Augmenting a Microprocessor with Reconfigurable Hardware

by

John Reid Hauser

B.S. (North Carolina State University) 1987
B.S. (North Carolina State University) 1987
M.S. (University of California, Berkeley) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor Randy H. Katz
Professor John Strain

Fall 2000

The dissertation of John Reid Hauser is approved:

Chair	Date
	Date
	Date

University of California, Berkeley

Fall 2000

Augmenting a Microprocessor with Reconfigurable Hardware

Abstract

Augmenting a Microprocessor with Reconfigurable Hardware

by

John Reid Hauser

Doctor of Philosophy in Computer Science
University of California, Berkeley

Professor John Wawrzynek, Chair

As VLSI technology continues to improve, configurable hardware devices such as PLDs are progressively replacing many specialized digital integrated circuits. Field-programmable gate arrays (FPGAs) are one class of such devices, characterized by their ability to be reconfigured as often as desired. Lately, FPGAs have advanced to the stage where they can host large computational circuits, giving rise to the study of *reconfigurable computing* as a potential alternative to traditional microprocessors. Most previous reconfigurable computers, however, have been ad hoc designs that are not fully compatible with existing general-purpose computing paradigms.

This thesis examines the problem of combining reconfigurable hardware with a conventional processor into a single-chip device that can serve as the core of a general-purpose computer. The impact of memory cache stalls, of multitasking context switches, and of virtual memory page faults on the design of the reconfigurable hardware is considered. A possible architecture for the device is defined in detail and its implementation in VLSI studied. With basic development tools and a full-fledged simulator, several benchmarks are tested on the proposed architecture and their performance compared favorably against an existing Sun UltraSPARC. Some additional experiences with the architecture are also related, followed by suggestions for future research.

Professor John Wawrzynek
Dissertation Committee Chair

To my recently departed mother, who very
much wanted to see me get my degree.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisor, John Wawrzynek, for his generous support and encouragement over the last several years. Without his unflagging confidence this project would never have been completed. Thanks also to the Berkeley Computer Science faculty for graciously extending me the time to bring this dissertation to fruition.

Thanks to my cohort on the Garp project, Tim Callahan, for his quiet friendship and for stoically listening to my various monologues and also my regular gripings about poor software. With his work on the Garp C compiler, Tim has helped make the project a success and increased its impact in the community.

Equal thanks to all my other officemates, notably Will Chang, Ngeci Bowman (the dancing Shmoo), William Tsu, Joe Gebis, and the two Greeks, Christoforos Kozyrakis and Stelios Perissakis, for many distracting discussions about religion, politics, language, culture, history, travel, photography, the stock market, intellectual property law, and sometimes even computer architecture and software. Before he left to join the faculty at MIT, Krste Asanović helped sharpen my thinking about computer architecture and other subjects. Thanks also to Dave Johnson and the other folks at ICSI and the Cal Hiking and Outdoor Society (CHAOS), to the people I played ultimate with, and to my friend Dave Blackston, for helping me keep life in perspective. Thanks to Jim Beck for his always enjoyable commentary about everything.

Graduate assistants Kathryn Crabtree and Peggy Lau deserve applause for helping all us graduate students negotiate the rules and paperwork accompanying every phase of our progress towards that coveted degree.

My graduate work has been funded in part by DARPA's Adaptive Computing Systems program, grant DABT63-96-C-0048, by ONR grant N00014-92-J-1617, by NSF grant CDA 94-01156, as well as by donations from Xilinx and Hewlett-Packard.

Last but not least, I would like to recognize the continued support of my family over these many years.

Chapter 1

Introduction

Throughout its history in the last fifty years, digital electronics technology has improved exponentially over time, doubling in performance roughly every 18 months while device sizes and costs have shrunk correspondingly. In line with this growth, the number of transistors available for constructing a commodity microprocessor currently doubles every two years or so. With such persistent evolution in computer components, computer architecture must be constantly reexamined and reinvented. Designs that were excellent only a decade ago may be hopelessly simplistic today, while techniques that were once prohibitively expensive have today become the norm. Intuition based on the current state of the art may not be a good judge of what the future will bring.

In the last decade, out-of-order superscalar processors have developed as the standard for desktop microprocessors. It is widely believed, however, that we are fast reaching the limits of this paradigm. As superscalar issue width grows, its overhead increases quadratically, while at the same time, opportunities for exploiting more instruction-level parallelism seem to grow ever scarcer. As we approach the day when a single chip holds 100 million transistors—literally enough to pack 30 vintage Intel Pentiums onto one die—it seems doubtful that superscalar designs by themselves will make the most of the available potential.

One alternative being considered for the future is based on the technology of *field programmable gate arrays* (FPGAs). It should be obvious that every application would be best served by custom circuitry targeted specifically for it; and, in fact, application-specific integrated circuits (ASICs) are often made in response to special needs. But no one can afford to turn out a custom chip for every application he wants to run; and even when

they are feasible, state-of-the-art ASICs become more expensive every day. As technology has improved, a market has grown up instead for versatile off-the-shelf parts that can be programmed to emulate arbitrary digital circuits in place of ASICs. FPGAs are one class of such devices, distinguished by their ability to be reprogrammed (reconfigured) any number of times.

The versatility and reprogrammability of FPGAs comes at a price. Only a few years ago, the algorithms that could be implemented in a single FPGA chip were fairly small. In 1995, for example, the largest FPGAs could be programmed for circuits of about 15,000 logic gates at most. Since a fast 32-bit adder requires a couple hundred gates, the capabilities of such devices were somewhat bounded.

More recently, though, FPGAs have reached a size where it is possibile to implement reasonable subpieces of an application in a single FPGA part. This has led to a new concept for computing: if a processor were to include one or more FPGA-like devices, it could in theory support a specialized application-specific circuit for each program, or even for each stage of a program's execution. The unlimited reconfigurability of an FPGA permits a continuous sequence of custom circuits to be employed, each optimized for the task of the moment. Because FPGAs scale better than superscalar techniques, such designs have the potential to make better use of continuing advances in device electronics in the long term.

The idea of *reconfigurable computing* has been a subject of research for a decade, but most projects have investigated the potential of connecting one or more commercial FPGAs to an existing microprocessor via a standard external bus such as the PCI bus. If reconfigurable computing is really to become the computing paradigm of the future, the main parts must be brought closer together. Only a few studies have considered integrating a processor and FPGA into a single device, with the two tailored to cooperate closely with each other; and so there remain important questions about how such a device might be built and programmed, and how it would fit within an existing general-purpose computing framework. Such questions must be addressed before the bigger issue of whether reconfigurable computing is really a good model can be answered.

This thesis attempts to make progress on the question of whether reconfigurable computing will be a viable option for future general-purpose computers. The succeeding chapters contain the following:

- A look at the fundamentals of computing machines, followed by a summary of past work on reconfigurable computing and an articulation of the purpose of this research project.
- An exploration of numerous issues arising from the integration of reconfigurable hardware into a processor.
- The presentation of a plausible architecture for a reconfigurable-enhanced processor, and an examination of the feasibility of implementing it efficiently in VLSI.
- A summary of the development and simulation tools created for the proposed architecture, and then the quantitative results from a handful of benchmarks by which the design was evaluated.
- Lastly, a brief retrospective of weaknesses in the proposed architecture and lessons learned in the design.

The thesis closes with conclusions and some opinions about the direction of research in reconfigurable computing.

Chapter 2

Background and Motivation

This chapter first reviews the fundamentals of computation and considers why reconfigurable computing might be important in the future. It then lays out the focus of the research, and ends by summarizing related work.

2.1 The potential for reconfigurable computing

2.1.1 Computing devices

Any computation can be represented as a combination of abstract data-flow and control-flow graphs, with the nodes in the graphs being *primitive operations* such as integer addition or comparison. Figure 2.1 has an example of a data-flow graph for a short expression. The primary function of a computer is to evaluate such graphs mechanically so as to accomplish some goal. Of course, real computer processors do not operate on such abstract graphs directly; instead, programs are encoded as a collection of machine instructions which can be executed one after another in a specific sequence. But this is just an artifact of the design of the machine, intended originally to simplify the processor's task (and perhaps the programmer's, too). Modern processors, in fact, re-expose instruction level parallelism by dynamically decoding short sequences of machine instructions into their corresponding data- and control-flow forms before executing them. Regardless of how a program is physically encoded, data-flow and control-flow graphs represent the *true* computation being performed.

To evaluate the computational primitives in the graphs (addition, multiplication,

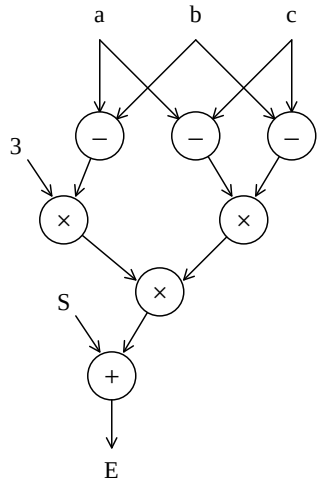


Figure 2.1: A data-flow graph for a simple expression.

etc.), processors include one or more *functional units*, each capable of performing a certain class of functions. A simple processor may have a single all-purpose functional unit known as an ALU (arithmetic and logic unit) that can only execute one operation at a time. More sophisticated processors (superscalar, VLIW) attempt to utilize multiple functional units of different kinds simultaneously to execute programs faster.

In concept, the functional units are all a computer needs to evaluate the operations in a data-flow graph. In practice, a computer must also support the physical movement of data among functional units, as well as to and from memory. Computers with multiple functional units clearly have to move data between them, and this not always as trivial as it might sound. But even for a simple processor with a single ALU, there is never more than a small amount of data that can be stored very close to the ALU at one time. A practical general-purpose computer must have a memory hierarchy, with the fastest and smallest memory (usually the registers) closest to the functional units, and increasingly slower but more capacious memory correspondingly farther away. Moving data between different levels of memory, either explicitly or implicitly, is thus another indispensible computer operation.

Figure 2.2 shows a simple diagram of a superscalar or VLIW processor with many functional units. For each clock cycle, an attempt is made to execute as many primitive operations as possible on the available functional units. A *forwarding crossbar* carries the output of all units back around to the functional unit inputs, so that the next set of operations can be executed in the next clock cycle with minimum delay. A multi-ported

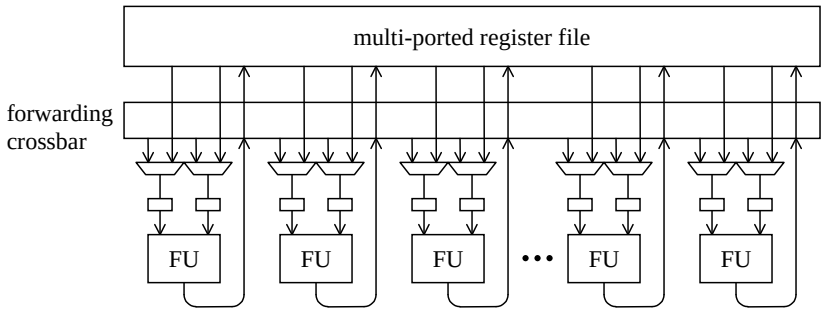


Figure 2.2: The core of an aggressive superscalar or VLIW processor. The boxes labeled *FU* are the functional units. The small rectangles are registers preceding the functional units; there may be other such registers not shown.

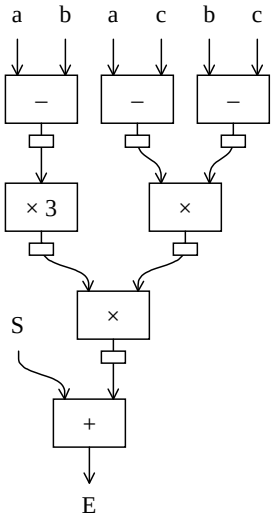


Figure 2.3: An application-specific pipeline for computing the graph in Figure 2.1.

register file forms the highest level of the memory hierarchy and can be used to store values over multiple cycles. Although the ideal is to use every functional unit every clock cycle, no processor can achieve this for all programs because of the data and control dependencies inherent in practical algorithms. The flexibility of the functional units, forwarding crossbar, and register file is what makes a standard processor *programmable*, capable of executing an arbitrary application with decent performance, even applications not thought of when the processor was built.

The antithesis of a programmable processor is an ASIC (application-specific integrated circuit). In an ASIC, functional units can be dedicated to individual program

operations and wired together to match precisely the calculation being performed. Figure 2.3 shows, for example, how an application-specific pipeline might be layed out to perform the calculation of the data-flow graph of Figure 2.1. The advantages an ASIC has over a programmable processor are threefold:

- Considerably less overhead is needed to control the mapping of functional units to operations and the routing of data values between them. On a programmable processor this overhead is manifest both in time and in die area.
- With smaller specialized functional units and less overhead circuitry around each one, more functional units can be fit into the same die area.
- Because the operation of each functional unit is known and planned out in advance, functional unit idleness can be minimized. On a programmable processor, certain kinds of functional units might never be used by a specific application.

Obviously, by their very definitions, a programmable device cannot be an ASIC and vice versa. However, as we shall see, *reconfigurable* devices such as FPGAs share characteristics of both processors and ASICs. On the one hand, FPGAs can implement ASIC-style circuits, while on the other, they are infinitely reprogrammable and thus immanently general-purpose. This leads to the question of whether reconfigurable hardware can capture some of the advantages of ASICs within a general-purpose computing environment.

2.1.2 Utilizing growing hardware resources

In the past, FPGAs were too modest to compete with relatively efficient superscalar processors, but this relationship may be changing. Electronic devices continue to grow smaller and faster every day. The benefits of faster transistors and wires are obvious; that they are also smaller means we get more of them for the same cost. Market forces provide a strong incentive to find a way to use these added resources to improve each new processor generation.

Most of the real work of a processor is done in the functional units, so that is an obvious place to focus. Additional transistors can be employed in the functional units in at least three ways:

- Individual functional units can be made faster by reengineering them for speed at the expense of die area. For example, a one-bit-at-a-time iterative multiplier could be

replaced by a much larger and faster array multiplier. There are physical limits to this approach, however. If the processor already has array multipliers, opportunities for improving multiplication further will be less dramatic.

- New functional units can be added for functions that previously required a sequence of other operations. This is exactly what happened, for example, when floating-point functional units were originally added to processors, and the same is happening again today with small-SIMD operations (MMX, VIS, etc.). By expanding the different kinds of functional units, however, the likelihood is increased that some will be unused by an application.

- More copies of existing kinds of functional units can be added. This is the easiest route to contemplate, but the hardest to make actually productive. Aside from the increased hardware complexity of having more functional units to juggle, new degrees of parallelism must be found in the software, or the new units will sit perpetually idle.

Despite the difficulties, increased parallelism is the only viable path once opportunities for the first two options have petered out. Modern superscalar and VLIW processors are already committed to 6- to 8-way instruction issue, and more is being considered.

Software contains roughly three classes of parallelism that can be exploited:

- *Thread parallelism* is between independent threads of execution, each executing a separate sequence of instructions. One example would be when a subroutine contains two or more separate loops that have no dependencies between them, in which case the loops could all be executed simultaneously.

- *Inter-iteration parallelism* exists when the iterations of a single loop are all mutually independent of one another and thus can be executed in parallel. This is also known as *data parallelism* or *vector parallelism*, being the kind of parallelism that vector processors profit by. A classic matrix multiply or fast Fourier transform has inter-iteration parallelism that grows with the sizes of the operand arrays.

- *Instruction-level parallelism* (ILP) exists among operations within a single thread of control, such as within a single loop iteration. The short expression in the example of Figure 2.1, for instance, has 3-way ILP between the three subtractions, allowing the three operations to be executed at the same time.

2.1.3 Limitations of superscalar and VLIW processors

In the 1990's, desktop processors adopted superscalar techniques to exploit ILP. By definition, a superscalar processor accepts a sequential instruction stream and discovers parallelism among the instructions dynamically and automatically. The basic model of Figure 2.2 is followed, with a forwarding crossbar and multi-ported register file. An extra-large register file and automatic register renaming are also commonly used to overcome false dependencies among the registers.

For all their popularity, superscalar machines are among the least efficient at exploiting parallelism. At 6-way issue, more effort is usually expended in testing instruction dependencies and controlling instruction issue than actually executing operations. This issue overhead grows quadratically with the number of functional units. Also under pressure to grow quadratically are the forwarding crossbar and register file (the register file because both its size in bits and number of ports must be increased). A 16-way superscalar processor would thus require tremendous overhead just to keep all its units busy.

VLIW (very long instruction word) processors eliminate much of the instruction issue overhead by relegating to the programmer or compiler the task of scheduling instruction execution to take maximum advantage of ILP. The work that a superscalar processor does on the fly to find ILP and avoid false dependencies (by register renaming) is done in advance before the program is run. Wide VLIW processors are also known to resist quadratic growth of the register file and crossbar, typically segmenting them such that the "crossbar" is not fully connected and not every register is accessible to every functional unit. These limitations make the VLIW processor more difficult to schedule for but also more efficient.

Nevertheless, the circuitry needed to issue new instructions to the functional units and reroute the crossbar every clock cycle remains a potential source of inefficiency even for VLIW processors. Furthermore, neither superscalar nor VLIW designs can take advantage of thread parallelism. Since the greatest sources of parallelism in programs are often inter-iteration and thread parallelism as opposed to simple ILP, this will eventually be exposed as a serious shortcoming.

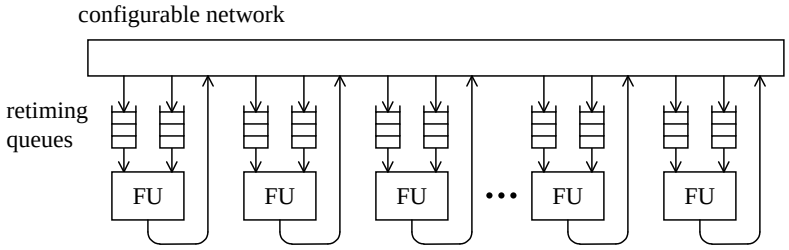


Figure 2.4: A model of a reconfigurable device analogous to Figure 2.2 for a traditional processor. An important aspect not visible in the figure is the fact that a reconfigurable device generally does not take a cycle-by-cycle instruction stream but instead is *reconfigured* between spurts of execution.

2.1.4 Reconfigurable computing as a new model

Reconfigurable computing is one alternative to the superscalar and VLIW paradigms. Figure 2.4 illustrates a reconfigurable device along the lines of the previous processor diagram. The main distinction between a reconfigurable device and a standard processor is in the instruction stream: in its purest form, a reconfigurable device has no cycle-by-cycle instruction stream. Rather, the device is *configured* by loading a complete specification of the function of each part of the device at once. Once configured, the intention is for the device to run in that configuration for a decent interval before being reconfigured. Each configuration mimics an ASIC-like circuit, like that of Figure 2.3, specialized for the particular task at hand. Changing configurations might take anywhere from a few clock cycles to a few thousand cycles. In accordance with the simpler programming mechanism, the dynamic forwarding crossbar is replaced by a less flexible *configurable network* for making static connections among the functional units; and short queues of *retiming registers* associated with each functional unit take the place of the traditional processor's shared, multi-ported register file.

