment to pack designs into fewer routing contexts at the cost of potentially increasing the delay. Table 12.4 shows the number of contexts into which each of the designs in Table 12.2 can be packed both with and without expanding their delay. Figure 12.9 shows how routed delay of several benchmarks increases as the designs are packed into fewer routing contexts.

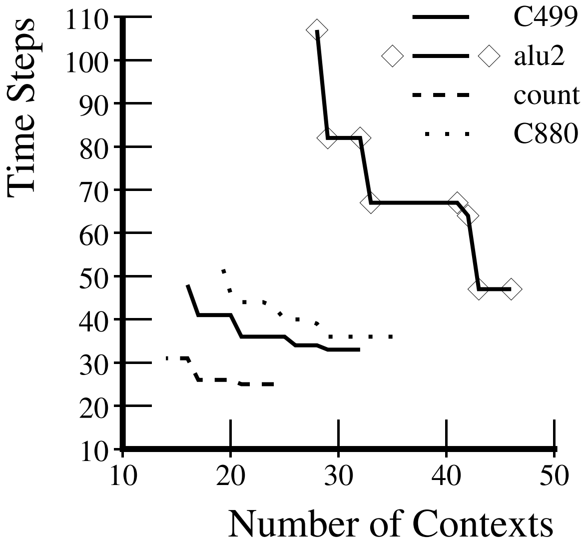


Figure 12.9: Sample Delay Increases with Context Packing

12.8 Related Work

Dharma [BCK93] time-switched two monolithic crossbars. It made the same basic reduction as the DPGA – that is, rather than having to simultaneously route all connections in the task, one only needed to route all connections on a single logical evaluation level. To make mapping fast, Dharma used a single monolithic crossbar. For arrays of decent size, the full crossbar connecting all LUTs at a single level can still be prohibitively large. Further, Dharma had a rigid assignment of LUT evaluation and hence routing resources to levels. As we see in TSFPGA, it is not always worth dedicating all of one's routing resources, an entire routing context, to a single evaluation timestep. Dharma deals with retiming using a separate flow-through crossbar. While the flow through retiming buffers are cheaper than full LUTs, they still consume active routing area which is expensive. As noted in Chapter 11, it is more area efficient to perform retiming, or temporal transport, in registers than to consume active interconnect.

VEGA [JL95], noted in Sections 10.2 and 11.5, uses a 1024 deep context memory, essentially eliminating the spatial switching network, and uses a register file to retime intermediate data. The VEGA architecture allows similar partitioning and greedy routing heuristics to be used for mapping. However, the heavy multicontexting and trivial network in VEGA means that it achieves its simplified mapping only at the cost of a 1024$\times$ reduction in active peak capacity and 100$\times$ penalty in typical throughput and latency over traditional FPGA architectures.

PLASMA [ACC^{+}96] was built for fast mapping of logic in reconfigurable computing tasks. It uses a hierarchical series of heavily populated crossbars for routing on chip. The existence of rich, hierarchical routing makes simple partitioning adequate to satisfy interconnect constraints. The heavily populated crossbars make the routing task simple. The basic logic primitive in PLASMA is the PALE, which is roughly a 2-output 6-LUT. The PLASMA IC packs 256 PALEs into 16.2mm$\times$16.2mm in a 0.8μ CMOS process, or roughly 1.6Gλ^2. This comes to 6.4Mλ^2 per PALE. If we generously assume each PALE is equivalent to 8 4-LUTs, the area per 4-LUT is 800Kλ^2, which is commensurate with conventional FPGA implementations, or about 2-2.5$\times$ the size for the TSFPGA design point described above. In practice, the TSFPGA LUT density will be even greater since it is not always the case that 8 4-LUTs can be packed into each PALE. PLASMA's size is a direct result of the fact that it builds all of its rich interconnect as active switches and wires. Routing is not pipelined on PLASMA, and critical paths often cross chip boundaries. As a result, typical PLASMA designs run with high latency and a moderately slow system clock rate (1-2 MHz). This suggests a time-switched device with a smaller amount of physical interconnect, such as TSFPGA, could provide the same level of mapping speed and mapped performance in substantially less area.

Virtual Wires [BTA93] employs time-multiplexing to extract higher capacity out of the I/O and inter-chip network connections only. Virtual Wires folds the FPGA and switching resources together, using FPGAs for inter-FPGA switching as well as logic. Since Virtual Wires is primarily a technique used on top of conventional FPGAs, it does accelerate the task of routing each individual FPGA or provide any greater use of on-chip active switching area.

Li and Cheng's **Dynamic FPID** [LC95] is a time-switched Field-Programmable Interconnect Device (FPID) for use in partial-crossbar interconnection of FPGAs. Similarly, they increase switching capacity, and hence routability and switch density, by dynamically switching a dedicated FPID.

UCSB researchers [cLCWMS96] consider adding a second routing context to a conventional FPGA routing architecture. Using a similar circuit benchmark suite, they find that the second context reduces wire and switching requirements by 30%. Since they otherwise use a conventional FPGA architecture, there is no reduction in mapping complexity for their architecture.

12.9 Conclusions

We have developed a new, programmable gate-array architecture model based around time-switching a modest amount of physical interconnect. The model defines a family of arrays with varying amounts of active interconnect. The key, enabling feature in TSFPGA is an input register which performs a wide range of signal retiming, freeing the active interconnect from performing data retiming or conveying data at rigidly defined times. Coupling the flexible retiming with reusable interconnect, we remove most of the constraints which make the place and route task difficult on conventional FPGA architectures. Consequently, even large designs can be mapped onto a TSFPGA array in seconds. More sophisticated mapping can be used, at the cost of longer mapping times, to achieve the lowest delay and best resource utilization. We demonstrated the viability of this fast mapping scheme by developing experimental mapping software for one design point and mapping traditional benchmark circuits onto these arrays. At the heavily time-switched design point which we explored in detail, the basic LUT size is half that of a conventional FPGA LUT while mapped design latency is comparable to the latency on fully levelized DPGAs.

12.10 Open Issues

At this point, we have left a number of interesting issues associated with TSFPGA unanswered.

- Performance using traditional place and route strategies – The fast mapping which we used above employs fast heuristics which are purposely limited to linear mapping complexity. Traditional mapping software uses different techniques, such as simulated annealing, which will consider simultaneous constraints to minimize resource usage and routed path length. It will be worthwhile to understand how well tasks can be mapped to members of the TSFPGA architecture when we are willing to take the time to perform a quality mapping job. In normal usage, one might use the fast mapping during design development and debug, then use the slower, higher quality mapping once a design becomes stable.

- Explore defined architectural space – We have focussed on a single point in the defined architectural space. It will be worthwhile to map tasks across various architectural points to determine the level of connectivity required to meet typical throughput and latency requirements, and to determine the most area efficient implementation points.

- Multichip extension – The specifics explored here focus on single-chip implementations, but there is a natural extension to multiple chip systems. The dimensional routing organization used on-chip should extend between chips when an array of TSFPGA components is employed. Partitioning, placement, and routing amongst components will be very similar to partitioning, placement, and routing amongst the subarrays on a single TSFPGA component. The boundary i/o will provide a more severe bottleneck between subarrays on distinct chips, requiring heavier time-multiplexing. Inter-TSFPGA routes will require more pipeline stages than inter-subarray routes.

13. MATRIX

Throughout this work, we have seen the central role which instructions play in general-purpose computing architectures. In Section 8.6, we saw a large architectural space characterized by the number of distinct control streams, datapath granularities, and instruction depth. In Chapters 4, 8, and 9, we reviewed this rich architectural space for general-purpose computing devices. We saw that the choices made in these parameters are what distinguish conventional general-purpose architectures, and we saw that it is these choices that define the circumstances under which a given general-purpose architecture is most efficient. In Section 9.5, we saw that even limiting ourselves to datapath granularity and instruction depth, it is not possible to select a single pair of these parameters which yielded a *robust* architecture – that is, there is no single selection point whose area requirement will be above a bounded fraction of the optimal selection of these two parameters for any task.

Every conventional general-purpose architecture reviewed in Chapter 4 and summarized in Table 8.1 takes a stand on instruction resources by selecting:

1. control stream to instruction ratio
2. local instruction depth
3. instruction to datapath element ratio

These selections are made and fixed at fabrication time and characterize the device for its entire lifetime. Unfortunately, most real computations are neither purely regular nor irregular, and real computations do not work on data elements of a single data size. Typical computing tasks spend most of their time in a very small portion of the code. In the kernel where most of the computational time is spent, the same computation is heavily repeated making it very regular such that a shallow instruction store is appropriate. The rest of the code is used infrequently making it irregular such that it is suited to a deep instruction store. Further, in systems, a general-purpose computational device is typically called upon to run many applications with differing requirements for datapath size, regularity, and control streams. This broad range of application requirements makes it difficult, if not impossible, to achieve robust and efficient performance across entire applications or application sets by selecting a single computational device which has a rigidly selected instruction organization.

In this chapter, we introduce MATRIX, a novel, general-purpose computing architecture which does not take a pre-fabrication stand on the assignment of space, distribution, and control for instructions. Rather, MATRIX allows the user or application to determine the actual organization and deployment of resources as needed. *Post-fabrication* the user can allocate instruction stores, instruction distribution, control elements, datapaths, data stores, dedicated and fixed data interconnect, and the interaction between datastreams and instruction streams.

We introduce MATRIX and the concepts behind it. We ground the abstract concepts behind the MATRIX architecture with:

- a concrete microarchitecture
- an illustrative application example
- model estimates and prototype implementation highlights
- architecture efficiencies for sample image processing tasks

MATRIX was developed jointly by Ethan Mirsky and André DeHon. André oversaw the architecture and guided the architectural definition, while Ethan defined the detailed microarchitecture and developed the VLSI implementation. MATRIX was first described publicly in [MD96] and portions of this chapter are taken from that description. Ethan details the MATRIX microarchitecture in his thesis [Mir96].

13.1 MATRIX Concepts

MATRIX is designed to maintain flexibility in instruction control. Primary instruction distribution paths are not defined at fabrication time. Instruction memories are not dedicated to datapath elements. Datapath widths are not fully predetermined. MATRIX neither binds control elements to datapaths nor predetermines elements that can only serve as control elements.

To provide this level of flexibility, MATRIX is based on a uniform array of primitive elements and interconnect which can serve instruction, control, and data functions. A single network is shared by both instruction and data distribution. A single integrated memory and computing element can serve as an instruction store, data store, datapath element, or control element. MATRIX's primitive resources are, therefore, **deployable**, in that the primitives may be deployed on a per-application basis to serve the role of instruction distribution, instruction control, and datapath elements as appropriate to the application. This allows tasks to have just as much regularity, dynamic control, or dedicated datapaths as needed. Datapaths can be composed efficiently from primitives since instructions are not prededicated to datapath elements, but rather delivered through the uniform interconnection network.

The key to providing this flexibility is a **multilevel configuration** scheme which allows the device to control the way it will deliver configuration information. To first order, MATRIX uses a two level configuration scheme. Traditional "instructions" direct the behavior of datapath and network elements on a cycle-by-cycle basis. *Metaconfiguration* data configures the device behavior at a more primitive level defining the architectural organization for a computation. Metaconfiguration data can be used to define the traditional architectural characteristics, such as instruction distribution paths, control assignment, and datapath width. The metaconfiguration "wires up" configuration elements which do not change from cycle-to-cycle including "wiring" instruction sources for elements whose configuration does change from cycle-to-cycle.

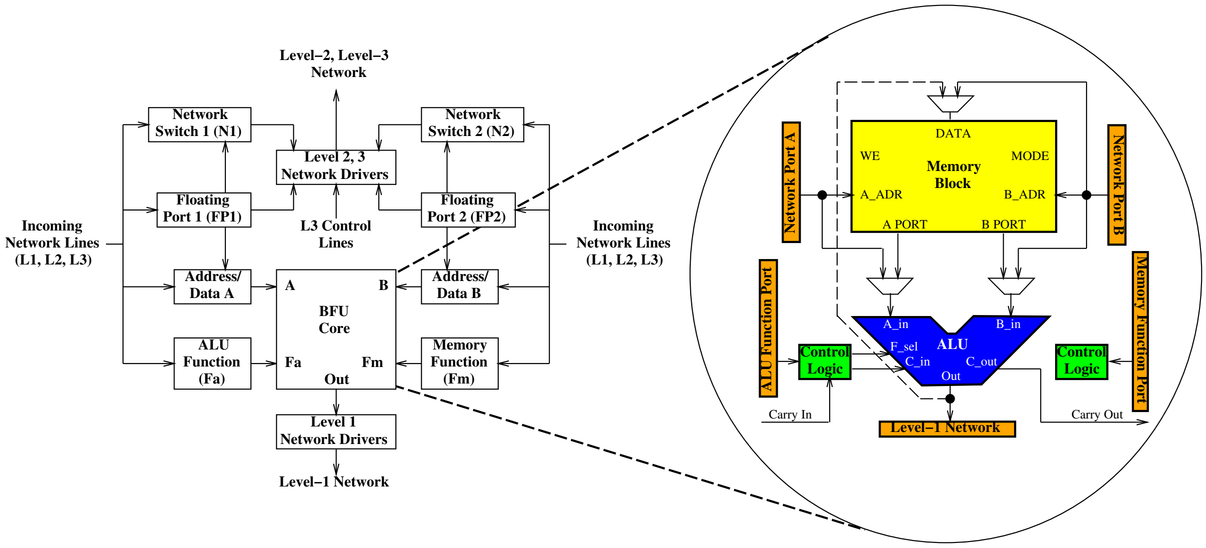


Figure 13.1: MATRIX BFU

13.2 MATRIX Architecture Overview

In this section we ground the more abstract concepts of the previous section with a concrete MATRIX microarchitecture. This concrete microarchitecture will be the focus of the remainder of the chapter. The concrete microarchitecture is based around an array of identical, 8-bit primitive datapath elements overlayed with a configurable network. Each datapath element or functional unit contains a 256×8-bit memory, an 8-bit ALU and multiply unit, and reduction control logic including a 20×8 NOR plane. The network is hierarchical, supporting three levels of interconnect. Functional unit port inputs and non-local network lines can be statically configured or dynamically switched.

13.2.1 BFU

The Basic Functional Unit (BFU) is shown in Figure 13.1. The BFU contains three major components:

- **256×8 memory** – the memory can function either as a single 256-byte memory or as a dual-ported, 128×8-bit memory in register-file mode. In register-file mode the memory supports two reads and one write operation on each cycle.
- **8-bit ALU** – the ALU supports the standard set of arithmetic and logic functions including NAND, NOR, XOR, shift, and add. With optional input inversion, this extends to include OR, AND, XNOR, and subtract. A configurable carry chain between adjacent ALUs allows cascading of ALUs to perform wide-word operations. The ALU also includes an 8×8 multiply-add-add operation; the multiply operation takes two operating cycles to deliver its

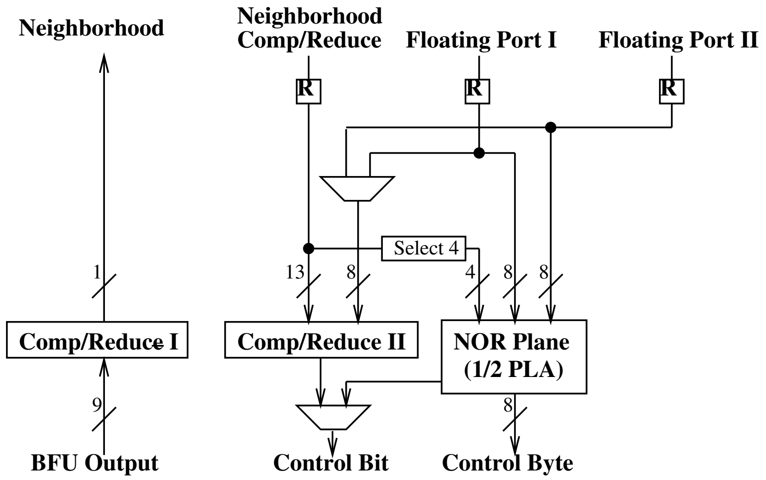


Figure 13.2: BFU Control Logic

results over the 8-bit BFU output, delivering the low 8 bits of the product on the first cycle and the high 8 bits on the second cycle.

- **Control Logic** – the control logic is composed of: (1) a local pattern matcher for generating local control from the ALU output (Figure 13.2 Left), (2) a reduction network for generating local control (Figure 13.2 Middle), and (3) a 20-input, 8-output NOR block which can serve as half of a PLA (Figure 13.2 Right). The local pattern matcher is used to reduce the datapath value to a condition bit such as zero detect, positive or negative test, or carry detect. The control bit produced by the reduction network is used to select among control contexts which we described in Section 13.2.4. The NOR plane allows us to perform programmable, bit-wise logically functions. This is the primary place where the datapath can be broken down to bits or composed from bits. Since the NOR plane acts on the bit level, it can be used to permute the bits in a byte or perform extract or deposit operations between two bytes.

MATRIX operation is pipelined at the BFU level with a pipeline register at each BFU input port. A single pipeline stage includes:

1. Memory read
2. ALU operation
3. Memory write and local interconnect traversal – these two operations proceed in parallel

The BFU can serve in any of several roles:

- **I-store** – Instruction memory for controlling ALU, memory, or interconnect functions
- **Data memory** – Read/Write memory for storage and retiming of data
- **RF+ALU slice** – Byte slice of a register-file-ALU combination
- **ALU function** – Independent ALU function

The BFU's versatility allows each unit to be deployed as part of a computational datapath or as part of the memory or control circuitry in a design.

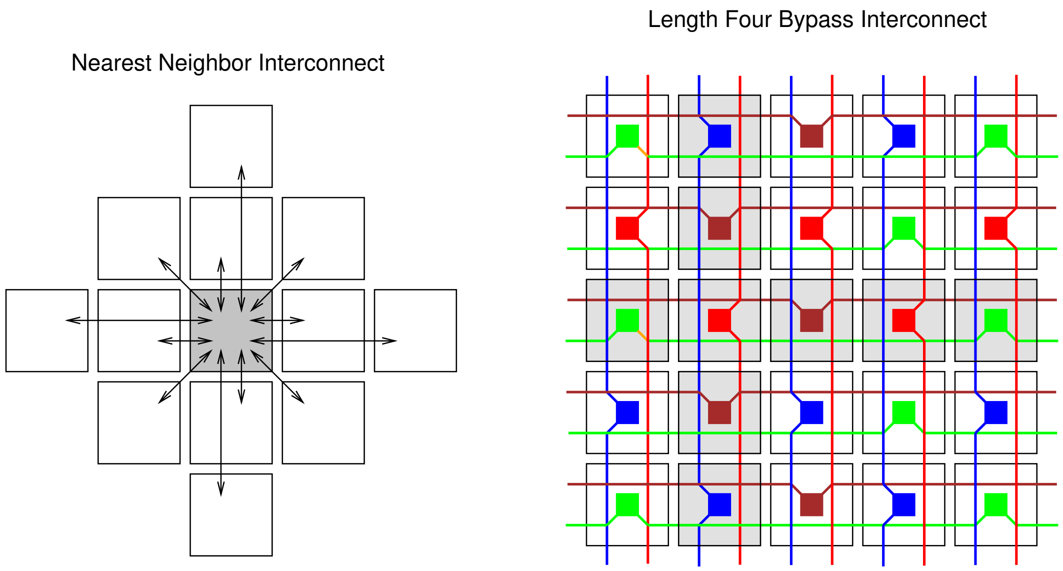


Figure 13.3: MATRIX Network

13.2.2 Network

The MATRIX network is a hierarchical collection of 8-bit busses. The interconnect distribution resembles traditional FPGA interconnect. Unlike traditional FPGA interconnect, MATRIX has the option to dynamically switch network connections. The network includes:

1. **Nearest Neighbor Connection** (Figure 13.3 Left) – A direct network connection is provided between the BFUs within two manhattan grid squares. Results transmitted over local interconnect are available for consumption on the following clock cycle.

2. **Length Four Bypass Connection** (Figure 13.3 Right) – Each BFU supports two connections into the level two network. The level two network allows corner turns, local fanout, medium distance interconnect, and some data shifting and retiming. Travel on the level two network may add as few as one pipeline delay stage between producer and consumer for every three level two switches included in the path. Each level two switch may add a pipeline delay stage if necessary for data retiming.

3. **Global Lines** – Every row and column supports four interconnect lines which span the entire row or column. Travel on a global line adds one pipeline stage between producer and consumer.

Notice that the *same* network resources deliver instructions, data, addresses, and control to the BFU ports. All of the eight BFU input ports (Figure 13.1) are connected to this same network, and all BFU outputs are routed through this network.

13.2.3 Port Architecture

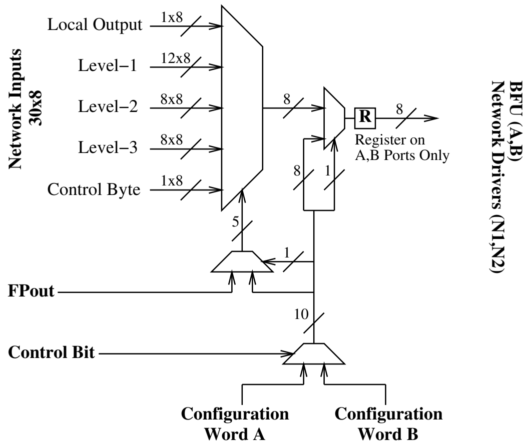


Figure 13.4: BFU Port Architecture

The MATRIX port configuration is one of the keys to the architecture's flexibility. The input ports are the primary source of MATRIX's *metaconfiguration*. Figure 13.4 shows the composition of the BFU network and data ports. Each port can be configured in one of three major modes:

1. **Static Value Mode** – The value stored in the port configuration word is used as a static value driven into the port. This is useful for driving constant data or instructions into a BFU. BFUs configured simply as I-Stores or memories will have their ALU function port statically set to pass memory output data. BFUs operating in a systolic array might also have their ALU function port set to the desired operation. For regular operations a BFU may be dedicated to that function and, in so doing, requires no instruction memory be allocated for control.

2. **Static Source Mode** – The value stored in the port configuration word is used to statically select the network bus providing data for the appropriate port. This configuration is useful in wiring static control or datapaths. Static port configuration is typical of FPGA interconnect.

3. **Dynamic Source Mode** – The value stored in the port configuration word is ignored. Instead the output of the associated floating port (see Figure 13.1) controls the input source on a cycle-by-cycle basis. This is useful when datapath sources need to switch during normal operation. For example, during a relaxation algorithm, a BFU might need to alternately take input from each of its neighbors.

The floating port and function ports are configured similarly, but only support the static value and static source modes.

13.2.4 Port Contexts

Matrix metaconfiguration information is also multicontext in two ways.

Control As shown in Figure 13.4, each port actually has two configuration words selected by a *control bit*. This control bit is generate by the NOR plane or reduction network (Comp/Reduce II) in the control portion of the BFU (Figure 13.2). This arrangement allows control data to locally affect each BFU's operation.

One common use of this control function is in a BFU which operates as the program counter. A typical program counter holds its value (`PC`) on the BFU output. In normal operation, the BFU simply increments its current value (`PC=PC+1`). When a branch test succeeds, the program counter BFU loads its value from its own memory (`PC=mem[PC]`) rather than incrementing. To arrange this, control logic is set to route the "take branch" condition on the control bit. One control context is used for the not taken branch case and simply configures the BFU to increment the PC. The other control context is used for the taken branch condition and configures the BFU to use the current PC as an address into memory for a read operation.

Since the control bit can come from the NOR plane, it can be slaved to any bit on any bus distributed to the BFU. This allows a controller to use a BFU or collection of BFUs as two context devices. A single datapath byte can control up to eight such BFUs independently if each BFU is configured to select a distinct bit from the control byte.

Global Additionally, the entire metaconfiguration data is replicated multiple times and controlled by a single, array-wide context select similar to the DPGA (Chapter 10). In our current microarchitecture we have four global context, two of which are hardwired and two of which are programmable. The hardwired contexts are intended for bootstrapping and device programming. They configure the entire device into a known configuration of datapaths so that metaconfiguration, configuration, and initial data can be loaded into the array. The two programmable contexts allow:

1. background loading of metaconfiguration data
2. assembly of new global context data without affecting the current, operating context
3. atomic swap between assembled configuration

The global contexts can also be used to provide DPGA-style multicontext swapping between configurations. Coupling the two programmable contexts with the two control contexts, the entire array can be treated as a four context device without dedicating BFU memory for context data.

13.2.5 Metaconfiguration Configuration

The metaconfiguration data for each BFU can be written by a BFU write operation. The metaconfiguration data is in a different address space from the BFU local memory. Access to the metaconfiguration data versus the normal BFU memory is controlled by the the instruction issued to the BFU memory function port (Figure 13.1). This arrangement allows the metaconfiguration to be loaded in one of several ways:

1. A hardwired context can make the entire device look like a memory so that an external device can perform memory-mapped writes to configure metaconfiguration and configuration data.
2. A hardwired context can setup the device to bootstrap load a configuration from a slave memory.

3. A controller can be configured on the array which can write metaconfiguration to other BFUs. There can be any number of controllers controlling any subsets of the array limited only by raw resource availability. More controllers can increase reconfiguration bandwidth at the expense of taking BFU resources away from datapath computations.

4. A BFU may write to its own metaconfiguration. This usually requires some assist from other BFUs. However, with the control contexts, there are useful configurations where a single BFU may reconfigure portions of itself. For example, a BFU in a systolic datapath could be configured primarily to perform one, fixed ALU operation. When a control event is signaled, its second control context could reload the ALU function port configuration from its local memory. When it returns to datapath operation, the BFU now performs the new operation. This basic reconfiguration scheme allows MATRIX to efficiently handle a variety of quasistatic instruction streams.

Note that the existence of two programmable, global contexts is useful for providing atomic, coordinated, array-wide context swaps. In typical use, the array would operate in one context while writing new configuration data into an unused, programmable configuration context. Once that context was fully programmed, the global context select would change effecting the array-wide switch.

13.2.6 Time-Switching

MATRIX ports can also operate in a time-switched mode, inspired by the time-switched input register (Section 12.1). In Chapter 12, we saw that the ability to latch and hold input values at designated microcycles, along with switched interconnect, allowed us to minimize the constraints required during design mapping and thereby perform physical mapping quickly. Each MATRIX port has a time matching unit as does memory write back. When metaconfiguration sets a BFU into time-switched mode, each input is loaded only on its programmed microcycle as with TSFPGA. The *timestep* for MATRIX is broadcast along a designated global line. In time-switching mode, the metaconfiguration dedicates these global lines and provides for the proper distribution of a timestep value. Typically, the remaining global lines will be dynamically switched to provide the necessary interconnect between BFUs. In situations where light multiplexing is all that is required, the control contexts may provide sufficient switched routing. For more heavily shared switching resources, global and bypass lines can be time-switched, with each getting its own BFU instruction store to control its operation. Time-switched routing will, of course, slow down MATRIX operation. This mode is intended primarily for fast, hands-off, automatic mapping during early development.

13.2.7 Resource Deployment Granularity

The primitives in the architecture do define a granularity at which resources must be deployed. Datapaths and non-local control paths can only come in 8-bit multiples. Context memories come in 256 instruction deep chunks. Compute elements come as 8-bit ALUs with 128-word register files.

Due to the flexible instruction distribution introduced above and discussed further in Section 13.4, MATRIX's granularity does not have the same kind of effects as conventional architectures (Chapter 9). For task requirements below 8-bits, the datapath suffers similar to traditional architectures. For task requirements above 8-bits, at most 7-bits of the datapath ever go wasted, and MATRIX does not waste space on instruction stores holding redundant data as would conventional 8-bit architectures.

13.2.8 Additional Information

For additional detail on the MATRIX microarchitecture see [Mir96].