The familiar 90-10 rule asserts that 90% of execution time is consumed by about 10% of a program's code, that 10% generally being inner loops. Reconfigurable devices excel in those cases where the computation represented by a configuration is repeated many times, so that the time required to load a configuration can be amortized over a long execution time and/or overlapped with other execution. When all of an application's important loop bodies can be configured to fit within the reconfigurable machine (one at a time), there would seem to be no need for the overhead of a fully dynamic instruction fetch and issue

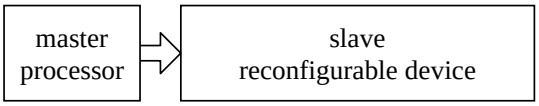


Figure 2.5: A more practical hybrid machine, combining a traditional processor with a reconfigurable device.

mechanism, allowing the machine to be leaner and more efficient.

By reducing the hardware to just the essentials needed to support computation, the reconfigurable design scales better to larger sizes than the more complex superscalar and VLIW styles. Although a naive expansion of the configurable network would cause it to grow quadratically with the number of functional units, it actually only needs to grow enough to support the connectivity required by real applications. Furthermore, unlike a superscalar or VLIW machine, reconfigurable hardware can easily exploit not only simple ILP but also inter-iteration and thread parallelism, making reconfigurable computing well-poised to work with very large numbers of functional units.

2.1.5 The hybrid machine

The corollary of the 90-10 rule is that 90% of a program's code accounts for only 10% of its execution time. In practice, reconfigurable devices get bogged down on the large parts of programs that are never executed with enough repetition to justify the time it takes to load a configuration for them. A practical compromise, therefore, is to couple a reconfigurable device with a traditional-style processor as in Figure 2.5, in order to exploit the strengths of each. With this organization, the reconfigurable hardware is used to execute the innermost loops (or *kernels*) of an application, while the modest traditional processor handles the mass of code between kernels.

In such a hybrid architecture, it makes sense to have the reconfigurable part be subservient to the traditional processor for at least two reasons: (1) the traditional processor executes the control code which logically ties together the various kernels the reconfigurable device will perform, and (2) execution on the traditional processor then becomes the default condition, allowing the new machine to fit in more easily with existing computing practice. The master processor can even be made instruction-set-compatible with an existing processor architecture to leverage existing development tools and operating systems.

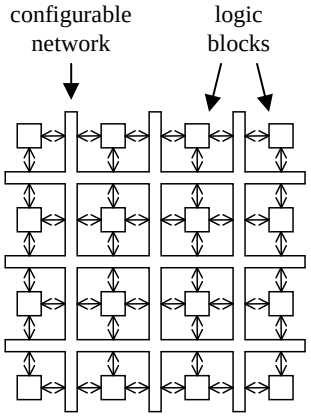


Figure 2.6: The reconfigurable structure from Figure 2.4 as it appears in a commercial FPGA.

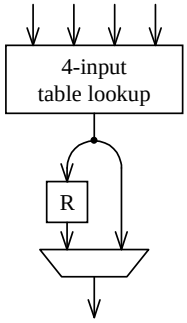


Figure 2.7: A canonical logic block function. Four single-bit inputs are taken from the configurable network and used to index into a 16-entry table of possible outputs. The box labeled R is a one-bit register in which the output can be optionally latched before being sent back over the configurable network to other logic blocks.

2.2 FPGAs

The most common reconfigurable devices today are FPGAs; these are independently packaged parts marketed both as prototyping platforms and as reconfigurable alternatives to ASICs. In a commercial FPGA, the basic structure from Figure 2.4 is reshaped into two dimensions to look more like Figure 2.6, with the functional units of the earlier diagram broken into a larger number of *logic blocks* that are individually rather small. The canonical logic block is often considered to be a lookup table that takes four bits of input and generates one bit of output, as shown in Figure 2.7. By filling in the table with the right bits, any four-input logic function can be realized. Various studies have suggested that four inputs is a good size for these lookup tables, trading off utility (how powerful the blocks are) against utilization (what fraction of their power ends up idle) [65, 66].

Logic blocks in actual FPGAs tend to be more complex than a single lookup table; Figure 2.8 has a similar diagram for a Xilinx 4000-series logic block, which has two four-input lookup tables and an extra three-input table, for a total of eleven bits of input and four bits of output [71, 81]. Dedicated carry chain circuitry at the top of the figure makes it easy to gang together a line of logic blocks to form a relatively fast multi-bit adder. This diagram in fact ignores many additional details, such as the way Xilinx's two 16-bit lookup tables can be used together as 32 bits of random access memory, or the options available

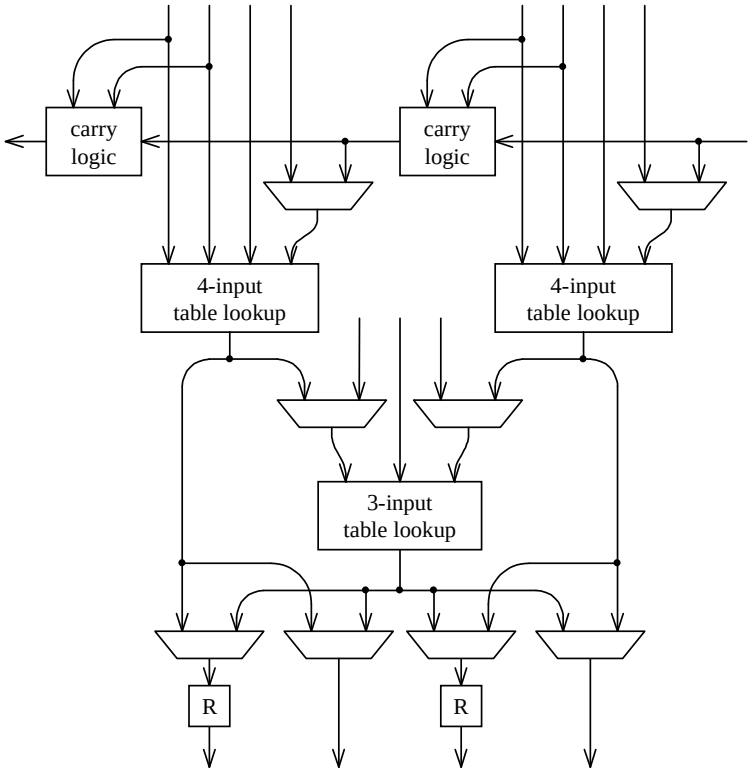


Figure 2.8: Simplified view of a Xilinx 4000-series logic block. The block has eleven input bits and four output bits, plus special carry chain hardware for basic arithmetic operations.

		number of 4-input lookup tables	bytes of on-chip data RAM
3.3 V	XC4036XLA	6,156	0
	XC4052XLA	9,196	0
	XC4085XLA	14,896	0
2.5 V	XC40150XV	24,624	0
	XC40250XV	40,204	0
2.5 V	XCV300	6,144	8,192
	XCV600	13,824	12,288
	XCV1000	24,576	16,384
1.8 V	XCV812E	18,816	143,360
	XCV1000E	24,576	49,152
	XCV2000E	38,400	81,920
	XCV3200E	64,896	106,496

Table 2.1: A sampling of FPGA chips available from Xilinx in 2000. All the devices have additional capabilities beyond just the lookup tables and data memory listed.

for controlling the clocking of the two single-bit registers.

Even a complex Xilinx logic block is quite small compared to the usual functional units of a computer. But in large numbers, small logic blocks can add up to considerable compute power. Table 2.1 lists some of the current offering of FPGAs from Xilinx, currently the most popular vendor (`xilinx.com`). The new XCV3200E, for example, has enough hardware to implement over *two thousand* 32-bit adders. Depending on how the device is configured, that equates to hundreds of 32-bit variable shifters or dozens of fully pipelined 32-bit multipliers. The die sizes of the largest parts are generally at the boundary of what can be manufactured, but this of course is not true of the smaller parts, and the future is expected to bring only greater densities.

2.3 Previous FPGA-based systems and their applications

In the past decade, many projects have been reported that use FPGAs to obtain speedups over traditional computers for particular applications. The literature includes quite a few examples where a collection of off-the-shelf FPGAs have been connected together to achieve performances hundreds to thousands of times faster than a workstation-type computer working on the same problem. Early work with Splash 2, for example, found that the FPGA-based machine could calculate genome (DNA) edit distances 2500 times faster than a contemporary SPARC 10, and could perform a median filter of a grey-scale image almost 140 times faster than the SPARC 10 [1, 9]. Such numbers have stirred interest in reconfigurable computing even as a general-purpose reconfigurable architecture has always seemed at least a few years out of reach.

It must be pointed out, however, that one Splash 2 board contains 17 FPGA parts, whereas a SPARC 10 is built around a microprocessor on a single chip. A fairer comparison would normalize performance to the number of compute chips, since a special board with 17 SPARC processors might also compute faster than a single SPARC 10 alone. When the speedups above are divided by the number of active FPGAs used, the per-chip speedup factors work out to be 147 for the genome edit distance and just under 10 for the grey-scale median filter. These numbers are obviously still interesting but put FPGA speedups into better perspective.

Some recent examples of FPGA applications are listed in Table 2.2. In general, the best speedups tend to be in the range of 10 to 30 times per chip, with only a few

application	reconfigurable machine	compared against	speedup per chip	year	reference
military target recognition	Splash-2 board, 16 Xilinx 4010's	110 MHz HP 770	7.5	1997	[64]
finding Golomb rulers	board with 20 Xilinx 5215's	167 MHz UltraSPARC	1.35	1998	[17]
10,000-bit multiply, divide, square root	1 Xilinx 4044XL	UltraSPARC	2–14	1998	[70]
frequency-domain sonar beamforming	1 Xilinx 4062XL	40 MHz ADI SHARC	8	1998	[24]
computing Goldbach partitions	1 Xilinx 40125	195 MHz MIPS R10000	>13 (1)	1998	[47]
rendering Bézier curves	PCI card with 1 Xilinx 6216	desktop PC	48 (2)	1998	[53]
genome matching, generalized profiles	VME card with 8 Xilinx 4013's	SPARC 20	7.9	1999	[59]
infrared military target recognition	PCI card with 16 Xilinx 4020's	180 MHz Pentium	1.24	1999	[40]

Notes: (1) Estimated based on experience with a PeRLe-1 board with 16 Xilinx 3090s.
(2) The FPGA operated on curves 14 times longer and thus with less overhead per rendered pixel.

Table 2.2: Some recent applications utilizing off-the-shelf FPGAs. Only FPGA chips actually performing computation have been counted in the speedup numbers.

exceptional applications doing even better. If the arguments of earlier sections are right, some of these numbers should improve as the technology advances. Memory and other support chips have not been counted in the comparisons, partly because they are not the subject of interest, but also partly because it is rare that enough information is provided to make it even possible. It will have to be assumed that the memory system is not the bottleneck in any of the examples.

Current research is indebted to early projects done at DEC's Paris Research Laboratory and at the Supercomputing Research Center in Maryland. The DECPeRLe-1 board—the third generation of DEC's Programmable Active Memories (PAM) project [74]—contained 24 Xilinx 3090 chips and connected to a host DEC workstation over the TURBOchannel bus [6]. Similarly, the Supercomputing Research Center's Splash 2 board sported 17 Xilinx 4010's and plugged into a SPARCstation SBus [1, 9]. Up to 16 of the Splash boards could be attached to a single SPARCstation (though apparently no one ever actually used that many). In the first half of the 1990s, these experimental boards were used to accelerate a collection of applications, including genome (DNA) pattern matching [9, 48], determining stereo vision from paired images [6], human fingerprint minutia matching [9], solving three-dimensional heat equations [6], and Hough transforms and Gaussian/Laplacian pyramid generation for images [1]. The PeRLe-1 board even assisted with particle detection at CERN's Large Hadron Collider [6]. The new reconfigurable machines could often solve large problems hundreds of times faster than the workstations of the time. However, they also needed dozens of FPGA chips to do so. A full-size board with over a dozen large processing chips will never be a competitive replacement for a single-chip microprocessor.

The FPGA parts available back then were too small to allow interesting problems to fit into just one chip. This situation has improved over time, so that beginning in the later 1990s various companies have marketed plug-in boards with only a few off-the-shelf FPGAs, sometimes as few as one. These boards typically plug into a computer expansion slot such as the PCI bus and can be used as generic accelerator cards for specially programmed applications. The most visible current vendors include Nallatech Limited (`www.nallatech.com`), Alpha Data Parallel Systems (`www.alphadata.co.uk`), Annapolis Micro Systems (`annapmicro.com`), and Virtual Computer Corporation (`vcc.com`).

Although these boards are useful, people have regularly complained about two shortcomings: (1) the slow reconfiguration times of the FPGAs, and (2) the overhead of shipping data back and forth over the connecting bus [2, 20, 39, 46, 52, 53, 69]. The

generally slow configuration time of commercial FPGAs is a reflection of their intended market. A Xilinx XCV1000E, for instance, though not one of the largest parts, still takes 12.5 ms to load a complete configuration. An older, slower XC4036XLA, with one-fourth the logic blocks, needs 10.5 ms to load a configuration. When an FPGA is being used as a flexible alternative to an ASIC, it might be configured only once when a system is booted, in which case configuration times of a dozen milliseconds are inconsequential. When acting as a computational accelerator for a 300 MHz processor, on the other hand, the same configuration overhead means each kernel the FPGA works on must execute for at least 15 million clock cycles to have any chance of achieving a speedup factor as much as 5. Reconfigurable hardware with less configuration overhead would have far greater applicability.

As for the overhead of moving data back and forth, Singh and Slous found for example that a Xilinx 6200 FPGA on a Virtual Computer Corporation board could perform a 3×3 FIR filter on an image 80 times faster than the PCI bus could transmit the original and filtered images onto and off the board [69]. While memory bottlenecks can be a problem no matter where a computation is performed, it is a safe bet that the main processor in a computer has more bandwidth to its DRAM than to devices on the other side of a PCI bus. If the reconfigurable hardware is going to execute program kernels faster than the main processor, it deserves to have equal or better bandwidth to the memory than the processor itself.

2.4 Focus of the research

To recap, a few truths can be stated:

- Reconfigurable devices may do well with small, highly repetitious kernels in applications, but standard processors are still superior for the remaining code that is irregular and/or rarely repeated.
- For reconfigurable computing to be economically viable for more than a small fraction of the market, the number of parts involved has to be squeezed down to at most one chip, and preferably less. Existing microprocessors are implemented in a single chip, and the pressure in the market is always for fewer parts, not more.
- Reconfigurable hardware will not meet its full potential unless reconfiguration time is

minimized and the reconfigurable hardware also has access to the computer's memory that is at least as good as that available to the main processor.

To better integrate a reconfigurable device into a computer, various researchers have called for combining a traditional microprocessor with an FPGA-like device onto one die to form a new kind of processor [4, 16, 29]. Although it used to be that it took more than one FPGA chip to do anything interesting, we have reached the stage where a very powerful reconfigurable device can be fit into only a partial die. As superscalar and VLIW designs fail to capitalize on increasing transistor counts, the one-chip hybrid machine provides a way to employ greater VLSI densities for something more than just larger caches. It remains uncertain, though, just how effective such a machine would be in practice as a general-purpose platform.

That is not to say it was ever hard to predict that a processor and an FPGA could share the same die in a commercial product. In the last year or so, Triscend (`www.triscend.com`), Chameleon Systems (`www.cmln.com`), and Altera (`www.altera.com`) have each announced or begun shipping such parts. But these products are intended for the embedded market, where an ad-hoc coupling between the FPGA, processor, and memory can be tolerated. The more challenging question is whether a hybrid architecture could be successful in a general-purpose market, such as that of desktop PCs now dominated by Intel processors. By definition, a general-purpose computer must be prepared to run software written by arbitrary third-party sources, for purposes unknown at the time the computer is created. Special features of general-purpose environments include:

- Preemptive multitasking: Multiple processes or threads of execution can be active at the same time, and a running process may be suspended at any time to give another process a chance to run for a while.

- Virtual memory: Memory accesses are translated between virtual and physical addresses; and moreover, at any given time some allocated virtual memory may reside not in physical RAM but instead on a much slower device such as a hard disk. A memory access may necessitate the suspension of the running process until the requested memory page can be retrieved from the hard disk.

- Interprocess protection: To better isolate faults and to ensure security between users, processes are prevented from performing actions that could interfere with each other.

- Binary compatibility: More-or-less the same user-level executable code should run on any one of a family of architecture implementations. Binary compatibility protects customers' investments in software when they upgrade to better machines.

Before the advantage of using reconfigurable hardware in a general-purpose processor can be measured, one first has to have an idea what the machine will look like. It is not enough just to know that the two parts are on the same die, since performance will surely be affected by the way the pieces are interconnected. Better access to memory was, after all, part of the point of moving the reconfigurable hardware onto the same die in the first place. And the issues listed above must be addressed. Making an FPGA amenable to preemptive multitasking, for instance, could have performance implications that the embedded market has not had to face.

This dissertation project has attempted to shed some light on the value of augmenting a traditional general-purpose processor with reconfigurable hardware, by first defining a prospective architecture and then assessing its performance on a sampling of applications. To this end, a number of design issues are presented in the next chapter relating to the performance of the reconfigurable part and its adaption to the general-purpose environment. An effort has also been made to verify that the proposed architecture can reasonably be implemented in VLSI. Finally, by running programs on a simulator, the new architecture has been compared against an ordinary superscalar processor to see what speedups might be achieved for typical applications.

To evaluate any new processor design, one would like to benchmark it rigorously against existing processors for a broad sampling of workloads. However, since there is not yet any commonly accepted framework for integrating reconfigurable hardware into a processor, this project has had to do a little bit of everything necessary to design a reasonable architecture and evaluate its performance. Considering that current processor design has benefited from decades of refinement by hundreds of researchers, it would be ambitious to expect this one project to generate absolutely definitive conclusions about the efficacy of reconfigurable computing. The goal instead has been to find trends and narrow the focus for further research.

2.5 Related work

Researchers have been discussing marrying reconfigurable hardware to a traditional processor for several years, and a number of prototypes and partial experiments have been tried with varying degrees of success.

2.5.1 Concept prototypes

Probably the earliest prototypes were Virginia Polytechnic Institute's PRISM machines [2, 4, 76]. The first, PRISM-I, consisted of a board with four Xilinx 3090's plugged into a host system based around a Motorola 68010. Although PRISM-I's reconfigurable hardware was essentially a multi-FPGA board like those discussed in Section 2.3, the focus with PRISM was on making the board a fully transparent extension of the host processor. (The name *PRISM* is an acronym for *Processor Reconfiguration through Instruction-Set Metamorphosis*.) A C compiler was created that, with some minimal assistance from the programmer, automatically picked out candidate subroutines and compiled them for the reconfigurable hardware instead of the processor. Compiled programs ran partly on the host processor and partly on the attached FPGA board.

The second prototype, PRISM-II, brought the host processor and FPGAs closer together, attaching an AMD Am29050 directly to three Xilinx 4010's in an irregular network. For both machines, the sizes of the FPGAs at the time limited the example kernels to very small functions such as Hamming distance, bit reversal, and finding the first 1 bit in a 32-bit word. For just these small kernels, speedups over the host processor ranged in factors from 7 to 86, or from about 2 to 25 per FPGA chip. Although encouraging, the small kernels could not be said to be representative of many real applications.

Another very similar project was conducted at the École Polytechnique Fédérale in Switzerland [37]. The Spyder machine extended a custom processor with three Xilinx 4010's acting as *reconfigurable execution units*. With Spyder, the programmer was responsible for dividing a program between the main processor and the reconfigurable units and programming each in a special subset of C++.

At Brigham Young University, the prototyping of a hybrid machine was taken one step further with the Dynamic Instruction Set Computer (DISC) [78, 79]. In both the original DISC and DISC-II, the entire combined processor—main processor and reconfigurable component together—were constructed within FPGA parts. The first DISC was made with

two National Semiconductor CLAy31's. A primitive main processor was squeezed into part of one CLAy31, with the majority of the same chip supplying the prototype reconfigurable component. The second CLAy31 FPGA served only to control the loading of configurations on the first one. With DISC-II, the main processor was moved onto a separate, third CLAy31 and made more powerful. Lacking a special compiler, DISC kernels had to be separated out by hand and programmed using the usual FPGA tools. DISC was an advance primarily in the handling of configuration loading. The reconfigurable component was treated as a small cache, with a "miss" in the cache resulting in an automatic stall while the needed configuration was loaded.

2.5.2 Reconfigurable functional units in a processor

After the early prototypes, a few researchers have looked at the possibility of integrating the reconfigurable hardware so closely that it is just another functional unit within the pipeline of the processor. Such projects include the PRISC design of Razdan at Harvard University [63], Chimaera at Northwestern University [29, 83], and ConCISe at Philips Research Laboratories [41]. In these designs, the reconfigurable hardware has no separate data state of its own; instead, like other functional units in a processor, data inputs are obtained from the register file and results stored back there. Without any internal registers, the reconfigurable functional units support only combinatorial circuits (no loops in the circuit).

For all three systems, a C compiler has been extended to find common sequences of instructions to implement on the reconfigurable units. Like DISC, the systems all load configurations in a cache-like manner. A program requests the configuration it wants, and if it is not already ready, execution stalls automatically while the requested configuration is loaded.

Unfortunately, the speedups demonstrated by these systems have been less than impressive, rarely more than a factor of 2. Fundamentally, this technique is limited by its ability to exploit parallelism, as will be explored later in Section 3.1.1.

2.5.3 Streaming reconfigurable hardware

In a different tack, other research has dedicated the reconfigurable hardware primarily for *streaming* applications. Streams present data in a sequential order, and stream

processing is generally done on a first-in-first-out basis. Examples of streaming applications include audio and video filtering, compression, decompression, encryption, and decryption. Streaming hardware can also be used on problems that can be decomposed into vector operations on unit-stride vectors (contiguous in memory), since such vectors can be treated as streams.

One stream-oriented architecture is the University of Washington's RaPiD [15, 18, 19]. The RaPiD reconfigurable unit is composed primarily of 16-bit functional units and registers connected by a configurable network, much in the original image of Figure 2.4. Although this part of the reconfigurable unit is not stream-specific, input and output to the reconfigurable unit is required to be in the form of streams to and from memory only. At least three streams are supported. There is no data connection specified between the main processor and the reconfigurable unit except through memory.

OneChip-98 is another stream-oriented architecture [38]. (The original OneChip had a different organization that was never fully fleshed out [80].) Only a single input stream and a single output stream are supported, and there are other restrictions on stream length and alignment that make the OneChip-98 design less than compelling.

Many streaming applications can be decomposed into an assembly-line sequence of smaller operations that also act on streams. This characteristic opens the door to a way of *virtualizing* the reconfigurable hardware, whereby a streaming application can be run on a range of reconfigurable units of varying sizes, with execution speed proportional to the amount of reconfigurable hardware available. Keeping the assembly-line analogy, it is easy to understand how an assembly line with fifty workers ought to be able to turn out the same product as one with a hundred; the team of fifty would simply take twice as long to generate the same quantity of output.

The most interesting work along these lines has been Carnegie Mellon University's PipeRench reconfigurable unit [10, 22, 23, 46, 58]. A compiler for PipeRench breaks up a streaming application into as long a sequence of component stream operations as it can, and this is then how the configuration is represented within a program. When it comes time to run the program, if a particular PipeRench implementation is not large enough to contain the entire sequence at once, individual pieces of the reconfigurable hardware are regularly recycled to do all the different parts of the long sequence automatically. The design of the PipeRench hardware, however, does not permit it to work on problems that cannot be adequately represented in terms of streams in this way.

2.5.4 Novel reconfigurable hardware for computation

The design of commercial FPGAs is generally acknowledged to have a historic bias toward "random" control logic—often called "glue logic"—rather than toward the core arithmetic operations such as addition and multiplication found in much computation. When reconfigurable hardware is adopted into a general-purpose processor, it becomes appropriate to consider alternative designs. Some of the projects already mentioned have invented new reconfigurable hardware for their processors. Other studies have focused exclusively on the reconfigurable array without specifying exactly how it is connected to memory or the main processor.

One direction taken has been to re-optimize an FPGA's logic blocks for bit-serial arithmetic. This approach is represented by NEC's experimental Sea Of Processors (SOP) [82], and by a reconfigurable array developed at the Tokyo Institute of Technology [60].

A more common notion is to replace the lookup-table-based logic blocks of an FPGA with larger ALUs capable of 4-bit, 8-bit, or larger operations. The CHESS reconfigurable array described by Marshall et al. has 4-bit logic blocks [54], whereas 8-bit functions have been favored for PipeRench [22, 23] and for the MATRIX array designed at the Massachussetts Institute of Technology [56]. The reconfigurable hardware of RaPiD is divided into two parallel tracts, one with 16-bit functional units for data and a separate control tract with FPGA-style lookup tables [15, 18, 19].

Many other ideas and structures have been proposed, too many to list here. As one example, Haynes and Cheung describe a novel array with 4-bit blocks that can be combined directly to form multiply-accumulate operators [35]. It should also be noted that RaPiD, CHESS, and MATRIX break with reconfigurable tradition by allowing their functional units to receive instructions as well as data over the configurable network. Certainly there is no consensus yet as to the best structure for a reconfigurable accelerator, other than a shared theme that existing FPGA hardware may not be the best choice for doing the type of computation associated with application software.

Chapter 3

Design Issues

General-purpose computers have distinctive features and limitations, including support for multitasking, virtual memory, and structured programming. Multitasking implies the ability to perform context switches. Virtual memory requires that memory addresses be translated, and memory accesses can cause page faults that must be serviced transparently by the system. Structured programming assumes that functional implementation can be hidden from the clients of functions, even to the extent of separate compilation of function and client. Each of these aspects must be considered in designing reconfigurable hardware for augmenting a general-purpose processor.

This chapter covers the gamut of issues, starting first with high-level processor-reconfigurable integration, and then later moving to low-level details of the reconfigurable unit itself.

3.1 Integrating reconfigurable hardware into a computer

By bringing the reconfigurable hardware closer to the processor, the two can be made to interact more tightly. The main advantage is the proximity of the reconfigurable array to the processor's memory system, eliminating the need to copy data back and forth laboriously over an external I/O bus. The closer connection to memory and the improved coordination with the processor should permit the reconfigurable array to be employed for a larger number of smaller tasks. To achieve the greatest cooperation, many options for the specifics of the coupling need to be explored.

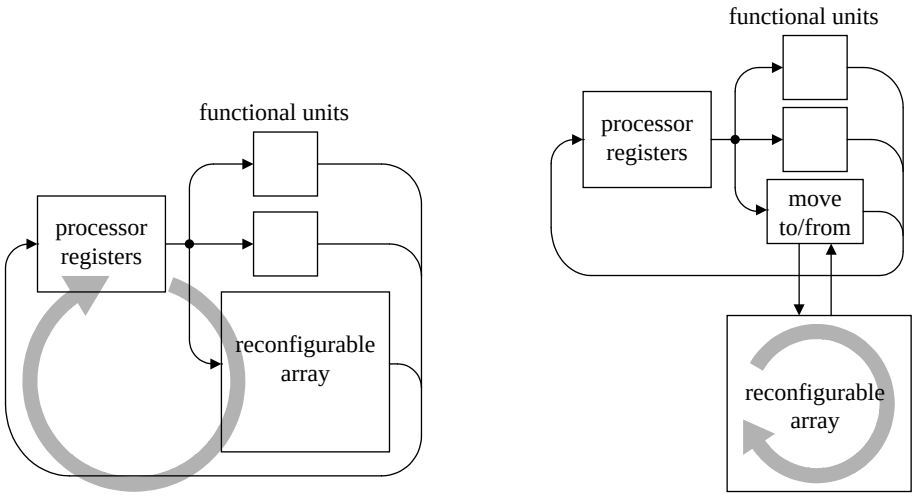


Figure 3.1: (a) Reconfigurable array as another functional unit in the main processor pipeline, using the existing register file. (b) Separating out the reconfigurable unit from the processor pipeline by a coprocessor-style register-transfer interface. Internal array state permits execution to proceed within the array independently of the main pipeline.

3.1.1 Level of integration

Given the freedom to integrate reconfigurable hardware with the processor, some designers have gone all the way and added a reconfigurable array as another functional unit in the processor pipeline. As noted in Section 2.5.2, examples in the literature include PRISC [63], Chimaera [29, 83], and ConCISe [41]. With such a design, array inputs are naturally taken from processor registers and results written back to the register file, just as for other pipeline functional units (Figure 3.1(a)). By adding one or more processor instructions of the form `rd = reconfigop(ra,rb)`, the reconfigurable array can provide unique, tailor-made operations for each program. But this seemingly straightforward approach harbors some pitfalls.

First we face the question of whether the array can hold internal data state, or whether it must be stateless like other functional units and dependent on the main register file for storing data. If the array contains no internal data registers, it will not be able to execute independently for an extended length of time, because there is only so much useful combinatorial calculation that can be done on a few words read from the register file to generate at most a couple words to write back. The consequence can be seen in the speedups

reported for the previous examples. Razdan's PRISC achieves speedups averaging only about 12% on most of the SPECint92 benchmarks, the lone exception being the `eqntott` benchmark which saw a speedup of almost a factor of 2 [63]. Chimaera speedups have also been less than a factor of 2 with a few exceptions, mostly benifiting from short 32-bit SIMD techniques [29, 83]. And ConCISe speedups for DES encryption and the A5 stream cipher have been 44% and 40% respectively [41]. In contrast, based on experience with PRISM-II, Agarwal et al. argued that the reconfigurable unit must be able to execute for more than a few cycles to maximize its advantage [2].

The parallelism that can be achieved when the reconfigurable unit has no data state is strictly bottlenecked by the number of read and write ports in the register file, preventing the array from realizing tremendous speedups over a traditional processor (Figure 3.1(a)). The situation can be improved with the addition of many more ports into the register file; however, the better, more direct solution is to allow the array to contain internal state, analogous to but separate from the traditional register file. This state should presumably be spread throughout the array, so that the array can keep a large number of operations running simultaneously for an extended period of time, which is the only way impressive speedups will be obtained. One complication is that it will be necessary to suspend array execution midway if a context switch occurs while the array is computing. This trouble is essentially unavoidable, and is addressed later in Section 3.1.7.

Given that a computation on the array can take indefinite time, instructions of the form `rd = reconfigop(ra,rb)` cause a problem for the processor pipeline. Assume for the moment that the results of such instructions are interlocked, so that if a following instruction takes as input a register written by a reconfigurable array instruction, the later instruction will stall until array execution has completed. Otherwise, in the absence of an interlock stall, subsequent processor instructions continue to execute in parallel with the reconfigurable array in the usual manner of a processor pipeline. Difficulties arise if a context switch occurs while an array instruction is in progress:

- Precise interrupts are impractical, even with a mechanism for backing up processor state such as a reorder buffer, because it would be necessary to restore the state to a point before the `reconfigop` instruction began. In addition to backing out of the array's work, the effects of all instructions completed while the array was executing would have to be undone. Even if we wanted to take the risk of discarding potentially

thousands of cycles of forward progress, the state that would need to be saved is prohibitive. (The saved state would necessarily cover all memory accesses during that time.)

- Ordinarily, the context switch software will save the registers with a simple sequence of register store instructions. If a register value is pending from an ongoing array computation, the store of that register will interlock, blocking the context switch indefinitely.

- The information about which if any registers are pending from an array instruction must be saved as part of the context switch state and restored when the context is resumed if register interlocks are to work properly when the process is resumed.

As the designers admit, it is partly in recognition of these difficulties that the reconfigurable units in PRISC and ConCISe are limited to a few clock cycles of execution [41, 63]. But throttling the power of the reconfigurable hardware is not the answer.

Some of these concerns would be moot if there was no interlock on the result of reconfigurable array instructions. But this option causes more problems than it solves. Without an automatic interlock, instructions must be statically scheduled by the compiler (or a human programmer). Static instruction scheduling over the hundreds or thousands of clock cycles the array might execute would surely be tricky to get right. But even worse is the fact that, to make static scheduling even possible, the exact instructions the processor will execute each cycle must be strictly determinable from the architecture definition, which means it cannot be improved by later implementations. If the architecture specifies that integer multiplication takes 16 clock cycles, that number cannot be reduced without rescheduling all software for the new implementation; otherwise, instruction execution might get out of sync with the execution of a function on the reconfigurable array.

It is possible to have full interlocking and still find a way to do context switches with a functional unit that takes indefinite time to complete. But it is easier simply to discard the `rd = reconfigop(ra,rb)` style of instruction which causes the trouble in the first place. A long-executing functional unit has no need to be in the processor pipeline; it can be separated out as a "coprocessor," with distinct instructions used to transfer data between it and the main register file (Figure 3.1(b)). The previous instruction form is then split into multiple ones that: (1) move argument data to the coprocessor, (2) start

coprocessor execution, and (3) move the result back. The overhead of executing the extra instructions to transfer data in and out will be small if the reconfigurable hardware actually executes for more than a few cycles at a time.

To avoid static scheduling, the instruction to copy data back from the coprocessor must be interlocked on the completion of the computation there. But this interlock is not problematic because it is explicitly tied to the completion of the array computation, whereas in the earlier case the connection was *implicit* through the use of a register. Section 3.1.7 covers what is required for context switches in the coprocessor model.

3.1.2 Programming paradigm

To reap the greatest speedups, a reconfigurable functional unit must execute for more than a few clock cycles at a stretch; yet the amount of static program code that can be implemented in the reconfigurable array at one time is usually rather small. Hence, of all the code in a program, the best candidates for implementing on the array are inner loops with small static code size but potentially large dynamic instruction counts. If enough loops can be transferred to the array that together account for a majority of program execution time, a signficant speedup can be achieved for an entire program.

The simplest programming model assumes that when program execution reaches an inner loop, the appropriate configuration is loaded onto the array, input data is copied to the array, and array execution is started. When the loop is done, results can be copied out and instruction execution resumed on the main processor, at least until the next loop. Because most interesting loops repeatedly access memory (through pointers or arrays), some means must also be provided for getting memory data in and out during loop execution; this will be discussed in Section 3.1.6.

In an experiment by Jantsch et al., eight real programs were found to contain 2523 inner loops that were potential candidates for acceleration [39]. However, most such candidates do not execute for very long, risking that the time to load a configuration for them will exceed the time saved from using the reconfigurable hardware.

To hide the time it takes to load configurations, it is often proposed that a reconfigurable unit be able to preload the next configuration needed while the current one is in use [28, 38]. One obstacle to doing this on a general-purpose computer is the way in which structured programming decomposes programs into independent subroutines whose

implementations are supposed to be hidden from one another and may in fact be separately compiled. Without reworking this paradigm, it will often not be known that a configuration needs to be loaded until the subroutine that needs it has been called, and by then it may be too late to overlap the load with much other computation. Section 3.1.6 explains why there also may not be excess bandwidth to memory for preloading a configuration while the array is executing.

Instead of preloading, other techniques can be used to combat configuration loading overheads: (1) configurations can be encoded more densely; (2) the bandwidth to memory for loading configurations can be widened; and (3) configurations can be cached in the array for reuse.

3.1.3 Configuration encoding and loading

Part of the reason it takes a Xilinx XCV1000E 12.5 ms to load a full configuration (Section 2.3) is the configuration's size of over 800 kB—more than 256 bits for each four-input lookup table. Existing FPGAs have often used a relatively decoded representation for configurations. To take a simple case, most logic block inputs can be connected to a choice of nearby wires in the network, only one of which can be selected for the input. If there are n wires to choose from, a typical FPGA might encode this choice with n bits, most of which will be 0's indicating no connection.[1] A more compact encoding would use $\log n$ bits to choose a wire to connect to a logic block's input. It will be seen later (Section 4.1.3) that the encoded form in this case is not always more costly for the array hardware.

Besides a dense encoding, outright compression can be employed, since there is often obvious redundancy among the logic blocks in a configuration. A 16-bit-wide arithmetic operation, for instance, might use 16 contiguous logic blocks, all configured exactly the same. Such redundancy could be compressed out with a simple run-length encoding, for example. Configuration compression has been studied by Hauck and his students, as well as others [31, 32, 50, 61].

Although compression can reduce the bandwidth needed to read a configuration from memory, maintaining a compressed form *within* the array adds hardware cost without much benefit. Certainly the array must have sufficient storage space for a fully uncompressed configuration, since legitimate configurations need not have any redundancy; every logic

[1] Due to the way FPGA networks work, a connection might be made between more than one wire and the logic block input. In general, though, most potential connections will not be selected.

block might be configured differently. Adding support for run-length encoding in the array would require extra wires and multiplexors to distribute configuration bits when they could simply be stored redundantly where they are needed. To avoid this cost, a compressed configuration must be decompressed as it is read into the array. If the bottleneck for loading configurations is not at the memory system but instead at the wires into the array, then run-length compression may be of little value.

With configuration sizes running several kilobytes at a minimum, it makes no sense to load configurations through a first-level (L1) cache that is probably smaller than the typical configuration. Configuration loading should therefore bypass the L1 data and instruction caches. With synchronous DRAM, the second-level (L2) cache could be bypassed as well, because bandwidth, not latency, is the critical factor when loading kilobytes at a time. However, not all memory systems are set up to support bypassing the L2 cache. If configurations can be kept down to a few kilobytes, there is less reason to be concerned about polluting an L2 cache that is probably a half megabyte or more.

A typical 500 MHz general-purpose computer could easily sustain 8 GB/s of bandwidth from its L2 cache. In comparison, commercial FPGAs often load configurations through as little as a 1-bit-wide shift chain, which (optimistically) at 500 MHz could load 63 MB/s. Newer FPGAs have wider configuration paths; the Xilinx XCV1000E, for instance, loads 8 configuration bits at a time. Nothing really prevents a custom reconfigurable unit from having a configuration interface equal in width to the L2 cache data bus in order to match the available memory bandwidth. L2 cache buses are commonly 128 to 256 bits wide.

With many FPGAs there exist invalid *parasitic* configurations that can destroy the device. In the configurable network between logic blocks, wires are often physically connected to more than one tri-state driver output; a valid configuration must select just one driver for each wire at any one time. If every tri-state driver is controlled by its own configuration bit, it is easy for multiple drivers to be selected accidentally for the same wire, causing the FPGA to burn out.