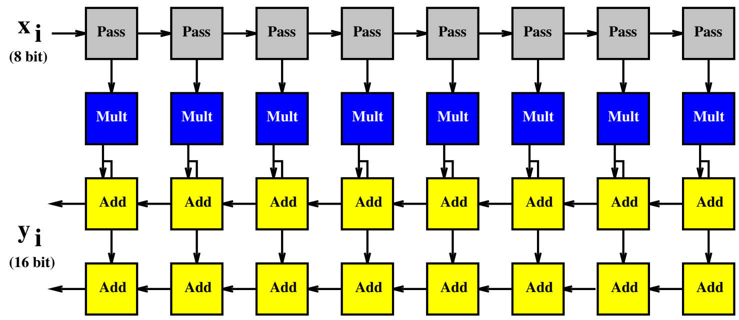


Figure 13.5: Systolic Convolution Implementation

13.3 Usage Example: Finite-Impulse Response Filter

In this section we present a range of implementation options for a single task, convolution, in order to illustrate MATRIX usage and further ground the features of this architecture. The convolution task is as follows: Given a set of k weights $\{w_1, w_2, \ldots w_k\}$ and a sequence of samples $\{x_1, x_2, \ldots\}$, compute a sequence of results $\{y_1, y_2, \ldots\}$ according to:

$$y_i = w_1 \cdot x_i + w_2 \cdot x_{i+1} + \cdots + w_k \cdot x_{i+k-1} \tag{13.1}$$

Systolic Figure 13.5 shows an eight-weight ($k = 8$) convolution of 8-bit samples accumulating a 16-bit result value. The top row simply carries sample values through the systolic pipeline. The middle row performs an 8×8 multiply against the constants weights, w's, producing a 16-bit result. The multiply operation is the rate limiter in this task requiring two cycles to produce each 16-bit result. The lower two rows accumulate y_i results. In this case, all datapaths (shown with arrows in the diagram) are wired using static source mode (Figure 13.4). The constant weights are configured as static value sources to the multiplier cells. Add operations are configured for carry chaining to perform the required 16-bit add operation. For a k-weight filter, this arrangement requires $4k$ cells and produces one result every 2 cycles, completing, on average, $\frac{k}{2}$ 8×8 multiplies and $\frac{k}{2}$ 16-bit adds per cycle.

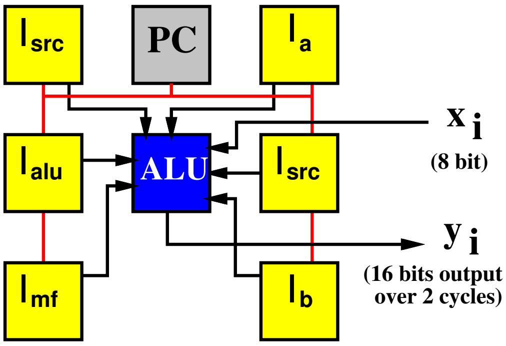
In practice, we can:

1. Use the horizontal level-two bypass lines for pipelining the inputs, removing the need for the top row of BFUs simply to carry sample values through the pipeline.

2. Use both the horizontal and vertical level-two bypass lines to retime the data flowing through the add pipeline so that only a single BFU adder is needed per filter tap stage.

3. Use three I-stores and a program counter (PC) to control the operation of the multiply and add BFUs, as well as the advance of samples along the sample pipeline.

The k-weight filter can be implemented with only $2k + 4$ cells in practice.

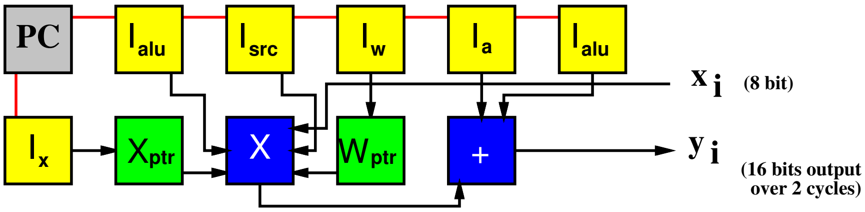


Label	ALU Op	PC
newsample	Rxp ← Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont1
	< Rxp > ← new x_i	(pipelined branch slot)
	Rxp ← 65	
xpcont1	< Rxp > ← new x_i	
	Rs ← < Rxp >	
	Rwp ← 1	
	Rw ← < Rwp >	
	Rs ← Rs × Rw	
	Rw ← ×-continue	
	Rl ← Rs; Match false	BNE enterloop
	Rh ← Rw	(pipelined branch slot)
innerloop	Rs ← Rs × Rw	
	Rw ← ×-continue	
	Rl ← Rs + Rl	
	Rh ← Rw +-continue Rh	
enterloop	Rxp ← Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont2
	Rs ← < Rxp >	(pipelined branch slot)
	Rxp ← 65	
	Rs ← < Rxp >	
xpcont2	Rwp ← Rwp + 1 ; Match $(k+1)$ (6 bits)	BNE innerloop
	Rw ← < Rwp >	(pipelined branch slot)
last	read Rl ; Match false	BNE newsample
	read Rh	(pipelined branch slot)

Figure 13.6: Microcoded Convolution Implementation

Microcoded Figure 13.6 shows a microcoded convolution implementation. The k coefficient weights are stored in the ALU register-file memory in registers 1 through k and the last k samples are stored in a ring buffer constructed from registers 65 through $64 + k$. Six other memory location (Rs, Rsp, Rw, Rwp, Rl, and Rh) are used to hold values during the computation. The ALU's A and B ports are set to dynamic source mode. I-store memories are used to drive the values controlling the source of the A and B input (two I_{src} memories), the values fed into the A and B inputs (I_a,I_b), the memory function (I_{mf}) and the ALU function (I_{alu}). The PC is a BFU setup to increment its output value or load an address from its associated memory as described in Section 13.2.4.

The implementation requires 8 BFUs and produces a new 16-bit result every $8k + 9$ cycles. The result is output over two cycles on the ALU's output bus. The number of weights supported



Label	Xptr unit	Wptr unit	PC	MPY unit	+-unit
firstsample	Xptr$\leftarrow$64	Wptr$\leftarrow$0			
	output Xptr	output Wptr		$<$ Xptr $>$ $\leftarrow$ new x_i	
nextsample	Xptr++ MOD $k \mid 64$	Wptr++		$<$ Xptr $>$ $\times$ $<$ Wptr $>$	
	output Xptr	output Wptr		$\times$-continue	Rlow $\leftarrow$ MPY-result
	Xptr++ MOD $k \mid 64$	Wptr++		$<$ Xptr $>$ $\times$ $<$ Wptr $>$	Rhigh$\leftarrow$ MPY-result
	output Xptr	output Wptr		$\times$-continue	Rlow $\leftarrow$ Rlow + MPY-result
innerloop	Xptr++ MOD $k \mid 64$	Wptr++; Match k	BNE innerloop	$<$ Xptr $>$ $\times$ $<$ Wptr $>$	Rhigh$\leftarrow$ Rhigh + MPY-result
	output Xptr	output Wptr	(pipelined branch slot)	$\times$-continue	Rlow $\leftarrow$ Rlow + MPY-result
last	output Xptr	output Wptr		$<$ Xptr $>$ $\times$ $<$ Wptr $>$	Rhigh$\leftarrow$ Rhigh + MPY-result
	Xptr++ MOD $k \mid 64$	Wptr$\leftarrow$0; Match false	BNE nextsample	$\times$-continue	$\boxed{\text{Rlow}}$ $\leftarrow$ Rlow + MPY-result
	output Xptr	output Wptr	(pipeline branch slot)	$<$ Xptr $>$ $\leftarrow$ new x_i	$\boxed{\text{Rhigh}}$ $\leftarrow$ Rhigh + MPY-result

Boxed values in last are the pair of y_i output bytes at the end of each convolution.

Figure 13.7: Custom VLIW Convolution Implementation

is limited to $k \leq 61$ by the space in the ALU's memory. Longer convolutions (larger k) can be supported by deploying additional memories to hold sample and coefficient values.

Custom VLIW (Horizontal Microcode) Figure 13.7 shows a VLIW-style implementation of the convolution operation that includes application-specific dataflow. The sample pointer (Xptr) and the coefficient pointer (Wptr) are each given a BFU, and separate ALUs are used for the multiply operation and the summing add operation. This configuration allows the inner loop to consist of only two operations, the two-cycle multiply in parallel with the low and high byte additions. Pointer increments are also performed in parallel. Conventional digital signal processors are generally designed to handle this kind of filtering problem well, and, not coincidentally, the datapath used here is quite similar to modern DSP architectures. Most of the I-stores used in this design only contain a couple of distinct instructions. With clever use of the control PLA and configuration words, the number of I-stores can be cut in half making this implementation no more costly than the microcoded implementation.

As shown, the implementation requires 11 BFUs and produces a new 16-bit result every $2k + 1$ cycles. As in the microcoded example the result is output over two cycles on the ALU output bus. The number of weights supported is limited to $k \leq 64$ by the space in the ALU's memory.

VLIW/MSIMD Figure 13.8 shows a Multiple-SIMD/VLIW hybrid implementation based on the control structure from the VLIW implementation. As shown in the figure, six separate convolutions are performed simultaneously sharing the same VLIW control developed to perform a single convolution, amortizing the cost of the control overhead. To exploit shared control in this manner,

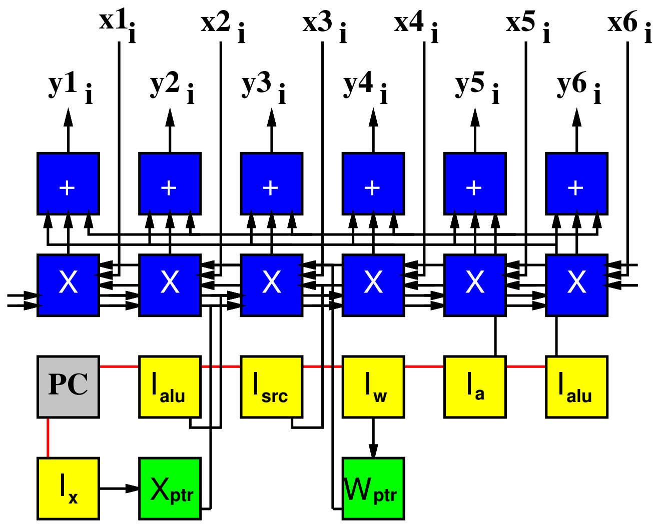


Figure 13.8: VLIW/MSIMD Convolution Implementation

the sample data streams must receive data at the same rate in lock step.

When sample rates differ, separate control may be required for each different rate. This amounts to replicating the VLIW control section for each data stream. In the extreme of one control unit per data stream, we would have a VLIW/MIMD implementation. Between the two extremes, we have VLIW/MSIMD hybrids with varying numbers of control streams according to the application requirements.

Comments Of course, many variations on these themes are possible. The power of the MATRIX architecture is its ability to deploy resources for control based on application regularity, throughput requirements, and space available. In contrast, traditional microprocessors, VLIW, or SIMD machines fix the assignment of control resources, memory, and datapath flow at fabrication time, while traditional programmable logic does not support the high-speed reuse of functional units to perform different functions.

13.4 Flexible Instruction Distribution

MATRIX supports flexbile allocation of instruction control resources as a consequence of the BFU, network, and port architecture described in Section 13.2.

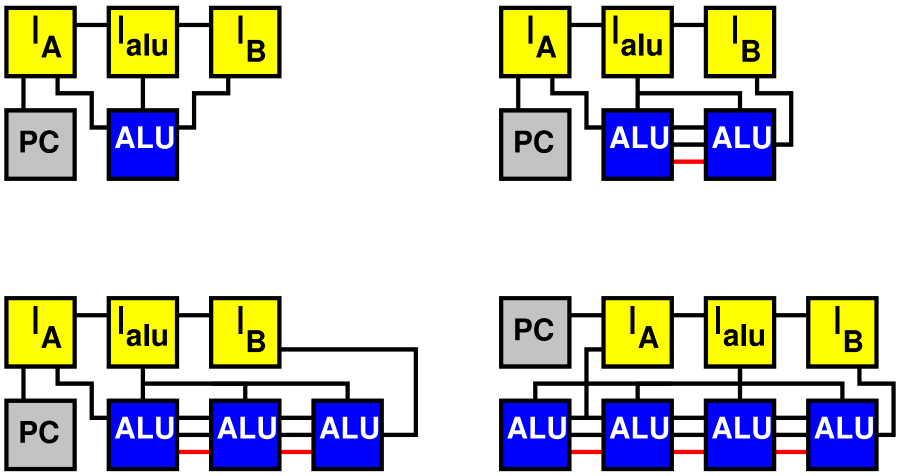
Instruction Depth We can directly select an instruction depth of 1 or $n \cdot 256$. If the instruction does not need to change, we can directly configure it via static value in the port metaconfiguration. If the instruction changes between only two values, we can use the control context. In certain situations, we can use the global contexts to support up to four contexts using the metaconfiguration contexts. When instruction need to change among more than a few values, we can allocate a BFU as an instruction store and use static source mode to configure said distribution. If more than 256 instructions are needed, we can use dynamic source mode to expand the selection to one of several different memory sources, allowing a large instruction space.

Note that conventional FPGAs are characterized by an instruction depth of one, while an instruction depth of 256-1024 is typical for conventional processor architectures.

Datapath Granularity We can control the datapath granularity to any multiple of 8-bits. The network allows fanout at all three levels of the hierarchy. To build an $8n$-bit wide datapath, we need only configure the n BFUs used as datapath elements to take their instructions from the same instruction memories (See Figure 13.9). Notice that this is not the same as having a conventional microarchitecture with $w = 8$, as introduced in Chapters 8 and 9. In a conventional case, each 8-bit datapath element would have its own instruction memory, whereas, for MATRIX, we get to use a single instruction memory for all n datapath elements (See Figure 13.10).

Notice also that the ability to assign instruction memories to composed datapaths is also different from the *segmentable* datapaths in modern multimedia processors (Section 4.7), multigauge SIMD architectures (*e.g.* [Sny85] [BSV+95]), or the Kartashev dynamic architecture [KK79]. In these architectures, all the bit processing elements in a predefined datapath perform the same operation. These generally exhibit SIMD instruction control for the datapath, but can be dynamically or quasistatically reconfigured to treat the n bit datapath as k, $\frac{n}{k}$-bit words, for certain, restricted, values of k. MATRIX does not have to perform the same ALU function across all datapath segments like these architectures.

Instruction Streams The number of instruction streams on a MATRIX component is limited only by the availability of resources. If the entire operation is efficiently handled by a systolic architecture, no resources, BFUs or interconnect need be sacrificed to control. For highly regular operations where SIMD control is effective, MATRIX need only dedicate a single set of BFUs to broadcast the instructions to the rest of the array. As the application needs more, independent instruction streams, more BFUs can be allocate to provide separate instruction streams. Like MSIMD (*e.g.* [Bri90, Nut77]) or MIMD multigauge [Sny85] designs, the array can be broken into units operating on different instructions. Synchronization between the separate functions can be lock-step VLIW or completely orthogonal depending on the application. Unlike traditional MSIMD or multigauge MIMD designs, the control processors and array processors are built out of the same

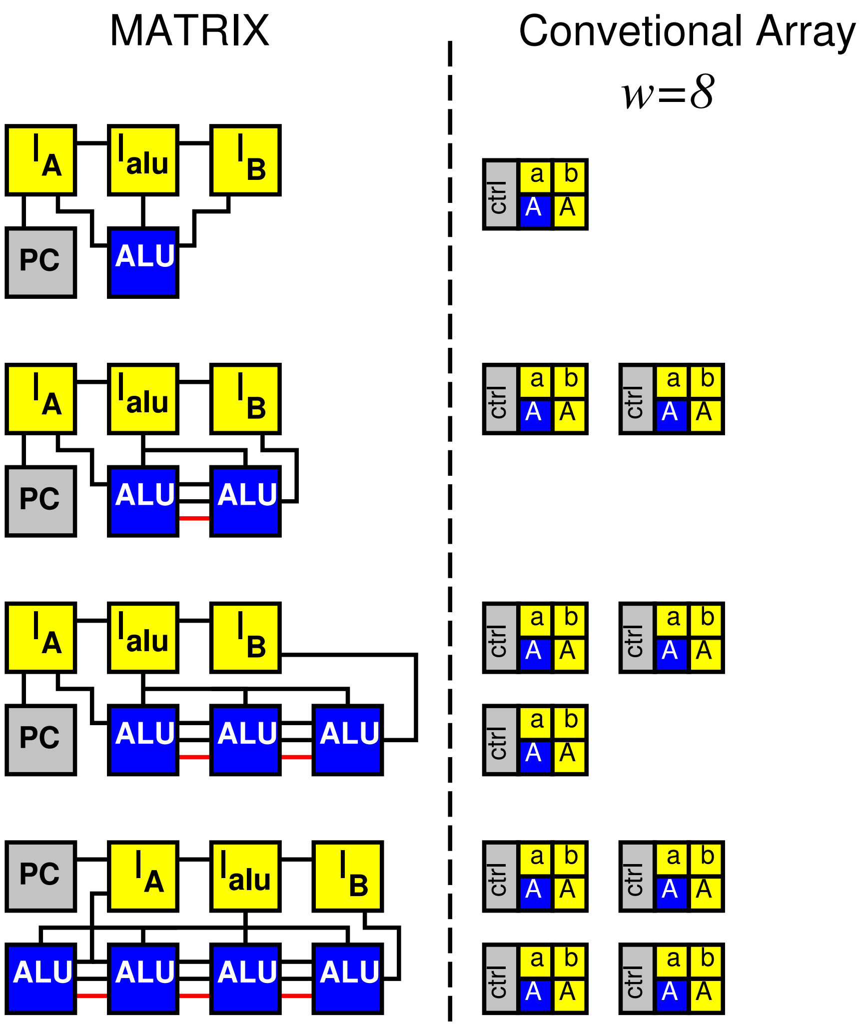


Here we show 8-, 16-, 24-, and 32-bit datapaths built on top of MATRIX. Configurable instruction distribution allows multiple datapath BFUs to share a single set of instruction stores.

Figure 13.9: Configurable Datapaths

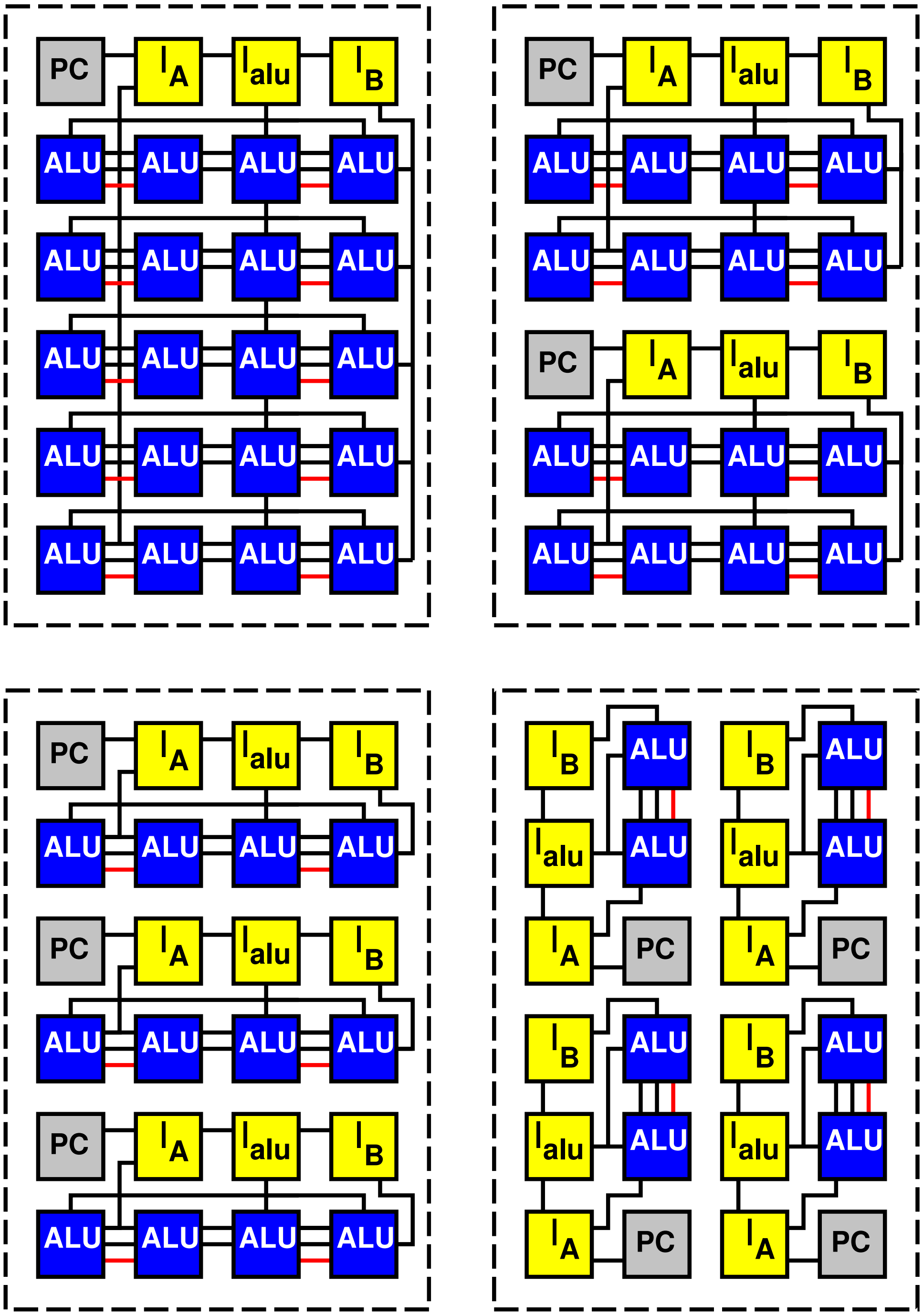
building block resources and networking. Consequently, more array resources are available as less control resources are used (See Figure 13.11).

Control Streams Similarly, MATRIX can handle any number of independent control streams. Each can have their own program counter realizing a MIMD architecture, or they can all be slaved to a single program counter realizing a VLIW architecture (See Figure 13.12). Between these extremes, any number of instruction streams may be associated with each program counter in the same way that any number of datapath elements can be slaved to a single instruction stream. Again, as we noted for datapath granularity and instruction streams, control resources come from the same pool of resources as datapath elements – as an application can be described with fewer control streams, more BFUs are available to serve as datapath elements.



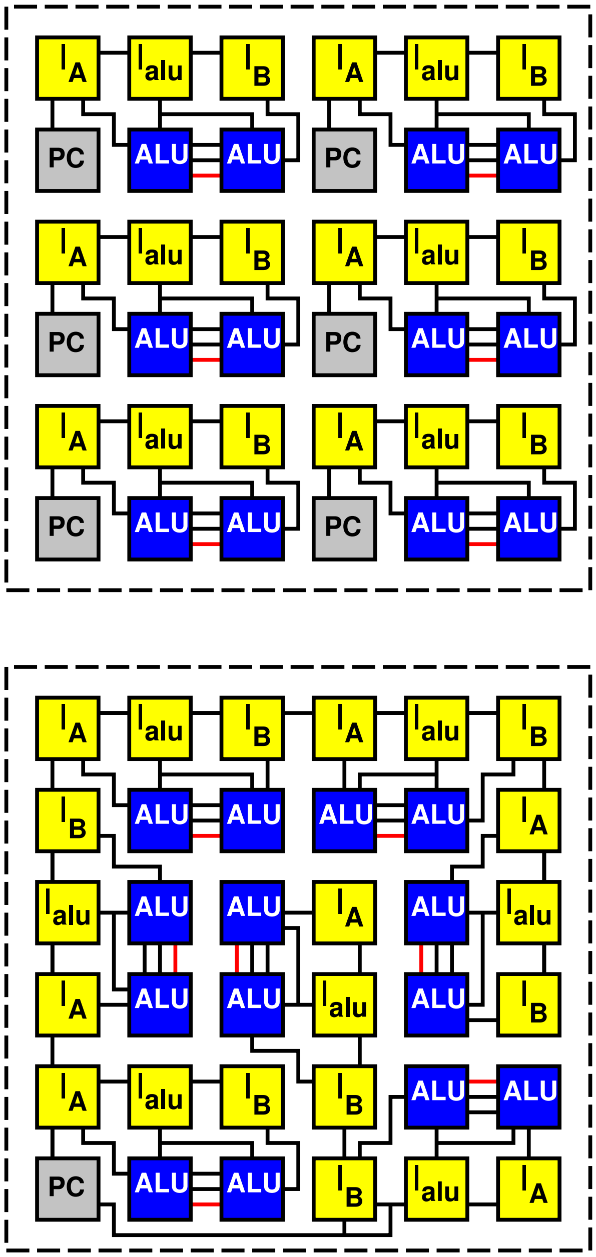
While using an 8-bit primitive datapath element, the MATRIX microarchitecture is very different from a conventional architecture with $w = 8$. Conventional architectures rigidly bind instructions and control with datapaths, whereas MATRIX deploys the resources separately. Consequently, MATRIX can share control and instruction memory across a composed datapath, whereas conventional architectures do not allow such sharing.

Figure 13.10: Datapath Composition: MATRIX versus Conventional $w = 8$ Architecture



Here we show one, two, three, and four instruction streams controlling a set of 16-bit datapaths. Given a fixed array size, as the number of independent instruction streams decrease, more array resources can be dedicated to SIMD datapaths.

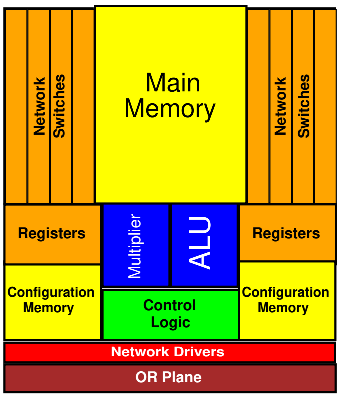
Figure 13.11: Configurable Instruction Streams



TOP – MIMD control with six, 16-bit data streams, each with indepdent control

Bottom – VLIW control with seven, 16-bit data streams directed by a single control unit

Figure 13.12: Configurable Control Streams



Technology	0.5μ CMOS
BFU Size	1.5mm $\times$ 1.2mm
	($1.8\text{mm}^2 \approx 29\text{M}\lambda^2$)
Data Width	8-bit
Memory	256×8
Cycle	10 ns (estimate)

Figure 13.13: MATRIX BFU Composition

Unit	Fraction
Main Memory	26%
ALU+multiplier	7%
Switches and Drivers	42%
Registers including time-switch	6%
Port Config	9%
Control (with config)	10%

Table 13.1: Area Breakdown for Prototype MATRIX BFU Implementation

13.5 MATRIX Implementation

Figure 13.13 shows the composition of the prototype BFU developed by Ethan Mirsky [Mir96], along with its size and projected performance. Table 13.1 shows the area breakdown from the prototype implementation. As described in Section 13.2, MATRIX operation is pipelined at the BFU level allowing high speed implementation. With only a small memory read, an ALU operation, and local network distribution, the basic cycle rate can be quite small – at least comparable to microprocessor clock rates. 100 MHz operation is the target for the prototype design. At 1.8mm^2, 100 BFUs fit on a $17\text{mm}\times14\text{mm}$ die. A 100 BFU MATRIX device operating at 100MHz has a peak performance of 10^{10} 8-bit operations per cycle (10 Gop/s).

Unit	Elements	Element Size	Total Size	Fraction
Port Switching	30×8×8=1920	2.5Kλ^2	4.8Mλ^2	46%
Main Memory	256×8=2048	1.2Kλ^2	2.5Mλ^2	24%
Config. Memory	135×8=1080	1.2Kλ^2	1.3Mλ^2	12%
(NOR)	20×8			
Control (match)	+9×1			
(control bit)	+21×1 = 190	3Kλ^2	0.6Mλ^2	6%
ALU	8b	20Kλ^2	0.2Mλ^2	2%
MPY	8×8=64	7Kλ^2	0.5Mλ^2	5%
Registers	8×8+16=80	4Kλ^2	0.3Mλ^2	3%
TS	9	16Kλ^2	0.15Mλ^2	1%
Sum			10-11Mλ^2	

Table 13.2: MATRIX BFU Composition Estimate

MATRIX is sufficiently different from conventional architectures that our model from Chapter 9 does not quite apply. We can account for the specific composition of our microarchitecture. Table 13.2 summarizes the constituent elements of the MATRIX BFU along with estimated areas. The MATRIX size estimate is about one-third the size of the prototype implementation, suggesting there is considerable room for improvement relative to the prototype design. The prototype is a first-generation, one student, university prototype of a novel architecture. As such, it is not surprising that it is not the most compact design.

Nevertheless, both area views agree on rough area proportions. Switches and drivers occupy roughly 45% of the area. The main BFU memory accounts for 25% of BFU area. Metaconfiguration makes up roughly 10% of the BFU. The ALU and multiplier composes only 7% of the area.

13.6 Building Block Efficiency

The MATRIX BFU serves several roles. It is interesting to consider its efficiency in each of these roles.

13.6.1 Memory

MATRIX packs 2048 RAM bits into 28.8Mλ^2 in the prototype or, perhaps, 10Mλ^2 in an optimized design. If we only use the BFU for its memory array, each memory bit cell is effectively 14Kλ^2, or 5Kλ^2, respectively. Of course, the MATRIX memory only comes in 256×8 blocks and will, therefore, be less dense as smaller memories or memories which are not even multiples of this size are needed.

Versus Custom Memory Since memory only accounts for 25% of the MATRIX array, MATRIX memory is only one-fourth as dense as a custom memory of the same size.

Versus Gate Array Memory A RAM cell implemented in a gate-array process is roughly 6Kλ^2 (*e.g.* [Fos96]). This size comparable to the amortized bit area according to the model (5Kλ^2) and is half of the size of the amortized bit cell area in the current prototype (14Kλ^2).

Versus Xilinx 4K Memory The Xilinx 4K series [Xil94b] can use each CLB as 32 bits of RAM. From Table 4.13, we know a Xilinx 4K CLB is 1.25Mλ^2, making each memory bit roughly 39Kλ^2, which is 3-7$\times$ larger than the amortized MATRIX RAM bit area.

13.6.2 Datapath Elements

Versus Hardwired ALU The ALU and multiplier make up only 7% of the BFU area. This suggests a datapath of BFUs could be a good 10-20$\times$ less dense than a full custom implementation of the same task.

Versus Systolic, Reconfigurable ALUs In systolic dataflow applications the ALU may be used as a functional unit without the memory. The 25% of the BFU area which is in the memory sits idle, as well as the $\left(\frac{1}{8}\right) \cdot 45\% \approx 6\%$ for the extra function port. The control logic constitutes another 6-10% of the BFU area. Consequently, the MATRIX BFU is roughly 1.5$\times$ larger than we might see in a pure mesh of reconfigurable ALUs, or perhaps in an architecture where the memory and ALU were independent resources.

ALU Bit Ops In Table 4.24, we have already noted that the prototype MATRIX achieves 28 ALU Bit Ops/λ^2s which is roughly 3$\times$ the computational density of processors (See Table 4.2). At the same time this is 4$\times$ lower than the peak computational density offered by single-context FPGAs (See Table 4.13). If we can realize the compaction suggested by the model, MATRIX can achieve 80 ALU Bit Ops/λ^2s, bringing its peak density almost comparable to FPGAs.

Adder A cascaded 16-bit add can occur in one cycle on 2 MATRIX BFUs. Assuming the BFUs are used only for the add, this consumes a capacity of $28.8\text{M}\lambda^2 \cdot 2 \cdot 10\text{ns} = 0.58\lambda^2\text{s}$. An XC4005-5 performs a 16-bit add in 20.5ns on 9 CLBs, taking a capacity of $0.23\lambda^2\text{s}$, which is only $2\times$ more efficient than the MATRIX prototype.

Multiply In Chapter 5 we reviewed custom and programmable multiply implementations. Even with the large prototype area, MATRIX achieved comparable multiply density to the best programmable devices (See Table 5.3). MATRIX is 10-100$\times$ less computationally dense at multiplication than full custom multiplies, which is consistent with the fact that multiplier occupies only 3% of the BFU area. At the same time the hardwired multiplier makes the MATRIX prototype 3-10$\times$ more computationally dense then FPGAs on multiply operations.

13.7 Image Processing Examples

To get a concrete view of MATRIX application performance, we will examine several image processing primitives implemented in custom and semi-custom silicon and compare them to MATRIX, FPGA, and microprocessor implementations of the same task. LSI's real-time DSP chip set [Rue89] is used to define the tasks and provide the custom implementations. The real-time chip set includes:

1. Variable Shift Register (VSR)
2. Rank Value Filter (RVF)
3. Binary Filter and Template Matcher (BFIR)
4. Multibit FIR Filter (MFIR)

We use area and timing from the prototype for the purposes of conservative MATRIX comparisons.

13.7.1 VSR

LSI's variable shift register takes in byte wide data and delays it a specified number of clock cycles. It provides eight, equidistant outputs. The maximum delay supported by the LSI component is $8 \cdot 512 = 4096$ clock cycles. That is, given a sequence of inputs:

$$x_n, x_{n+1}, x_{n+2}, \ldots$$

On the cycle n when x_n arrives, the VSR outputs eight values:

$$\begin{aligned} y1_n &= x_{(n-1\cdot(4\cdot l+8))} \\ y2_n &= x_{(n-2\cdot(4\cdot l+8))} \\ &\vdots \\ y8_n &= x_{(n-8\cdot(4\cdot l+8))} \end{aligned}$$

Here l is a value between 0 and 126. LSI implements their VSR in 64mm^2 in a 1.5μ CMOS process (114Mλ^2) using a semicustom standard cell methodology. The LSI VSR runs at a 26 MHz clock rate (38.5ns clock).

A MATRIX implementation providing the full, worst-case functionality of the VSR requires two BFUs to implement each 512 byte tap and two BFUs to implement a 9-bit modulo counter, for a total of 18 BFUs (See Figure 13.14). The memory BFUs implement the shift register by alternately reading and writing from their main memory. The control contexts are programmed to support the two instructions, read and write. The counter counts on every cycle from zero to $4 \cdot (l + 2) - 1$. The low bit of the counter is selected as the control bit on the memories while the high 8 bits serve as the memory address. The match unit on the counter is set to look for $4 \cdot (l + 2) - 1$. When a match occurs, the counter executes a load zero control context instead of the normal increment context. The 18 BFUs take 28.8M$\lambda^2 \cdot 18 =$ 518.4Mλ^2. Operating on the two clock macrocycle, the MATRIX VSR can run at 50MHz (20ns macroclock).

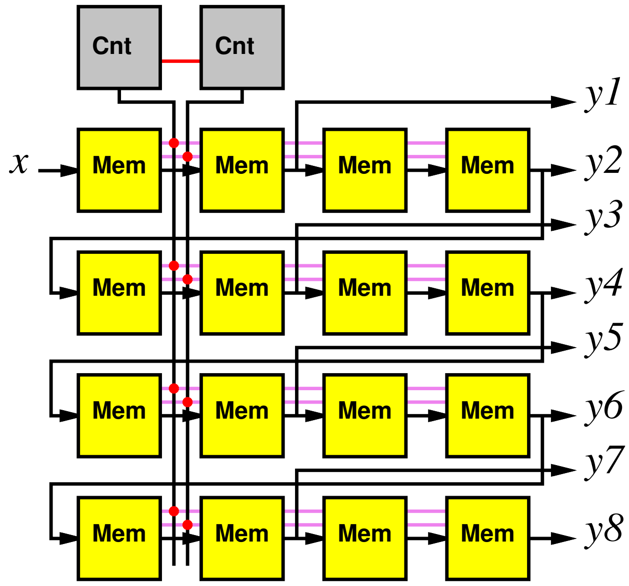


Figure 13.14: MATRIX Implemenation of Full 8-TAP, 4096 shift, VSR

```
innerloop   addi r1,#1,r1
            ld BUFF[r1],r2
            st OUTPUT,r2
            ld INPUT,r2
            st BUFF[r1],r2
            blt r1,r3,innerloop
            ----------------------
            move r0,r1
            bne r0,r0,innerloop
```

Figure 13.15: Processor Implementation of VSR

A typical processor implemenation of VSR (See Figure 13.15) takes 6 instructions per tap in a tight loop. For the full 8 tap VSR, the processor implementation requires 48 instructions. MIPS-X [HHC+87], one of the highest capacity processors we reviewed in Table 4.2, is 68Mλ^2. With a 50ns clock cycle, the 48 instructions will dictate, at least, a 2400ns macroclock.

An FPGA implementation would be dominated by data memory. A pure 4-LUT design would require up to 4096$\times$8=32K cells. At 600Kλ^2, a low-end estimate for 4-LUT size (See Table 7.1), this is 19.7Gλ^2. Exploiting the memory in an XC4000 part, we can pack 16$\times$2 bits per CLB, requiring 256$\times$4=1K CLBs or 1.28Gλ^2. The full shift register approach is trivial and should be very fast, so we will assume 100MHz operation. Exploiting the XC4000 memories will require both a read and a write operation as with MATRIX so we will assume it can achieve 50MHz operation.

Implementation	LSI	MATRIX	MIPS-X	4-LUT	XC4K
Area	$114\text{M}\lambda^2$	$518\text{M}\lambda^2$	$68\text{M}\lambda^2$	$19.7\text{G}\lambda^2$	$1.28\text{G}\lambda^2$
Cycle	38.5ns	20ns	2400ns	10ns	20ns
Capacity	$4.4\lambda^2$s	$10.4\lambda^2$s	$163\lambda^2$s	$197\lambda^2$s	$25.6\lambda^2$s
Ratio	1.0	2.4	37	45	5.9

Table 13.3: VSR Implementation Comparison

Table 13.3 compares the VSR implementations. The MATRIX implementation is 2.4$\times$ larger than the semicustom LSI implementation, 2.5$\times$ smaller than the XC4000 implementation, and 16$\times$ smaller than the processor implementation. If the shift register requires less than 2048 delay slots, MATRIX can implement each tap with a single BFU and use a single counter. This cuts the implementation area and capacity in half, bringing it within 20% of the capacity of the LSI implementation. Smaller shift registers with fewer taps will allow further reduction in BFUs for the MATRIX implementation. Capacity requirements for the FPGA implementations similarly reduce with total shift register length. The capacity required for the processor implementation will decrease with the number of taps.

Implementation	LSI	MATRIX	MIPS-X
Area	235Mλ^2	11117Mλ^2	68Mλ^2
Cycle	37ns	20ns	32450ns
Capacity	8.7λ^2s	222λ^2s	2221λ^2s
Ratio	1.0	26	260

Table 13.4: RVF Implementation Comparison

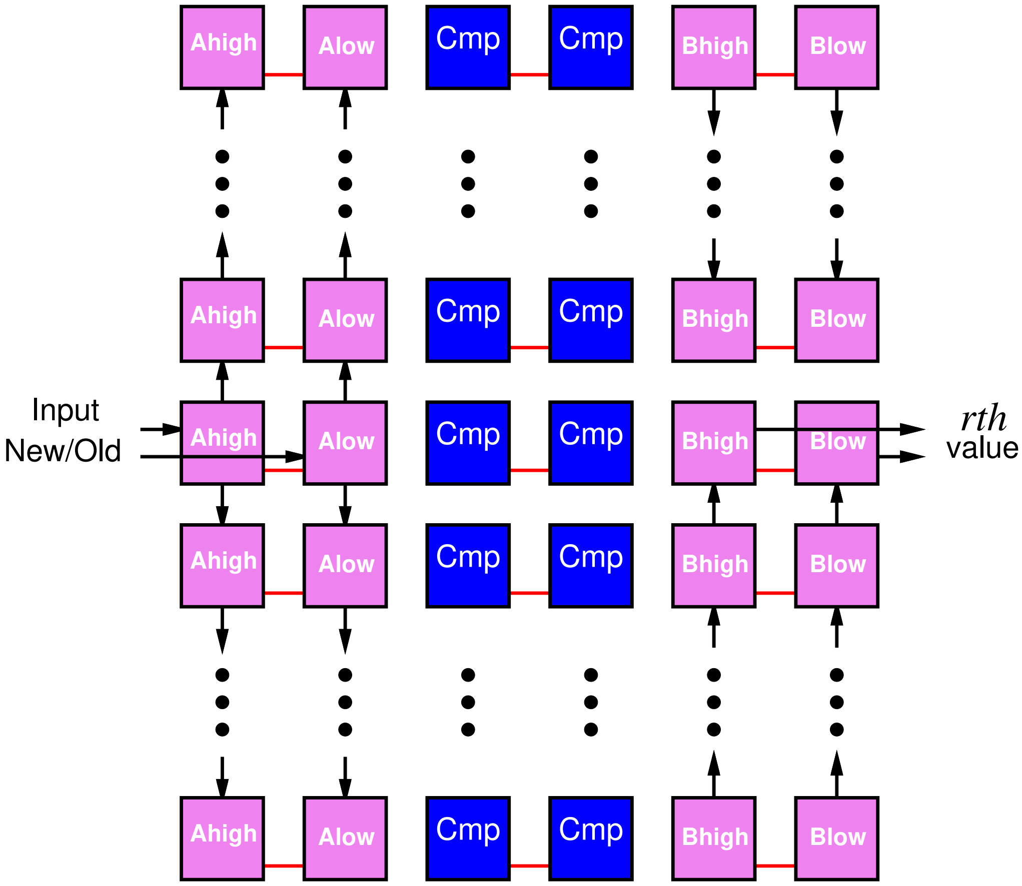
13.7.2 RVF

LSI's rank value filter selects the rth largest 12-bit value within a 64 sample window. That is, on each cycle, the component takes in a new 12-bit sample, x_i. It looks at the previous 64 values (x_i, x_{i-1}, ..., x_{i-63}), and selects the rth largest, which it outputs as y_i. If $r = 1$, it implements a maximum filter; if $r = 64$, it implements a minimum filter, and if $r = 32$, it implements a median filter. The LSI implementation occupies 132mm^2 in a 1.5μ CMOS process (235Mλ^2) using an array design methodology. The RVF runs at a 27 MHz clock rate (37ns clock).

The MATRIX implementation of RVF maintains a completely ordered list of the 64 window values using a systolic priority queue scheme similar to [Lei79]. The systolic priority queue allows it to compute incremental updates to the list ordering rather than recalculating the entire ordering on each cycle. To simulate the 64 tap window scheme, the systolic queue supports both an insert and a delete operation. Each macrocycle requires two microcycles – one in which the old value is deleted and one in which the new value is inserted. A fixed delay register scheme like the VSR is used to retime the old value for deletion 64 macrocycles later.

Using this style, an n-tap, w-bit wide MATRIX RVF implementation requires $3n\left\lceil\frac{w}{8}\right\rceil + 2$ BFUs, or 386 BFUs for the 64 tap, 12-bit case as implemented in the LSI filter. Each tap requires two active data swap registers (A and B) and a comparator, each of which needs to be as wide as the sample data. Figure 13.16 shows the basic array structure for the 12-bit sample case where two BFUs are required for each register and comparator. The additional two BFUs are used for the retiming memory and its associated counter. Figure 13.17 shows details of the datapath for a tap slice and its adjacent elements. The A registers are used to propagate insert and delete values while the B registers are used to hold sorted values. A values propagate away from the rth item and B values propagate toward it. By inserting data at the rth value location, we obtain an update latency of only one macrocycle or two primitive MATRIX cycles. The logic for a datapath slice is described in Figure 13.17. Note that the logic and datapath shown are for a tap position below the rth position in the array. The logic and flow are reversed for tap positions above the array. Figure 13.18 shows the control setup used to implement the datapath logic providing single cycle throughput for each comparison and swap operation.

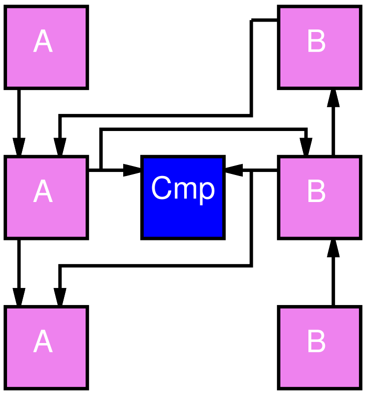
We use a similar insert and delete structure for the processor RVF implementation which is shown in Figure 13.19. For any width less than the processor word size, the processor implementation requires $10n + 9$ instructions in a tight loop. For the full 64 tap VSR, the processor implementation requires 649 instructions. Again, using the MIPS-X processor this requires 68Mλ^2 and $649 \cdot 50\text{ns} = 32450\text{ns}$.



Internal datapath connections omitted – See Figure 13.17.

Figure 13.16: MATRIX RVF Array

Table 13.4 compares the RVF implementations. The MATRIX implementation is 26× larger than the custom implementation and 10× smaller than the processor implementation. If less taps are required, both the matrix and the processor implementation decrease linearly in the number of taps. For 8-bit or smaller sample values, the MATRIX implementation will halve its datapath requirements. If one only wants to filter for the maximum or minimum value, a straightforward shift and compare reduce scheme will only require $2nw$ BFUs and operate at 100MHz throughput. For a maximum or minimum filter, the MATRIX implementation requires less capacity than the LSI RVF for 8-bit filters with less than 16 taps or 12-bit filters with less than 8 taps.



if	(old_sample)	
	if ($A = B$)	
		$B \leftarrow B_{prev}$
		$A_{next} \leftarrow$ MIN-VALUE
	else	$B \leftarrow B$
		$A_{next} \leftarrow A$
else	if ($A > B$)	
		$B \leftarrow A$
		$A_{next} \leftarrow B$
	else	$B \leftarrow B$
		$A_{next} \leftarrow A$

Figure 13.17: RVF Dataslice and Logic for Cells Below rth Postion

Cmp	Control Bit	old bit
B	Control Bit	Compare unit's match output
	NOR computes	old $\wedge$ (Select B_{prev} Source) $\vee$ $\overline{\text{old}}$ $\wedge$ (Select A Source)
	Control Context 0	Statically Route B from B
	Control Context 1	NOR plane specification selects B input
A	Control Bit	Compare unit's match output
	NOR computes	old $\wedge$ (Select MIN-VALUE Source) $\vee$ $\overline{\text{old}}$ $\wedge$ (Select B Source)
	Control Context 0	Statically Route A from A_{prev}
	Control Context 1	NOR plane specification selects A input

Figure 13.18: Control for MATRIX RVF for Cells Below rth Postion

```
//r3 – number of taps
//r5 – delay ring buffer head
//OLD – ring buffer head
//BUFF – sorted result
new          ld new,r4
             mov r0,r1
             ld OLD[r5],r6
             st OLD[r5],r4
             beq r5,r3,resetoldp
             addi,r5,#1,r5
findloop     ld BUFF[r1],r2
             ble r2,r4,insert
             addi,r1,#1,r1
             blt r1,r3,findloop
insert       st BUFF[r2],r4
             addi,r1,#1,r1
             ld BUFF[r1],r4
             blt r1,r3,insert
findold      mov r3,r1
             ld BUFF[r1],r4
             addi,r1,#1,r1
removeloop   ld BUFF[r1],r2
             st BUFF[r1],r4
             beq r2,r6,done
             mov r4,r2
             addi,r1,#1,r1
             blt r1,r3,removeloop
             b next
resetoldp    mov,r0,r5
             b findloop
```

Figure 13.19: Processor Implementation of RVF

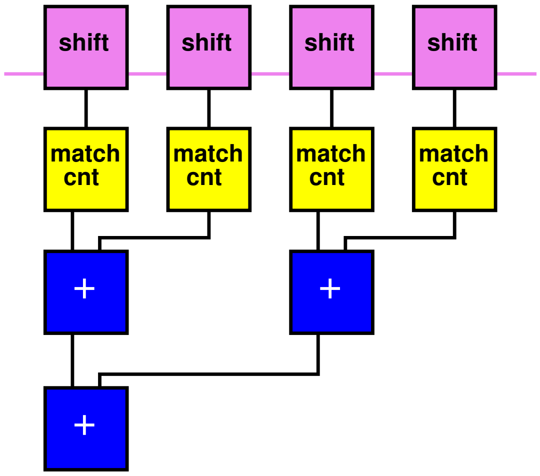


Figure 13.20: MATRIX BFIR Datapath

13.7.3 BFIR

LSI's binary filter and template matcher performs binary template matching across a 1024 bit template. That is:

$$y(n) = \sum_{i=0}^{1023} \left(c_i \text{ AND } \left(v_i \text{ XOR } x(n-i)\right)\right) \tag{13.2}$$

Here v is a vector of 1024 bit match values and c is a mask indicating which positions are "don't care" values and should be ignored. LSI implements their VSR in 88mm^2 in a 1.5μ CMOS process (156Mλ^2) using a full custom design methodology. The LSI BFIR runs at a 27 MHz clock rate (37ns clock).

The MATRIX implementation comes in three parts shown in Figure 13.20. A set of shift registers provide the bit level samples. A set of BFUs use their memories to perform matching and counting, starting with 8 bits of input and producing a 4-bit sum of the number of matches. Finally, an adder tree reduces the partial sums to a single result. To handle the 1024 tap problem, MATRIX requires $\frac{1024}{8} = 128$ BFUs for bitwise shifting and another 128 BFUs for matching. The sum tree is 7 stages deep. Since the final two stages add 9- and 10-bit sums, they each require 2 BFUs per addition, while each of the others requires a single BFU per sum, making for a total of 130 BFUs in the adder tree. Together, the MATRIX implementation requires 386 BFUs (11.1Gλ^2) and can operate at the full 100MHz basic cycle rate.

The processor implementation shown in Figure 13.21 stores and masks data in 32-bit units to exploit its datapath. It also uses a programmed lookup table to count ones. The processor only counts ones a byte at a time so that the count one's lookup table can fit in a reasonably sized data cache. The main loop takes 25 instructions per word. For a 1024 tap problem, this makes $\left(\frac{1024}{32}\right) \cdot 25 = 32 \cdot 25 = 800$ total instructions. The MIPS-X processor implementation then is 68Mλ^2 and $800 \cdot 50\text{ns} = 40000\text{ns}$.

```
//r14 – number of taps
//r15 – byte mask
//BUFF – stored input
//CARE – mask bits to check
//MASK – values to check
//CNTONES – lookup table to count ones in a byte
new  mov r0,r1
     ld BUFF[r1],r2
     addi,r0,TAPS,r14
     addi,r0,#0xff,r15
     addi,r0,r0,r6
top  ld BUFF[r1],r3
     sh r2,r3,r2,#1
     ld MASK[r1],r4
     xor r4,r2,r5
     ld CARE[r1],r4
     and r4,r5,r5
     and r5,r15,r4
     ld CNTONES[r4],r4
     add r4,r6,r6
     asr r5,r5,#8
     and r5,r15,r4
     ld CNTONES[r4],r4
     add r4,r6,r6
     asr r5,r5,#8
     and r5,r15,r4
     ld CNTONES[r4],r4
     add r4,r6,r6
     asr r5,r5,#8
     and r5,r15,r4
     ld CNTONES[r4],r4
     add r4,r6,r6
     st BUFF[r1],r2
     move r3,r2
     addi r1,#1,r1
     ble r1,r14,top
```

Figure 13.21: Processor Implementation of BFIR

Implementation	LSI	MATRIX	MIPS-X	XC4K
Area	$156\text{M}\lambda^2$	$11.1\text{G}\lambda^2$	$68\text{M}\lambda^2$	$2.32\text{G}\lambda^2$
Cycle	37ns	10ns	40000ns	10ns
Capacity	$5.8\lambda^2\text{s}$	$111\lambda^2\text{s}$	$2720\lambda^2\text{s}$	$23\lambda^2\text{s}$
Ratio	1.0	19	470	4.0

Table 13.5: BFIR Implementation Comparison

An FPGA BFIR could take a similar form to the MATRIX implementation. 1024 LUTs would compose the shift register. $\left\lceil \frac{1024}{3} \right\rceil \cdot 2 = 684$ 4-LUTs compose the match and initial reduce. The sum tree requires slightly over 1000 full adder bits – 1000 XC4K CLBs or 2000 4-LUTs. In total, an XC4K implementation would require 1850+ CLBs, or $2.3\text{G}\lambda^2$. Using the fast carry on the XC4K, and pipelining the adder stages, the basic cycle could be as low as 10ns assuming an optimal physical layout.

Table 13.5 compares the BFIR implementations. The MATRIX implementation is 19$\times$ larger than the custom implementation, 4.8$\times$ larger than the Xilinx implementation, and 24$\times$ smaller than the MIPS-X implementation. If the "care" region is sparse, the FPGA implementation can easily take advantage of it, using less match and sum reduce units (*e.g.* [VSCZ96]). If the sparsity is in 8-bit chunks, MATRIX can similar exploit the sparseness. The processor implementation can exploit sparseness, as well, but requires even larger chunks for it to be beneficial. Resource requirements for all the programmable implementations are proportional to the template size, so their areas decrease with the number of binary taps.

Architecture	Reference	area and time	$\frac{\text{Filter TAPs}}{\lambda^2 s}$
16b DSP	[WDW+85]	100ns/TAP	0.65
	[vMWvW+86]	125ns/TAP	0.090
	[KNK+87]	50ns/TAP	0.057
	[CBBF87]	60ns/TAP	0.082
	[PML+89]	100ns/TAP	0.051
	[SKYH92]	50ns/TAP	0.072
	[USO+93]	47ns/TAP	0.041
32b RISC MSTEP	MIPS-X [HHC+87]	10+ cycles/TAP	0.029
32b RISC/DSP	[NHK95]	40ns/TAP	0.022
64b RISC	Alpha [GBB+96]	2.3ns/TAP	0.064
	(systolic)	3 BFUs, 20ns/TAP	0.56
MATRIX	Section 13.3 (VLIW)	12 BFUs, 20ns/TAP	0.14
	(microcoded)	8 BFUs, 90ns/TAP	0.048
Full Custom	LSI [Rue89]	45ns/64 TAPs	3.6

Table 13.6: FIR Survey – 8×8 multiply, 24-bit Accumulate

13.7.4 MFIR

The LSI multibit finite-impulse response filter is a 64-tap, 8-bit FIR filter:

$$y(n) = \sum_{i=0}^{63} h_i \cdot x(n-i)$$

The MFIR is implements in 225mm^2 in a 1.5μ CMOS process (400Mλ^2) using a full custom design methodology. The LSI MFIR runs at a 22 MHz clock rate (45ns clock).

In Section 13.3, we have already seen several MATRIX FIR implementations. To handle the same generality as the LSI MFIR, we need to handle a 24-bit accumulate instead of the 16-bit accumulate used in the examples shown in Section 13.3. This adds one cycle per tap to the microcoded implementation, one BFU to the VLIW implementation, and one BFU per tap to the systolic implementation. Table 13.6 compares the LSI and MATRIX implementations along with processor and DSP implementations. For the table, we use an application-specific metric and report the area-time capacity required per TAP in each of the implementations.

The systolic MATRIX implementation is 6× larger than the full-custom LSI implementation, 20× smaller than the MIPS-X processor implementation, and 9× smaller than the Alpha implementation. Note also that the VLIW MATRIX implementation, which resembles modern DSP architectures, is 2× smaller than modern DSPs. The systolic version is 8× smaller than the DSPs. The capacity requirements for the processors, DSPs, and MATRIX will decrease with the number of taps, while the LSI implementation is fixed. At 10 filter taps, the systolic MATRIX implementation uses less capacity than the LSI MFIR.

Table 13.7 provides an expanded table for FIRs with 16-bit accumulates. Here, we see more clearly that the systolic MATRIX implementation is on par with reconfigurable implementations such as PADDI and FPGAs. The MATRIX VLIW is comparable to DSPs. The MATRIX microcoded yields performance comparable to microprocessor implementations. It is this versatility to efficiently span such a wide range of raw performance requirements which makes MATRIX an interesting and powerful general-purpose architecture.

13.7.5 Image Processing Summary

Across the four tasks, we see that the MATRIX implementation is roughly an order of magnitude larger than the custom implementation ($6\times$, $19\times$, $26\times$, and $2.4\times$). Since it remains general-purpose, MATRIX retains the ability to deploy resources to adapt to the problem size. For many instances of problems the area-time penalty will be much less.

At the same time, we saw that MATRIX provided an order of magnitude smaller implementations than conventional processors ($16\times$,$10\times$,$24\times$,$20\times$). The variation in the benefits is somewhat telling. The one task where MATRIX only had a $10\times$ advantage is the one task which required a 16-bit datapath, while all the others essentially used 8-bit datapaths. Combining that observations with our earlier observation that MATRIX has $3\times$ the raw computational density of modern processors, we can decompose MATRIX's capacity advantage over processors as: roughly as:

- $3\times$ raw computational capacity
- $4\times$ versus 8-bit, $2\times$ versus 16-bit – granularity (datapath deployability)
- 1.5-$2\times$ elimination of overhead operations

For the highest throughput implementations of these tasks, aggressive FPGA or DPGA implementations may approach the MATRIX implementation. We saw cases where MATRIX was 2-$10\times$ smaller than optimistic FPGA implementations. We also saw naturally bit-level tasks where MATRIX might be 4-$5\times$ worse than an FPGA implementation.