Parasitic configurations cannot be tolerated on a general-purpose computer, where any application might load a configuration of random bits. A few FPGAs such as the Xilinx 6200 have encoded their configurations in a way that guarantees no possible configuration is parasitic. While this sounds attractive, it has a hardware cost. The 6200 replaced simple network bus wires with safer but more expensive multiplexors, perhaps resulting in an

increase in FPGA size of 20 to 30%.

An alternative solution is to verify a configuration's validity as it is being loaded. The processor could simply refuse to load a configuration that fails to pass the test. This approach should require less die space than the multiplexor-based solution because a smaller amount of checking hardware can be used repeatedly (time-multiplexed) as each piece of the configuration is streamed into the reconfigurable array. (This technique will be promoted in Section 4.3.1.)

In addition to loading configurations in whole, some FPGAs allow the array's configuration to be modified through *editing* operations. Again on the Xilinx 6200, for example, individual logic block configurations could be updated without the need to load a complete configuration for the entire FPGA. Although not a problem on the 6200, configuration editing is dangerous when parasitic configurations are possible. Since there would seem to be no way to revalidate an edited configuration without streaming it all past the checking hardware again, editing must be considered incompatible with the technique of validating configurations during loading.

The only time an array configuration needs to be written back to memory is on a context switch. Even then, if the current configuration was never edited, and if its copy in memory (from which it was originally loaded) is still intact, the write can be skipped; the configuration can simply be reloaded from the original when the process resumes. If array configurations *never* needed to be written to memory, context switches could avoid the write and thus be a little faster. To ensure that configurations never need to be written out, the architecture only needs to forgo support for configuration editing and also require that configurations not be modified in memory while loaded in the array.

It is not clear how valuable configuration editing might be in practice. Assuming configurations are used for inner loops in a program, there are two circumstances under which editing might be appropriate:

- If configurations for two independent loops share a common structure, the second one could be "loaded" by merely editing the first one.
- When an inner loop is nested inside another loop, the inner configuration will be called upon again and again. This configuration might benefit from small edits as the value of a variable changes in the outer loop.

The second of these seems more compelling. However, when weighed against the hardware

expense of ensuring that there are no parasitic configurations and the time expense of writing configurations out to memory on context switches, support for configuration editing seems dispensable.

3.1.4 Caching of configurations

To save loading time when multiple configurations are regularly reused, the array can include a cache of recently loaded configurations. A cache is especially valuable for the expected programming model, in which a configuration might not be loaded until just before it is needed. When a program requests to load a configuration that is already in the cache, the "load" should be able to activate the cached configuration in only a few clock cycles.

Like other caches, a configuration cache should be dynamically managed in hardware, in part to simplify programming, but also so that the cache's size can vary with implementation without programs having to be explicitly adjusted to correspond. A dynamically managed cache can often be more efficient, such as when two multitasking processes each have a working set of configurations that fills half the cache size. A cache that was statically managed by each program would be more difficult to share.

The only reasonable place to store the cache is to distribute it among the array logic blocks. Assuming the reconfigurable array is roughly square in size with area A, its perimeter grows only as $\sqrt{A}$. If the configuration cache were kept outside this perimeter, the available bandwidth per array block would be proportional to $1/\sqrt{A}$, decreasing as the array grows. Already, FPGAs do not have enough wires across their periphery to load a configuration that way in only a few clock cycles. There is simply not enough internal bandwidth in an FPGA to move that many bits very far that quickly.

A configuration cache distributed within a reconfigurable array can be divided into equal *planes*, where the number of planes is the number of full configurations the cache can hold (Figure 3.2). As not all useful configurations will need the entire array, to achieve better cache utilization it must be possible to keep multiple less-than-full-size configurations in each plane. But it would be pointless to store a configuration at a particular location in a cache plane if the configuration could not also be *executed* at that location in the array; otherwise, the problem of quickly moving a configuration a large distance within the array is resurrected. Consequently, the array must be able to load and execute small configurations

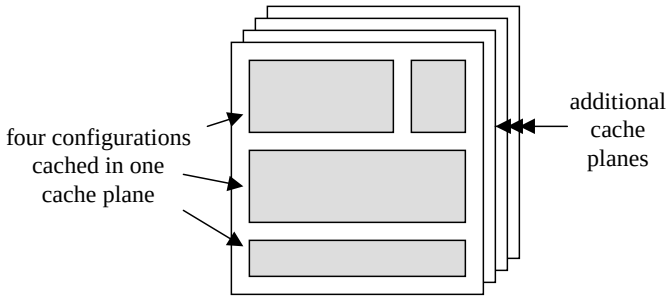


Figure 3.2: A configuration cache with multiple cache planes. Each plane can store one full-size configuration or multiple smaller configurations.

at various different positions so that a greater number of configurations can be kept in the cache.

The exact positions where smaller configurations can be loaded into the array will depend on the array design. The "perfect" array would let a configuration be arbitrarily shifted to any position. In reality, the wire network between logic blocks often does not look the same with respect to every logic block. Some FPGAs, for example, divide logic blocks into groups of 4×4, with a richer interconnect available within a group than between groups. On such an array, a configuration can only be shifted by multiples 4 blocks (at best) without all logic block interconnections being thoroughly rerouted.

If the cache is managed automatically in hardware, the position in which a small configuration gets placed in the array ought to be invisible to the program that loads it. The "first" logic block of a configuration should always appear as logic block 0 to the program, regardless of where the configuration is actually placed. The hardware should thus translate between *logical* block addresses and *physical* block addresses, just as it translates between logical and physical memory addresses for virtual memory.

To save a little on die area, cached configurations might be stored using dynamic memory. Certainly, a cache access time on the order of 10 clock cycles would be acceptable if configurations really execute for substantially longer on average. Assuming there are only a few cache planes, the dynamic memory circuitry need not be very complex. Of course, a dynamic memory cache would require regular refreshing, even while a configuration is running. Moreover, if parasitic configurations are a possibility, it is important to remember that dynamic storage cells are susceptible to upsets. Some redundancy would be necessary to guard against catastrophes. If the probability of errors is small enough, it should be

sufficient to abort the current process in the event such an error is detected, the goal being solely to protect the hardware against an unlikely disaster.

3.1.5 Array clocking

Commercial FPGAs allow each configuration to have a different clock frequency, or even multiple clocks at different frequencies. It is a development task (either for the tools or for a human designer) to ensure that no signal path exceeds its maximum allowed delay. Often the clock frequency is simply set to the highest value that works for a particular configuration.

In practice, the relationships between the different FPGA components' delay times varies with each FPGA implementation, making it hard to predict in advance the speeds at which two versions from the same FPGA family will execute the same configuration. This is a problem for upward compatibility of software on a general-purpose computer, where all configurations originally created on one version of a processor must be runable at a known clock speed on a newer version of the processor without having to be retuned for the new part. Moreover, allowing the reconfigurable unit to have a flexible clock speed would necessitate complex *synchronizers* between the reconfigurable array and the rest of the system, introducing additional latency and a risk of metastability.

Rather than specify component delays as precise times that could change with each processor generation, delays can be defined in terms of the sequences that can be fit within some fixed *array clock cycle* determined by the implementation. Such rules can be set once-and-for-all by the architecture, with implementations responsible for ensuring that configurations that follow the rules run properly. For example, the architecture could specify that a signal can propagate a certain distance between logic blocks within one array clock cycle. An implementation is responsible for setting the array clock speed slow enough to guarantee this rule. The array clock need not be identical to the processor clock, but any actual implementation would surely use a simple ratio between the two.

One disadvantage to a fixed clock is a potential loss of efficiency due to the forced quantitization of time. If every path between registers in a specific configuration happens to be short enough to propagate in less than a full clock period, the configuration could run faster with a faster clock. On the other hand, this type of loss can already be suffered in the main processor. The processor pipeline must assign some integer number of clock cycles to

the execution of each functional unit, even though a functional unit's actual delay might be 0.7 or 1.1 cycles. A program that performs many bitwise logical operations, for example, might accumulate a large proportion of "unproductive" time from the fact that a logical operation does not require the full clock period allocated to it. Some efficiency is usually worth sacrificing to simplify the architecture and thus permit a range of implementations of varying performance and cost.

Aside from simplifying the overall architecture, a fixed clock enables implementation techniques that would be impractical otherwise. Precharged circuits are one example: with a known fixed clock period, the array hardware could implement a precharged carry chain that would likely be faster than other carry propagation circuits. Such techniques might even compensate for the inefficiency caused by a nonadjustable clock.

Unlike a superscalar processor, which executes instructions at a variable and often unspecified rate, the architecture of a reconfigurable array must include a precise conception of time as measured in array clock cycles. To perform a certain computation, a configuration must execute for a specific number of clock cycles; any more or less might not perform the exact computation desired. To make it more interesting, the number of clock cycles needed will not always be known in advance, because, in particular, the number of iterations through a loop can be data-dependent. To handle such cases, the array needs a way to signal that computation is complete and array execution should be halted. If the processor is waiting (interlocked) on array completion, this will also indicate that processing can continue in the main processor pipeline. Freezing the array when it is inactive can be accomplished in hardware by *gating* the array clock, thus forcing array registers to hold their last values indefinitely.

Besides disabling the array when it is inactive, it may be necessary at times to stall array execution for a few cycles while active. If the configuration cache uses dynamic storage, for example, cache refreshes may require regular suspension of array execution. The next section also discusses stalling the array to hide the latency of memory accesses made from the array. In both these cases, array execution must be delayed due to irregular or hidden events. Rather than add complexity to every configuration to handle such occurences, the array can be stalled transparently by the hardware by briefly gating the clock for the necessary number of cycles. Array clock cycles in this way become *logical* execution increments not necessarily connected to real time.

3.1.6 External interface and access to memory

The main reason for bringing the reconfigurable unit closer to the processor is to bring it closer to the processor's data, and that especially means the memory system where most of the data is stored. If the reconfigurable array is in the main pipeline, it makes sense to use the existing processor load and store instructions to transfer data between memory and the main register file, from which values can be forwarded to the reconfigurable unit the same as for any functional unit. However, Section 3.1.1 argued that putting the array in the pipeline and depending on the main register file for storage will limit the speedups obtainable by the array.

Even with the array attached via a coprocessor interface, we could continue to let the main processor perform memory accesses, copying to and from the array for each access. This might be practical when the sequence of memory accesses is easily predictable. In other cases, though, it would be necessary to communicate information from the array to the main processor for each access, resulting in the following steps for a memory load: (1) copy address (or other information) from the array to the main register file, (2) load from memory to a register, and (3) copy the loaded value to the array. Using this scheme, the memory bandwidth available to the array might be half that of the main processor, with double or more latency.

If the reconfigurable unit is actually going to compute faster than the main processor it will need *more* bandwidth to memory, not less. A good connection to memory is also needed for loading configurations quickly. Assuming the array already contains paths for copying the contents of array registers back and forth to the main processor, essentially all that remains is to allow those paths to be diverted onto the bus over which configurations are loaded from memory (Figure 3.3). The main complication is supplying an address and the necessary control signals to the memory hierarchy.

Inside the array, physical wires might be needed for four distinct purposes:

1. interconnecting logic blocks,
2. bringing configurations in from memory,
3. transfering data to and from the main processor, and
4. transfering data to and from the memory system.

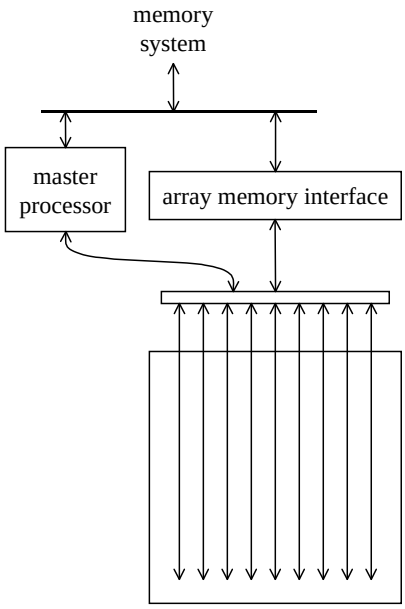


Figure 3.3: Bus through the array for loading configurations and for moving data to and from memory and the main processor register file.

An array design does not have to contain distinct wires for each of these purposes, however. Commercial FPGAs typically support only the first two cases, with separate sets of physical wires for each. Because the interconnect between logic blocks is configured by loading a configuration from memory, it probably does not make sense to rely on the former to do the latter, so there is a legitimate claim for keeping cases 1 and 2 distinct. On the other hand, when it comes to communicating with the processor or memory, either of the first two networks might do the job of moving values into and out of registers within the array.

If the array includes a configuration cache, it was argued earlier that configurations less than full size need to be loadable at multiple different locations in the array in order to make better use of the cache. If a *configuration bus* is run across the entire array, configurations can be loaded quickly to any array location (Figure 3.3). This bus would naturally be connected to the full bandwidth of the memory system at one end. It would be logical, therefore, to use the same bus to support memory accesses from the array while the array is executing. By sharing the connection to memory, this arrangement might preclude the possibility of preloading the next configuration while the current one is running. However, memory bandwidth is a limited resource; if an executing configuration can saturate the available bandwidth, none would be left over for preloading anyway, so the

sharing of a bus to memory would not really be the bottleneck.

Since it would be convenient to give the main processor random access to all the register state in the array as well, values could be transfered to and from the processor's register file over the same bus, at least while the array is not active. All told, this would make the proposed bus the main external connection for the array, with direct random access to any logic block. The configurable wire network between logic blocks would remain a separate entity.

When a configuration requests to load a value from memory, we can expect there to be a few cycles of latency before the value can be returned. There is no point in forcing the array to sit idle during that time; however, if a configuration is to continue executing in the meantime, it will need to know the array clock cycle at which the memory value will be returned. Preferably, the memory latency can be known at the time the configuration is created, so extra control circuitry does not have to be included for synchronizing the configuration's calculations to a variable memory latency. But in truth, the memory latency can vary from one implementation to another, and in fact can vary depending on whether the requested value is in one of the memory caches. Memory loads can even complete out of order on some systems.

These complications can be avoided with a simple compromise: The configuration can specify the *exact* latency it expects for all memory loads, and the array's interface to memory can be responsible for ensuring that those expectations are met. When a memory load requires more time than the configuration indicates (for example, because of a cache miss), the array clock can be automatically stalled to let the memory system catch up. Conversely, if the memory returns data more quickly than expected, the memory interface must hold onto it for the intervening cycles. With this technique, the array can initiate a new memory access every array clock cycle, with the results returned in a predictable pipelined fashion with respect to array execution.

3.1.7 Multitasking

To support multitasking, it must be possible to freeze and swap out execution on the reconfigurable unit for a context switch. The first step of any context switch is a processor interrupt or trap. This can proceed as normal, except for one special case: At the time an interrupt occurs, the processor could be interlocked awaiting completion of

execution on the reconfigurable array. Assuming the array is outside the main processor pipeline as proposed in Section 3.1.1, the interlocking instruction must be an attempt to read a result from the array. But in this case, the interlock can simply be ignored and the interrupt taken at the point in the instruction stream just before the interlocking instruction. When the process resumes, the same instruction to read from the array will be re-executed and the interlock reinvoked.

Once the interrupt or trap has been taken, the operating system's context-switch software must suspend and save the current array state and also restore and resume that of the process being swapped in. A new instruction will be needed to permit the processor to force a halt to array execution, although this might employ essentially the same mechanism the array uses to halt itself when done. If configurations cannot be edited in the array (Section 3.1.3), there is no need to write out the current configuration. However, the memory address from which the current configuration was originally loaded (and thus the address of the copy presumably still in memory) *will* have to be saved, and for this reason the hardware has to keep a record of the memory address from which the current configuration was loaded.

If the array can access memory directly and can operate in pipeline with memory loads (as proposed in the previous section), then halting the array leaves in-progress loads suspended in limbo. Memory operations can continue to completion from the perspective of the memory system, but values that have been loaded from memory and not yet accepted by the array must be saved and restored. This will require additional support in the architecture, most likely involving numerous special registers not accessible from ordinary "user mode" programs.

Lastly, if the main processor has random access to all array data registers, array data state can be saved and restored using ordinary processor instructions. The complete sequence for swapping out array execution to memory is then the following:

1. Force a halt of array execution.

2. Save the state of memory loads still in the pipeline, using special architectural support provided for this step.

3. For each array register, read the register value into the main register file and store it to memory.

4. Save the memory address of the current configuration.

Restoring a previously saved context is mostly the reverse process. One interesting twist concerns signal propagation after the array's register state has been restored. Unless explicitly outlawed, the propagation path between two registers can be made arbitrarily long in a configuration; this is legitimate so long as the configuration always holds the values of source registers constant for the full time it takes signals to propagate through to the destination registers. When a context is swapped out, the array might be halted at the moment just before a long-propagating signal was ready to be latched at its destination register. In that case, when the same context is later swapped back in, array execution will not be ready to proceed until signal propagation has had a chance to retraverse the path to the destination register. The operating system must ensure that this occurs before array execution is resumed.

Rather than try to ascertain the exact time needed for any configuration, it is easier simply to put a time limit on the propagation between registers in a valid configuration. A reasonable limit might be as short as 8 array clock cycles, which would certainly not be a burden to wait in a context switch. Actually, such a short time could easily be overlapped with other context switch activities such as restoring the main register file.

The complete sequence for swapping array execution back in is:

1. Load the configuration from memory (possibly still in the configuration cache).
2. Restore all array registers by reading from memory and copying to the array.
3. Restore the state of outstanding memory loads, using special architectural support provided for this step.
4. Allow signals time to propagate in the array. (The architecture or programming conventions place a limit on how long this is required to be.)
5. Resume array execution.

3.1.8 Servicing page misses

As is often the case, the most troublesome context switches are page misses. Unlike typical RISC instructions, if an array-initiated memory access causes a page miss, array execution cannot be backed up to reinitiate the access after the relevant memory page has

been brought in from disk by the operating system. Thus, failed memory accesses can only be completed with the participation of the hardware.

Assume that the failed access is a store, causing a trap that initiates a context switch. Since array execution cannot be moved backward, the array will not reinitiate the store when it resumes; the operating system will have to complete the store itself after the missing page has been brought in from disk. To do this, the operating system needs only the failed address and the data to be written, both of which could be made accessible through additional special registers existing in the interface between the array and memory. (The array memory interface cannot simply remember this information and automatically reattempt the access when the process is resumed because with multiple processes running there might be any number of page misses in the process of being serviced at any one time.)

If a page miss occurs on a load access, the operating system can complete the load, but the data loaded must be inserted into the pipeline of outstanding loads in the array memory interface. It was already suggested in the previous section that this state must be made visible to the operating system for context switches, so it could be that no more hardware is needed to support completion of failed loads.

A distinction needs to be made between a memory request that fails because of a nonresident virtual memory page and one that fails because of an invalid address. To support deep pipelining of operations on a stream of data it is important to be able to load data in advance of operating on it. Before processing has completed on the last of the input stream, additional speculative loads past the end of the input stream will often be performed, and these could be at invalid addresses.

To prevent programs from being terminated because of speculative loads to invalid addresses, it is possible simply to ignore invalid virtual address exceptions and return arbitrary data. Ignoring these exceptions can make debugging more difficult but cannot cause a correct program to fail. In contrast, speculative loads at virtual addresses that *are* valid but not currently resident must definitely cause a trap to the operating system so that it can service the page fault.