Architecture	**Reference**	**area and time**	$\frac{\text{Filter TAPs}}{\lambda^2 s}$
16b DSP	[WDW+85]	100ns/TAP	0.65
	[vMWvW+86]	125ns/TAP	0.090
	[KNK+87]	50ns/TAP	0.057
	[CBBF87]	60ns/TAP	0.082
	[PML+89]	100ns/TAP	0.051
	[SKYH92]	50ns/TAP	0.072
	[USO+93]	47ns/TAP	0.041
32b RISC MSTEP	MIPS-X [HHC+87]	10+ cycles/TAP	0.029
32b RISC/DSP	[NHK95]	40ns/TAP	0.022
64b RISC	Alpha [GBB+96]	2.3ns/TAP	0.064
XC4K	[GN94]	67 CLBs, 184ns/16-TAPs†	1.0
	[CME93]	400 CLBs, 100ns/4-TAPs	0.080
PADDI2	[YR95]	5 EXUs, 20ns/TAP	0.93
	(systolic)	2 BFUs, 20ns/TAP	0.87
MATRIX	Section 13.3 (VLIW)	11 BFUs, 20ns/TAP	0.16
	(microcoded)	8 BFUs, 80ns/TAP	0.054
Gate Array fixed coefficient	[YJY+90]	10ns/64 TAPs	21
Full Custom	[Rue89]	45ns/64 TAPs‡	3.6
	[CLRA90]	25ns/4 TAPs§	0.68
	[GNC+90]	33ns/16 TAPs	3.5
	[RK92]	50ns/10 TAPs	2.4
fixed coefficient	[LS92]	6.7ns/43 TAPs§	57

† – symmetric filter only
‡ – 24-bit accumulate
§ – 16×16 architecture

Table 13.7: FIR Survey – 8×8 multiply, 16-bit Accumulate

13.8 Summary

All conventional, general-purpose computing architectures set the resources for instruction distribution and control and bind datapaths to instructions at fabrication time. This, in turn, defines the efficiency of the architecture at handling tasks with a given wordsize, throughput, and control structure. Large applications typically work with data items of multiple sizes and subtasks with varying amounts of regularity. Application sets have an even wider range of computational task characteristics. Consequently, no single, fixed, general-purpose architectural point can provide robust performance across the wide range of application requirements.

To efficiently handle the wide range of application characteristics seen in general-purpose computing, we developed MATRIX, a novel general-purpose architecture which uses multilevel configuration and a single pool of network and datapath elements to defer until application run time:

1. binding of primitive elements to roles such as data memories, instruction stores, datapath elements, or control units
2. binding of datapaths to instructions
3. interconnection of primitive elements

Using metaconfiguration, MATRIX can deploy primitive resources and interconnect to various roles as best suits the application. In this manner, MATRIX can provide as much dynamic instruction control, instruction sharing, static dataflow, or independent control flow as required by the task. MATRIX's *post-fabrication* configurability of instruction organization is unique, differentiating it from all previous general-purpose computing architectures.

An ongoing prototyping effort shows promising results. While the VLSI implementation has considerable room for improvement, the prototype has $3\times$ the raw computational density of conventional processors and achieves $10\times$ the yielded computational density on regular, byte-level computing tasks. At the same time, the prototype holds its own on less regular tasks, achieving performance comparable to conventional processors.

13.9 Area for Improvement

The concrete microarchitecture presented here has been our initial vehicle for studying the basic concepts behind MATRIX and providing a concrete grounding for them. In these respects the concrete microarchitecture has been very successful. However, this microarchitecture fails to achieve the full breadth of performance robustness promised by the MATRIX architectural style.

Figure 13.22 shows the efficiency of the MATRIX microarchitecture at handling tasks with various instruction depths and datapaths widths. Shown alongside MATRIX is the efficiency for a conventional architecture with fixed instruction distribution. These graphs are similar to the one shown in Section 9.5. The efficiency is the ratio between the size of the implementation in the target architecture versus the size of the conventional architecture with the instruction depth and datapath width perfectly matched to the task. We assume here that MATRIX must deploy eight BFU instruction stores per independent datapath for control. That is, we assume all eight MATRIX ports must be fed with dynamic instructions.

It is not surprising that MATRIX does not have the peak performance of the fixed architecture at its optimal design point. However, the poor efficiency across such a broad range of space is disappointing. We can identify several effects from the graph:

- The performance cliff between the path length of two and four arises since we can handle two contexts with the control contexts, but four or more require that we deploy BFU instruction stores. For datapaths of eight bits or less in width, we go from one BFU per datapath element to nine as the task goes from two instructions to four.

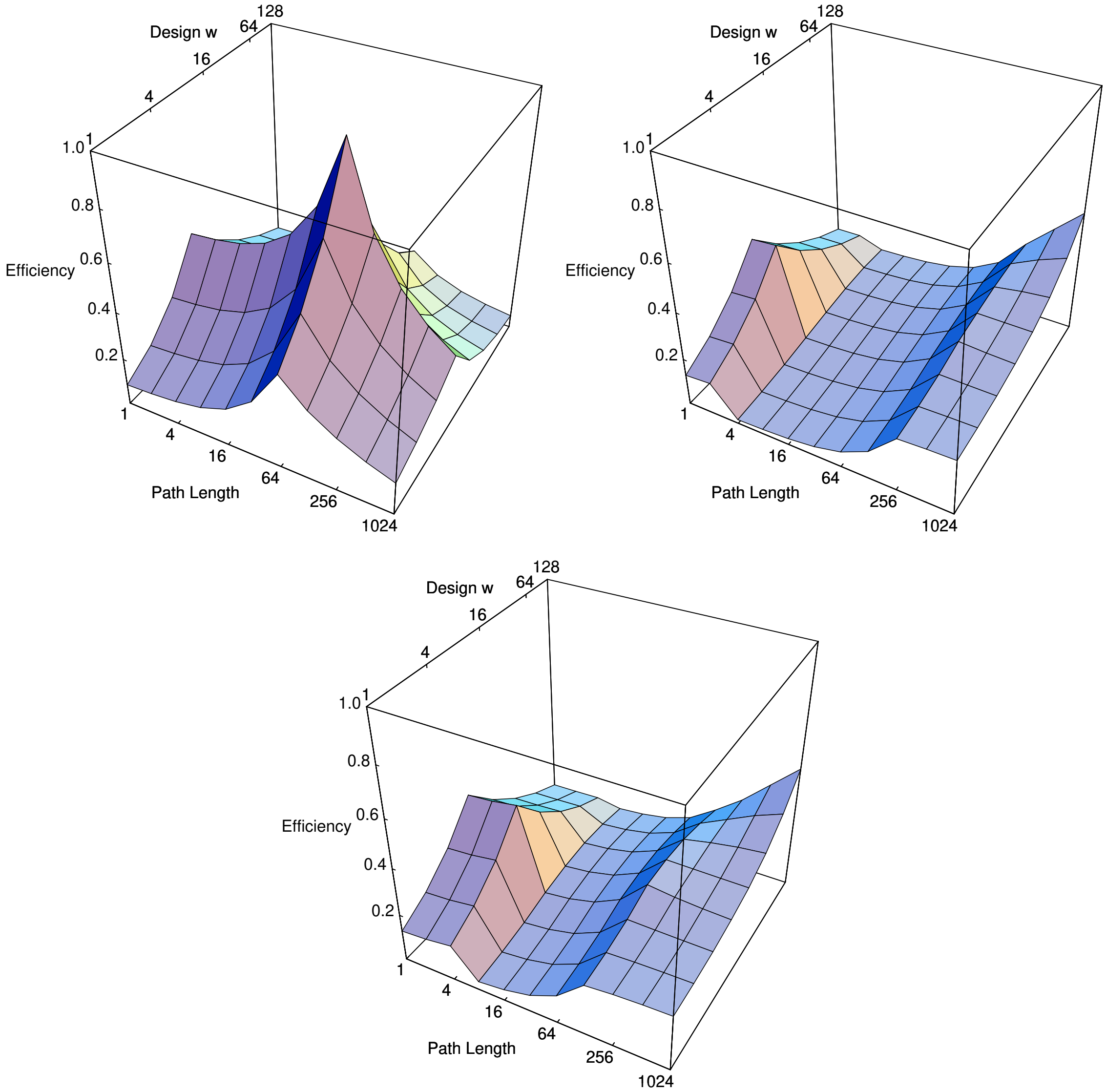
- At large path lengths ($c > 256$), and small datapaths, we asymptotically approach 25% efficiency. We noted earlier (Section 9.2) that the instruction memory dominates the interconnect and compute area in this region. We also noted that the MATRIX memory makes up 25% of the BFU area (Section 13.13), so we are seeing the implementation efficiency being dictated by the instruction memory efficiency.

- Unlike traditional architectures MATRIX implementations continue to become more efficient with larger task datapath width. As we saw in Section 13.4, MATRIX does not need to deploy additional instruction memories to handle larger datapaths. As the datapath grows, the instruction memory overhead is amortized over a greater number of elements, improving the overall implementation efficiency.

One thing which may be unfair in the comparison in Figure 13.22 is the interconnect Rent parameter, p. The MATRIX microarchitecture under discussion has fully populated input switches. Also note that this comparison is strictly based on varying instruction depth and datapath width and does not account for variations in control requirements. The fixed architecture will suffer more than MATRIX as the number of natural task control streams varies.

Also shown in Figure 13.22 is a MATRIX$'$ architecture which lessens the BFU overhead penalty for cases with a path length between 2 and 256. MATRIX$'$ assumes that it can use each BFU memory as two 128×8 instruction stores, bringing both memory read ports out to routed lines and allowing path lengths less than $c = 256$ to use only four BFUs per datapath. MATRIX$'$ also assumes the addition of two more control contexts.

These graphs suggest:



Top Left – $k = 2$, $n = 2$, $p = 0.70$, $w = 8$, $c = d = 64$, $N_{ctrl} = 0$, $N_p = 2048$

Top Right – MATRIX model

Bottom – MATRIX′ – 4 control contexts, use BFU I-store memory as $2 \times 128 \times 8$

Figure 13.22: Efficiency for MATRIX and Fixed 8-bit Architecture ($p = 0.70$)

- The microarchitecture may be too coarse-grained in its deployment of resources. Context memory deployment suffers particularly from the large chunk size for memory deployment.
- Metaconfiguration and interconnect overheads are particularly large compared to the memory size.

Part V

Review and Extrapolation

Special-Purpose Computing We build computing devices to algorithmically transform raw input data into results. Special-purpose computing devices are designed with one particular transformation embedded into their architecture and implementation. Each such device can solve only the particular transformation problem, and that problem is set prior to device fabrication. Conventional fabrication techniques require long turn-around (weeks to months) to produce devices, high up front costs for setup, and large volume sales to amortize out fixed costs for design, tooling, and equipment.

Many of the characteristics which come with special-purpose computing devices are undesirable or untenable in numerous situations.

- Device dedicated to a single function
  - Device can be quickly oboslesced as functional requirements often change, transformations are tuned, algorithms advance, and missions and tasks evolve.
  - When the function needed by a task is time or data dependent the special-purpose devices for functions which are not needed at some point in time sit idle and cannot be used for any other function which may be required by the task.
  - When lower throughput is required from the device than its native capability, the device has spare capacity which cannot be put to productive use.
- High up front cost
- Long delay from concept to delivery
- Economical only in volume

General-Purpose Computing General-purpose devices are our alternative to these fixed function devices. Here, we build computing devices which can be configured to solve a variety of computing problems. Instead of building a device with exactly the computational units and hardwired dataflow necessary to solve a single problem, we build a device with a set of primitive computational elements interconnected via a flexible interconnect. Post-fabrication, we control the behavior of the device with instructions, extra inputs which tell the device what computations to perform and how to route data during the computation. As a result, we:

- Make a single device appealing for a wide-range of tasks. While each, individual task may lack the volume required for a dedicated device to be economical, the general applicability across many tasks provides the volume necessary to make the general-purpose device economical.
- Eliminate the fabrication delay necessary to put a new computational task into use.

- Eliminate the up front cost associated with producing custom hardware for a task.
- Make it possible to customize a single device to perform any of a large number of different computing task, allowing the device to adapt to changes in requirements, or share its capacity among a variety of computing tasks.

The *RP*-space defined here models a large domain of reconfigurable architectures within the general-purpose architecture space.

Reconfigurable Computing Costs Reconfigurable devices gain their breadth of use at a cost in computational density. Reconfigurable devices must add:

1. Flexible interconnect or data flow
2. Instructions to control compute units and data flow

Additionally, the computational units in these devices must be more general than in the special-purpose devices where each compute unit may perform a single, focussed computation.

Replacing fixed interconnect with flexible interconnect is the most costly single addition for reconfigurable architectures. A decent amount of programmable interconnect may add two orders of magnitude in size to the reconfigurable implementation compared to the fully special-purpose implementation of the same task.

Instructions In contrast, the area required to hold a single, device-wide configuration is, itself, an order of magnitude smaller than the interconnect. That is, the area taken by a single instruction is generally an order of magnitude smaller than the active interconnect which it controls. However, if we allocate space to hold tens of instructions per active compute element, the total instruction memory area can easily equal the active compute and interconnect area. By the time we add hundreds of instructions, the instruction memory area can dominate even the flexible interconnect. With this additional order of magnitude in overhead, such a reconfigurable device can easily be three orders of magnitude larger per computational element than its special-purpose counterpart.

Since instruction area can quickly come to dominate even the flexible interconnect, when building reconfigurable computing architectures we often look for structure in typical computational problems which will allow us to reduce the instruction size. One common technique is to control several pieces of interconnect and computational elements with a single instruction. That is, we assemble wide datapaths which are controlled together. This reduces the size of the configuration by reducing the number of instructions required to specify device behavior at any point in time.

Consequently, when we build a reconfigurable computing device, we must make decisions about:

- How many primitive computational elements are directed by each instructions?
- How many instructions are controlled by each controller?
- How many instructions are stored on chip?
- How rapidly can the instructions change, chip-wide?

The answers to these questions place a particular reconfigurable device in the *RP*-space. The answers to each of these questions also determines the size of the reconfigurable device and its efficiency on various tasks.

- If the task has data elements of width of $w_{task} > w_{arch}$, the architecture provides finer instruction control than necessary and pays an overhead for redundant instruction memory.
- If the task has data elements of width of $w_{task} < w_{arch}$, the architecture does not allow control over the compute element at the fine granularity of the task, and computational capacity in the architecture goes to waste.
- If the task needs to cycle through only a few different instructions, but the architecture provides large instruction memories, the reconfigurable device is unnecessarily large for the task, wasting area in unused memories.
- If the task needs to cycle through a large number of different instructions at different times but the architecture provides small instruction memories, the reconfigurable device will not be able to store all the instructions logically associated with each computational element. Extra computational elements will be required simply to hold all of the task's instructions, but these extra computational elements will effectively sit idle during computation.
- If the task requires more independent control of computing resources than provided by the architecture, either resources will go unused since they cannot be controlled or memory requirements will increase greatly to compensate for the lack of control independence.
- If the task requires less independent control than the architecture supplies, the additional controllers and resources are redundant and add to device overhead.
- If the task requires rapidly changing instructions, but the architecture does not meet the required bandwidth, computational resources sit idle, paced by task description bandwidth not the availability of computing resources.
- If the task can handle slowly changing instructions, but the architecture dedicates significant area to providing high instruction delivery bandwidth, much of the dedicated area is overhead making the device larger than necessary for the task.

Interconnect In devices where the ratio between instructions and compute elements is low, flexible interconnect will remain the dominant area feature in reconfigurable devices. Here, a device must decide how richly to interconnect the compute elements. Rich interconnect makes the routing area even greater, while inadequate interconnect can make it impossible to make use of the available computing elements. The choice in interconnect richness determines where the architecture will be most efficient.

- If the interconnect is richer than needed by the task, the device will be larger than necessary.
- If the interconnect is not as rich as required by the task, the task must be laid out sparsely on the architecture. Portions of the interconnect and compute resources are wasted as they cannot be used.

In all computing devices there are two components associated with routing data between producers and consumers:

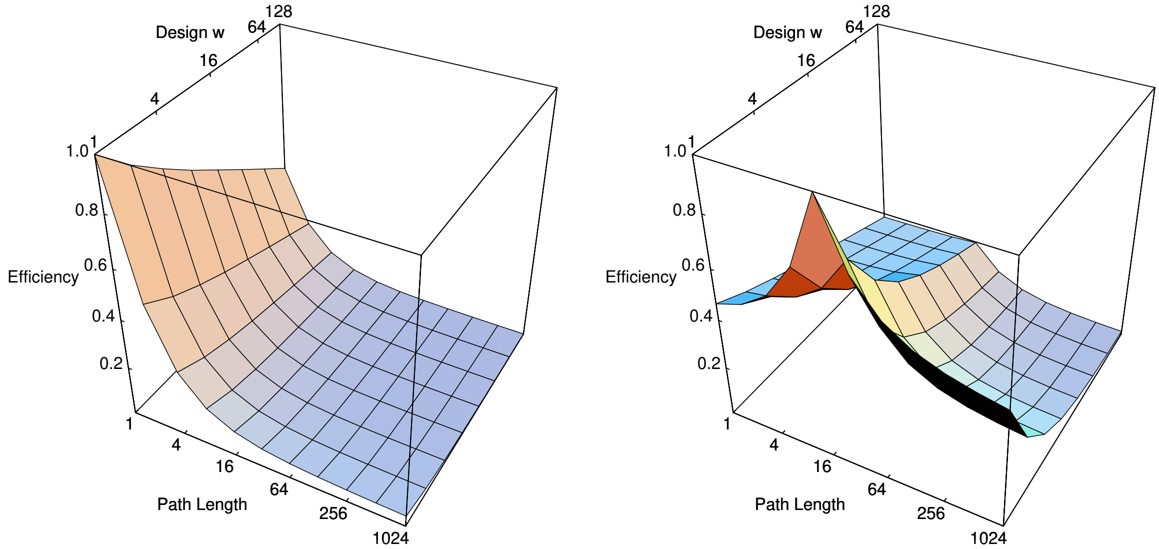
1. Spatially routing intermediates from the compute element which produced them to those which consume them
2. Retiming the intermediates for the time when the consumer is ready to use them

Particularly, in reconfigurable devices with expensive, flexible interconnect, memories can hold values for retiming more cheaply than active interconnect.

Degrees of Generality and Reconfigurability There are, of course, degrees of "generality" between fully special-purpose devices and general-purpose devices. Some special-purpose devices are given limited configurability to broaden there use – *e.g.* a typical UART can be configured to handle different data sizes, data rates, and parities. Some devices are targeted at being "general" within very specific domains. Digital signal processors are one of our most familiar examples of a general-purpose, domain-optimized device. The domain may dictate the typical data element size or desirable instruction depth. Further, the domain may allow a more structured programmable interconnect to suffice. Nonetheless, to the extent that we have post-fabrication control over the computations which a device performs, the device will have some form of instructions and will generally have some level of flexible interconnect. With these features it exhibits reconfigurable characteristics, and many of the the architectural characteristics, relations, and issues we have identified in our, more ideal, *RP*-space.

FPGAs Conventional FPGAs fall at a moderately extreme point in our *RP*-space with single bit wide datapaths and single instruction deep instruction memories. At this point, they are efficient on the highest throughput, fine-grained computing tasks and their efficiency drops rapidly as the task throughput requirements diminishes and the word size increases.

Beyond FPGAs in the Reconfigurable Computing Space Beyond FPGAs there is a rich reconfigurable architecture space. Our DPGA represents one different point in this architectural space (See Figure 14.1). The DPGA retains the bit-level granularity of FPGAs, but instead of holding a single instruction per active array element, the DPGA stores several instructions per array element. The memory necessary to hold each instruction, is small compared to the area comprising the array element and interconnect which the instruction controls. Consequently, adding a small number of on-chip instructions does not substantially increase die size or decrease computational density. The addition does, however, substantially increase the device's ability to efficiently handle lower throughput, more irregular computational tasks. At the same time, a large number of on-chip instructions is not as clearly beneficial. While the instructions are small, their size is not trivial – supporting a large number of instructions per array element (*e.g.* tens to hundreds) would cause a substantial increase in die area decreasing the device efficiency on regular tasks. Consequently, we see that we can achieve a design point which is moderately robust across a wide range of throughput variations by balancing the instruction memory area with the fixed area for interconnect and computational units.



FPGA $c = d = 1$ | DPGA $c = d = 16$

$w = 1$, $k = 4$, $n = 2$, $p = 0.5$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 14.1: FPGA and DPGA efficiency in *RP*-space

The importance of efficiently supporting retiming of intermediates was most clearly demonstrated in the context of the DPGA design. Here, we saw that the benefits of deeper instruction memories were substantially reduced if we forced retiming to occur on active interconnect. However, when we provided architectural registers so that retiming could take place in registers, DPGAs were able to realize typical computing tasks in one-third the area required by conventional FPGAs.

While we did not detail them in this thesis, multiple context components with moderate datapaths also come down essentially in this reconfigurable architectural space. Pilkington's VDSP [Cla95] has an 8-bit datapath and space for four instruction per datapath element. UC Berkeley's PADDI [CR92] and PADDI-II [YR95] have a 16-bit datapath and eight instruction per datapath element. All of these architectures were originally developed for signal processing applications and can handle semi-regular tasks on small datapaths very efficiently. Here, too, the instructions are small compared to the active datapath computing elements so including 4-8 instructions per datapath substantially increases device efficiency on irregular applications and robustness to throughput variations with minimal impact on die area.

Flexible Deployment of Instruction Resources While architectures such as these are often superior to the conventional extremes of FPGAs, any architecture with a fixed datapath width, on-chip instruction depth, and instruction distribution area will always be less efficient than the architecture whose datapath width, local instruction depth, and instruction distribution bandwidth exactly matches the needs of a particular application. Unfortunately, since the space of allocations is large and the requirements change from application to application, it will never make sense to produce every such architecture and, even if we did, a single system would have to choose one of them. Flexible, *post fabrication*, assembly of datapaths and assignment of routing channels and memories to instruction distribution enables a single component to deploy its resources efficiently, allowing the device to realize the architecture best suited for each application. Our MATRIX design represents the first architecture to provide this kind of flexible instruction distribution and deployable resources. Using an array of 8-bit ALU and register-file building blocks interconnected via a byte-wide network, our focus MATRIX design point has $3\times$ the raw computational density of processors and can yield $10\times$ the computational density of conventional processors on high throughput tasks.

15. Projections

In Parts III and IV, and Chapter 14, we focussed on reconfigurable, general-purpose computing devices roughly characterized by *RP*-space. In that focussed domain, we were able to look closely at area costs, computational density, and efficiencies. General-purpose devices, more broadly, also share many of the characteristics (*e.g.* instruction depth and width, interconnect richness, data retiming) which we identified as key architectural parameters in *RP*-space and in the more detailed architectural studies. In this chapter, we speculate more broadly on what the relationships developed while focusing on reconfigurable devices in *RP*-space might tell us about general-purpose architectures, in general. We emphasize that these extrapolations may overly trivialize important architectural aspects which did not arise in *RP*-space, and we attempt to identify those aspects during the discussion.

15.1 Role of Memory in Computational Devices

In our computing architectures, we have seen memory show up in two roles:

1. instruction storage
2. data retiming

Neither appears to be really fundamental for computing, but both are of pragmatic value as they facilitate resource sharing and reuse which allows us to implement computing functions in less area when throughput requirements are limited. In special-purpose computing architectures we did not need instructions. For ease of construction, we often use clocked registers to tolerate variable delays through primitive blocks, but otherwise memory for retiming arises primarily from serialization and reuse of common resources.

15.1.1 Memory for Instructions

Instruction memories reduce hardware requirements in two ways by allowing:

1. a fabricated resources to perform any of several functions
2. a resource to be shared among several different functions during a single computation

Select Function In our general-purpose devices, a single resource can perform any of a number of different functions. This allows us not to have a single, dedicated piece of hardware for every possible function ever desired. For an application or device requiring n primitive component computations, this realizes an important compression from "all possible computing functions made of n primitives" to "all n primitive computing functions required by this application." Here, each primitive computing element needs a configuration memory to tell it what computation to perform and where its inputs are produced among the computing elements. The per computing element overhead we pay for this reduction is high, mostly in terms of flexible interconnect, but this quickly balances the exponential reduction realized by only having to implement the n functions required by this task.

We can return to our pedagogical 4-LUTs to see this reduction more concretely. There are $O\left(\left(2^{(2^4)}\right)^n\right)$ different functions which can be implemented with n, four-input gates. So, even with the $100\times$ area overhead per gate required to support flexible interconnect, our programmable, n 4-LUT device is significantly smaller than implementing all possible n input functions for anything other than trivial values of n.

Shared Function Our general-purpose devices also allow us to share each piece of hardware among multiple functions within a single computing task. This aspect allows us to compress area requirements further from "all n primitive computing functions required by this application" to "all m computing functions required at one point in time in order for this application to achieve the requisite computational throughput." Here, we take advantage of the fact that the configuration memory to describe a computing function is smaller than the active area required to route its inputs and compute the result. In the extreme, this allows us to reduce the area required for a computation from the area required for n programmable compute primitives and their associated interconnect to the area required to store the description of the computation and interconnect performed by n programmable compute elements.

What we trade for this reduction is computational throughput. With only m active computing functions, it requires us, at least, $\frac{n}{m}$ cycles to perform the computation of the n primitive computing functions in the original task. Sometimes, the original task already had a dependency structure such that this reduction comes for free or at minor costs. Other times, we are trading increased evaluation time for reduced implementation area. In the limit, where we have a single computing element with instruction memory to hold n instructions, the task can take n cycles to evaluate.

We often talk about *virtualizing* hardware resources. The virtualization really substitutes a less expensive resources (*e.g.* an instruction in memory, state in memory, cheaper forms of memory) for a more expensive one (*e.g.* a piece of hardware to actually perform a function, fast access memory). Behind all of these virtualizations, we must ultimately have some form of physical memory to hold the description of the virtualized resources and their state.

Notice that we can continue to push technology and structure in order to reduce this last limit, but it cannot be avoided. We can apply aggressive memory technology, such as DRAM or flash memory, to reducing storage cell size. We can store data on different media, such as magnetic disks or tape. We can exploit structure in the task description to compress the number of bits required down to the Kolmogorov complexity limit. *In the limit, however, we ultimately require sufficient area to store the description of the computing task and no further reduction is possible.*

We noted in Section 4.4 that memory can be used as a general-purpose computing element. That role of memory is a special case of role of memory as instructions. The memory contents act as an instruction which configures the memory array to provide the desired computational transformation between the address inputs and the data outputs. In Section 4.5, we saw that computational portion of conventional FPGAs, the LUTs, were programmed in exactly this way.

15.1.2 Memory for Retiming of Intermediate Data

Once we begin to reuse primitive compute functions for different roles at different times, we introduce the need to assure that the right data arrives at the inputs of the function at the right time. This need is particularly acute when we serialize execution and use a single primitive to perform multiple different functions, but it also appears when we reuse a primitive to perform exactly the same function on logically different data. Since programmable interconnect is expensive, we use memories as an inexpensive way to provide the temporal retiming necessary for correct execution.

The use of memory for retiming is pragmatic. We could get away with little more than pipeline registers on interconnect. However, it is cheaper to transport data forward in time through memory than over interconnect. If we do not take advantage of this, much of the area savings potentially associated with serializing execution and sharing primitive compute elements cannot be realized.

The requirements for data retiming depend on the interconnect structure of the problem, not the number of compute elements in the task. The amount of retiming does depend on the amount of serialization. With more parallelism, more data can be consumed as soon as it has been spatially routed avoiding the need for retiming. As we compress size requirements by converting task compute primitives into instructions sharing a small number of physical compute elements, we must ultimately have space to store all computation intermediates at the widest point in the computation flow. That is, we ultimately need space for all the live intermediates in a computation. The number of such intermediates depends on the task and its mapping. The mapping should try to minimize the number of such intermediates.

Note that all non-instruction uses of memory fall into this category.

- Register File – We have already seen that register files perform the same functions as our input retiming registers, transporting results in time from the point of production to the point of consumption.

- Main memory, including data caches – All the data results stored in memories are being transported between the point of production and consumption.

- Buffers an FIFOs – These are explicitly retiming the arrival of data to a time when a portion of the system is ready to consume the data.

If we had not sequentialized execution and shared computational resources among multiple tasks, we would not need these memories.

Even special-purpose devices often sequentialize their processing of data so that a few, fixed compute elements can serve to process data with nominally different roles. The most common example of this is in audio, video, or image processing. Rather than dedicating a separate computational unit to each pixel in a frame, many pixels are processed on the same computational unit. The pixel data stream is serialized into and out of the special-purpose device. The pixels within the frame often need to be retimed so that the right pixel values are presented to the compute elements at the right time. For example, when pixels are fed in by rows, it is often necessary to perform row-wise retiming on data so that the compute element can calculate column-wise interactions between pixel elements. If all the data necessary for the computation were presented simultaneously and all of the output was produced at once, this retiming would not be necessary. However, serialization and reuse is often necessary to make the amount of hardware resources, including component input and output bandwidth, tractable. The serialization allows us to share all of the hardware resources, but requires that we provide unique storage space for intermediate data so that we perform the correct computation on the shared resources.

15.1.3 Implications

There are two important ideas to take away from these observations on the role of memory:

1. Memories in computer architectures facilitate the sharing and reuse of expensive resources. It is the pragmatic fact that the memory necessary to hold an intermediate or an instruction is smaller in conventional technologies than the active computing and interconnect elements which process the data according to the instruction which makes it worthwhile to use memory to reduce implementation area requirements.

2. As we go to heavier sharing, each doubling of our sharing factor does not result in a halving of implementation area because we always leave behind a memory residue composed of (1) instructions and (2) intermediate data. In the limit, the size of our computing element for a task is dictated by the area to hold the instructions to describe the task and the intermediate, live data which must be stored as the task computes.

15.2 Reconfiguration: A Technique for the Computer Architect

Device architects are often faced with the dilemma of balancing semantic expressiveness with instruction distribution bandwidth. In processors, only a few bits are allocated to instruction specification limiting (1) the number of different computations which can be selected and (2) the number of different sources which can be expressed. The latter manifests itself as limited address space and limited size register files, while the earlier is often taken for granted. Architects are reluctant to increase instruction width because it entails added costs in (1) on- and off-chip storage space for all instructions, (2) distribution bandwidth, and (3) power for instruction distribution. However, limited semantic expressiveness can force the processor to issue a large number of instructions to perform the desired computation, resulting in even great losses in time and power efficiency.

Conventional processors generally support an ALU which performs basic operations on 2 or 3, word-wide data inputs. Today we see typical word sizes of 32 and 64 bits. Conventional processors further limit their instruction size to the word size to limit instruction bandwidth requirements. As a consequence of this limitation, it can often take a large number of instructions to specify an operation which is not inherently difficult for the active silicon to perform.

To appreciate the magnitude of the semantic disparity here, we notice that there are:

$$N_{alu2ops} = 2^{\left(w2^{(2w)}\right)}$$

functions from two w-bit wide inputs to one w-bit wide output. If we limit the specification of our function to w bits, we can only address 2^w functions with this instruction. Thus, if all of the $N_{alu2ops}$ were equally likely, on average, it would take at least $2^{(2w)}$ cycles to compute a function.

In practice, a good fraction of the w bits are dedicated to operand selection, increasing the severity of the instruction limitation. While all operations are not equally likely, in practice, this disparity demonstrates that conventional processor design makes an *early binding*, pre-fabrication time, decision on the effective cost of basic operations. *Many applications cannot use the active silicon area on conventional processors efficiently since they cannot directly issue the instructions native to the task.*

Reconfiguration is a technique which allows us to find a resolution to this dilemma. Reconfiguration allows us the semantic expressiveness of very large instructions without paying commensurate bandwidth and deep storage costs for these powerful instructions. What we give up in this solution is the ability to change the entire instruction on every cycle. Rather, the rate of change of the full instruction is determined by the instruction bandwidth we are willing to expend. The distinction between instruction bandwidth which delivers all the semantic content on every cycle and instruction bandwidth that can be used to load a larger semantic instruction is an important one because configured instruction bits which can be used for many operational cycles do not require additional instruction bandwidth once loaded. Returning to our simple calculation above, it may take us $2^{(2w)}$ cycles to load a specification for an instruction the first time it is encountered. However, if this value is loaded into configuration memory, subsequent uses can operate using the loaded data, avoiding the time required to redundantly specify the operation. An architecture without configuration would require the $2^{(2w)}$ cycles each time the computation is required. Reconfiguration thus allows us to compress instruction distribution requirements in cases where the instruction changes slowly or infrequently.

Reconfiguration opens a middle ground, or an intermediate binding time, between ‘behavior which is hardwired at fabrication time’ and ‘behavior which is specified on a cycle by cycle basis.’ This middle ground is useful to consider in the design of any kind of of computing device not just conventional FPGAs. When designing a device with any general-purpose capabilities, the architect’s decision can extend beyond what expressiveness to include or omit based solely instruction size and bandwidth. Rather, the architect should consider the expressiveness which may be required for efficient task implementations and the rates at which various parts of the task description change. Characteristics of the task which change infrequently can be configured rather than broadcast.

15.3 Projecting General-Purpose Computing onto *RP*-space

Our *RP*-space model articulated in Chapter 9 provided architecture implementation area estimates based on a few major parameters. Instruction depth (c), data width (w), interconnect richness (p), and intermediate data retiming support (d) have been the focus of our discussion in Parts III and IV. More broadly, these parameters have rough analogs in all general-purpose architectures. One can, thus, generally project a general-purpose architecture into a point in *RP*-space by identifying these parameters and abstracting away architecture characteristics not covered in the *RP*-space model.

15.3.1 General Hazards

The more general projection to *RP*-space may be hazardous as it ignores many detailed characteristics of real architectures in the broader general-purpose architecture space, such as:

- No special-purpose capacity – we explicitly assumed only general-purpose building blocks for *RP*-space. Most nominally "general-purpose" architectures include blocks of special-purpose logic. The special-purpose blocks do not provide general-purpose capacity, but can provide high density to applications when the specialized structures match the task requirements of the application. The multiply example reviewed in Chapter 5 is the most common instance of a special-purpose block added to general-purpose architectures.

- Homogeneous processing arrays – we explicitly assumed homogeneous arrays. Because of the mixed processing requirements in most computing tasks, a hybrid array which mixes processing blocks with different parameters may be quite interesting. Extending the model to reasonably encompass mixed architectures is an interesting direction for future work.

- No boundary effects – we assumed single chip implementations for all of our comparisons, in effect, assuming that full task implementations for all alternatives fit onto a single die. Since we are looking at 10's to 100's of $G\lambda^2$ of silicon area in the near future, a large class of computing tasks or primary subtasks can be reasonably placed on a single die such that the assumption seems reasonable. However, we also saw that inefficient design points can easily be two orders of magnitude larger than efficient points. With this much area variation, it is not really reasonable to assume that both implementations are single die implementations. The larger implementation is likely to require a multiple-chip solution and will suffer further degradation in latency and bandwidth due to chip crossings.

  Another consequence of ignoring boundary effects is that the model trivializes limited device i/o effects between different components that might make up the core of a general-purpose processing system. Notably, systems have traditionally placed bulk memory on different ICs from the processing. As a result, care must be taken to prevent the limited boundary i/o between compute and memory devices from being the performance limiting bottleneck. This care often shows up as additional mechanism and memories on the processing chips to make most effective use of the limited interchip i/o bandwidth and high interchip i/o latency.

15.3.2 Processors, FPGAs, and *RP*-space

For years, microprocessors have been our canonical example of single-chip, general-purpose computing devices. It is tempting to try to understand the relation between processors, FPGAs, and *RP*-space. In Part II, we took a broad, empirical look at these devices and made a few, high-level observations on their relative efficiencies. In this section, we revisit this comparison projecting both architectures into *RP*-space.

Conventional processors have:

1. moderately wide datapath which have been growing larger over time (*e.g.* 16, 32, 64 bits)
2. support for large on-chip instruction caches which have also been growing larger over time and can now hold hundreds to thousands of instructions (contexts)
3. high bandwidth instruction distribution so that one or several instructions may be issued per cycle at the cost of dedicating considerable die area for instruction distribution
4. a single thread of computation control

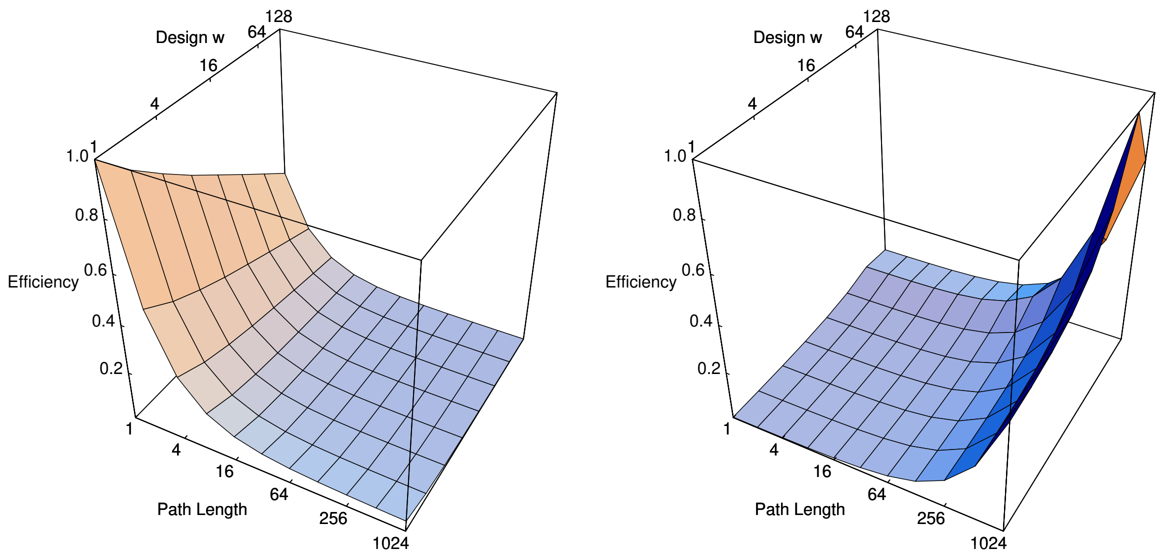
As a consequence these devices are efficient on wide word data and irregular tasks – *i.e.*, those tasks which need to perform a large number of distinct operations on each datapath processing element. On tasks with narrow data items, the active computing resources are underutilized, wasting computing potential. Processors pay overhead for their deep instruction memories. On very regular computational tasks, the on-chip space to hold a large sequence of instructions goes largely unused. Processors exploit wide datapaths to reduce the cost per instruction, but even so, with instruction stores typical supporting thousands of instructions, instruction and retiming memories dominate, leaving their peak general-purpose computational density three orders of magnitude lower than special-purpose devices and one order of magnitude below FPGAs.

Looking at modern scalar, superscalar, and VLIW, processors, then, we might abstract a modern processor as: $k = 2$, $w = 64$, $c = 1024$. Processors use ALU bit-slices in lieu of lookup tables. Each ALU bit-slice takes in two data inputs and a carry bit. As such, they provide less than a full 2- or 3-LUT's capacity per ALU bit, in general, but can provide an add, subtract, or compare operation per bit which would require a pair of 3-LUTs. Processors also include:

- special-purpose capacity (*e.g.* multipliers, floating-point units)
- complicated flow control (*e.g.* branch prediction, bypassing)
- memory controllers to deal with boundary bottlenecks between compute and bulk memory components (*e.g.* cache-controllers, TLBs)

These items tend to make the area of a processor larger than that predicted by the model in Chapter 9. As we have seen in Table 4.1 and Section 4.1, when performing traditional ALU ops, processors generally provide less bit operations per ALU bit than a small LUT. These effects will tend to make the *RP*-space projection of the processor optimistic in terms of area; that is, the real processor will be larger and provide less computational capacity per unit area. On the other hand, the specialized capacity in processors allow them to handle fixed and floating point arithmetic operations more efficiently than would be predicted by the *RP*-space projection.

We have already seen that conventional FPGAs have:



FPGA
$k = 4$, $c = d = 1$, $w = 1$

RP-space mapped Processor
$k = 2$, $c = d = 1024$, $w = 64$

$n = 2$, $p = 0.5$, $N_{ctrl} = 0$, $N_p = 16384$

Figure 15.1: Comparing efficiency of FPGA and Processor idealizations in *RP*-space

- narrow datapath (*e.g.* almost always one bit)
- on-chip space for only one instruction per compute element – *i.e.* the single instruction which tells the FPGA array cell what function to perform and how to route its inputs and outputs
- minimal die area dedicated to instruction distribution such that it takes hundreds of thousands of compute cycles to change the active set of array instructions

As a consequence these devices are efficient on bit-level data and regular tasks – *i.e.*, those tasks which need to repeatedly perform the same collection of operations on data from cycle to cycle. On tasks with large data elements, these fine-grain devices pay excessive area for interconnect and instruction storage versus a coarser-grain device. On very irregular computational tasks, active computing elements are underutilized – either the array holds all subcomputations required by a task, but only a small subset of the array elements are used at any point in time, or the array holds only the subcomputation needed at each point in time, but must sit idle for long periods of time between computational subtasks while the next subtask's array instructions are being reloaded. The peak computational density for FPGAs is two orders of magnitude lower than special-purpose devices because they pay overhead primarily for the flexible interconnect.

Figure 15.1 shows graphically this idealized comparison projected into *RP*-space in the style used in Section 9.5. As noted before, the FPGA is less than 1% efficient at the cross point of wide task data words and long path lengths. Similarly, the modeled processor is less than 1% efficient processing single bit data items at a path length of one. Certainly, if the processor needs to perform bit operations that do not match its special-purpose support, the inefficiency will be at least this large – and may be greater due to the effects noted above which make the real processor larger than the model.

15.3.3 General-Purpose Computing Space

We have already noted that *RP*-space is large such that we can see two or more orders of magnitude in efficiency loss when the application requirements are mismatched with the architecture structure for fixed instruction architectures. Our comparison in the previous sections underscores that the general-purpose architectural space is even larger making it paramount that one understand the realm of efficiency for each "general-purpose" computing device when selecting a device for an application. They underscore the room for intermediate architectures such as the DPGA, PADDI, or VDSP to cover parts of the space which are not covered well by either conventional extremes of processor or FPGAs. They also underscore the desirability of architectures like MATRIX which allow some run-time reallocation of resources to provide more robust yielded performance across the computational space.

Hybrid Since many tasks have a mix of irregular and regular computing components and a mix of native data sizes, a hybrid architecture which tightly couples arrays of mixed datapath sizes and instruction depths along with flexible control may be able to provided the most robust performance across the entire application. While this thesis focussed on characterizing the implications of each pure architectural point, it should be clear from the development here how a hybrid architecture might be better suited to the mix of datasizes and regularities seen in real applications. In the simplest case, such an architecture might couple an FPGA or DPGA array into a conventional processor, allocating the regular, fine-grained tasks to the array, and the irregular, coarse-grained tasks to the conventional processor. Such coupled architectures are now being studied by several groups (*e.g.* [DeH94] [Raz94] [WC96]).

15.4 Trends and Implications for Conventional Architectures

In summary, we see that conventional, general-purpose device architectures, both microprocessors and FPGAs, live far apart in a rich architectural space. As feature sizes shrink and the available computing die real-estate grows, microprocessors have traditionally gone to wider datapaths and deeper instruction and data caches, while FPGAs have maintained single-bit granularity and a single instruction per array element. This trend has widened the space between the two architectural extremes, and accentuated the realm where each is efficient. A more effective use of the silicon area now becoming available for the construction of general-purpose computing components may lie in the space between these extremes. In this space, we see the emergence of intermediate architectures, architectures with flexible resource allocation, and architectures which mix components from multiple points in the space. Both processors and FPGAs stand to learn from each other's strengths. In processor design, we will learn that not all instructions need to change on every cycle, allowing us to increase the computational work done per cycle without correspondingly increasing on-chip instruction memory area or instruction distribution bandwidth. In reconfigurable device design, we will learn that a single instruction per datapath is limiting and that a few additional instructions are inexpensive, allowing the devices to cope with a wider range of computational tasks efficiently.

15.4.1 Microprocessors

Over the past two decade, microprocessors have steadily increased their word size and their cache size. While these trends allow larger tasks to fit in on-chip caches and allow processors to handle larger word operations in a single cycle, the trends also make processors less and less efficient in their use of die area. While some large word operations are required, a larger and larger fraction of the operations executed by modern processors use only a small portion of the wide datapath. The computationally critical portions of programs occupy only small portions of the instruction and data cache.

We can continue to improve aggregate processor performance by using more silicon in this manner, but the performance per unit area will steadily decrease. To the extent that silicon area is inexpensive, task recompilation is hard or unacceptable, and various forms of parallelism are difficult to achieve, the current trends have their value.

However, to the extent we wish to engineer better silicon system which do more with less resources, these trends are now yielding diminishing returns. We can manage more programmable compute elements than a single, central word-wide, ALU on modern IC dies. Reconfiguration allows us to do this without paying a prohibitive costs for increased instruction distribution as we go to more, independently controlled computing units.

15.4.2 Multiprocessors

The conventional view of multiprocessing is that we replicate the entire microprocessor and place these replicas on the same board or die. At best, this allows aggregate performance to improve with additional area dedicated to additional processors. However, it entails a large amount of unnecessary cost, replicating entire processors when many portions of the processor may not need to be replicated. Further, coupling between processors is poor, at best, entailing 10's to 100's of cycles of latency to move data from one processing element to another and significant overhead to coordinate the activities of multiple processing units.

Most of the task which have generally been "good" multiprocessor applications are very regular computing tasks for which configured, systolic dataflow can provide more area efficient implementations. For the sake of intuition, consider an image processing task where we need to perform 100 operations on each pixel. We can divide this task among n conventional processors, where each processor must have memory to hold the 100 operations and must pay overhead cycles for communication, as necessary, among the n processors. Alternately, we can configure a hardware pipeline to process the data. If we allocate 100 compute elements, each compute element in the configured pipeline needs to only execute its one operation. Direct connections between computing elements transport data avoiding additional overhead cycles. To get the same throughput as the 100 element systolic design, the multiprocessor implementation would need, at least, 100 processors. In terms of instruction memory alone, the multiprocessor implementation requires memory area to hold 9900 more instructions than the systolic implementation, making it significantly larger just to support the same throughput.

The traditional strength of microprocessors has been their ability to pack large computations into small area by reusing central computing resources. This tight packing of functionality comes at the cost of a decrease in computational density as we saw in Chapter 4 and Section 15.3.2. When we are willing to pay area to increase throughput, the traditional microprocessor architecture is not efficient since it brings with it the baggage of a large investment in instruction distribution, instruction memory, and control which are unnecessary for highly regular tasks. Further, the i/o structure of conventional processors is designed around heavy sequentialization, creating an interconnect bottleneck which makes high throughput usage impractical.

16. Review of Major Concepts

After reading this thesis, you should appreciate the following major concepts:

- Our reconfigurable computing space, *RP*-space, is largely characterized by architectural choices surrounding the storage, distribution, binding, and control of instructions. [Chapters 8 and 9]
- These choices about instruction resources, in turn, are largely responsible for defining the circumstances under which a given architecture within the *RP*-space is most efficient. [Chapter 9]
- Using a multilevel configuration scheme, the deployment of chip resources, including those for instructions, can be deferred until run-time. Consequently, resource allocation, instruction distribution, and control can be tailored to the needs of the application, making such a device efficient over a broader range of application characteristics than architectures whose resources are bound at fabrication time. [Chapter 13]
- There are three primary consumers of area on reconfigurable components: (1) instructions, (2) interconnect, and (3) intermediate data.
  - Task descriptions (instructions) are small compared to their physical realizations. [Chapter 7, Chapter 9, and Section 10.4]
  - Nonetheless, instruction storage space is not trivial. A large number of instructions (typically 10-100) often take up as much space as the active interconnect and computational elements required to actually perform the instruction. [Chapter 7, Chapter 9, and Section 10.4]
  - We can compress the area for an implementation by increasing the instruction to active area ratio, but the benefits diminish past the point where the total area for stored instruction and data equal the active area on which they are evaluated. [Chapter 9]
  - The "optimal" amount of each of these resources arise from different sources. [Section 10.1]
    - Instructions and intermediates are dictated by the computational task to be performed.
    - Active interconnect and, to a lesser extent active compute resources, are dictated by the ratio between desired computational throughput and primitive computational speed.
- Interconnect is the dominant feature determining device area in conventional FPGAs. [Sections 7.1, 7.6, and 7.7]

- Interconnect requirement growth is superlinear in array size. Consequently, either interconnect area will continue to grow relative to non-interconnect area, or gate utilization will decrease as array sizes grow. [Sections 7.6 and 7.7]

- Since the non-interconnect area is trivial compared to network area for conventional FPGAs, optimizing for gate utilization is often short sighted and can result in unnecessarily large implementations. [Section 7.7]

- There are two interconnect functions typically required to realize a computation – spatial transport and temporal transport. To use silicon area most efficiently, these should be separated and handled via different mechanisms. [Chapter 11, especially Section 11.1]

  - Data values can be transported forward in time through registers or memories. While this ties up register area for the period of transport, it is much cheaper than tying up critical active, routing resources which occupy much more area.
  - Active interconnect can easily be the dominant area feature on a general-purpose device. It is used most efficiently when its resources are pipelined and reused at their capacity level – *i.e.* wires and switches should not sit idle holding a value once it has propagated past them. Rather, they should be redeployed to route new data once they have performed their spatial transport task.

- Memory plays two fundamental roles in reconfigurable computing architectures: (1) storage for instructions, (2) retiming of intermediate data. Both roles arise from the sharing of expensive, active hardware resources among multiple logical functions. [Identified in Chapters 9 through 11 and summarized in Section 15.1]

- Since interconnect is the major consumer of space on FPGAs, conventional architectures limit the interconnect by depopulating interconnect switches as much as possible. [Chapter 7 especially Sections 7.4 and 7.5]

- Physical place and route on devices with limited interconnect is computationally difficult because it is necessary to simultaneous satisfy a large number of constraints in order to find a valid mapping of the design netlist onto the physical network. [Chapter 12]

- We can alleviate the place and route problem in several different ways, each with different costs:

  - Provide rich interconnect (*e.g.* HP PLASMA). Easier mapping comes at the cost of greater cell area and lower computational density. [Section 12.8]
  - Provide rich, time-switched interconnect (*e.g.* UCB DHARMA). Rigid evaluation levels and lack of retiming can make this an expensive solution, as well, especially for larger arrays. [Section 12.8]
  - Provide rich retiming and time-switching (*e.g.* TSFPGA). Cell area can actually be lower than conventional FPGAs, but is higher than in DPGAs. This scheme sacrifices the high, peak computational throughput of traditional FPGAs. [Chapter 12]

- Eliminate interconnect (*e.g.* University of Toronto VEGA). This approach saves some additional area over DPGAs, but at the cost of significantly lower computational throughput and density than all other options. [Section 12.8]

Our focus and demonstration of these characteristics has been within the limited realm of *RP*-space. Nonetheless, most of the features which characterize *RP*-space show up more generally in general-purpose computational devices. Consequently, many of the characteristics identified here may have broader application to the extent they are not dominated by effects abstracted away in the *RP*-space model.

Terminology

τ see tau.

λ see lambda.

active computing resources The portions of a general-purpose architecture which actually compute results or transport data – *e.g.* ALUs, switches, wires. The term is typically used to distinguish such resources from overhead resources used to store descriptions or intermediate data.

active interconnect Switches and wires which actually produce a physical connection between a source and a destination. The term is used to distinguish resources used to actually perform switching from descriptions of switching operations or storage for intermediate data. Chapter 7 is primarily focussed on active interconnect, while Chapters 11 and 12 introduce forms of switched interconnect where the distinction becomes quite important.

bit processing element A generic term for the primitive computational unit which produces one bit of result. Conventionally, each FPGA LUT is a bit processing element, as is each bit-slice in an SIMD ALU datapath. See Chapter 8.

context A generic term used to refer to a slice of instructions and intermediate data used by a general-purpose device on a single cycle. See configuration context and data context.

control stream An independent thread of execution. When the computation varies with time and data, the control stream determines which sets of instructions are executed on a give cycle. A computational device may support a single control stream (*e.g.* processors, SIMD, pure VLIW) or multiple control streams (*e.g.* MSIMD, MIMD). See Section 8.5.

configurable computing Computing by configuring interconnect between programmable function units to wire up computations spatially. See Sections 1.3 and 2.3.

configurable computing architectures Architectures where there is only one or a few instructions loaded per active computing element and there is limited bandwidth to reload an entire configuration context. These architectures are used for configurable computing where the computation is typically arranged via spatial interconnect of computing elements as opposed to programmable computing architectures which realize computation by rapid temporal reuse of a few, central active computing resources. See Section 2.3.

computational density See functional density.

computational throughput Computations performed per unit time. *i.e.* Operations completed per unit time.

configuration context The collection of bits which describe the behavior of a general-purpose machine on one operation cycle. Equivalently, the collection of all instructions required to specify the behavior of a general-purpose device at one point in time. See Section 2.3.

data context The data used by a general-purpose device on one cycle of execution.

distance delay The critical path delay through a placed circuit taking into account the distance between logically adjacent functional units. See Section 12.6.

datapath granularity Datapath width. The number of bit processing elements or interconnect switches controlled in SIMD fashion by a single instruction. See Section 8.3.1.

deployable resources Resources whose role can be determined at run-time. *e.g.* A memory which can be used as an instruction store or as a data store; Interconnect which can be used to distribute instructions or to deliver data between functional units. Distinguished from resources which are dedicated to a single function at fabrication time. See Section 13.1.

dynamic Marked by a continuous usually productive activity or change. In this context usually used to distinguish quantities, particularly, instructions, which change on a cycle-by-cycle basis. Contrast with static and quasistatic. See Section 10.3.4.

dynamic instruction distribution Instruction distribution allowing instructions to change on a cycle-by-cycle basis. See Section 10.3.4.

DPGA Dynamically Programmable Gate Array – Fine-grained programmable array where each processing element has a small, local configuration memory allowing processing elements to change instructions, array-wide, on a cycle-by-cycle basis. See Chapters 10 and 11.

FPGA Field Programmable Gate Array – A collection of configurable processing units embedded in a configurable interconnection network. See Sections 2.4 and 4.5.

functional density Computations performed per unit space-time. Usually measured in $\mathrm{Ops}/\lambda^2\mathrm{s}$. See Section 2.6.1.

functional diversity The number of different functions which are resident and rapidly accessible from a unit of computational area. The density of instructions stored on a general-purpose computing device. See Section 2.6.2.

general-purpose computing Computing using devices which can be configured to solve any number of computing tasks. See Section 2.1.

iDPGA Dynamically Programmable Gate Array with input retiming registers – A DPGA including input retiming registers. See Chapter 11.

input depth The temporal range of input retiming registers in the iDPGA or similar architectures. See Chapter 11.

input folding A style for reducing the amount of active switching interconnect by sharing crossbar inputs among multiple sources. See Section 12.2.

instruction The set of bits which describe the behavior of one computational unit and its associated interconnect. See Section 2.2.2.

instruction context See configuration context.

instruction density See functional diversity.

instruction depth Number of instructions per compute element stored local to the compute element.

irregular computing task Task which require a large sequence of different computations and where operations are heavily data-dependent. See Section 2.5.

Kolmogorov complexity Of all programs which can be used to calculate a particular set of values, the length of the smallest such one. Ultimately, this is the least number of bits into which a piece of data can be described. Kolmogorov complexity is, primarily, a conceptual description of the lower bound as there is no algorthimic way to find such the bound. See any information theory text such as [CT91].

lambda (λ) – half the minimum feature size in a silicon process. Lambda is used to normalize out the effects of different process sizes when comparing implementations. Area normalized to λ^2 units is roughly comparable between processes which differ primarily in feature size. See Section 2.6.1.

low instruction entropy Computing tasks which require a limited set of operations with very regular flow, admitting to heavy compression of instruction distribution requirements. See Section 8.3.

lookup table A small, typically programmable, memory where the address bits act as inputs and data read out serves as an output. An n-input, m-output lookup table can implement any, deterministic mapping between n input bits and m output bits. We frequently refer to a k-input, 1-output lookup table as a k-LUT. See Section 2.4.

LUT Look Up Table – see lookup table.

MATRIX Multiple ALU architecture with Reconfigurable Interconnect – A flexible general-purpose computing architecture which defers binding of instructions and instruction resources until use. Instruction storage and distribution resources are unified with datapath compute, memory, and interconnect resources, allowing the basic instruction architecture to be defined at run-time. See Chapter 13.

metaconfiguration A higher and more primitive level of configuration than traditional instructions which defines the sources and distribution paths for dynamic control including instructions. See multi-level configuration. See Section 13.1 and 10.8.2.

microcycle One primitive machine cycle on architectures which evaluate logical tasks over several smaller clock cycles. See Section 10.5.1. Microcycle evaluation is a common theme in Chapters 10 through 13.

multicontext Having more than one configuration for the entire general-purpose device. Usually used to refer to devices or architectures which hold multiple such configurations on chip. Also used to describe evaluation schemes which compute a result using more than one device-wide configuration. See Chapter 10.

multi-level configuration Hierarchical configuration where higher levels of configuration describe the architecture, behavior, and distribution used by lower levers of configuration. See metaconfiguration. See Section 13.1.

output folding A style for reducing the amount of active switching interconnect by sharing crossbar outputs among multiple sinks. See Section 12.2.

partial reconfiguration The ability for individual or small numbers of processing units to change instructions without requiring an entire reload of all instructions across a general-purpose computing device. See Sections 8.3.3 and 10.3.4.

quasistatic Changing, but on an time scale much slower than standard operation. An intermediate point of activity between dynamic and static.

quasistatic instruction distribution Instructions which change during an application, but do so slowly compared to the rate of execution. A quasistatic instruction might be in effect for hundreds of cycles before changing. See Section 10.3.4.