3.2 Designing reconfigurable hardware for computation

Almost any FPGA design could be adapted for the reconfigurable unit. However, since commercial FPGAs are generally not intended to be processor functional units, they

may not be best suited for that purpose. This section considers how reconfigurable hardware might be better tailored for accelerating software loops.

3.2.1 Dominance of wires

One issue with a major bearing on array design is the extent to which FPGAs are normally dominated by their configurable networks, both in terms of circuit area and delay. In 1993, Rose et al. asserted that "the area for routing is usually larger than the active area ... representing from 70 to 90% of the total area" [66]. And the significance of the network only increases as FPGAs grow. Six years later, Betz and Rose have reconfirmed that "the delay of a circuit implemented in an FPGA is mostly due to routing delays, rather than logic block delays, and most of an FPGA's area is devoted to programmable routing" [7].

This situation favors the inclusion of special-case circuitry to reduce the number of logic blocks needed for common tasks and/or to provide faster short-cut connections between logic block, even at the expense of making the logic blocks themselves somewhat larger and slower. The prototypical special-case circuitry is the carry chain support found in most commercial FPGAs. While it is obvious how such special paths bypass the slow general network, there are secondary effects that are also beneficial:

- With direct connections among neighbors, logic blocks require fewer physical connections to the network, which allows the general network to be more efficient. (Fewer network taps could mean less loading on the wires, for example.)

- The general network will have fewer signals to carry, again allowing it to be smaller and more efficient. (Because fewer wires are needed, the network taps can be smaller and faster because there are fewer wires to choose from.)

The benefits compound when special logic block circuitry reduces the number of blocks needed to perform a common operation:

- With fewer logic blocks in the paths between circuit inputs and outputs, fewer network traversals are needed, thereby reducing total function delay.

- By fitting configured circuits into a smaller area of the reconfigurable hardware, each network traversal between two blocks may have less distance to travel and thus may be individually faster.

- Conversely, with denser packing of configurations, more functionality can be configured into the same die area.

However, as with anything, more is not always better when it comes to logic block complexity. If special-purpose circuitry does not get used—presumably because the functionality is rarely needed—it only eats up space and acts as a drag on performance. The special features added to a logic block must be chosen judiciously. For reconfigurable hardware intended for computation, the standard arithmetic operations would seem to be a good place to start.

3.2.2 Bit-serial, bit-parallel, and bit-pipelined arithmetic

The basic arithmetic operations can be performed in various ways, corresponding to different engineering tradeoffs. A major distinction between the different techniques is the way in which numeric values are represented and delivered from one operator to the next. If we restrict ourselves to the usual binary representation, the primary categories are *bit-serial*, *bit-parallel*, and *bit-pipelined*. Each style can lead to different optimization structures in the reconfigurable array, so it is worth attempting to assess the relative advantages of each. (For more along the lines of the following discussion, see Kollig and Al-Hashimi [44].)

Bit-serial corresponds to the way addition is taught in grade school: The least significant digits are added first; then the next pair of digits are added along with any carry out from the previous place; then the next, etc., until all digit positions have been processed in turn. Implemented in hardware (Figure 3.4(a)), a bit-serial adder is little more than a bit adder with the carry looped back around. The two operands are each fed into the adder over a single wire, one bit (binary digit) per clock cycle, starting with the least significant bit first. The sum is generated at the same rate. For n-bit operands, it obviously takes n clock cycles to form the complete sum. On the other hand, if the adder output is used as an input to another adder, the second addition does not need to wait the full n cycles for the first to complete but can begin as soon as the least significant sum bit from the first addition has been computed. By mostly overlapping the operations in this way, two back-to-back additions can be completed in $n + 1$ clock cycles, only one cycle more than a single addition. This overlap effect extends to any number of additions, and the same applies to subtractions, comparisons, and—with more hardware—multiplications. For this reason, bit-serial arithmetic has often been used to obtain a minimal-area implementation of functions

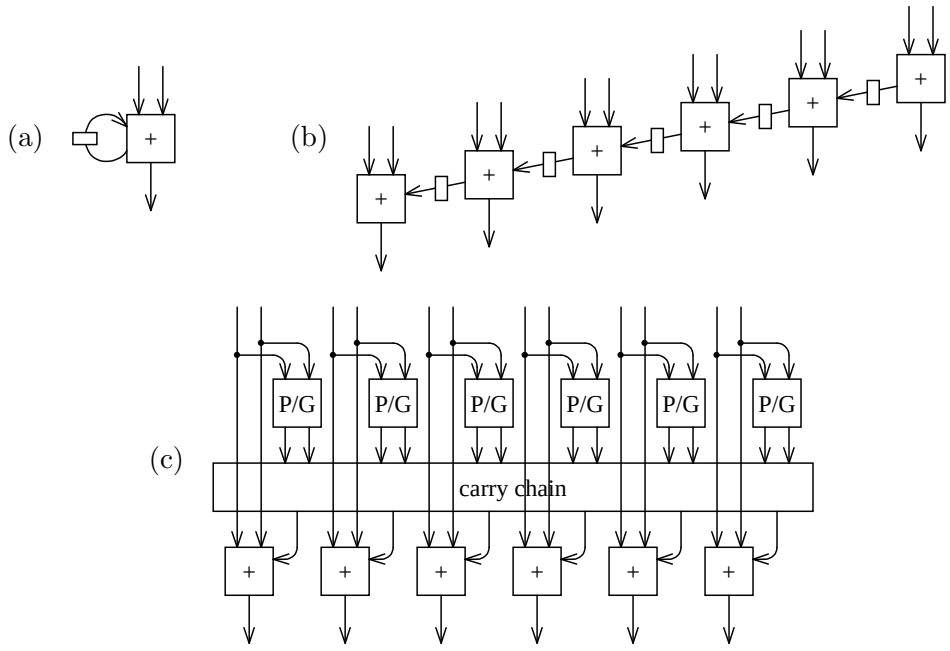


Figure 3.4: (a) Bit-serial addition. (b) Bit-pipelined addition. (c) Bit-parallel addition. The boxes labeled *P/G* calculate the *propagate* and *generate* signals for the carry chain.

such as digital FIR filters that require only addition, subtraction, and multiplication.

Even with maximum overlap, bit-serial arithmetic can only output one n-bit final result every n clock cycles. A bit-pipelined adder (Figure 3.4(b)) is similar in concept to a bit-serial adder, but instead of making a lone bit adder add each bit in turn, n individual bit adders are arranged in an assembly line to form a full n-bit adder. An addition operation occurs much as it did before, except that now each bit position is handled by its own dedicated bit adder. Now as soon as the first (rightmost) bit adder has summed the two least significant bits, it is free to start another addition. Consequently, n additions can be in progress at any one time, giving an overall rate of one n-bit addition per clock cycle. Aside from the n-factor increase in hardware and throughput, bit-pipelined arithmetic shares the same characteristics as bit-serial arithmetic.

Standard processors perform a full 32- or 64-bit addition in a single processor clock cycle. If the intermediate carry registers were removed from Figure 3.4(b) and the clock made n times slower, the result would be a ripple adder that could complete in a single (but much longer) clock cycle. Such an adder could be called bit-parallel because all the bits of an input or output value are latched together on a single clock edge. However, ripple adders are not the best; modern bit-parallel adders employ a more sophisticated technique

illustrated in Figure 3.4(c). The critical part of a bit-parallel adder is the carry propagation, which can be factored out as a separate, highly-optimized entity. At each bit position, either the carry out is known immediately from the two operand bits at that position, or the carry out must be the same as the carry in from the position one to the right. Each bit position calculates *propagate* and *generate* signals that determine the carry out as follows:

propagate	generate	carry out
0	0	0
0	1	1
1	–	same as carry in

The job of the carry chain unit is to propagate carry values as quickly as possible across all positions for which *propagate* = 1. For a 32-bit addition, this can be done much faster than a ripple adder, albeit using considerably more hardware. Overall, a bit-parallel adder operates with less total latency for each addition but also with less potential throughput in terms of additions per unit time than a bit-pipelined adder.

The bit adders for bit-serial and bit-pipelined addition are simple enough to be easily implemented by the lookup tables of typical FPGAs. The main issue is the number of logic blocks needed for each bit adder. If each logic block has exactly one lookup table and one output to the wire network as in Figure 2.7, then it will take two logic blocks to make each bit adder: one to calculate the bit sum, and the other to determine the carry out to the next bit position. Squeezing this down to a single logic block requires that a block be able to output two independent bit values. The obvious way to do this is by providing each block with two drivers onto the general wire network. However, addition occurs often enough that, for reasons explained in the last section, it is usually more efficient to build in a dedicated carry connection between neighboring blocks along every row or column of the array. Figure 3.5 shows two possible forms this carry connection could take.

Most commercial FPGAs include support for bit-parallel adders, although not necessarily anything better than good ripple adders. Typically, hardware for calculating and propagating a carry from one logic block to another is specified in the FPGA architecture, but the physical FPGA might implement this in alternate ways. (Note Xilinx's carry chain in Figure 2.8, for example.) This leaves open the possibility that a sophisticated carry chain unit could be sereptitiously employed in some existing FPGA chips. Whether this has actually been done or not, the existence of good bit-parallel carry units in FPGAs is not outside the realm of possibility.

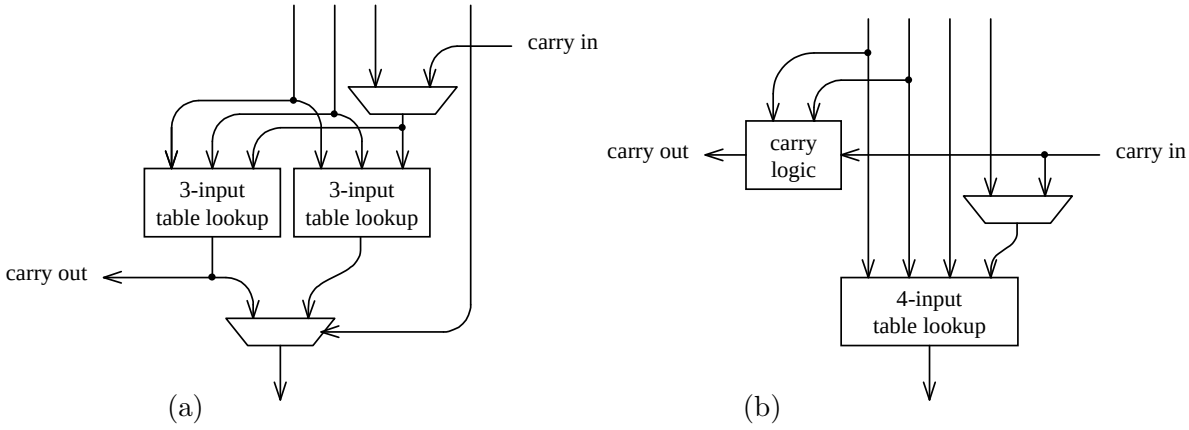


Figure 3.5: (a) Simple extension to a basic four-input-lookup-table-based logic block to implement a dedicated carry connection between neighboring blocks. When configured as a bit adder, one table calculates the sum bit and the other the carry out. To form a four-input lookup table instead, the two three-input tables are multiplexed down by the fourth logic block input. (b) Support for bit adders in the Xilinx 4000-series logic blocks. The main lookup table determines the sum bit, while dedicated circuitry generates the carry out appropriate for an addition or subtraction. This arrangement allows for a faster ripple adder than the construction on the left. (As seen in Figure 2.8, two such bit adders fit in each Xilinx logic block.)

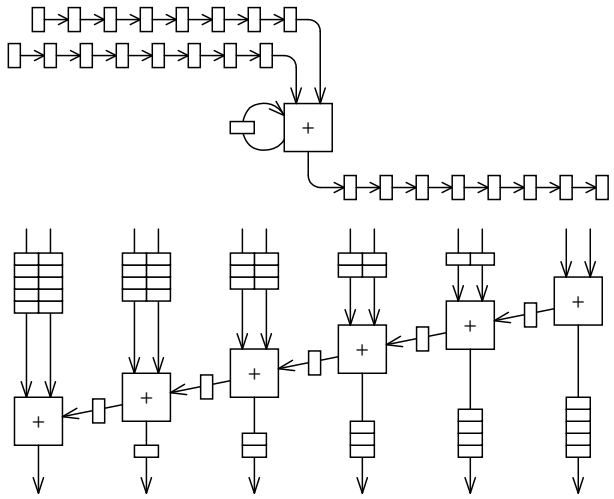


Figure 3.6: Skew conversion from bit-parallel to serial or pipelined form and back again. Skew conversion is required for all initial inputs and final outputs of a computation.

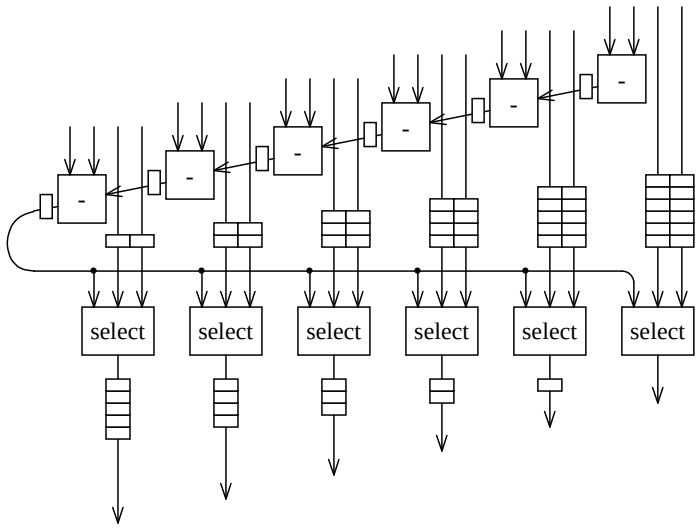


Figure 3.7: A comparison followed by a multiplexor, implementing the C expression `(` a `<` b `) ?` c `:` d. Skew conversions are needed for c, d, and the result, totalling roughly $\frac{3}{2}n^2$ bit registers for bit width n.

With appropriate assistance, all three forms—bit-serial, bit-pipelined, and bit-parallel—have a common statistic: n logic blocks can perform an average of one n-bit addition every clock cycle. The clock speed for bit-serial and bit-pipelined can be significantly faster than that of bit-parallel, however, causing bit-parallel to lag behind, at least according to this metric. Even so, bit-pipelined is not optimal in all circumstances, due to the following effects:

- Interaction with the rest of the system ordinarily must be done in bit-parallel form, necessitating *skew conversions* at function inputs and outputs (Figure 3.6). On a memory access, for example, all bits of the address must be delivered simultaneously, and the data must be read or written the same way. The same is true when values are transfered to or from the main processor. A circuit with a lot of such interaction will require constant skewing and deskewing of inputs and outputs, nullifying any throughput advantage bit-pipelined has over bit-parallel.

- Any feedback from high-order bits to low-order bits introduces a delay equal in clock cycles to the bit-width. Figure 3.7 shows the effect for the simple C expression

  `(` a `<` b `) ?` c `:` d.

  In this case, the entire comparison must complete before any bit of the selection can

proceed. A similar thing occurs for shift operations, depending on the shift distance. As with skew conversion, numerous bit registers are needed. The figure shows that for data values 6 bits wide, 57 bits of registers are needed. When the data is 32 bits wide, a total of 1616 bits of registers are needed, or the equivalent of fifty 32-bit registers. Moreover, if any of these operations are part of a feedback loop, the latency of the entire loop is considerably reduced, making bit-parallel techniques actually faster.

Tables 3.1 through 3.3 attempt to analyze these differences in more detail. The first table presents rough formulas for area, latency, and throughput of the three styles for a simple addition and for the comparison/multiplexor example. *Bit latency* is the time before a subsequent operation can start work on the result of a previous operation; while *turnaround* refers to the time before the same hardware can begin operating on the next set of values coming down the pipeline. Clock frequency differences are accounted for in the formulas. The area cost of skew conversion is also tabulated in the bit-serial and bit-parallel styles.

Table 3.2 simplifies these formulas for the following conditions and assumptions:

- The operation bit width is 32. (Although data values may often be of smaller size, any application that manipulates pointers or addresses will need to operate on 32-bit or even 64-bit quantities.)
- The wire network consumes two-thirds of the area of the reconfigurable array.
- Traversing the network between nearby logic blocks takes the same time as performing a simple lookup-table operation.
- Traversing the network a distance of 32 logic blocks takes three times as long as for nearby blocks.
- The area needed in each block to implement a decent bit-parallel carry propagation unit is double that of the rest of the lookup-table-based logic block.
- Using such a carry unit, a 32-bit bit-parallel addition takes three times as long to execute as a simple table lookup.

Under these assumptions, Table 3.2 makes it easier to judge the relative merits of each form. Some differences between bit-pipelined and bit-parallel arithmetic worth noting:

bit-serial

	area	bit latency	turnaround
addition	$W + L + R$	$V + K$	$n(V + K)$
comparison + mux	$2W + 2L + (2n + 2)R$	$(n + 1)(V + K)$	$n(V + K)$
skew conversion	$(n - 1)R$		

bit-pipelined

	area	bit latency	turnaround
addition	$nW + nL + nR$	$V + K$	$V + K$
comparison + mux	$2nW + 2nL + \left(\frac{3}{2}n^2 + \frac{5}{2}n\right) R$	$(n + 1)(V' + K)$	$V' + K$
skew conversion	$\frac{1}{2}(n^2 - n)R$		

bit-parallel

	area	bit latency	turnaround
addition	$nW + nL' + nR$	$V + K'$	$V + K'$
comparison + mux	$2nW + 2nL' + (3n + 1)R$	$2(V' + K')$	$V' + K'$

n : data size in bits
W: area of wire network corresponding to a single logic block
L : area of a simple logic block function configurable as a bit adder or multiplexor
L' : area of a logic block including a carry chain supporting bit-parallel addition
R : area of an externally accessible clocked register
V : time to traverse the network between close logic blocks
V': time to traverse the network between logic blocks n bits apart
K : time to perform a table-lookup logic block function
K': time to perform an n-bit parallel addition

Table 3.1: Approximate formulas for area, latency, and turnaround for the different arithmetic styles. All operands are assumed to become available at the same time.

bit-serial

	area	bit latency	turnaround
addition	$A + R$	T	$32\,T$
comparison + mux	$2\,A + 66\,R$	$33\,T$	$32\,T$
skew conversion	$31\,R$		

bit-pipelined

	area	bit latency	turnaround
addition	$32\,A + 32\,R$	T	T
comparison + mux	$64\,A + 1616\,R$	$66\,T$	$2\,T$
skew conversion	$496\,R$		

bit-parallel

	area	bit latency	turnaround
addition	$53\frac{1}{3}\,A + 32\,R$	$2\,T$	$2\,T$
comparison + mux	$106\frac{2}{3}\,A + 97\,R$	$6\,T$	$3\,T$

Table 3.2: Same as Table 3.1 with $W + L = A$, $V + K = T$, and the following additional assumptions: $W = \frac{2}{3}A$, $L = \frac{1}{3}A$, $V = \frac{1}{2}T$, $K = \frac{1}{2}T$, $n = 32$, $L' = 3L$, $V' = 3V$, and $K' = 3K$.

		number of retiming registers in each logic block					
		1	2	4	8	16	32
bit-serial	addition	1.025	1.05	1.1	1.2	1.4	1.8
	comparison + mux	67.7	35.7	19.8	12.0	8.4	7.2
	skew conversion	31.8	16.8	8.8	4.8	2.8	1.8
bit-pipelined	addition	32.8	33.6	35.2	38.4	44.8	57.6
	comparison + mux	1656.4	873.6	492.8	307.2	224.0	201.6
	skew conversion	508.4	268.8	149.6	91.2	64.4	55.8
bit-parallel	addition	54.7	56.0	58.7	64.0	74.7	96.0
	comparison + mux	218.7	168.0	117.3	128.0	149.3	192.0

Table 3.3: Relative total area under the same assumptions as the previous table and also supposing that each configurable retiming register (R) takes 2.5% as much area as the rest of the logic block ($W + L$). An integral number of logic blocks is used in every case, with the size of the logic blocks varying depending on the number of retiming registers they contain.