Rent's Rule An empirical relationship between the number of i/o's in and out of a cluster of logic and the number of logical elements inside the logic ($N_{io} = CN_{gates}^{p}$). See Section 7.6.

regular computing task Tasks which need to repeatedly perform the same collection of operations to a large amount of data with little data-dependent flow control. See Section 2.5.

retiming Changing the time at which particular events occur. In this work, used largely to describe the transportation of signals forward in time between the point in time when they are generated to the point in time when they are consumed. See Section 10.1. Retiming is a major theme in Chapters 10 through 12.

robust architectural points Design points where we can bound the inefficiency to some constant percentage when the task has different characteristics from the architecture. See Chapter 9 starting in Section 9.3.

***RP*-space** A high-level abstraction of the reconfigurable computing design space parameterized by key instruction and interconnect features. See Chapter 9.

run-time reconfiguration The ability to change device configuration during a computational task.

segmentable datapath A SIMD controlled n-bit datapath which can be dynamically or quasistatically reconfigured to treat the datapath as k, $\frac{n}{k}$-bit words, for certain, restricted, values of k. See Section 13.4.

subarray An organizational unit in array architectures composed of multiple processing elements but not the entire device. In the DPGA and TSFPGA, the subarray defines the extent of local interconnect and the set of processing elements which share common resources such as decoders and instruction distribution. See Section 10.4.1.

spatial transport Movement of intermediate data in space from the point of production to the point of consumption. See Section 11.1.

static Showing little change; characterized by a lack of movement, animation or progression. In this context used primarily to distinguish values and instructions which do not change during an operational epic. Contrast with static and quasistatic. See Section 10.3.4.

static instruction distribution Instruction distribution where instructions are set at the beginning of a computational task and do not changed during execution. See Section 10.3.4.

programmable computing architectures General-purpose computing architectures which heavily and rapidly reuse a single or small number of active computing resources for many different functions (*e.g.* conventional microprocessors). See Section 2.3.

tau (τ) The delay parameter for a process. One τ is the delay required for one inverter to drive a single, equally large inverter.

temporal pipelining Reusing general-purpose resources in time to evaluate different components of a single logical task. Like spatial pipelining, the result is produced after traversing a number of pipelining stages. Unlike spatial pipelining, the same physical resources are used to evaluate each stage of the pipeline. Temporal pipelining reduces spatial requirements, whereas spatial pipelining increases throughput. See Sections 10.1 and 10.5.1.

temporal transport Movement of intermediate data in time from the microcycle on which the value is produced to the one where it is consumed. See Section 11.1.

timestep A particular microcycle in the evaluation of a computing task. See Section 12.1.

time-switched input register An input register supporting data retiming on architectures which time-switch their interconnect. The input register loads the value from its associated network output only when the current timestep matches a programmed value. See Section 12.1.

TSFPGA Time-Switched Field Programmable Gate Array – Fine-grained programmable array where the physical interconnect is shared and switched in time. See Chapter 12.

yielded computational density The effective computational density which an application or task extracts from a computational device. Mismatches in datapath granularity, interconnect richness, or control may cause a device to provide computational capacity below its peak. See Section 2.6.1 and examples given in Chapter 4.

Bibliography

[ABI+95] K. Asanovic, J. Beck, B. Irissou, D. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 Vector Microprocessor. In *Hot Chips VII Proceedings*, August 1995.

[ACC+96] Rick Amerson, Richard Carter, W. Bruce Culbertson, Phil Kuekes, and Greg Snider. Plasma: An FPGA for Million Gate Systems. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 10–16, February 1996.

[ADD90] Creigton Asato, Christoph Ditzen, and Suresh Dholakia. A Data-Path Multiplier with Automatic Insertion of Pipeline Stages. *IEEE Journal of Solid-State Circuits*, 25(2):383–387, August 1990.

[AFM+89] Kazutami Arimoto, Kazuyasu Fujishima, Yoshio Matsuda, Masaki Tsukude, Tukasa Oishi, Wataru Wakamiya, Shin-Ichi Satoh, Michihiro Yamada, and Takao Nakano. A 60-ns 3.3-V-Only 16-Mbit DRAM with a Multipurpose Register. *IEEE Journal of Solid-State Circuits*, 24(5):1176–1183, October 1989.

[AKY+96] Yoshiharu Aimoto, Tohru Kimura, Yoshikazu Yabe, Hideki Heiuchi, Youetsu Nakazawa, Masato Motomura, Takuya Koga, Yoshihiro Fujita, Masayuki Hamada, Takaho Tanigawa, Hajime Nobusawa, and Kuniaki Koyama. A 7.68GIPS 3.84GB/s 1W Parallel Image-Processing RAM Integrating a 16Mb DRAM and 128 Processors. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 372–373. IEEE, February 1996.

[AL94] Aditya A. Agarwal and David Lewis. Routing Architectures for Hierarchical Field Programmable Gate Arrays. In *Proceedings 1994 IEEE International Conference on Computer Design*, pages 475–478. IEEE, October 1994.

[Alg90] Algotronix Ltd., Edinburgh, UK. *The Configurable Logic Data Book*, 1990.

[Alt94] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *FLEX 8000 Handbook*, May 1994.

[Alt95] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *Data Book*, March 1995.

[Alt96] Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020. *Digital Signal Processing in FLEX Devices*, January 1996.

[ANAB+92] Fuad Abu-Nofal, Rick Avra, Kanti Bhabuthmal, Rob Bhamidipaty, Greg Blanck, Andy Charnas, Peter DelVecchio, Joe Grass, Joel Grinberg, Norm Hayes, George Haber, Jim Hunt, Govind Kizhepat, Adam Malamy, Al Marston, Kaushal Mehta, Sunil Nanda, Hoa Van Nguyen, Rajiv Patel, Andy Ray, Jim Reaves, Alan Rogers, Stefan Rusu, Tom Shay, Irwan Sidharta, Terry Tham, Peter Tong, Richard Trauben, Anthony Wong, David Yee, Naeem Maan, Don Steiss, and Lynn Youngs. A Three-Million-Transistor Microprocessor. In *1992 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 108–109. IEEE, February 1992.

[ANH+88] Masakazu Aoki, Yoshinobu Nakagome, Masashi Horiguchi, Hitoshi Tanaka, Shin'ichi Ikenaga, Jun Etoh, Yoshifumi Kawamoto, Shin'ichiro Kimura, Eiji Takeda, Hideo Sunami, and Kiyoo Itoh. A 60-ns 16-Mbit CMOS DRAM with a Transposed Data-Line Structure. *IEEE Journal of Solid-State Circuits*, 23(5):1113–1119, October 1988.

[AOT+94] Mikio Asakura, Tsukasa Ooishi, MAsaki Tsukude, Shigeki Tomishima, Takahisa Eimori, Hideto Hidaka, Yoshikazu Ohno, Kazutani Arimoto, Kazuyasu Fujishima, Tadashi Nishimura, and Tsutomu Yoshihara. An Experimental 256-Mb DRAM with Boosted Sense-Ground Scheme. *IEEE Journal of Solid-State Circuits*, 29(11):1303–1308, November 1994.

[AS93] Peter Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[ASO+90] Shingo Aizaki, Toshiyuki Shimizu, Masayoshi Ohkawa, Kazuhiko Abe, Akane Aizaki, Manabu Ando, Osamu Kudoh, and Isao Sasaki. A 15-ns 4-Mb CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 25(5):1063–1067, October 1990.

[Atm94] Atmel Corporation, 2125 O'Nel Drive, San Jose, CA 95131. *Configurable Logic Design and Application Book*, 1994.

[ATT94] ATT Microelectronics, 555 Union Boulevard, Room 21Q-133BA, Allentown, PA 18103. *Immplementing and Optimizing Multipliers in ORCA FPGAs*, November 1994.

[ATT95] ATT Microelectronics, 555 Union Boulevard, Room 21Q-133BA, Allentown, PA 18103. *AT&T Field-Programmable Gate Arrays Data Book*, April 1995.

[AWG94] Lalit Agarwal, Mike Wazlowski, and Sumit Ghosh. An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs. In Duncan Buell and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 101–1100, Los Alamitos, California, April 1994. IEEE Computer Society, IEEE Computer Society Press.

[BAB+95] William Bowhill, Randy Allmon, Shane Bell, Elizabeth Cooper, Dale Donchin, John Edmondson, Timothy Fischer, Paul Gronowski, Anil Jain, Patricia Kroesen, Bruce Loughlin, Ronald Preston, Paul Rubinfeld, Michael Smith, Stephen

Thierauf, and Gilbert Wolrich. A 300MHz 64b Quad-Issue CMOS RISC Microprocessor. In *1995 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 182–183. IEEE, February 1995.

[BBB+95] David Bearden, Roger Bailey, Brad Beavers, Carlos Gutierrez, Chin-Cheng Kau, Kurt Lewchuk, Paul Rossback, and Mike Tabom. A 133MHz 64b Four-Issue CMOS Microprocessor. In *1995 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 174–175. IEEE, February 1995.

[BCE+94] Jeremy Brown, Derrick Chen, Ian Eslick, Edward Tau, and André DeHon. A 1μ CMOS Dynamically Programmable Gate Array. Transit Note 112, MIT Artificial Intelligence Laboratory, November 1994. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn112.ps.Z`.

[BCH+84] Erich K. Baier, Rainer Clemen, Werner Haug, Walter Fischer, Rolf Mueller, Wolf Dieter Loehlein, and Horst Barsuhn. A Fast 256K DRAM Designed for a Wide Range of Applications. *IEEE Journal of Solid-State Circuits*, 19(5), October 1984.

[BCK93] Narasimha B. Bhat, Kamal Chaudhary, and Ernest S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device. UCB/ERL M93/42, University of California, Berkeley, June 1993.

[BDK94] Michael Bolotski, André DeHon, and Thomas F. Knight, Jr. Unifying FPGAs and SIMD Arrays. In *FPGA Workshop*, 1994. proceedings not available outside of the workshop; paper available as Transit Note #95 Anonymous FTP `transit.ai.mit.edu:transit-notes/tn95.ps.Z`. Anonymous FTP `transit.ai.mit.edu:papers/dpga-fpga94.ps.Z`.

[BDN84] John J. Barnes, Armando L. DeJesus, and David Novosel. Circuit Techniques for a 25 ns 16K$\times$1 SRAM Using Address-Transition Detection. *IEEE Journal of Solid-State Circuits*, 19(4):455–460, August 1984.

[BFRV92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts, 02061 USA, 1992.

[Bha93] Narasimha B. Bhat. Novel Techniques for High Performance Field Programmable Logic Devices. UCB/ERL M93/80, University of California, Berkeley, November 1993.

[BLMR83] Ted Burggraff, Al Love, Richard Malm, and Ann Rudy. The IBM Los Gatos Logic Simulation Machine Hardware. In *Proceedings of the International Conference on Computer Design*, pages 584–587, October 1983.

[BMNW87] Gerald Boudun, Pierre Mollier, Jean Nuez, and Franck Wallart. A 30ns-32b Programmable Arithmetic Operator. In *1987 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 54–55. IEEE, February 1987.

[Bri90] Timothy Bridges. The GPA Machine: A Generally Partitionable MSIMD Architecture. In *Proceedings of the Third Symposium on The Frontiers for Massively Parallel Computations*, pages 196–202. IEEE, 1990.

[Bro92] Stephen Brown. *Routing Algorithms and Architectures for Field-Programamble Gate Arrays*. PhD thesis, University of Toronto, January 1992.

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to Programmable Active Memories. PRL Report 3, DEC Paris Research Laboratory, 85, Av. Victor Hugo, 92563 Rueil-Malmaison Cedex, France, June 1989.

[BRV92] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Programmable Active Memories: A Performance Assessment. Prl report, DEC Paris Reserch Laboratory, 85, Av. Victor Hugo, 92563 Rueil-Malmaison Cedex, France, June 1992.

[BSV+95] Michael Bolotski, Thomas Simon, Carlin Vieri, Rajeevan Amirtharajah, and Thomas F. Knight Jr. Abacus: A 1024 Processor 8ns SIMD Array. In *Advanced Research in VLSI 1995*, 1995. Anonymous FTP `ftp.ai.mit.edu:pub/users/misha/arvlsi95.ps.gz`.

[BTA93] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Los Alamitos, California, April 1993. IEEE Computer Society, IEEE Computer Society Press.

[CBBF87] Craig Caren, Bruce Benjamin, James Boddie, and Michael Fuccio. A 60ns CMOS DSP with On-Chip Instruction Cache. In *1987 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 156–157. IEEE, February 1987.

[CC86] Remi Cissou and Remy Chapelle. A High-Speed 640kbit CMOS RAM. *IEEE Journal of Solid-State Circuits*, 21(3):390–396, June 1986.

[CCS+91] Terry Chappell, Barbara Chappell, Stanley Schuster, James Allan, Stephen Klepner, Rajiv Joshi, and Robert Franch. A 2-ns Cycle, 3.8ns Access 512-kb CMOS ECL SRAM with a Fully Pipelined Architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1577 ff., November 1991.

[CD96] Derrick Chen and André DeHon. TSFPGA: A Fine-Grain Reconfigurable Architecture with Time-Switched Interconnect. Transit Note 134, MIT Artificial Intelligence Laboratory, January 1996. Anonymous FTP `transit.ai.mit.edu:transit-notes/tn134.ps.Z`.

[CDd+95] A. Charmas, A. Dalal, P. deDood, P. Ferolito, B. Frederick, O. Geva, D. Greenhill, H. Hingarh, J. Kaku, L. Kohn, L. Lev, M. Levitt, R. Melanson, S. Mitra, R. Sundar, M. Tamjidi, P. Wang, D. Wendell, R. Yu, and G. Zyner. A 64b Microprocessor with

Multimedia Support. In *1995 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 178–179. IEEE, February 1995.

[CDF+86] William S. Carter, Khue Duong, Ross H. Freeman, Hung-Cheng Hsieh, Jason Y. Ja, John E. Mahoney, Luan T. Ngo, and Shelly L. Sze. A User Programmable Reconfigurable Logic Array. In *IEEE 1986 Custom Integrated Circuits Conference*, pages 233–235. IEEE, May 1986.

[CDF+95] Jonathan Change, Anand Dharmaraj, Michael Filardo, Astushi Ike, Bala Joshi, Takeshi Kitahara, Anand Krishnamoorthy, Simon Li, Sanjay Mansingh, Osamu Moriyama, Arvind Narayan, Kesiraju Rao, Murugappan Ramaswami, Farnad Sajjadian, Mike Simone, Gene Shen, Ravi Swami, John Szeto, Viji Thirumalaiswamy, Shalesh Thusoo, and DeFrost Tovey. SPARC64+: HaL's Second Generation 64-bit SPARC Processor. In *Proceedings of Hot Chips VII*, page 3.2, August 1995. `http://www.hal.com/docs/PS/sparc64_plus.ps`.

[CDH+88] Sow Chu, Jan Dikken, Cornelis Hartgring, Frans List, John Raemaekers, Simon Bell, Brendan Walsh, and Roelof Salters. A 25-ns Low-Power Full-CMOS 1-Mbit (128K$\times$8) SRAM. *IEEE Journal of Solid-State Circuits*, 23(5):1078–1084, October 1988.

[CH84] Larry F. Childs and Ryan T. Hirose. An 18 ns 4K$\times$4 CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 19(5):545–551, October 1984.

[Cha93] Kenneth David Chapman. Fast Integer Multipliers fit in FPGAs. *EDN*, 39(10):80, May 12 1993. Anonymous FTP `www.ednmag.com:EDN/di_sig/DI1223Z.ZIP` .

[Cho89] Paul Chow, editor. *The MIPS-X RISC Microprocessor*. Kluwer Academic Publishers, 1989.

[CKC+89] Daeje Chin, Changhyun Kim, Yunho Choi, Dong-Sun Min, Hong Sun Hwang, Hoon Choi, Sooin Cho, Tae Young Chung, Chan J. Park, Yunseung Shin, Kwangpyuk Suh, and Yong Park. An Experimental 16-Mbit DRAM with Reduced Peak-Current Noise. *IEEE Journal of Solid-State Circuits*, 24(5):1191–1198, October 1989.

[Cla95] Peter Clarke. Pilkington Preps Reconfigurable Video DSP. *Electronic Engineering Times*, page 16, August 7 1995. Online briefing `http://www.pmel.com/dsp.html`.

[cLCWMS96] Chih chang Lin, Douglas Chang, Yu-Liang Wu, and Malgorzata Marek-Sadowska. Time-Multiplexed Routing Resources for FPGA Design. In *Proceedings of the Custom Integrated Circuits Conference*, May 1996.

[CLRA90] Mike Cai, Daniel Luthi, Peter Ruetz, and Peng Ang. A 40 MHz Programmable and Reconfigurable Filter Processor. In *Proceedings of the 1990 Custom Integrated Circuits Conference*, pages 13.2.1–13.2.4. IEEE, May 1990.

[CME93] Chi-Jui Chou, Satish Mohanakrishnan, and Joseph B. Evans. FPGA Implementation of Digital Filters. In *International Conference on Signal Processing Applications and Technology*, 1993. Anonymous FTP `ftp.tisl.ukans.edu:pub/projects/DSP/FPGA/Digital_Filters.ps`.

[CR92] Dev C. Chen and Jan M. Rabaey. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[CSA+91] Paul Chow, Soon Ong Seo, Dennis Au, Terrence Choy, Bahram Fallah, David Lewis, Cherry Li, and Jonathan Rose. A 1.2μm CMOS FPGA using Cascaded Logic Blocks and Segmented Routing. In Will Moore and Wayne Luk, editors, *FPGAs*, pages 91–102. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon, OX14 1NV, UK, 1991.

[CT91] Thomas Cover and Joy Thomas. *Elements of Information Theory*. John Wiley and Sons, Inc., New York, 1991.

[CTK+89] Shizuo Chou, Tsuneo Takano, Akio Kita, Fumio Ichikawa, and Masaru Uesugi. A 60-ns 16-Mbit DRAM with a Minimized Sensing Delay Caused by Bit-Line Stray Capacitance. *IEEE Journal of Solid-State Circuits*, 24(5):1176–1183, October 1989.

[D+92] William J. Dally et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[DeH94] André DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994. Anonymous FTP `transit.ai.mit.edu:papers/dpga-proc-fccm94.ps.Z`.

[Den82] Monty Denneau. The Yorktown Simulation Engine. In *19th Design Automation Conference*, pages 55–59. IEEE, 1982.

[DMNSV88] Srinivas Devadas, Hi-Keung Ma, A.R. Newton, and Alberto Sangiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(12):1290–1300, December 1988.

[Don74] Wilm E. Donath. Equivalence of Memory to "Random Logic". *IBM Journal of Research and Development*, 18(5):401–407, September 1974.

[Don79] Wilm E. Donath. Placement and Average Interconnection Lengths of Computer Logic. *IEEE Transactions on Circuits and Systms*, 26(4):272–277, April 1979.

[Dur94] Serge Durand. FPGA DLX processor. August 22, 1994 posting to `comp.arch.fpga`. Author may be reached at `durand@lslsun4.epfl.ch`, December 1994.

[DWA+92] Daniel Dobberpuhl, Richard Witek, Randy Allmon, Robert Anglin, Sharon Britton, Linda Chao, Robert Conrad, Daniel Dever, Bruce Gieseke, Gregory Hoeppner, John Kowaleski, Kathryn Kuchler, Maureen Ladd, Michael Leary, Liam Madden, Edward McLellan, Derrick Meyer, James Montanaro, Donald Priore, Vidya Rajagopalan, Sridhar Samudrala, and Sribalan Santhanam. A 200MHz 64b Dual-Issue CMOS Microprocessor. In *1992 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 106–107. IEEE, February 1992.

[EG95] Andrew Essen and Stephen Goldstein. Performance Evaluation of the Superscalar Speculative Execution HaL SPARC64 Processor. In *Proceedings of Hot Chips VII*, page 3.1, August 1995. `http://www.hal.com/docs/PS/sparc64_perf.ps`.

[EH94] James G. Eldredge and Brad L. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180–188, Los Alamitos, California, April 1994. IEEE Computer Society, IEEE Computer Society Press.

[Eps95] Dave Epstein. Chromatic Raises the Multimedia Bar. *Microprocessor Report*, 9(14):23 ff., October 23 1995. `http://www.chipanalyst.com/report/report9_14/page23.html`.

[FA93] Jahil Fadavi-Ardekani. $M \times N$ Booth Encoded Multiplier Generator Using Optimized Wallace Trees. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(2):120–125, June 1993.

[FHR94] Allan Fisher, Peter Highnam, and Todd Rockoff. A Four-Processor Building Block for SIMD Processor Arrays. *IEEE Journal of Solid-State Circuits*, 25(2):369–375, April 1994.

[FHT+92] Hiroshige Fujii, Chikahiro Hori, Tomoji Takada, Naoyuki Hatanaka, Tatsuhiko Demura, and Goichi Ootomo. A Floating-Point Cell Library and a 100-MFLOPS Image Signal Processor. *IEEE Journal of Solid-State Circuits*, 27(7):1080–1088, July 1992.

[FKM83] Allan L. Fisher, H. T. Kung, and Louis M. Monier. Architecture of the PSC: A Programmable Systolic Chip. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 48–53, June 1983.

[FKS91] Richard Forsyth, Bob Krysiak, and Roger Shepherd. T9000 – Superscalar Transputer. In *Proceedings of Hot Chips III*, pages 8.15–8.25, August 1991.

[Fly66] Michael J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[FOS+89] Syuso Fujii, Masaki Ogihara, Mitsuru Shimizu, Munehiro Yoshida, Kenji Numata, Takahiko Hara, Shigeyoshi Watanabe, Shizuo Sawada, Tomohisa Mizuno, Junpei Kumagai, Susumu Yoshikawa, Seiji Kaki, Yoshikazu Saito, Hideaki Aochi, Takeshi Hamamoto, and Koichi Toita. A 45-ns 16-Mbit DRAM with Triple-well Structure. *IEEE Journal of Solid-State Circuits*, 24(5):1170–1175, October 1989.

[Fos96] Richard Foss. Implementing Application Specific Memory. In *1996 IEEE International Solid-State Circuits Conference*, pages 260–261. IEEE, February 1996.

[FOW+86] Tohru Furuyama, Takashi Ohshawa, Yohji Watanabe, Hidemi Ishiuchi, Toshiharu Watanabe, Takeshi Tanaka, Kenji Natori, and Osamu Ozawa. An Experimental 4-Mbit CMOS DRAM. *IEEE Journal of Solid-State Circuits*, 21(5):605 ff., October 1986.

[FPH+90] Stephen Flannagan, Perry Pelley, Norman Herr, Bruce Engles, Taisheng Feng, Scott Nogle, John Eagan, Robert Dunnigan, Lawrence Day, and Roger Kung. 8-ns CMOS 64K×4 and 256K×1 SRAM's. *IEEE Journal of Solid-State Circuits*, 25(5):1049–1054, October 1990.

[Fra92] Robert Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.

[Fre94] Philip Freidin. R16: A 20MHz 16-bit RISC Processor in a XC4005. Informal presentation at FCCM'94 and `comp.arch.fpga` posting. Author may be reached at `fliptron@netcom.com`, April 1994.

[FRV+86] Stephen Flannagan, Paul Reed, Peter Voss, Scott Nogle, Lawrence Day, David Sheng, John Barnes, and Roger Kung. Two 13-ns 64K CMOS SRAM's with Very Low Active Power and Improved Asynchronous Circuit Techniques. *IEEE Journal of Solid-State Circuits*, 21(5):692–703, October 1986.

[FSO+86] Syuso Fujii, Shozo Saito, Yoshio Okada, Masayuki Sato, Shizuo Sawada, Satoshi Shinozaki, Kenji Natori, and Osamu Ozawa. A 50-μA Standby 1M×1/256K×4 CMOS DRAM with High-Speed Sense Amplifier. *IEEE Journal of Solid-State Circuits*, 21(5):643–647, October 1986.

[Gam81] Abbas El Gamal. Two-Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits. *IEEE Transactions on Circuits and Systems*, 28(2):127–138, February 1981.

[GBB+96] Paul Gronowski, Peter Bannon, Michael Bertone, Randel Blake-Campos, Gregory Bouchard, William Bowhill, David Carlson, Ruben Castelino, Dale Donchin, Richard Fromm, Mary Gowan, Anil Jain, Bruce Loughlin, Shekhar Mehta, Jeanne

Meyer, Robert Mueller, Andy Olesin, Tung Pham, Ronald Preston, and Paul Robinfeld. A 433MHz 64b Quad-Issue RISC Microprocessor. In *1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 222–223. IEEE, February 1996.

[GGA+85] Abbas El Gamal, David Gluss, Peng-Huat Ang, Jonathan Greene, and Justin Reyneri. A CMOS 32b Wallace Tree Multiplier-Accumulator. In *1985 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 194–195. IEEE, February 1985.

[GHH+96] Henry Green, Scott Harper, Rhett Hudson, Wencheng Li, Daniel Lough, Qiang Lu, Shah Musa, Brenda O'Connor, Kevin Paar, and Peter Athanas. The Hokie Instant RISC Microprocessor. WWW `http://www.ee.vt.edu/courses/ee6504_athanas/rapid.html`, 1996.

[GHK+91] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel Lopresti. Building and Using a Highly Programmable Logic Array. *IEEE Computer*, 24(1):81–89, January 1991.

[GHS+87] Will Gubbels, Cornelis Hartgring, Roelof Salters, Jos Lammerts, Michael Tooher, Patrick Hens, Joseph Bastiaens, Jan Dijk, and Marc Sprokel. A 40-ns/100-pF Low-Power Full-CMOS 256K (32K$\times$8) SRAM. *IEEE Journal of Solid-State Circuits*, 22(5):741 ff., October 1987.

[GK89] John Gray and Tom Kean. Configurable Hardware: A New Paradigm for Computation. In Charles Seitz, editor, *Advanced Research in VLSI: proceedings of teh Decennial Caltech Conference on VLSI*, pages 279–295, March 1989.

[GM93] Maya Gokhale and Ron Minnich. FPGA Computing in a Data Parallel C. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, Los Alamitos, California, April 1993. IEEE Computer Society, IEEE Computer Society Press.

[GN94] Greg Goslin and Bruce Newgard. *16-TAP, 8-Bit FIR Filter Applications Guide*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, November 1994. `http://www.xilinx.com/appnotes/fir_filt.pdf`.

[GNAB93] Jeffrey Gray, Andrew Naylor, Arthur Abnous, and Nader Bagherzadeh. VIPER: A VLIW Integer Microprocessor. *IEEE Journal of Solid-State Circuits*, 28(12):1377–1382, December 1993.

[GNC+90] Carla Golla, Fulvio Nava, Franco Cavallotti, Alessandro Cremonesi, and Giulio Casagrande. 30-MSamples/s Programmable Filter Processor. *IEEE Journal of Solid-State Circuits*, 25(6):1502–1509, December 1990.

[GOI95] Eric Gayles, Robert Owens, and Mary Jane Irwin. The MGAP-2: A Micro-Grained Massively Parallel Array Processor. In *Eith Annual IEEE International ASIC Conference and Exhibit*, pages 333–337, April 1995.

[GOK+92] Hiroyuki Goto, Hiroaki Ohkubo, Kenji Kondou, Masayoshi Ohkawa, Hitoshi Mitani, Shinichi Horiba, Masakazu Soeda, Fumihiko Hayashi, Yutaro Hachiya, Toshiyuki Shimizu, Manabu Ando, and Zensuke Matsuda. A 3.3-V 12-ns 16-Mb SRAM. *IEEE Journal of Solid-State Circuits*, 27(11):1490–1496, November 1992.

[Gol87] Alex Goldberger. A High Performance, Easy to Program DSP for General Purpose Applications. In *Mini/Micro Northeast Conference Record*, pages 27/3 1–10, April 1987.

[Gra94] Jan Gray. homebuilt processors using FPGAs (long). December 11, 1994 posting to `comp.arch.fpga`. Author may be reached at `jsgray@ix.netcom.com`, December 1994.

[Gra96] Jan Gray. j32 FPGA Processor. Personal communications `jsgray@ix.netcom.com`, February 1996.

[Gro87] Robert Grondalski. A VLSI Chip Set for Massively Parallel Architecture. In *IEEE International Solid-State Circuits Conference*, pages 198–199, 1987.

[GSNS92] Gensuke Goto, Tomio Sato, Masao Nakajima, and Takao Sukemura. A 54×54-b Regularly Structured Tree Multiplier. *IEEE Journal of Solid-State Circuits*, 27(9):1229–1236, July 1992.

[HAH+92] Hideto Hidaka, Kazutami Arimoto, Kazutoshi Hirayama, Masanori Hayashikoshi, Mikio Asakura, Masaki Tsukude, Tsukasa Oishi, Shinji Kawai, Katsuhiro Suma, Yasuhiro Konishi, Koji Tanaka, Wataru Wakamiya, Yoshikazu Ohno, and Kazuyasu Fujishima. A 34-ns 16-Mb DRAM with Controllable Voltage Down-Converter. *IEEE Journal of Solid-State Circuits*, 27(7):1020 ff., July 1992.

[Has87] Chuck Hastings. When is a Memory Not a Memory. In *Proceedings of the Electro/87 Mini/Micro Northeast*, pages 1132, 4/5/1–18, 1987.

[Haw91] David Hawley. Advanced PLD Architectures. In Will Moore and Wayne Luk, editors, *FPGAs*, pages 11–23. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon, OX14 1NV, UK, 1991.

[HBD94] Robert Heaton, Donald Blevins, and Edward Davis. A Bit-Serial VLSI Array Processing Chip for Image Procesing. *IEEE Journal of Solid-State Circuits*, 25(2):364–368, April 1994.

[HDJ+88] Hung-Cheng Hsieh, Khue Duong, Jason Y. Ja, Roy Kanazawa, Luan T. Ngo, Liane G. Tinkey, Ross H. Freeman, and William S. Carter. A 9000-Gate User-Programmable Gate Array. In *IEEE 1988 Custom Integrated Circuits Conference*, pages 15.3.1–7. IEEE, May 1988.

[HFML85] Dennis A. Henlin, Michael T. Fertsch, Moshe Mazin, and Edard T. Lewis. A 16×16 Bit Pipelined Multiplier Macrocell. *IEEE Journal of Solid-State Circuits*, 20(2):542–547, April 1985.

[HHC+87] Mark Horowitz, John Hennessy, Paul Chow, Glenn Gulak, John Acken, Anant Agarwal, Chorng-Yeung Chu, Scott McFarling, Steven Przybylski, Steven Richardson, Arturo Salz, Richard Simoni, Don Stark, Peter Steenkiste, Steven Tjiang, and Malcom Wing. A 32b Microprocessor with On-Chip 2K byte Instruction Cache. In *1987 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 30–31. IEEE, February 1987.

[HKKM96] Makoto Hanawa, Kenji Kaneko, Tatsuya Kawashimo, and Hiroshi Maruyama. A 4.3 ns 0.3μm CMOS 54$\times$54 Multiplier Using Precharged Pass-Transistor Logic. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 364–365. IEEE, February 1996.

[HKM+90] Toshihiko Hirose, Hirotada Kuriyama, Shuji Murakami, Kojiro Yuzuriha, Takao Mukai, Kazuhito Tsutsumi, Yasumasa, Nishimura, Yoshio Kohno, and Kenji Anami. A 20-ns 4-Mb CMOS SRAM with Hierarchical Word Decoding Architecture. *IEEE Journal of Solid-State Circuits*, 25(5):1068–1074, October 1990.

[HOW+86] Fumio Horiguchi, Mitsugi Ogura, Shigeyoshi Watanabe, Koji Sakui, Naokazu Miyawaki, Yasuo Itoh, Kei Kurosawa, Fujio Masuoka, and Hisakazu Iizuka. A High-Performance 1-Mbit Dynamic RAM with a Folded Capacitor Cell. *IEEE Journal of Solid-State Circuits*, 21(6):1076–1082, December 1986.

[HP90] John Hennessey and David Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[HS84] Kye S. Hedlund and Lawrence Snyder. Systolic Architectures – A Wafer Scale Approach. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 604–610. IEEE, IEEE Computer Society Press, October 1984.

[HT95] Hannes Hassler and Naofumi Takagi. Function Evaluation by Table Look-up and Addition. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 10–16, July 1995.

[ID95] Tsuyoshi Isshiki and Wayne Wei-Ming Dai. High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 167–173. ACM, February 1995.

[IIF+95] Hiroyuki Igura, Masanori Izumikawa, Koichiro Furuta, Tohru Mogami, Tadahiko Horiuchi, and Masakazu Yamashina. 100MHz, 0.55mm^2, 2mW, 16-b Stacked-CMOS Multiplier-Accumulator. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 597–600. IEEE, May 1995.

[IKM+94] Koichiro Ishibashi, Kunihiro Komiyaji, Sadayuki Morita, Toshiro Aoto, Shuji Ikeda, Kyoichiro Asayama, Atsuyosi Koike, Toshiaki Yamanaka, Naotaka Hashimoto, Haruhito Iida, Fumio Kojima, Koichi Motohashi, and Katsuro Sasaki.

A 12.5-ns 16-Mb CMOS SRAM with Common-Centroid-Geometry-Layout Sense Amplifiers. *IEEE Journal of Solid-State Circuits*, 29(4):411 ff., April 1994.

[IYK+88] Michihiro Inoue, Toshio Yamada, Hisakazu Kotani, Hiroyuki Yamauchi, Atsushi Fujiwara, Junko Matsushima, Hironori Akamatsu, Masanori Fukumoto, Masafumi Kubota, Ichiro Nakao, Nobuo Aoi, Genshu Fuse, Shin-Ichi Ogawa, Shinji Odanaka, Atsushi Ueno, and Hiroshi Yamamoto. A 16-Mbit DRAM with a Relaxed Sense-Amplifier-Pitch Open-Bit Line Architecture. *IEEE Journal of Solid-State Circuits*, 23(5):1104–1112, October 1988.

[JF72] J. Robert Jump and Dennis R. Fritsche. Microprogrammed Arrays. *IEEE Transactions on Computers*, 21(9):974–984, September 1972.

[JL95] David Jones and David Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 495–498. IEEE, May 1995.

[JOSV95] Chris Jones, John Oswald, Brian Schoner, and John Villasenor. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In Peter Athanas and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[KAI+86] Yoshifumi Kobayashi, Kazutami Arimoto, Yuto Ikeda, Masahiro Hatanaka, Koichiro Mashiko, Michihiro Yamad, and Takao Nakano. A High-Speed 64K×4 CMOS CRAM Using On-Chip Self-Timing Techniques. *IEEE Journal of Solid-State Circuits*, 21(5):655–661, October 1986.

[KCE+85] Howard L. Kaltzer, Pierre D. Coppens, Wayne F. Ellis, John A. Fifield, Daryl J. Kokoszka, Terry L. Leasure, Christopher P. Miller, Quan Nguyen, Ronald E. Papritz, Charles S. Patton, J. Michael Poplawski, Jr., Steven W. Tomashot, and Willem B. Van Der Hoeven. An Experimental 80-ns 1-Mbit DRAM with Fast Page Operation. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[KDK+90] Yasuhiro Konishi, Katsumi Dosaka, Takahiro Komatsu, Yoshinori Inoue, Masaki Kumanoya, Youichi Tobita, Hideki Genjyo, Masao Nagatomo, and Tsutomu Yoshihara. A 38-ns 4-Mb DRAM with A Battery-Backup (BBU) Mode. *IEEE Journal of Solid-State Circuits*, 25(5):1112–1117, October 1990.

[KDK+92] Toshiaki Kirihata, Sang Dhong, Koji Kitamura, Toshio Sunaga, Yasunao Katayama, Roy Scheuerlein, Akashi Satoh, Yoshinori Sakaue, Kentaroh Tobimatsu, Koji Hosokawa, Takaki Saitoh, Takefumi Yoshikawa, Hideki Hashimoto, and Michiya Kazusawa. A 14-ns 4-Mb DRAM with 300-mW Active Power. *IEEE Journal of Solid-State Circuits*, 27(9):1222 ff., September 1992.

[KDS+96] Shinichi Kozu, Masayuki Daito, Yukinori Sugiyama, Hiroaki Suzuki, Hiroshi Morita, Masahiro Nomura, Kouhei Nadehara, Souichiro Ishibuchi, Masako

Tokuda, Yoshihisa Inoue, Takashi Nakayama, Hisao Harigai, and Yoichi Yano. A 100MHz, 0.4W Processor with 200MHz Multiply-Adder, using Pulse-Register Technique. In *1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 140–141. IEEE, February 1996.

[Kea89] Tom Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. PhD thesis, University of Edinburgh, January 1989.

[KEK+85] Yasuo Kobayashi, Hirotsugo Eguchi, Osamu Kudoh, Toshio Hara, Hideyuki Ooka, Isao Sasaki, Manabu Andoh, and Masato Tameda. A 10-μW Standby Power 256K CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 20(5):935–940, October 1985.

[KFM+85] Masaki Kumanoya, Kazuyasu Fujishima, Hideshi Miyatake, Yasumasa, Nishimura, Kazunori Saito, Takayuki, Matsukawa, Tsutomu Yoshihara, and Takao Nakano. A Reliable 1-Mbit DRAM with Multi-Bit-Test Mode. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[KFO84] Robert A. Kertis, Kerlly J. Fitzpatrick, and Kul B. Ohri. A 60 ns 256K$\times$1 Bit DRAM Using LD^3 Technology and Double-Level Metal Interconnection. *IEEE Journal of Solid-State Circuits*, 19(5):585–590, October 1984.

[KHANW94] Alan Y. Kwentus, Hing-Tsun Hung, and Jr. Alan N. Wilson. An Architecture for High-Performance/Small-Area Multipliers for Use in Digital Filtering Applications. *IEEE Journal of Solid-State Circuits*, 29(2):117–121, February 1994.

[KHK+93] Goro Kitsukawa, Masashi Horiguchi, Yoshiki Kawajiri, Takayuki Kawahara, Takesada Akiba, Yasushi Kawase, Toshikazu Tachibana, Takeshi Sakai, Masakazu Aoki, Syoji Shukuri, Kazuhiko Sagara, Ryo Nagai, Yuzuru Ohji, Norio Hasegawa, Natsuki Yokoyama, Teruaki Kisu, Hisaomi Yamashita, Tokuo Kure, and Takashi Nishida. 256-Mb DRAM Circuit Technologies for File Applications. *IEEE Journal of Solid-State Circuits*, 28(11):1105–1112, November 1993.

[KHN+96] Masuyoshi Kurokawa, Akihiko Hashiguchi, Ken'ichiro Nakamura, Hiroshi Okuda, Koji Aoyama, Takao Yamazaki, Mitsuharu Ohki, Mitsuo Soneda, Katsunori Seno, Ichiro Kumata, Masatoshi Aikawa, Hirokazu Hanaki, and Seiichiro Iwase. 5.4GOPS Linear Array Architecture DPS for Video-Format Conversion. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 254–255. IEEE, February 1996.

[KIK+86] Shinpei Kayano, Katsuki Ichinose, Yoshio Kohno, Hirofumi Shinohara, Kenji Anami, Shuji Murakami, Tomohisa Wada, Yuji Kawai, and Yoichi Akasaka. 25-ns 256K$\times$1/64K$\times$4 CMOS SRAM's. *IEEE Journal of Solid-State Circuits*, 21(5):686–691, October 1986.

[KK79] Steven I. Kartashev and Svetlana P. Kartashev. A multicomputer Systems with Dynamic Architecture. *IEEE Transactions on Computers*, 28(10):704–720, October 1979.

[KKHY88] Shoji Kawahito, Michitaka Kameyama, Tatsuo Higuchi, and Haruyaso Yamada. A 32×32-bit Multiplier Using Multiple-Valued MOS Current-Mode Circuits. *IEEE Journal of Solid-State Circuits*, 23(1):124–132, February 1988.

[KNK+87] Kenji Kaneko, Tetsuya Nakagawa, Atsushi Kiuchi, Yoshimune Hagiwara, Hirotada Ueda, and Hitoshi Matsushima. A 50ns DSP with Parallel Processing Architecture. In *1987 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 158–159. IEEE, February 1987.

[Knu71] Donal E. Knuth. Empirical Study of FORTRAN Programs. *Software Practice and Experience*, 1(1):105–133, 1971.

[Knu81] Donal E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachusetts, 2nd edition, 1981.

[KOT+96] Hideyuki Kabuo, Minoru Okamoto, Isao Tanaka, Hiroyuki Yasoshima, Shinichi Marui, Masayuki Yamasaki, Toshio Sugimura, Katsuhiko Ueda, Toshihiro Ishikawa, Hidetoshi Suzuki, and Ryuichi Asahi. An 80-MOPS-Peak High-Speed Low-Power Consumption 16-b Digital Signal Processor. *IEEE Journal of Solid-State Circuits*, 31(4):494–503, April 1996.

[KSB+90] Howard Kalter, Charles Stapper, John Barth, Jr., John DiLorenzo, Charles Drake, John Fifield, Gordon Kelly, Jr., Soctt Lewis, Willem Van Der Hoeven, and James Yankosky. A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC. *IEEE Journal of Solid-State Circuits*, 25(5):1118 ff., October 1990.

[KSE+87] Katsutaka Kimura, Katsuhiro Shimohigashi, Jun Etoh, Masamichi Ishihara, Kazuyuki Miyazawa, Shinji Shimizu, Yoshio Sakai, and Kunihiro Yagi. A 65-ns 4-Mbit CMOS DRAM with a Twisted Driveline Sense Amplifier. *IEEE Journal of Solid-State Circuits*, 22(5):651–656, October 1987.

[KSY+84] Hiroshi Kawamoto, Takashi Shinoda, Yasunori Yamaguchi, Shinji Shimizu, Kanji Ohishi, Nobuyoshi Tnimura, and Tokumasa Yasui. A 288K CMOS Psedostatic RAM. *IEEE Journal of Solid-State Circuits*, 19(5):619–623, October 1984.

[KT93] Won Kim and Russ Tuck. MasPar MP-2 PE Chip: A Totally Cool Hot Chip. In *Proceedings of Hot Chips V*, MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA 94086, August 1993.

[KTO+87] Takaaki Komatsu, Hitoshi Taniguchi, Nobumichi Okazaki, Toshiyuki Nishihara, Shigeki Kayama, Naoya Hoshi, Jun-Ichi Aoyama, and Takashi Shimada. A 35-ns $128K \times 8$ CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 22(5):721–726, October 1987.

[Kun82] H. T. Kung. Why Systolic Architectures? *IEEE Computer*, 15(1):37–46, January 1982.

[KWA+88] Yoshio Kohno, Tomohisa Wada, Kenji Anami, Yuji Kawai, Kojiro Yuzuriha, Takayuki Matsukawa, and Shimpei Kayano. A 14-ns 1-Mbit CMOS SRAM with Variable Bit Organization. *IEEE Journal of Solid-State Circuits*, 23(5):1060–1066, October 1988.

[LBK+89] Nicky Lu, Gary Bronner, Koji Kitamur, Roy Scheuerlein, Walter Henkels, Sang Dhong, Yasunao Katayama, Toshiaki Kirihata, Hideto Niijima, Robert Franch, Wei Hwang, Motoo Nishiwaki, Frank Pesavento, T. V. Rajeevakumar, Yoshinori Sakaue, Yasusuke Suzuki, Yasunori Iguchi, and Eiji Yano. A 22-ns 1-Mbit CMOS High-Speed DRAM with Address Multiplexing. *IEEE Journal of Solid-State Circuits*, 24(5):1198 ff., October 1989.

[LC95] Jianmin Li and Chung-Kuan Cheng. Routability Improvement Using Dynamic Interconnect Architecture. In Peter Athanas and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 61–67, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[LCwH+88] Nicky Lu, Hu Chao, wei Hwang, Walter Henkels, T. V. Rajeevakumar, Hussein Hanafi, Lewis Terman, and Robert Franch. A 20-ns 128-kbit$\times$4 High-Speed DRAM with 330-Mbit/s Data Rate. *IEEE Journal of Solid-State Circuits*, 23(5):1140 ff., October 1988.

[LE94] Marianne E. Louie and Milos D. Ercegovac. A Variable Precision Multiplier for Field Programmable Gate Arrays. In *Second International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*. ACM, February 1994. proceedings not available outside of the workshop.

[LE96] Per Larsson-Edefors. A 965-Mb/s 1.o-μm Standard CMOS Twin-Pipe Serial/Parallel Multiplier. *IEEE Journal of Solid-State Circuits*, 31(2):230–239, February 1996.

[Lei79] Charles Leiserson. Systolic Priority Queues. CMU-CS-TR 115, Carnegie-Mellon University, Pittsbugh, Pennsylvania 15213, April 1979.

[Lev77] Lance Leventhal. Cut Your Processor's Computation Time. *Electronic Design*, 25(17):82–88, August 16 1977.

[LGC84] Claude P. Lerouge, Pierre Girard, and Joël S. Colardelle. A Fast 16 Bit Parallel Multiplier. *IEEE Journal of Solid-State Circuits*, 19(3):338–342, June 1984.

[LGS87] Josephy Y. Lee, Hugh L. Garvin, and Charles W. Slayman. A High-Speed High-Density Silicon 8$\times$8-bit Parallel Multiplier. *IEEE Journal of Solid-State Circuits*, 22(1):35–40, February 1987.

[LLNK96] Jon Lotz, Gregg Lesartre, Samuel Naffziger, and Don Kipp. A Quad-Issue Out-of-Order RISC CPU. In *1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 210–211. IEEE, February 1996.

[LR71] B. S. Landman and R. L. Russo. On Pin Versus Block Relationship for Partitions of Logic Circuits. *IEEE Transactions on Computers*, 20:1469–1479, 1971.

[LRSS84] Chris Lutz, Steve Rabin, Chuck Seitz, and Don Speck. Design of the MOSAIC Element. In Paul Penfield, Jr., editor, *Proceedings, Conference on Advanced Research in VLSI*, pages 1–10, Cambdrige, MA, January 1984.

[LS90] Junien Labrousse and Gerrit Slavenburg. A 50MHz Microprocessor with a Very Long Instruction Word Architecture. In *1990 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 44–45. IEEE, February 1990.

[LS92] Joe Laskowski and Henry Samueli. A 150-MHz 43-Tap Half-Band FIR Digital Filter in 1.2-μm CMOS Generated by Silicon Compiler. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pages 11.4.1–11.4.4. IEEE, May 1992.

[LS93] Fang Lu and Henry Samueli. A 200-MHz CMOS Pipelined Multiplier-Accumulator Using a Quasi-Domino Dynamic Full-Adder Cell Design. *IEEE Journal of Solid-State Circuits*, 28(2):123–132, February 1993.

[Mal94] Lisa Maliniak. Hardware Emulation Draws Speed From Innovative 3D Parallel Processing Based on Custom ICs. *Electronic Design*, pages 38–41, May 30 1994.

[MD96] Ethan Mirsky and André DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1996. Anonymous FTP `transit.ai.mit.edu:papers/matrix-fccm96.ps.Z`.

[Min67] Robert C. Minnick. A Survey of Microcellular Research. *Journal of the ACM*, 14(2):203–241, April 1967.

[Min71] Robert Minnick. A Programmable Cellular Array. In *Fifth Annual IEEE International Computer Society Conference: Hardware Software Firmware Trade-Offs*, pages 25–26. IEEE, September 1971.

[Mir96] Ethan Mirsky. Course-Grain Reconfigurable Computer. Master's thesis, Massachusetts Institute of Technology, 545 Technology Sq., Cambridge, MA 02139, June 1996. Anonymous FTP `transit.ai.mit.edu:papers/eamirsky-matrix-meng.ps.Z`.

[MKM+84] Koichiro Mashiko, Toshifumi Kobayashi, Hiroshi Miyamoto, Kazutami Arimoto, Yoshikazu Morooka, Masahiro Hatanaka, Michihiro Yamada, and Takao Nakano. A 70 ns 256K DRAM with Bit-Line Shield. *IEEE Journal of Solid-State Circuits*, 19(5), October 1984.

[MKS+84] Amr Mohsen, Roger I. Kung, Carl J. Simonsen, Joseph Schutz, Paul D. Madland, Esmatz Z. Hamdy, and Mark T. Bohr. The Design and Performance of CMOS 256K Bit DRAM Devices. *IEEE Journal of Solid-State Circuits*, 19(5):610–620, October 1984.

[MKS+92] Masato Matsumiya, Shoichiro Kawashima, Makoto Sakata, Masahiko Ookura, Toru Miyabo, Toru Koga, Kazuo Itabashi, Kazuhiro Mizutani, Hiroshi Shimada, and Noriyuki Suzuki. A 3.3-V 12-ns 16-Mb SRAM. *IEEE Journal of Solid-State Circuits*, 27(11):1497–1503, November 1992.

[MMK+89] Fumio Miyaji, Yasushi Matsuyama, Yoshikazu Kanaishi, Katsunori Senoh, Takashi Emori, and Yoshiaki Hagiwara. A 25-ns 4-Mbit CMOS SRAM with Dynamic Bit-Line Loads. *IEEE Journal of Solid-State Circuits*, 24(5):1213–1218, October 1989.

[MMM+91] Shigeru Mori, Hiroshi Miyamoto, Yoshikazu Morooka, Shigeru Kikuda, Makoto Suwa, Mitsuya Kinoshita, Atsushi Hachisuka, Hideaki Arima, Michihiro Yamada, Tsutomu Yoshihara, and Shimpei Kayano. A 45-ns 64-Mb DRAM with a Merged Match-Line Test Architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1486–1492, November 1991.

[MMN+90] Jiro Miyake, Toshinori Maeda, Yoshito Nishimichi, Joji Katsura, Takashi Taniguchi, Seiji Yamaguchi, Hisakazu Edamatsu, Shigeru Watari, Yoshiyuki Takagi, Kazuhiko Tsuji, Shigeo Kuninobu, Steve Cox, Douglas Duschatko, and Douglas MacGregor. A 40 MIPS (Peak) 64-bit Microprocessor with One-Clock Physical Cache Load/Store. In *1990 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 42–43. IEEE, February 1990.

[MMS+84] Osamu Minato, Toshiaki Masuhara, Toshio Sasaki, Keizo Matsumoto, Yoshio Sakai, Tetsuya, and Hayashida. A 20 ns 64K CMOS Static RAM. *IEEE Journal of Solid-State Circuits*, 19(6), October 1984.

[MNA+87] Koichiro Mashiko, Masao Nagatomo, Kazutami Arimoto, Yoshio Matsuda, Kiyohiro Furutani, Takayuki Matsukawa, Michihiro Yamada, Tsutomu Yoshihara, and Takao Nakano. A 4-Mbit DRAM with Folded-Bit-Line Adaptive Sidewall-Isolated Capacitor (FASIC) Cell. *IEEE Journal of Solid-State Circuits*, 22(5):643–650, October 1987.

[MNH+91] Junji Mori, Masato Nagamatsu, Masashi Hirano, Shigeru Tanaka, Makoto Noda, Yoshiaki Yoyoshima, Kazuhiro Hashimoto, Hiroyuki Hayashida, and Kenji Maeguchi. A 10-ns 54×54-b Parallel Structured Full Array Multiplier with 0.5μm CMOS Technology. *IEEE Journal of Solid-State Circuits*, 26(4):600–606, April 1991.

[MNS+96] Hiroshi Makino, Yasunobu Nakase, Hiroaki Suzuki, Hiroyuki Morinaka, Hirofumi Shinohara, and Koichiro Mashiko. An 8.8-ns 54×54-Bit Multiplier with

High Speed Redundant Binary Architecture. *IEEE Journal of Solid-State Circuits*, 31(6):773–783, June 1996.

[MOT+87] Masataka Matsui, Takayuki Ohtani, Jun-Ichi Tsujimoto, Hiroshi Iwai, Azuma Suzuki, Katsuhiko Sato, Mitsuo Isobe, Kazuhiko Hashimoto, Mitsuchika Saitoh, Hideki Shibata, Hisayo Sasaki, Tadashi Matsuno, Jun-Ichi Matsunaga, and Tetsuya Iizuka. A 25-ns 1-Mbit CMOS SRAM with Loading-Free Bit Lines. *IEEE Journal of Solid-State Circuits*, 22(5):733–740, October 1987.

[MSM+84] Jun-Ichi Miyamoto, Shinji Saito, Hiroshi Momose, Hideki Shibata, Koichi Kanzaki, and Tetsuya Iizuka. A High-Speed 64K CMOS RAM with Bipolar Sense Amplifiers. *IEEE Journal of Solid-State Circuits*, 19(5):557–564, October 1984.

[MWA+96] James Montanaro, Richard Witek, Krishna Anne, Andrew Black, Elizabeth Cooper, Dan Dobberpuhl, Paul Donahure, Jim Eno, Alejandro Farell, Gregory Hoeppner, David Kruckemyer, Thomas Lee, Peter Lin, Liam Madden, Daniel Murray, Mark Pearce, Sribalan Santhanam, Kathryn Snyder, Ray Stephany, and Stephen Thierauf. A 160MHz 32b 0.5W CMOS RISC Microprocessor. In *1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 210–211. IEEE, February 1996.

[MYM+87] Hiroshi Miyamoto, Tadato Yamagata, Shigeru Mori, Toshifumi Kobayashi, Shin-Ichi Satoh, and Michihiro Yamada. A Fast 256K$\times$4 CMOS DRAM with Distributed Sense and Unique Restore Circuit. *IEEE Journal of Solid-State Circuits*, 22(5):861–867, October 1987.

[MYO+96] Hiroaki Murakami, Naoka Yano, Yukio Ootaguro, Yukio Sugeno, Maki Ueno, Yukinori Muroya, and Tsuneo Aramaki. A Multiplier-Accumulator Macro for a 45 MIPS Embedded RISC Processor. *IEEE Journal of Solid-State Circuits*, 31(7):1067–1071, July 1996.

[NHK95] Kouhei Nadehara, Miwako Hayashida, and Ichiro Kuroda. *A Low-Power, 32-bit RISC Processor with Signal Processing Capability and its Multiply-Adder*, volume VIII of *VLSI Signal Processing*, pages 51–60. IEEE, 1995.

[Nic90] John Nickolls. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. In *Compcon Spring 90*, pages 25–28. IEEE, 1990.

[NNO+91] Takeshi Nagai, Kenji Numata, Masaki Ogihara, Mitsuru Shimizu, Kimimasa Imai, Takahiko Hara, Munehiro Yoshida, Yoshikazu Saito, Yoshiaki Asao, Shizuo Sawada, and Syuso Fujii. A 17-ns 4-Mb DRAM. *IEEE Journal of Solid-State Circuits*, 26(11):1538 ff., November 1991.

[NSLKE86] Tobias G. Noll, Doris Schmitt-Landsiedel, Heinrich Klar, and Gerhard Enders. A Pipelined 300-MHz Multiplier. *IEEE Journal of Solid-State Circuits*, 21(3):411–416, June 1986.

[NSS+86] Kazutaka Nogami, Takayasu Sakurai, Kazuhiro Sawada, Tetsunori Wada, Katsuhiko Sato, Mitsuo Isobe, Masakazu Kakumu, Shigeru Morita, Shunji Yokogawa, Masaaki Kinugawa, Tetsuya Asami, Kazuhiko Hashimoto, Jun-Ichi Matsunaga, Hiroshi Nozawa, and Tetsuya Iizuka. 1-Mbit Virtually Static RAM. *IEEE Journal of Solid-State Circuits*, 21(5):662–668, October 1986.

[NTT+91] Yoshinobu Nakagome, Hitoshi Tanaka, Kan Takeuchi, Eiji Kume, Yasushi Watanabe, Toru Kaga, Yoshifumi Kawamoto, Fumio Murai, Ryuichi Izawa, Digh Hisamoto, Teruaki Kisu, Takashi Nishida, Eiji Takeda, and Kiyoo Itoh. An Experimental 1.5-V 64-Mb DRAM. *IEEE Journal of Solid-State Circuits*, 26(4):465 ff., April 1991.

[Nut77] Gary J. Nutt. Microprocessor Implementation of a Parallel Processor. In *Proceedings of the Fourth Annual International Symposium on Computer Architecture*, pages 147–152. ACM, 1977.