- Bit-parallel addition has more latency and less throughput (slower turnaround) than bit-pipelined, but the difference is masked some by the time it takes to traverse the wire network between logic blocks.
- Bit-parallel forms suffer less variability in the latency of common operations. Although a bit-pipelined addition is twice as fast as a bit-parallel one, the bit-pipelined comparison-and-multipexor is an order of magnitude slower that its bit-parallel counterpart.
- As stated before, a large number of register bits are needed for bit-parallel skew conversion and for situations where high-order bits effect the outcome of low-order bits.

To get a better handle on the area costs of each scheme, Table 3.3 gives the relative total areas for the same operations under the assumption that adding a one-bit retiming register to each logic block adds 2.5% to the die area of the array. Both the per-logic-block area and the number of logic blocks needed by each operation vary with the number of retiming registers included in each logic block. As more retiming registers are added, the per-logic-block area increases but the number of logic blocks needed may decrease. Furthermore, with more registers in each logic block, the risk that registers will go unused also increases. This complex relationship plays out differently for different operations, as shown in the table. With bit-pipelined arithmetic, the optimal number of registers can be seen to be highly sensitive to the proportion of additions, multiplexors, and skew conversions in each application, yet the choice cannot be varied once committed in the physical hardware. Once again, the bit-parallel approach appears more robust overall despite being less efficient for addition alone.

With so many accumulated assumptions and with numerous second- and third-order effects not accounted for, it would be wrong to put too much stock in the exact figures in these tables. The hope is simply that there is enough sense in the numbers to steer design decisions in a productive direction. One issue not reflected in the tables, for instance, is the greater power cost of distributing a faster clock throughout the array. Since bit-pipelined and bit-serial techniques require a faster clock to have any chance of beating bit-parallel, clock distribution power is another factor that might tip the balance toward bit-parallel forms.

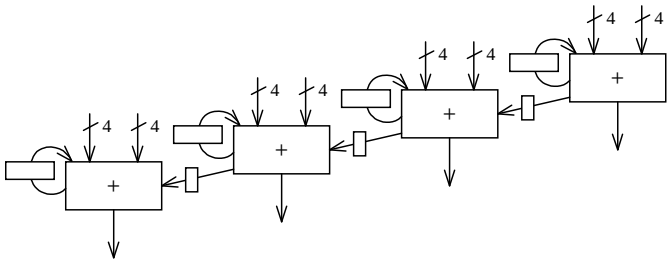


Figure 3.8: A mixture of parallel, pipelined, and serial techniques. Bit-parallel adders 4 bits wide are arranged into a pipeline to make a 16-bit adder, which can then be applied serially to add values that are multiples of 16 bits in size (such as 32 bits or 48 bits).

The preceding analysis compared only pure bit-serial, bit-pipelined, and bit-parallel techniques without allowing for anything in between. In fact, nothing prevents the techniques from being used together in concert. Figure 3.8 illustrates, for instance, an adder employing all three styles at once: 4-bit parallel adders are linked into a 4-stage pipeline to make a 16-bit adder, which can then be applied serially over any multiple of 16 bits. The choices presented in the preceding tables therefore are merely the extremes; the best balance is likely to occur somewhere in between. The array might provide hardware for 8- or 16-bit parallel adders, for example, but fall back on pipelined or serial styles for composing larger sizes. Finding the optimum balance would require a more refined cost model and a better understanding of the applications than is possible a priori. But even in commercial FPGAs, the current trend is clearly toward better bit-parallel support.

3.2.3 Array granularity

Array *granularity* is another core design issue. The granularity of a reconfigurable device is its default data unit size, similar in concept to the natural word width of a processor. Array granularity is determined primarily by the number of bits that the wire network requires be routed as a group from one logic block to another. Most commercial FPGAs can route individual bits independently and so have a granularity of 1 bit. In contrast, research designs such as MATRIX [56] operate only on units of 8 bits. The best choice depends on the mix of sizes of data and control values within applications. Note that compared to a fine-grained wire network, a network of granularity n has roughly $1/n$ the number of configuration bits per data bit routed. Array granularity has been studied for FPGAs in general by Cherepacha and Lewis [14] and for PipeRench in particular by Goldstein et al. [22, 23].

Larger-grain routing is generally associated with larger-grain data operations too. In arrays with single-bit granularity like FPGAs, logic blocks typically support a generic set of 1-bit-wide functions from inputs to outputs, implemented as lookup tables. With larger granularity, on the other hand, logic blocks are more likely to be limited to the usual arithmetic operations such as addition and multiplication, without any general table lookups. Hence, granularity is a characteristic that tends to pervade the entire array, both the wire network and the logic blocks.

Granularity should not be confused with support for bit-parallel operations discussed in the previous section. An array with 1-bit granularity can include hardware for constructing fast 16-bit parallel adders. The carry chain structures would simply stretch across multiple logic blocks, as they already do for many commercial FPGAs. In contrast, an array with 16-bit granularity cannot construct an adder *smaller* than 16 bits because that is the minimum data operand size.

The main tradeoffs between large and fine grain are:

- For bit-parallel operations on wide data widths, a fine-grained array can suffer from much wasted redundancy in the configuration representation. To implement a wide bit-parallel operator, a linear array of logic blocks is usually configured identically or nearly identically for each bit of data width, which underutilizes the flexibility of a fine-grained array. A large-grained array should be able to configure a wide operator—at least if it is one of the standard arithmetic operations—with fewer configuration bits. Denser configurations require less time to load into the array, and less area to store within the array once loaded.

- On the other hand, for control logic or bit-serial operations, large granularity suffers from much internal fragmentation in the allocation of resources in the array. If all operands and operations are single bits, an array with 8-bit granularity wastes a factor of 8 in wires and computation hardware over a 1-bit-granularity array. In terms of area, this is a stronger effect than the wasting of configuration bits in the previous point because wires and computation functions consume more hardware area than the configuration storage.

Many applications can be roughly divided into *data* and *control* sections, with the datapath comprising 8-bit-or-wider operations and the control involving mostly single-bit random logic. Taking this as a rule, we can construct a simple model for how relative

		α						
		1:25	1:15	1:10	1:6	1:4	2:5	2:3
	0.2	2	2	2	1	1	1	1
	0.3	4	2	2	2	1	1	1
c	0.4	4	4	2	2	1	1	1
	0.5	4	4	4	2	2	1	1
	0.6	8	4	4	2	2	2	1

c: proportion of area used for configuration storage in a logic block of granularity 1
α: ratio of control bit-ops to total bit-ops

Table 3.4: Granularity with least area per bit-op, according to a simple model.

die area is affected by three parameters: (1) the array granularity k; (2) the ratio, α, of control bit-ops to total bit-ops; and (3) the proportion of area, c, that configuration storage would consume in an equivalent array of granularity 1. Let A_k be the area per logic block of the granularity-k array, and define A_1 to be the area per logic block of a corresponding array of granularity 1. If we assume the granularity-k and granularity-1 arrays have the same number of configuration bits per logic block, we can estimate that $A_k \approx cA_1 + k(1-c)A_1 = (c+k(1-c))A_1$. For granularity k, the area per application bit-op is then

$$\alpha A_k + (1-\alpha)\frac{A_k}{k} \;=\; \left(\alpha + \frac{1-\alpha}{k}\right) A_k \;\approx\; \left(\alpha + \frac{1-\alpha}{k}\right)(c + k(1-c))A_1.$$

Supposing A_1 to be a constant independent of k, this formula can be used to estimate the relative die area consumed for different settings of the three parameters.

Given a particular α and c, the formula can be evaluated for different granularities k to find the granularity with the least area overall per bit-op. Table 3.4 summarizes the results of this exercise over a range of α and c, with granularity restricted to powers of 2. The table shows, for example, that if the ratio of control bit-ops to data bit-ops is 1:15, the optimal granularity is almost surely either 2 or 4, depending on the proportion of area consumed by the configuration bits.

The analysis so far has assumed the reconfigurable array is homogenous, with a single granularity throughout. If it were known that control never accounted for more than 10% of application bit-ops with the rest being wider, multi-bit data operations, it would be smarter to build 10% of the array with single-bit granularity for the control and the rest with a much coarser granularity for the datapath. This type of split has been advocated

by Cherepacha and Lewis [14] and Wittig and Chow [80], and, as noted earlier, has been adopted in the RaPiD reconfigurable array [15, 18, 19].

3.2.4 Multiplication elements

Among the usual arithmetic operations, multiplication is right behind addition and subtraction in importance. In many applications, multiplication occurs as frequently as addition. Unfortunately, it is also far more costly to implement than addition, a general n-bit $\times$ n-bit multiplication being essentially equivalent to summing n n-bit terms. Given its prevalance and impact, array design must consider how multiplication will be accomodated.

The simplest solution is just to build multipliers using the existing reconfigurable hardware without giving the hardware any additional capabilities for the task. The core of any multiplier is the ability to sum many terms together. Dedicated hardware multipliers are typically made as a tree of carry-sum adders, also known as 3–2 adders, which input three terms and output two numbers having the same sum as the inputs. Carry-save adders can be very small and fast, and when replicated often enough they can eventually reduce any number of input terms down to two. The last two terms are then added using a conventional adder to obtain the final product.

Figure 3.9(a) shows how a carry-save adder tree for summing five terms might be constructed in a reconfigurable array. (The "treeness" of the structure would be more apparent with more than five terms.) At each stage, one set of logic blocks calculates the carry result and the other the XOR (sum) result. The number of blocks can be cut almost in half as shown in Figure 3.9(b) if a single logic block can perform two logic operations and drive both outputs onto the network. But there is a potential cost to allowing this, since such dual outputs might have only limited use elsewhere and are sure to impact network delays negatively (Section 3.2.1).

Given the assumption that dedicated fast carry chains are already supported in the reconfigurable hardware, a set of terms can be summed with a simple binary tree of adders as in Figure 3.9(c). Naturally, each adder would be slower than the carry-save adders of the first two structures, but the height of a full adder tree is less than that of the carry-save tree for the same number of terms, which can mitigate the slowness of the full adders. A free-standing hardware multiplier would never be implemented this way if for no other reason than the wastefulness of laying down so many fast carry chains. But in the reconfigurable

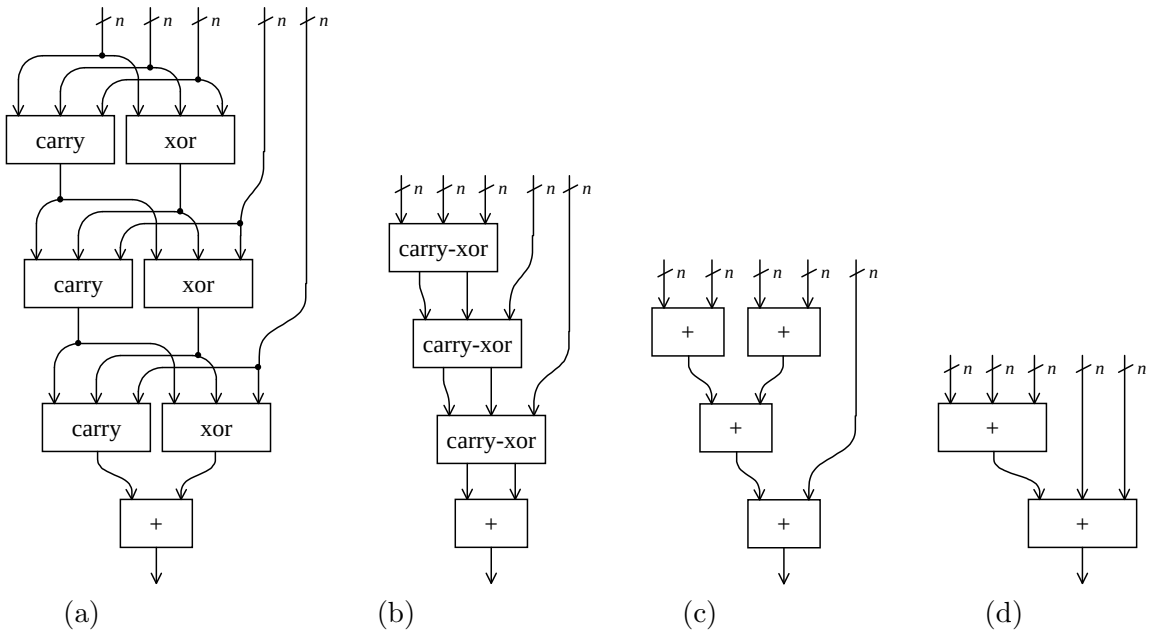


Figure 3.9: Various mutliplier structures. (a) A carry-save tree where the carry and XOR operations must be done by separate logic blocks. (b) The same carry-save tree if logic blocks can drive two outputs onto the configurable network. (c) A tree of adders making use of special carry chain circuitry within the reconfigurable array. (d) A tree of three-input adders. To implement three-input adders, logic blocks only need to perform an internal carry-save addition in advance of the hardwired carry chain. Each box shown is actually a collection of logic blocks sufficient to perform the operation on n operand bits.

		number of operators				tree height				approx. relative latency			
		(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
	4	5	3	3	2	3	3	2	2	4	4	4	5
	6	9	5	5	3	4	4	3	2	5	5	6	5
number	8	13	7	7	4	5	5	3	3	6	6	6	7.5
of	12	21	11	11	6	6	6	4	3	7	7	8	7.5
terms	16	29	15	15	8	7	7	4	3	8	8	8	7.5
	24	45	23	23	12	8	8	5	4	9	9	10	10
	32	61	31	31	16	9	9	5	4	10	10	10	10

Table 3.5: For each summation tree form in Figure 3.9, the number of n-bit operators, the tree height, and the approximate relative latency for numbers of terms ranging from 4 to 32. From Section 3.2.2, the latency of a bit-parallel addition is taken as twice that of a basic lookup table operation after wire traversal is taken into account. Correspondingly, a three-input addition is assumed to have a latency $2\frac{1}{2}$ times that of a simple table lookup.

hardware, the carry chains would presumably be there anyway and so can be used without compunction.

By employing even wider adders, the summation tree height can be reduced further as illustrated in Figure 3.9(d). A three-input adder is easily invented by giving logic blocks the ability to perform one carry-save addition internally in advance of a full addition. Significantly, this extension to the reconfigurable hardware is less drastic than increasing the number of outputs from logic blocks into the configurable network.

Table 3.5 shows that, under reasonable assumptions (recall Section 3.2.2), the latency of all four summation forms in Figure 3.9 is surprisingly about the same. On the other hand, there is a drastic difference in the number of logic blocks consumed that appears to favor overwhelmingly the last structure using three-input adders. Even if the latter form turns out to be not quite as fast as the table suggests, decreasing the disproportionate size of multipliers should have a positive impact all of its own, as explained back in Section 3.2.1.

3.3 Review

The main conclusions of this chapter can be summarized as follows:

- To achieve the best performance, the reconfigurable unit needs to contain its own data state and should be designed to execute for more than a few clock cycles at a time. The reconfigurable unit should not be integrated directly into the pipeline of the main processor but instead should be attached as an on-chip coprocessor. To simplify interlocking, processor instructions for initiating an operation on the reconfigurable unit can be made separate from those that attempt to retrieve any results.

- The system should avoid relying on configuration preloading to hide loading times, because separate compilation often defeats it and because there may not be any unused bandwidth to memory while the reconfigurable unit is executing. Instead, the path to memory should be as wide as possible, configurations should be encoded as densely as practical, and the system should cache configurations for reuse.

- By not allowing configurations to be edited once loaded in the reconfigurable array, context switches can omit having to copy the current configuration back to memory to save it. A prohibition against editing also solves the problem of careless edits creating parasitic configurations having more than one driver on a single wire. This in turn

permits the integrity of a configuration to be verified just once as it is loaded from external memory, using minimal checking hardware.

- If a configuration cache is supported, it should be distributed throughout the array to minimize the time needed to load a configuration from the cache. Configuration cache management ought to be done in hardware as it is for other processor caches. To maximize cache utilization, it should be possible to place small configurations at multiple alternative locations within the array and execute them correctly at any of those locations. The location at which a configuration is placed should be invisible to software.

- The clock within the array should be fixed by the implementation, with each clock cycle corresponding to a logical execution step in the reconfigurable hardware. The architecture can specify the computational "distance" that can be traversed in one clock cycle, allowing configurations to be created that run unchanged on a range of implementations. Whenever array execution must be stalled (such as for a cache miss), the array clock can be automatically delayed without affecting the sequence of logical steps executed by a configuration.

- To ensure the reconfigurable unit has sufficient memory bandwidth during execution, it should be given its own connection to memory that is not dependent on the main processor. Memory accesses can be performed via the same wires used to bring configurations into the array from memory. Loads from memory must be pipelineable so the reconfigurable hardware does not have to sit idle for several cycles during each load.

- To support multitasking, a context switch must be able to freeze and swap out the current configuration and restore it at a later time. Memory reads that are still in the pipeline at the time of the context switch must be saved and restored along with the visible data state. A limit must be placed on the maximum signal propagation distance in any configuration so that configurations can be safely resumed after having been swapped out.

- Page misses that occur due to memory accesses by the reconfigurable unit will have to be completed by the operating system, because the reconfigurable hardware cannot

be backed up to retry the access. To make pipelining easier in configurations, loads at invalid addresses should be ignored, returning arbitrary data.

- Standard reconfigurable hardware is dominated mainly by the configurable network between logic blocks, not by the logic blocks themselves. The reconfigurable unit can be made more efficient in general if logic blocks are expanded with additional functionality that succeeds in reducing the number of blocks needed for configurations.

- Bit-serial and bit-pipelined arithmetic can be more efficient than bit-parallel for basic additions, subtractions, and multiplications, whereas for other mixes of operations, bit-parallel is often superior. The bit-parallel style has less variance overall, and also does not require the profusion of registers often needed for skew-converting bit-serial and bit-pipelined arithmetic. To support bit-parallel arithmetic, a reconfigurable array must include fast carry chains in hardware.

- For reconfigurable hardware with a constant granularity throughout, a granularity of 2 or 4 bits is likely to be most efficient in terms of chip area.

- If the reconfigurable array already contains carry chain hardware, multipliers can be configured fairly efficiently as simple trees of basic adders. By extending logic blocks to include a dedicated carry-sum adder before the carry chain, three-input adders can be created, out of which even denser multiplier structures can be built.

A reconfigurable functional unit for a processor would presumably want to take account of all these concerns.