[OFW+87] Takashi Ohsawa, Tohru Furuyama, Yohji Watanabe, Hiroto Tanaka, Natsuki Kushiyama, Kenji Tsuchida, Yohsei Nagahama, Satoshi Yamano, Takeshi Tanaka, Satoshi Shinozaki, and Kenji Natori. A 60-ns 4-Mbit CMOS DRAM with Built-In Self-Test Function. *IEEE Journal of Solid-State Circuits*, 22(5):663–668, October 1987.

[OHK+90] Takayuki Ootani, Shigeyuki Hayakawa, Masakazu Kakumu, Akira Aono, Masaaki Kinugawa, Hideki Takeuchi, Kazuhiro Noguchi, Tomoaki Yabe, Katsuhiko Sato, Kenji Maeguchi, and Kiyofumi Ochii. A 4-Mb CMOS SRAM with a PMOS Thin-Film-Transistor Load Cell. *IEEE Journal of Solid-State Circuits*, 25(5):1082–1091, October 1990.

[OKH+84] Nobumichi Okazaki, Takaaki Komatsu, Naoya Hoshi, Kunihiko Tsuboi, and Takashi Shimada. A 16 ns 2K×8 Full CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 19(5):552–556, October 1984.

[ONN+88] Hiroaki Okuyama, Takeshi Nakano, Shuichi Nishida, Etsuro Aono, Hisahiro Satoh, and Shigeru Arita. A 7.5-ns 32K×8 CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 23(5):1054–1059, October 1988.

[OSS+95] Norio Ohkubo, Makoto Suzuki, Toshinobu Shinbo, Toshiaki Yamanaka, Akihiro Shimizu, Katsuro Sasaki, and Yoshinobu Nakagome. a 4.4 ns CMOS 54×54-b Multiplier Using Pass-Transistor Multiplexer. *IEEE Journal of Solid-State Circuits*, 30(3):251–257, February 1995.

[OTW+91] Yukihito Oowaki, Kenji Tsuchida, Yohji Watanabe, Daisaburo Takashima, Masako Ohita, Hiroaki Nakano, Shigeyoshi Watanabe, Akihiro Nitayama, Fumio Horiguchi, Kazunori Ohuchi, and Fujio Masuoka. A 33-ns 64-Mb DRAM. *IEEE Journal of Solid-State Circuits*, 26(11):1498–1505, November 1991.

[Ple90] Plessey Semiconductors, Cheney Manor, Sindown, Wiltshire SN2 2QW, UK. *ERA60100 Datasheet – Electrically Reconfigurable Array*, May 1990.

[PML+89] A. Picco, J. C. Michalina, B. Laurier, D. Fuin, P. Menut, and JL. Laborie. The ST18940/41: An Advanced Single-chip Digital Signal Processors. In *Proceedings of the 1989 IEEE International Symposium on Circuits and Systems*, pages 1559–1562. IEEE, May 1989.

[QC88] Le Quach and Richard Chueh. CMOS Gate Array Implementation of SPARC. In *Digest of Papers COMPCON'88*, pages 14–17. IEEE, Februrary 1988.

[Ram93] Rambus Inc. Architectural Overview. Produce Literature, 1993. Rambus Inc., 2465 Latham Steet, Mountain View, CA 94040.

[Raz94] Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard Univeristy, May 1994. Anonymous FTP `ftp.eecs.harvard.edu:users/smith/theses/razdan-thesis.tar.gz`.

[RB91] Jonathan Rose and Stephen Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *IEEE Journal of Solid-State Circuits*, 26(3):277–282, March 1991.

[RDB+94] Ehsan Rashid, Eric Delano, Michael Buckley, Jason Zheng, Francis Schumacher, Gordon Kurpanek, John Shelton, Tom Alexander, Nazeem Noordeen, Mark Ludwig, Alisa Scherer, Chaim Amir, Dan Cheung, Prasad Sabada, Ram Rajamani, Nick Fiduccia, Bill Ches, Kamyar Eshghi, Fred Eatock, Denny Renfrow, John Keller, Paul Ilgenfrizt, Ilan Krashinsky, Darryl Weatherspoon, Shrikant Ranade, Dave Goldberg, and William Byrg. A CMOS RISC CPU with On-Chip Parallel Cache. In *1994 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 210–211. IEEE, February 1994.

[RFLC90] Jonathan Rose, Robert Francis, David Lewis, and Paul Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, October 1990.

[RK92] Dirk Reuver and Heinrich Klar. A Configurable Convolution Chip with Programmable Coefficients. *IEEE Journal of Solid-State Circuits*, 27(7):1121–1123, July 1992.

[RPJ+84] Christopher Rowen, Steven Przbylski, Norman Jouppi, Thomas Gross, John Shott, and John Hennessey. A Pipelined 32b NMOS Microprocessor. In *1984 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 180–181. IEEE, February 1984.

[RS92] Poornachandra B. Rao and Alexander Skavantzos. New Multiplier Designs Based on Squared Law Algorithms and Table Look-ups. In *Conference Record of the Twenty-Sixth Asilomar Conference on Signals, Systems and Computers (volume 2)*, pages 686–690, October 1992.

[RS94] Rahul Razdan and Michael D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–180. IEEE Computer Society, November 1994. Anonymous FTP `ftp.eecs.harvard.edu:users/smith/papers/micro94.ps.gz`.

[RSV87] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 6(5):727–751, September 1987.

[Rue89] Peter Ruetz. The Architectures and Design of a 20-MHz Real-Time DSP Chip Set. *IEEE Journal of Solid-State Circuits*, 24(2):338–348, April 1989.

[SA90] Chip Sterns and Peng Ang. Yet Another Multiplier Architecture. In *Proceedings of the IEEE 1990 Custom Integrated Circuits Conference*, pages 24.6.1–4. IEEE, May 1990.

[SAI+85] Hirofumi Shinohara, Kenji Anami, Katsuki Ichinose, Tomohisa Wada, Yoshio Kohno, Yuji Kawai, Yoichi Akasaka, and Shinpei Kayano. A 45-ns 256K CMOS Static RAM with Tri-Level Word Line. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[Sch71] Mario R. Schaffner. A System with Programmable Hardware. In *Fifth Annual IEEE International Computer Society Conference: Hardware Software Firmware Trade-Offs*, pages 17–18. IEEE, September 1971.

[Sch78] Mario R. Schaffner. Processing by Data and Program Blocks. *IEEE Transactions on Computers*, 27(11):1015–1028, November 1978.

[SCLB84] Stanley E. Schuster, Barbara Chappell, Victor Di Lonardo, and Peter E. Britton. A 20 ns 64K (4K$\times$16) NMOS RAM. *IEEE Journal of Solid-State Circuits*, 19(5), October 1984.

[Sei92] Charles L. Seitz. Mosaic C: An Experimental Fine-Grain Multicomputer. In A. Bensoussan and J.-P. Verjus, editors, *Future Tendencies in Computer Science, Control and Applied Mathematics: Internantional Conference on the Occasion of the 25th Anniversary of INRIA*, pages 69–85. Sprinter-Verlag, December 1992.

[Seo94] Soon Ong Seo. A High Speed Field-Programmable Gate Array Using Programmable Minitiles. Master's thesis, University of Toronto, Ontario, Canada, 1994.

[SFO+85] Shozo Saito, Syuso Fujii, Yoshio Okada, Shizuo Sawada, Satoshi Shinozaki, Kenji Natori, and Osamo Ozawa. A 1-Mbit CMOS DRAM with Fast Page Mode and Static Column Mode. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[SGS+85] Lal C. Sood, James S. Golab, John Salter, John E. Leiss, and John J. Barnes. A Fast 8K$\times$8 CMOS SRAM With Internal Power Down Design Techniques. *IEEE Journal of Solid-State Circuits*, 20(5):941–950, October 1985.

[SH89] Mark R. Santoro and Mark A. Horowitz. SPIM: A Pipelined 64×64-bit Iterative Multiplier. *IEEE Journal of Solid-State Circuits*, 24(2):487–493, April 1989.

[SHU+88] Katsuro Sasaki, Shoji Hanamura, Kiyotsugo Ueda, Takao Oono, Osamu Minato, Yoshio Sakai, Satoshi Meguro, Masayoshi Tsunematsu, Toshiaki Masuhara, Masaaki Kubotera, and Hiroshi Toyoshima. A 15-ns 1-Mbit CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 23(5):1067–1073, October 1988.

[SIS+90] Katsuro Sasaki, Koichiro Ishibashi, Katsuhiro Shimohigashi, Toshiaki Yamanaka, Nobuyuki Moriwaki, Shigeru Honjo, Shuji Ikeda, Atsuyoshi Koike, Satoshi Meguro, and Osamu Minato. A 23-ns 4-Mb CMOS SRAM with 0.2-μA Standby Current. *IEEE Journal of Solid-State Circuits*, 25(5):1075–1081, October 1990.

[SIU+92] Katsuro Sasaki, Koichiro Ishibashi, Kiyotsugo Ueda, Kunihiro Komiyaji, Toshiaki Yamanaka, Naotaka Hashimoto, Hiroshi Toyoshima, Fumio Kojima, and Akihiro Shimizu. A 7-ns 140-mW 1-Mb CMOS SRAM with Current Sense Amplifier. *IEEE Journal of Solid-State Circuits*, 27(11):1511–1518, November 1992.

[SIY+89] Katsuro Sasaki, Koichiro Ishibashi, Toshiaki Yamanaka, Naotaka Hashimoto, Takashi Nishida, Katsuhiro Shimohigashi, Shoji Hanamura, and Shigeru Honjo. A 9-ns 1-Mbit CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 24(5):1219–1225, October 1989.

[SJ88] Naresh R. Shanbhag and Pushkal Juneja. Parallel Implementation of a 4×4 Multiplier Using Modified Booth's Algorithm. *IEEE Journal of Solid-State Circuits*, 23(4):1010–1013, August 1988.

[SKI+88] Hiroshi Shimada, Shoichiro Kawashima, Hideo Itoh, Noriyuki Suzuki, and Takashi Yabu. A 45-ns 1-Mbit CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 23(1):53–58, February 1988.

[SKK+91] Katsuyuki Sato, Kanehide Kenmizaki, Shoji Kubono, Toshio Mochizuki, Hidetomo Aoyagi, Michitaro Kanamitsu, Soichi Kunito, Hiroyuki Uchida, Yoshihiko Yasu, Atsushi Ogishima, Sho Sano, and Hiroshi Kawamoto. A 4-Mb Pseudo SRAM Operating at 2.6±1V with 3-μA Data Retention Current. *IEEE Journal of Solid-State Circuits*, 26(11):1556–1561, November 1991.

[SKPS84] Robert Sherburne, Jr., Manolis Katevenis, David Patterson, and Carlo Sequin. A 32b NMOS Microprocessor with a Large Register File. In *1984 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 168–169. IEEE, February 1984.

[SKS+93] Katsunori Seno, Kurt Knorpp, Lee-Lean Shu, Naoki Teshima, Hiroki Kihara, Hiroshi Sato, Fumio Miyaji, Minoru Takeda, Masayoshi Sasaki, Yoichi Tomo, Patrick Chuang, and Kazuyoshi Kobayashi. A 9-ns 16-Mb CMOS SRAM with Offset-Compenstated Current Sense Amplifier. *IEEE Journal of Solid-State Circuits*, 28(11):1119–1124, November 1993.

[SKYH92] M. Shiraishi, M. Koizumi, A. Yamaguchi, and H. Hoike. User Programmable 16Bit 50ns DSP. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pages 6.4.1–6.4.4. IEEE, May 1992.

[Sla95] Michael Slater. MicroUnity Lifts Veil on MediaProcessor. *Microprocessor Report*, 9(14):11 ff., October 23 1995. `http://www.chipanalyst.com/report/report9_14/page11.html`.

[SLM+89] Ramautar Sharma, Alexander D. Lopez, John A. Michejda, Steven J. Hillenius, John M. Andrews, and Arnold J. Studwell. A 6.75-ns 16×16-bit Multiplier in Single-Level-Metal CMOS Technology. *IEEE Journal of Solid-State Circuits*, 24(4):922–927, August 1989.

[SMI+84] Takayasu Sakurai, Junichi Matsunaga, Mitsuo Isobe, Takayuki Ohtani, Kazuhiro Sawada, Akira Aono, Hiroshi Nozawa, Tetsuya IIzuka, and Susumu Kohyama. A Low Power 46 ns 256 kbit CMOS Static RAM with Dynamic Double Word Line. *IEEE Journal of Solid-State Circuits*, 19(5):578–584, October 1984.

[SMK+94] Toshio Sunaga, Hisatada Miyatake, Koji Kitamura, Keishi Kasuya, Takaki Saitoh, Masahiro Tanaka, Norio Tanigaki, Yohtaro Mori, and Noritoshi Yamasaki. DRAM Macros for ASIC Chips. *IEEE Journal of Solid-State Circuits*, 30(9):1006–1014, September 1994.

[SNT+84] Shun'ishi Suzuki, Masumi Nakao, Toshio Takeshima, Masaaki Yoshida, Masanori Kikuchi, Kunio Nakamura, Takeshi Mizukami, and Masayuki Yanagisawa. A $128K\times8$ Bit Dynamic RAM. *IEEE Journal of Solid-State Circuits*, 19(5):624–626, October 1984.

[Sny85] Lawrence Snyder. An Inquiry into the Benefits of Multigauge Parallel Computation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 488–492. IEEE, August 1985.

[SPA+95] Gene Shen, Niteen Patkar, Hisashige Ando, David Chang, Charles Chen, Chien Chen, Frank Chen, Per Forssell, John Gmuender, Takeshi Kitahara, Hungwen Li, David Lyon, Robert Montoye, Leon Peng, Sunil Savkar, Jonathan Sherred, Mike Simone, Ravi Swami, DeFroset Tovey, and Ted Williams. A 64b 4-Issue Out-of-Order Execution RISC Processor. In *1995 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 170–171. IEEE, February 1995.

[SSL+92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. UCB/ERL M92/41, University of California, Berkeley, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, May 1992.

[SSN+92] Akinori Sekiyama, Teruo Seki, Shinji Nagai, Akihiro Iwase, Noriyuki Suzuki, and Masato Hayasaka. A 1-V Operating 256-kb Full-CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 27(5):776–782, May 1992.

[STN+93] Tadahiko Sugibayashi, Toshio Takeshima, Isao Naritake, Tatsuya Matano, Hiroshi Takada, Yoshiharu Aimoto, Koichiro Furuta, Mamoru Fujita, Takanori Saeki, Hiroshi Sugawara, Tatsunori Murotani, Naoki Kasai, Kentaro Shibahara, Ken Nakajima, Hiromitsu Hada, Takehiko Hamada, Naoaki Aizaki, Takemitsu Kunio, Eiichiro Kakehashi, Katsuhiro Masumori, and Takaho Tanigawa. A 30-ns 256-Mb DRAM with a Multidivided Array Structure. *IEEE Journal of Solid-State Circuits*, 28(11):1092–1098, November 1993.

[STT+88] Hiroshi Shimada, Yoshinao Tange, Kazuo Tanimoto, Michio Shiraishi, Noriyuki Suzuki, and Toshio Nomura. An 18-ns 1-Mbit CMOS SRAM. *IEEE Journal of Solid-State Circuits*, 23(5):1073–1077, October 1988.

[SUT+93] Katsuro Sasaki, Kiyotsugu Ueda, Koichi Takasugi, Hiroshi Toyoshima, Koichiro Ishibashi, Toshiaki Yamanaka, Naotaka Hashimoto, and Nagatoshi Ohki. A 16-Mb CMOS SRAM with a 2.3μm^2 Single-Bit-Line Memory Cell. *IEEE Journal of Solid-State Circuits*, 28(11):1125–1130, November 1993.

[SV93] Dinesh Somasekhar and V. Visvanathan. A 230-MHz Half-Bit Level Pipelinined Multiplier Using True Single-Phase Clocking. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(4):415–422, December 1993.

[SYN+94] Kazumasa Suzuki, Masakazu Yamashina, Takashi Nakayama, Masanori Izumikawa, Masahiro Nomura, Hiroyuki Igura, Hideki Heiuchi, Junichi Goto, Toshiaki Inoue, Youichi Koseki, Hitoshi Abiko, Kazuhiro Okabe, Atsuki Ono, Youichi Yano, and Hachiro Yamada. A 500MHz 32b 0.4μm CMOS RISC Processor LSI. In *1994 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 214–215. IEEE, February 1994.

[TEC+95] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995. Anonymous FTP `transit.ai.mit.edu:papers/dpga-proto-fpd95.ps.Z`.

[TFT+85] Yoshihisa Takayama, Shigeru Fujii, Tomoaki Tanabe, Kazuyuki Kawauchi, and Toshihiko Yoshida. A 1ns 20K CMOS Gate Array Series with Configurable 15ns 12K Memory. In *1985 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 196–197. IEEE, February 1985.

[TJ85] Ronald T. Taylor and Mark G. Johnson. A 1-Mbit CMOS Dynamic RAM with a Divided Bitline Matrix Architecture. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[TLB+90] Darius Tansalvala, Joel Lamb, Michael Buckley, Bruce Long, Sean Chapin, Jonathan Lotz, Eric Delano, Richard Luebs, Keith Erskine, Scott McMullen, Mark Forsyth, Robert Novak, Tony Gaddis, Doug Quarnstrom, Craig Gleason, Ehsan Rashid, Daniel Halperin, Leon Sigal, Harlan Hill, Craig Simpson, David Hollenbeck, John Spencer, Robert Horning, Hoang Tran, Thomas Hotchkiss, Duncan Weir, Donald Kipp, John Wheeler, Patrick Knebel, Jeffery Yetter, and Charles Kohlhardt. A 15 MIPS 32b Microprocessor. In *1990 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 52–53. IEEE, February 1990.

[TNH+96] Toshinari Takayanagi, Kazutaka Nogami, Fumitoshi Hatori, Naoyuki Hatanaka, Makoto Takahashi, Makoto Ichida, Shinji Kitabayashi, Tatsuya Higashi, Mike Klein, John Thomson, Roger Carpenter, Ravi Donthi, Denny Renfrow, Jason Zheng, Liane Tinkey, Brandi Maness, Jim Battle, Steve Purcell, and Takayasu Sakurai. 350MHz Time-Multiplexed 8-port SRAM and Word-Size Variable Multiplier for Multimedia DSP. In *1996 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 150–151. IEEE, February 1996.

[TNK+94] Yasuhiro Takai, Mamoru Nagase, Mamoru Kitamura, Yasuji Koshikawa, Naoyuki Yoshida, Yasuaki Kobayashi, Takashi Obara, Yukio Fukuzo, and Hiroshi Watanabe. 250 Mbytes/s Synchronous DRAM Using a 3-Stage-Pipelined Architecture. *IEEE Journal of Solid-State Circuits*, 29(4):426–431, April 1994.

[TTK+90] Toshio Takeshima, Masahide Takada, Hiroki Koike, Hiroshi Watanabe, Shigeru Koshimaru, Kenjiro Mitake, Wataru Kikuchi, Takaho Tanigawa, Tatsunori Murotani, Kenji Noda, Kazuhiro Tasaka, Koji Yamanaka, and Kuniaki Koyama. A 55-ns 16-Mb DRAM with Built-in Self-Test Function Using Microprogram ROM. *IEEE Journal of Solid-State Circuits*, 25(4):903–910, August 1990.

[TTS+86] Masahide Takada, Toshio Takeshima, Mitsuru Sakamoto, Toshiyuki Shimizu, Hitoshi Abiko, Takuya Katoh, Masanori Kikuchi, Sakari Takahashi, Yoshinori Sato, and Yasukazu Inoue. A 4-Mbit DRAM with Half-Internal-Voltage Bit-Line Precharge. *IEEE Journal of Solid-State Circuits*, 21(5), October 1986.

[TTT+94] Satoru Tanoi, Yasuhiro Tanaka, Tetsuy Tanabe, Akio Kita, Toshio Inada, Ryoji Hamazaki, Yoshio Ohtsuki, and Masaru Uesugi. A 32-Bank 256-Mb DRAM with Cache and TAG. *IEEE Journal of Solid-State Circuits*, 29(11):1330–1336, November 1994.

[TTU+91] Masao Taguchi, Hiroyoshi Tomita, Toshiya Uchida, Yasuhiro Ohnishi, Kimiaki Sato, Taiji Ema, Masaaki Higashitani, and Takashi Yabu. A 40-ns 64-Mb DRAM with 64-b Parallel Data Bus Architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1493–1497, November 1991.

[UKY84] Masaru Uya, Katsuyuki Kaneko, and Juro Yasui. A CMOS Floating Point Multiplier. *IEEE Journal of Solid-State Circuits*, 19(5):697–702, October 1984.

[USO+93] Katsuhiko Ueda, Toshio Sugimura, Minoru Okamoto, Shinichi Marui, Toshihiro Ishikawa, and Mikio Sakakihara. A 16b Low-Power-Consumption Digital Signal Processor. In *1993 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 28–29. IEEE, February 1993.

[VBB93] Joseph Varghese, Michael Butts, and Jon Batcheller. An Efficient Logic Emulation System. *IEEE Transactions on Very Large Scale Integration (VLSI) Syatems*, 1(2):171–174, June 1993.

[Vil82] W. Vilkelis. Lead Reduction Among Combinational Logic Circuits. *IBM Journal of Research and Development*, 26(3):342–348, May 1982.

[vMWvW+86] Jef van Meerbergen, Frank Welten, Frans van Wijk, Jan Stoter, Jos Huisken, Antoine Delaruelle, and Karel Van Eerdewijk. An 8 MIPS CMOS Digital Signal Processor. In *1985 IEEE International Solid-State Circuits Conference, Digst of Technical Papers*, pages 84–85. IEEE, February 1986.

[vN66] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966. Compiled by Arthur W. Burks.

[VPP+89] Peter Voss, Leo Pfennings, Cathal Phelan, Cormac O'Connell, Thomas Davies, Hans Ontrop, Simon Bell, and Roelof Salters. A 14-ns 256K$\times$1 CMOS SRAM with Multiple Test Modes. *IEEE Journal of Solid-State Circuits*, 24(4):874–881, August 1989.

[VSCZ96] John Villasenor, Brian Schoner, Kang-Ngee Chia, and Charles Zapata. Configurable Computer Solutions for Automatic Target Recognition. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*. IEEE, April 1996.

[WBEK+88] Todd Williams, Kenneth Beilstein, Badih El-Kareh, Roy Flaker, Gregory Gravenites, Robert Lipa, Hsing-San Lee, Joseph Maslack, John Pessetto, William F. Pokorny, Michael Roberge, and Harold Zeller. An Experimental 1-Mbit CMOS SRAM with Configurable Organization and Operation. *IEEE Journal of Solid-State Circuits*, 23(5):1085 ff., October 1988.

[WBS+87] Karl Wang, Mark Bader, Vince Soorholtz, Richard Mauntel, Horacio Mendez, Peter Voss, and Roger Kung. A 21-ns 32K$\times$8 CMOS Static RAM with a Selectively Pumped p-Well Array. *IEEE Journal of Solid-State Circuits*, 22(5):704–712, October 1987.

[WC96] Ralph D. Wittig and Paul Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, April 1996. IEEE Computer Society, IEEE Computer Society Press. `http://www.eecg.toronto.edu/~wittig/thesis.description.html`.

[WDW+85] Frank Welten, Antoine Delaruelle, Frans Van Wyk, Jef Van Meerbergen, Josef Schmid, Klaus Rinner, Karel Van Eedewijk, and Jan Wittek. A 2-μm CMOS 10-MHz Microprogrammable Signal Processing Core with an On-Chip Multiport Memory Bank. *IEEE Journal of Solid-State Circuits*, 20(3):754–760, June 1985.

[WH95] Michael J. Wirthlin and Brad L. Hutchings. A Dynamic Instruction Set Computer. In Peter Athanas and Ken Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Los Alamitos, California, April 1995. IEEE Computer Society, IEEE Computer Society Press.

[WHG94] Michael J. Wirthlin, Brad L. Hutchings, and Kent L. Gilson. The Nano Processor: a Low Resource Reconfigurable Processor. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, Los Alamitos, California, April 1994. IEEE Computer Society, IEEE Computer Society Press.

[WHS+87] Tomohisa Wada, Toshihiko Hirose, Hirofumi Shinohara, Yuji Kawai, Kojiro Yuzuriha, Yoshio Kohno, and Shimpei Kayano. A 34-ns 1-Mbit CMOS SRAM Using Triple Polysilicon. *IEEE Journal of Solid-State Circuits*, 22(5):727–732, October 1987.

[WOI+89] Shigeyoshi Watanabe, Yukihito Oowaki, Yasuo Itoh, Koji Sakui, Kenji Numata, Tsuneaki Fuse, Takayuki Kobayashi, Kenji Tsuchida, Masahiko Chiba, Takahiko Hara, Masako Ohta, Fumio Horiguchi, Katsuhiko Hieda, Akihiro Nitayama, Takeshi Hamamoto, Kazunori Ohuchi, and Fujio Masuoka. An Experimental 16-Mbit CMOS DRAM Chip with a 100-MHz Serial READ/WRITE Mode. *IEEE Journal of Solid-State Circuits*, 24(3):763–770, June 1989.

[Xil89] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Gate Array Databook*, 1989.

[Xil91] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC5200 FPGA Preliminary Prodcut Specification*, version 4.0 edition, June 1991. `http://www.xilinx.com/partinfo/5200.pdf`.

[Xil94a] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book*, 1989, 1994.

[Xil94b] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book*, 1994.

[Xil96] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC6200 FPGA Advanced Product Specification*, version 1.0 edition, June 1996. `http://www.xilinx.com/partinfo/6200.pdf`.

[YFJ+87] Jeff Yetter, Mark Forsyth, William Jaffe, Darius Tanksalvala, and John Wheeler. A 15 MIPS 32b Microprocessor. In *1987 IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, pages 26–27. IEEE, February 1987.

[YJY+90] Toshiaki Yoshino, Rajeev Jain, Paul Yang, Harvey Davis, Wanda Gass, and Ashwin Shah. A 100-MHz 64-Tap FIR Digital Filter in 0.8μm BiCMOS Gate Array. *IEEE Journal of Solid-State Circuits*, 25(6):1494–1501, December 1990.

[YKF+94] Nobuyuki Yamashita, Tohru Kimura, Yoshihiro Fujita, Yoshiharu Aimoto, Takashi Manabe, Shin'ichiro Okazaki, Kazuyuki Nakamura, and Masakazu Yamashina. A 3.84 GIPs Integrated Memory Array Processor with 64 Processing Elements and a 2-Mb SRAM. *IEEE Journal of Solid-State Circuits*, 29(11):1336–1343, November 1994.

[YKK+84] Takashi Yamanaka, Shigeru Koshimaru, Osamu Kudoh, Yakashi Ozawa, Nobuyuoka, Hiroshiito, Hidehiro Asai, Nobuyuki Harashima, and Shinichi Kikuchi. A 25 ns 64K Static RAM. *IEEE Journal of Solid-State Circuits*, 19(5), October 1984.

[YKMI88] Toshio Yamada, Hisakazu Kotani, Junko Matsushima, and Michihiro Inoue. A 4-Mbit DRAM with 16-bit Concurrent ECC. *IEEE Journal of Solid-State Circuits*, 23(1), February 1988.

[YNH+91] Toshio Yamada, Yoshiro Nakata, Junko Hasegawa, Noriaki Amano, Akinori Shibayama, Masaru Sasago, Naoto Matsuo, Toshiki Yabu, Susumu Matsumoto, Shozo Okada, and Michihiro Inoue. A 64-Mb DRAM with Meshed Power Line. *IEEE Journal of Solid-State Circuits*, 26(11):1506–1510, November 1991.

[YR95] Alfred K. Yeung and Jan M. Rabaey. A 2.4 GOPS Data-Drivern Reconfigurable Multiprocessor IC for DSP. In *Proceedings of the 1995 IEEE International Solid-State Circuits Conference*, pages 108–109. IEEE, February 1995.

[YTN+85] Sho Yamamoto, Nobuyoshi Tanimura, Kouichi Nagasawa, Satoshi Meguro, Tokumasa Yasui, Osamu Minato, and Toshiaki Masuhara. A 256K CMOS SRAM with Variable Impedance Data-Line Loads. *IEEE Journal of Solid-State Circuits*, 20(5), October 1985.

[YYN+90] Kazuo Yano, Toshiaki Yamanaka, Takashi Nishida, Masayoshi Saito, Katsuhiro Shimohigashi, and Akihiro Shimizo. A 3.8-ns CMOS 16$\times$16-b Multiplier Using Complementary Pass-Transistor Logic. *IEEE Journal of Solid-State Circuits*, 25(2):388–395, August 1990.