Chapter 4

Proposed Garp Design

An architecture has been proposed called *Garp* that attempts to address many of the issues in the previous chapter. Garp has been designed to fit into an ordinary processing environment that includes structured programs, software libraries, context switches, virtual memory, and multiple users. The Garp architecture is first presented along with a brief comparison with other related designs, and then a study is made of the suitability of Garp's reconfigurable hardware for implementation in VLSI.

4.1 Garp Architecture

Garp extends a MIPS-II-compatible processor with reconfigurable hardware designed specifically for accelerating kernels in application software. An overall view of Garp is provided by Figure 4.1. Garp's reconfigurable array is often just called "the array" for short.

The Garp main processor has complete control over the loading and execution of configurations on the reconfigurable array. Several instructions have been added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the array and the processor's own registers. Garp's reconfigurable array cannot read or write the main processor's registers itself, but the array does contain data registers of its own.

Garp makes external storage accessible to the reconfigurable array by giving the array access to the standard memory hierarchy of the main processor. This also provides immediate memory consistency between array and processor. Furthermore, Garp has been

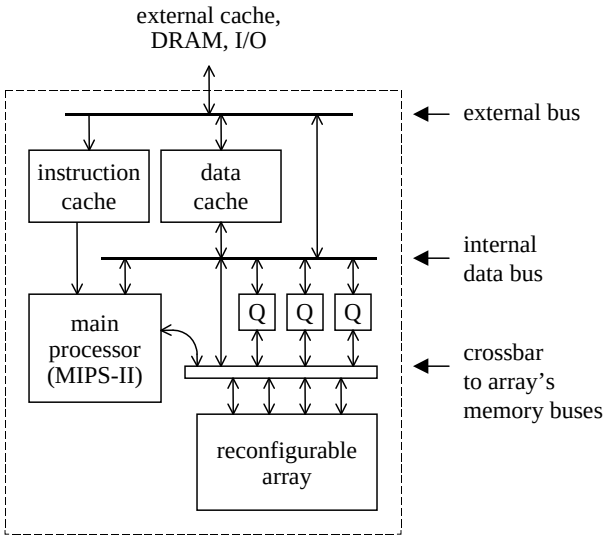


Figure 4.1: Overall organization of Garp. The boxes labeled Q are *memory queues* supporting streams to and from memory as discussed in Section 4.1.9.

defined to support strict binary compatibility among implementations, even for its reconfigurable hardware.

An overview of the array architecture and its integration with the processor and memory are given in the next several sections. Complete details of the Garp architecture are in Appendix A.

4.1.1 Array organization

Garp's reconfigurable hardware is a two-dimensional array of entities called *blocks* (Figure 4.2). One block on each row is known as a *control block*. The rest of the blocks in the array are *logic blocks*, which correspond roughly to the logic blocks of a commercial FPGA. The Garp architecture fixes the number of columns of blocks at 24. The number of rows is implementation-specific, but can be expected to be at least 32. The architecture is defined so that the number of rows can grow in an upward-compatible fashion.

The granularity of the array is set at 2 bits. Logic blocks operate on values as 2-bit units, and all wires are arranged in pairs to transmit 2-bit quantities. Operations on 32-bit values thus generally require 16 logic blocks. Multi-bit functions are naturally laid out along array rows (Figure 4.3). With 23 logic blocks per row, there is space on each row for an operation of 32 bits, plus a few logic blocks to the left and right for overflow checking,

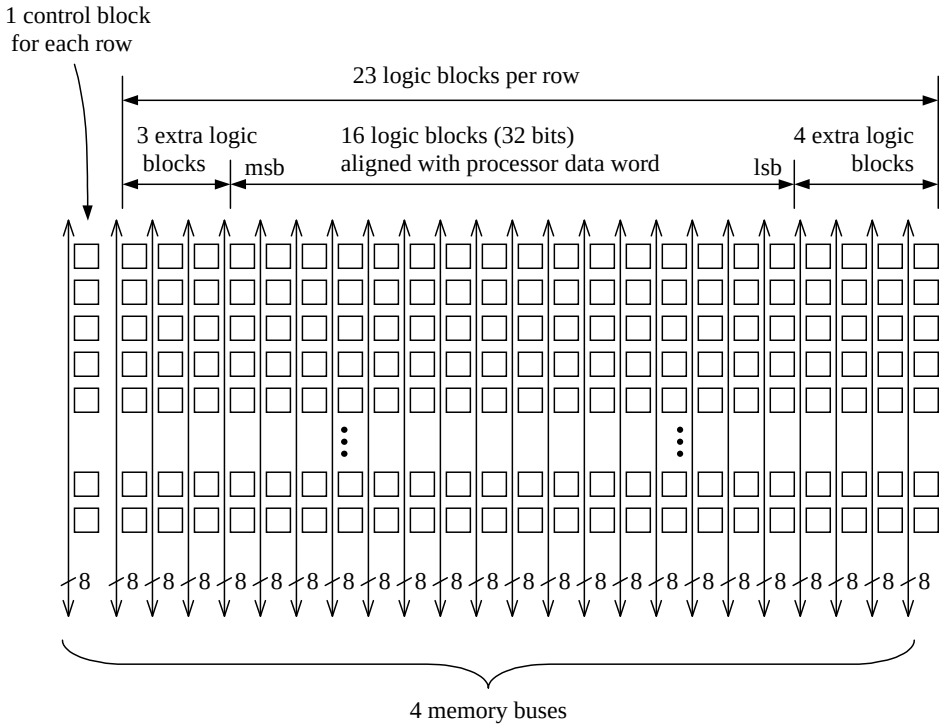


Figure 4.2: Garp array organization. In addition to the memory buses in the figure, a configurable network interconnects array blocks.

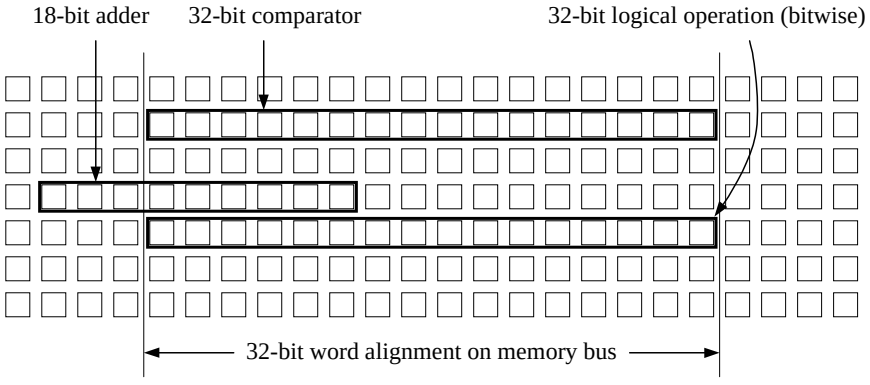


Figure 4.3: Typical natural layouts of multi-bit functions.

rounding, control functions, wider data sizes, or whatever is needed.

A relatively fine granularity was preferred for Garp because an array with small granularity can roughly emulate a larger granularity when desired, but the reverse is less true. Fine granularity is thus a safer choice for a research design. Reconfigurable hardware with 2-bit granularity also provides a starker contrast to the capabilities of a 32-bit processor. A granularity of 1 bit, on the other hand, was seen as unnecessarily extreme based on the analysis of Section 3.2.3. By doubling up bits, the size of configurations—and thus the time required to load configurations and the space taken up on the die to store them—is nearly cut in half at the cost of some loss of flexibility.

As proposed in Section 3.1.6, four *memory buses* run vertically through the rows for moving information into and out of the array. While the array is idle, the processor can use the memory buses to load configurations or to transfer data between processor registers and array registers. While the array is executing, it is the master of the memory buses and uses them to access memory. Memory transfers are restricted to the central portion of each memory bus, corresponding to the middle 16 logic blocks of each row (Figure 4.2). For loading configurations and for saving and restoring array data, the entire width of the memory buses is used.

A configurable network provides interconnection among the array blocks. Wires of various lengths run orthogonally vertically and horizontally. Vertical wires can be used to communicate between blocks in the same column, while horizontal wires can connect blocks in the same or adjacent rows. Unlike most FPGA designs, there are no connections from one wire to another except through a logic block. However, every logic block includes resources for potentially making one wire-to-wire connection, independent of its other obligations.

An individual configuration covers some number of complete rows of the array, which may be less than the total number of physical rows in the array. Distributed within the array is a cache of recently used configurations, so that programs can quickly switch between several configurations without the cost of reloading from memory each time. As with traditional memory caches, the size and management of the configuration cache is transparent to programs.

Data registers in the array are latched synchronously according to an *array clock*, whose frequency is fixed by the implementation. No relationship between the array clock and the main processor clock is required, although it is intended that the two clocks be the same. A *clock counter* governs array execution. While the clock counter is nonzero, it is

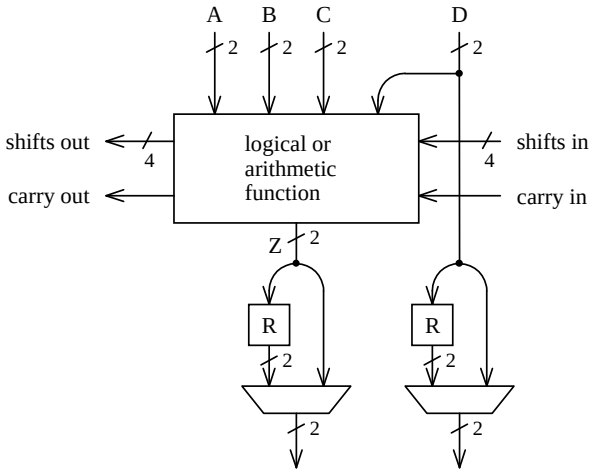


Figure 4.4: Simplified logic block schematic. (Compare with Figures 2.7 and 2.8.) Some of the available functions can be seen in subsequent figures.

decremented by 1 with every array clock cycle. When the clock counter is zero, updates of state in the array are stalled, effectively stopping the array. (Copies to the array by the main processor may still modify array state.) The main processor sets the array clock counter to nonzero to make the array execute for a specific number of array clock steps.

The array can also zero the clock counter itself if its computation is completed before the counter reaches zero (because of an exceptional condition, for example). A very large value in the array clock counter acts as an infinity which can only be returned to zero explicitly by the array or the processor. For cases where the number of clock steps needed to complete a computation is entirely data-dependent, the processor can set the counter to infinity and the array can zero it when done.

The *control blocks* at the end of every row serve as liaisons between the array and the outside world. Among other things, control blocks can interrupt the main processor, zero the clock counter, and initiate data memory accesses to and from the array.

4.1.2 Array logic blocks

Each logic block in the array can implement a function of up to four 2-bit inputs. Operations on data wider than 2 bits can be accomplished by adjoining logic blocks along a row (Figure 4.3). Construction of multi-bit adders, shifters, and other major functions is aided by hardware invoked through a few different logic block modes.

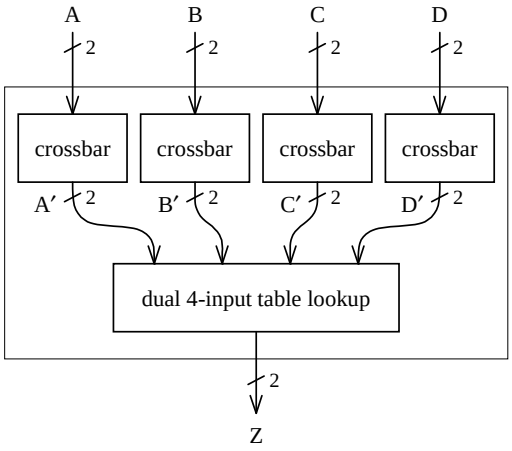


Figure 4.5: Simple table-lookup function for a logic block.

Figure 4.4 provides the basic diagram of a logic block. Four 2-bit inputs (A, B, C, D) are taken from adjacent wires and are used to derive two outputs. One output is calculated (Z), and the other is a direct copy of an input (D). Each output value can be optionally buffered in a register, after which the two 2-bit outputs can be driven onto wires to other logic blocks. The logic block registers can also be read or written over the memory buses. Often the D input is not needed for the logic block function, in which case the "D path" can be used to copy a value from one wire to another.

There are three primary modes for the logic block function:

- *Table mode* implements a four-input bitwise logical function using table lookups (Figure 4.5). A single 16-bit lookup table is independently applied to the high and low bits of A', B', C', and D' to generate the high and low bits of the result; that is, $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$, where the function f is determined by the 16-bit lookup table. The effect is to perform an arbitrary logical function bitwise on the four 2-bit inputs.

- Following the advice of Section 3.2.4, *triple-add mode* performs a three-input addition (Figure 4.6). The values A', B', and C' are reduced by a carry-save adder to two values which are then summed using a dedicated carry chain. To support fast 32-bit-wide additions, each row includes a fast carry chain "box" spread across all the logic blocks on a row, as depicted in Figure 4.7. A full-sized addition of three inputs can be performed in one array clock cycle. Triple-add mode is flexible enough to permit

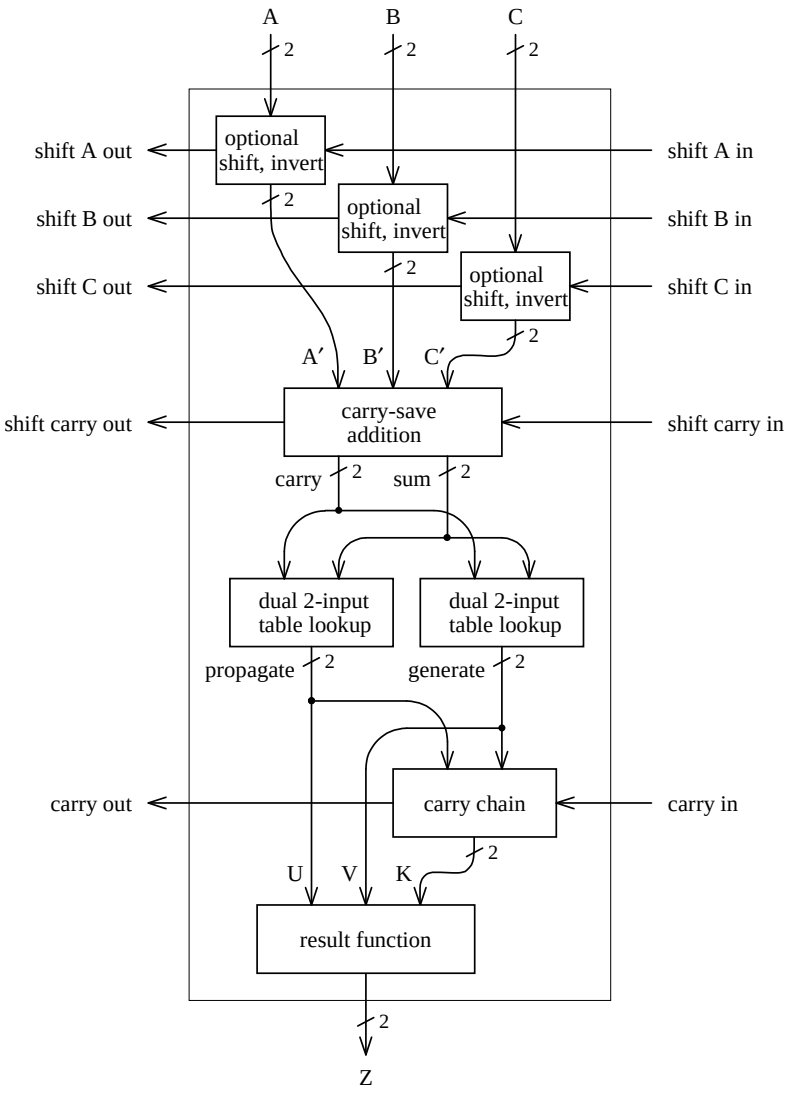


Figure 4.6: Logic block triple-add function.

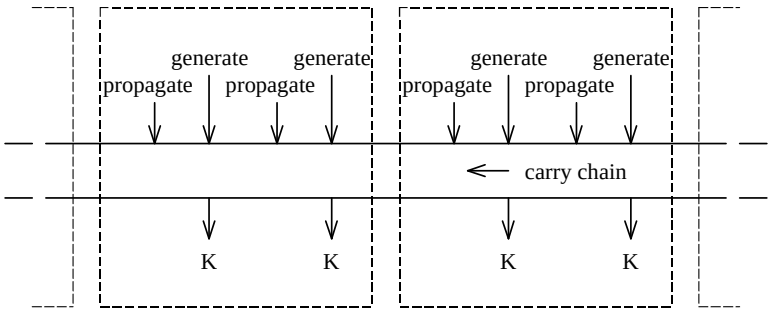


Figure 4.7: The carry chain across a row.

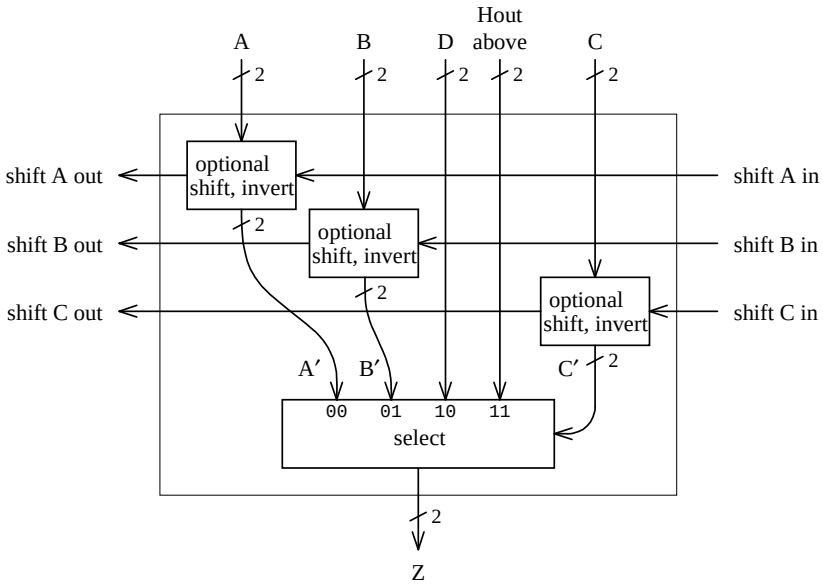


Figure 4.8: Logic block select function. Because this mode needs five inputs, the extra *Hout above* input comes directly from the logic block in the same column in the row above.

the negation of any of the inputs, so that an arbitrary sum or difference of the three inputs can be calculated.

- Because it is not well supported by the other modes, a special *select mode* implements a four-way multiplexor (Figure 4.8). This mode needs an extra fifth input, which it takes directly from the logic block in the same column in the row above.

All of the function modes include *permutation boxes* that can shift or permutate the inputs. The *crossbars* in table mode can perform an arbitrary permutation on the two bits of each input, while the *shift/invert* boxes in the other two modes can shift an input left one bit across a row and optionally complement it logically. The permutation boxes are essential since without them there would be almost no opportunity for the low and high bits of the 2-bit values to ever cross paths. (Interestingly, Cherepacha and Lewis adopted a similar feature with their 4-bit-granularity array [14].)

Each of the function modes already listed has a cousin that is a variation on the same theme:

- *Split table mode* is similar to table mode, but uses two separate three-input lookup tables to determine the high and low bits of the result. The D input is ignored.

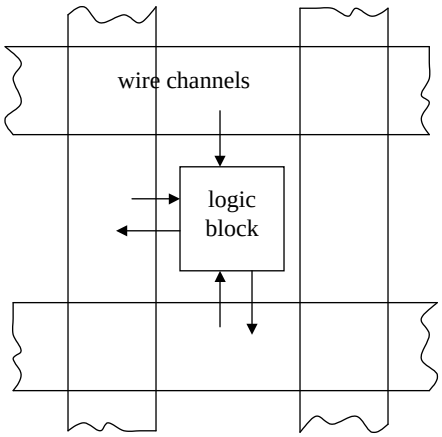


Figure 4.9: The wire channels that can be input and output by a logic block. Logic blocks can read from the horizontal wires above them and can read from and output to the horizontal wires below. Exactly one vertical channel is associated with each column of logic blocks.

- *Carry chain mode* dispenses with the carry-save addition of triple-add mode. All three inputs, A, B, and C, participate in the table lookups that determine the *propagate* and *generate* controls for the carry chain.
- *Partial select mode* is a variant of select mode useful in selecting partial products for multiplications.

4.1.3 Array wires

Vertical and horizontal wires exist within the array for moving data between logic blocks. As the array has 2-bit granularity, all array wires are grouped into pairs to carry 2-bit quantities. Each pair of wires can be driven by only a single logic block but can be read simultaneously by all the logic blocks spanned by the wires. The wire network is passive, in that a value cannot jump from one wire to another without passing through a logic block. Figure 4.9 shows the wire channels that can be input and output by a logic block.

Figure 4.10 illustrates the pattern of vertical wires (V wires) in a single column of 32 rows. By configuring the vertical wires in concert, multi-bit values are easily moved among array rows. Each wire pair can be driven by any one of the logic blocks it spans, and can be read by all of the logic blocks spanned.

Unlike the vertical wires, which are always associated with only a single column of blocks, horizontal wires exist *between* rows, and are accessible by logic blocks in the rows

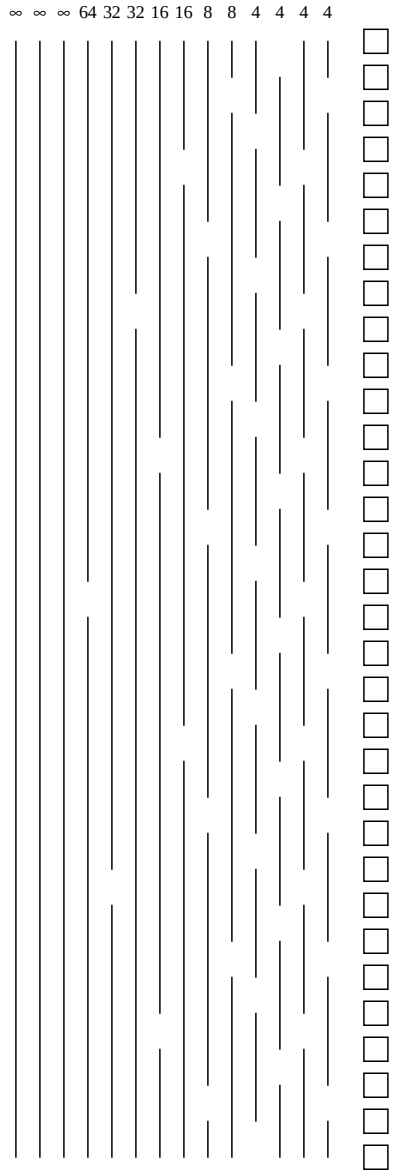


Figure 4.10: The pattern of vertical wires (V wires) in a single column of 32 rows. Each line drawn actually represents a pair of wires (2 bits).

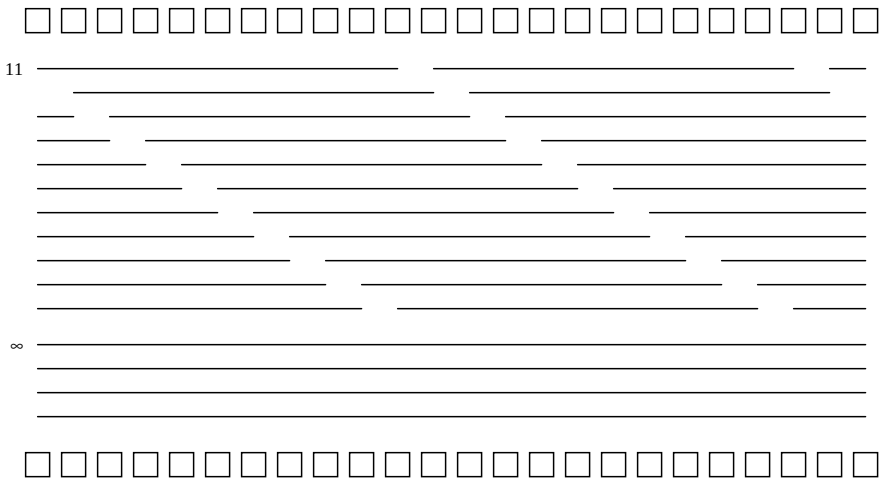


Figure 4.11: The horizontal wires (H wires and G wires) between two rows. Again, each line represents a pair of wires (2 bits).

both above and below the wires, as seen in Figure 4.11. There is a full set of pairs spanning 11 blocks (H wires), and 4 pairs of wires spanning the entire width of the array (G wires). Although horizontal wires can be read from blocks both above and below the wires, they can only be driven from the row above. The horizontal wires can be used to communicate among the columns of a single row, or from a logic block in one row to a different column in the row immediately below.

The horizontal and vertical wires have different patterns because they are optimized for different purposes. The shorter horizontal wires are tailored to multi-bit shifts across a row, while the vertical wires are oriented towards connecting functional units laid out horizontally. The long horizontal wires are typically used to broadcast control signals to all the logic blocks implementing a single multi-bit operation.

The driver of every wire is fixed by a configuration and cannot be changed without loading a new configuration. As discussed in Section 3.1.3, configurations are checked by the hardware when loaded to ensure that no wire has more than one driver. Configurations failing this test cannot be loaded.

4.1.4 Array timing

Delays within the Garp array are defined in terms of the sequences that can be fit within an array clock cycle. Only three sequences are permitted:

- short wire, simple function, short wire, simple function;
- long wire, any function not using the carry chain; or
- short wire, any function.

Any other sequence must be assumed to require multiple clock cycles. The *short* wires include all the shorter horizontal wires, plus vertical wires less than a certain length. A *simple* function is either a direct table lookup or a traversal of the independent "D path" in a logic block (Figure 4.4). At the end of a cycle, a computed value may be latched in a logic block register without affecting the timing.

Quantitizing time with a simple set of rules makes it easy to determine the number of clock cycles a computation will need. From an implementor's perspective, the rules delineate what is required of an implementation in order to run valid configurations correctly, thus facilitating the development of a family of implementations executing bit-identical configurations.

4.1.5 Support for computational primitives

As the Garp array has been designed expressly for computation, Table 4.1 lists the areas and speeds of various computational primitives configured in the array. Area is counted in array rows, and speed is given both as latency and turnaround time in array clock cycles. Note that multiplies and divides by small constants are especially dense because they can be configured as hard-wired shifts and adds using the horizontal wires between rows and triple-add mode.

Most of the operations in the table are for data 32 bits in size, because that is the most convenient data size for a full row. All of the operations are easily reduced to smaller sizes, of course, using less than the full width of the array rows. By design, the Garp array permits most of the simpler operations to be done without the need for any "extra" logic blocks to the left or right of the data width. This means, for example, that an 8-bit comparison will fit neatly within four logic blocks of a row, two bit positions per logic block (the array having 2-bit granularity, recall). This fact becomes significant because 32-bit words are read from and written to memory within exactly the middle 16 columns of the array (Section 4.1.1 and Figure 4.2). Should those 32 bits contain four 8-bit characters,

operation	rows	latency cycles	turn-around cycles
32-bit sum $A \pm B \pm C$	1	1	1
32-bit comparison (=, $\neq$, <, $\leq$, signed or unsigned)	1	1	1
32-bit fixed shift by up to 16 bits (logical or arithmetic)	1+	1	1
32-bit fixed shift, any distance (logical or arithmetic)	1+	1	1
32-bit variable shift (left or right, logical or arithmetic)	3+	3+	1
multiply, 32 bits $\times$ 5-bit constant $\rightarrow$ 32 bits	1+	1+	1
multiply, 32 bits $\times$ 8-bit constant $\rightarrow$ 32 bits	2+	2+	1
multiply, 32 bits $\times$ 16-bit constant $\rightarrow$ 32 bits	4+	3+	1
multiply, unsigned 16 bits $\times$ 16 bits $\rightarrow$ 32 bits	4	7	4
multiply, unsigned 16 bits $\times$ 16 bits $\rightarrow$ 32 bits	9	5	1
integer divide, unsigned 32 bits $\div$ 4-bit constant	4+	4+	1
integer divide, signed 32 bits $\div$ 4-bit constant	5	6	1
arbitrary table-lookup function, 3-bit index $\rightarrow$ 32 bits	1	1	1
arbitrary table-lookup function, 4-bit index $\rightarrow$ 32 bits	3	2	1
arbitrary table-lookup function, 5-bit index $\rightarrow$ 32 bits	5	2	1
32-bit four-input multiplexor (conditional operator)	1	1	1

Table 4.1: Examples of primitive operations implemented in Garp's reconfigurable array. *Latency* is the time from when inputs are supplied to when the result is ready, whereas *turnaround* measures how soon the implementation can accept another set of inputs for pipelined operation. A plus sign in the *rows* column indicates that the implementation depends on the row immediately above to drive its H wires with one of the operands to the operation. If the row immediately above cannot do this, the implementation needs an additional row for this purpose, and may also take an additional cycle as marked in the *latency* column.

	horizontal wires		split		partial	triple-	carry
operation type	H	G	table mode	select mode	select mode	add mode	chain mode
additions and subtractions						Y or Y	
equality comparisons (=, $\neq$)							Y
ordered comparisons (<, $\leq$)						Y or Y	
fixed shifts	Y						
variable shifts	Y	Y		Y			
constant multiplications	Y					Y	
general multiplications	Y	Y		Y	Y	Y	
constant divisions (unsigned)	Y					Y	
table-lookup functions		Y	Y	Y			
multiplexors		Y		Y			

Table 4.2: Array features employed by various operations. At each grid point, a Y indicates that the given operation type makes significant use of the listed mode or wire class.

parallel 8-bit operations can be done directly within each set of four columns without the individual characters having to be shifted apart slightly to make room for extra logic blocks.

Table 4.2 tells how the different logic block functions and two different types of horizontal wires are used by various operations. Garp has six function modes in total; ordinary table mode which implements bitwise logical operations is not listed. For its part, control logic tends to use table mode and split table mode almost exclusively.

4.1.6 Processor control of array execution

The main processor has a number of instructions for controlling the array. The most important are listed in Table 4.3. These include instructions for loading configurations, for copying data between the array and the processor registers, for manipulating the array clock counter, and for saving and restoring array state on context switches. Loading a configuration also initializes all the data registers in the array to zero by default.

As mentioned earlier, an array clock counter controls array execution. When the counter is nonzero the array is executing, and when it is zero the array is halted. To avoid restricting the main processor implementation, the Garp architecture does not specify how many main processor instructions might execute during each array clock cycle. Instead, to keep processor and array synchronized, many of the new processor instructions (Table 4.3) first wait for the clock counter to reach zero before performing their function. The simplest example is when the main processor needs to read the result of a computation performed by the array. After setting the array clock counter to the proper value, the processor can execute a `mfga` instruction at any time. As long as the array is not yet done, `mfga` will wait for the clock counter to become zero before attempting to copy the result over to the processor.

The `mtga` and `mfga` instructions copy to and from the middle 16 logic blocks of a row (recall Figure 4.2.) Additional instructions (not listed) give the processor access to data in the logic blocks at the edges of the array. Several instructions such as `gasave` and `garestore` exist primarily to support context switches.

4.1.7 Configurations

Each block in Garp's array requires 64 configuration bits (8 bytes) to specify the sources of inputs, the function of the block, and any wires driven with outputs. No

instruction	interlock	description
`gaconf` *reg*	yes	Load (or switch to) the configuration at address given by *reg* and zero all array registers.
`mtga` *reg*, *array-row-reg*, *count*	yes	Copy *reg* value to *array-row-reg* and set array clock counter to *count*.
`mfga` *reg*, *array-row-reg*, *count*	yes	Copy *array-row-reg* value to *reg* and set array clock counter to *count*.
`gabump` *reg*	no	Increase array clock counter by value in *reg*.
`gastop` *reg*	no	Copy array clock counter to *reg* and stop array by zeroing clock counter.
`gacinv` *reg*	no	Invalidate cache copy of configuration at address given by *reg*.
`cfga` *reg*, *array-control-reg*	no	Copy value of *array-control-reg* to *reg*.
`gasave` *reg*	yes	Save internal array state to memory at address given by *reg*.
`garestore` *reg*	yes	Restore previously saved internal state from memory at address given by *reg*.

Table 4.3: Basic processor instructions for controlling the reconfigurable array. The *interlock* column indicates whether the instruction first stalls waiting for the array clock counter to run down to zero. (Instructions can be interrupted while stalled.) The last three instructions are intended for context switches.

configuration bits are needed for the array wires, so a configuration of 32 rows requires exactly $8 \times 24 \times 32 = 6144$ bytes. Assuming a 128-bit path to external memory, loading a full 32-row configuration takes 384 sequential memory accesses. A typical processor external bus might need 50 μs to complete the load.

Since not all useful configurations will require the entire resources of the array, partial array configurations are allowed. The smallest configuration is one row, and every configuration must fill exactly some number of contiguous rows. When a configuration is loaded that uses less than the entire array, the rows that are unused are automatically made inactive.

A cache of recently used configurations may be distributed within the array, similar to an ordinary instruction cache. The size of this cache is implementation-dependent. A reasonable Garp might have a 4-deep cache at every logic block—sufficient to hold four 32-row configurations, or sixteen 8-row configurations, or any other combination of the same size.

To maximize cache utilization as discussed in Section 3.1.4, partial configurations are not necessarily loaded at the first physical row of the array. The hardware translates row numbers so that programs see all configurations as starting at logical row 0. Exactly where partial configurations can be placed in the array is dependent on the pattern of vertical wires (Figure 4.10). The vertical wires in Garp follow a repeated, recursive pattern so that partial configurations can be loaded at various offsets.

Two Garp configurations can never be active at the same time, no matter how many array rows might be left unused by a small configuration. This is analogous to there being only one thread of control—only one program counter—in the main processor. If two independently-written configurations could be active simultaneously, there is no way to guarantee they would not interfere with each other's use of the wires. If a program has a special need for making more than one configuration active at a time, it can easily load one larger configuration containing both the smaller ones.

Configurations are loaded in whole from memory. They cannot be edited within the array or written back to memory. Once a configuration has been loaded into the array, its copy in memory must not be modified until the main processor has explicitly purged it from the configuration cache using the `gacinv` instruction (Table 4.3).

4.1.8 Array access to memory

To maximize bandwidth to memory (Section 3.1.6), memory accesses can be initiated in the array without direct processor intervention. These memory accesses proceed in two phases: the first phase is the memory access *request*, and the second is the *data transfer*. For stores, the two phases occur on the same array cycle. For loads, the memory request necessarily precedes the data transfer. With some restrictions, the two phases can be pipelined so that a new memory access can be initiated every cycle.

Array memory accesses are controlled by the control blocks at the edge of the array. Parallel to the memory buses, an *address bus* also runs vertically through the rows. Control signals requesting memory accesses can be generated in the array logic blocks and forwarded by the control blocks to the memory system. A memory address is then read over the address bus from the registers of the row that just initiated the accesses. The data is transferred over a memory bus to/from another selected row, which is often a different row than the one that supplied the address. Up to four contiguous 32-bit words can be read or written with one request over the four memory buses.

The array sees the same memory hierarchy as the main processor. Misses in the on-chip data cache cause array execution to be stalled while the data is fetched from external memory. To reduce cache misses, the array can perform prefetching accesses that merely load the on-chip data cache. Page faults due to array memory accesses are also possible and cause the faulting process to be suspended while the page fault is serviced.

Many commercial FPGAs intersperse blocks of memory within the reconfigurable hardware or permit logic block lookup tables to be transformed into small memories. In contrast, the amount of data state in Garp is intentionally kept down—only four bits in each logic block—to help limit the time needed for context switches. The existing on-chip data cache provides ample temporary storage, although the limited bandwidth of the memory buses can certainly be a bottleneck.

Exploiting maximum parallelism in a program often entails executing memory loads speculatively, that is, earlier than they appear in the original code and thus before it is known for sure that they should have been executed. A speculative load that should not have occurred might very well be at an invalid virtual address. The Garp hardware supports speculative loads in that case by simply ignoring invalid virtual address exceptions and returning arbitrary data as suggested in Section 3.1.8. Loads at valid virtual addresses

cause a page fault as usual if the memory page is not resident; these are serviced in the manner also described in Section 3.1.8.

4.1.9 Memory queues

In addition to the mechanism for demand accesses just described, the array has available to it three memory queues for performing read-aheads and write-behinds on multiple data streams. Three streams are supported that can operate in either direction. All three streams can be read/written in the same cycle, using three of the memory buses concurrently. Memory queues are programmed by the main processor before a configuration is executed. From the array's perspective, queue accesses resemble other memory accesses, except that the array does not provide the address. Read response is also usually faster because the data is already waiting in the queue.

Given that the array can perform arbitrary memory accesses anyway, the behavior of the queues could be implemented by part of a configuration within the array. However, by providing dedicated hardware for this common task, more array resources are freed for the actual kernel computation. Similar memory stream hardware has been considered for standard processors by McKee et al. [55].

4.2 Contrast with other designs

Table 4.4 compares some of the features of Garp and a few other research designs. The other designs considered—OneChip-98, RaPiD, and PipeRench—all have reconfigurable coprocessors that do not operate on the main processor's registers and can execute for arbitrary numbers of clock cycles independently of the master processor. Like Garp, RaPiD and PipeRench define their own reconfigurble hardware. Of the four, Garp is the only one not limited exclusively to stream processing.

In the table, *data-flow feedback* refers to the reconfigurable hardware's ability to encode computations with feedback cycles. PipeRench's virtualization of its reconfigurable hardware (Section 2.5.3) limits its ability to support feedback: any feedback must all fit within one *stripe*, which is PipeRench's unit of virtualization. *Transparent stalling for sync.* means the reconfigurable hardware will stall automatically and transparently as needed to stay synchronized with external components such as the memory system. The OneChip-98 authors fail to mention how their FPGA unit maintains synchronization, presumably

feature	Garp	OneChip-98 [38]	RaPiD [15, 18, 19]	PipeRench [10, 22, 23]
granularity	2 b	1 b	1 b, 16 b	$\approx$ 8 b
data-flow feedback allowed	yes	yes	yes	limited
configuration preloading	no	yes	no?	no
fast configuration cache	yes	yes	no?	yes
preemptable (for multitasking)	yes	no	no?	no?
transparent stalling for sync.	yes	no?	yes	yes
arbitrary memory accesses	yes	no	no	no
no. of streams from/to memory	3	1/1	3	4?
virtual hardware (over streams)	no	no	no	yes
data-dependent loop exits	yes	no	yes	no

Table 4.4: Comparison of Garp with other research designs.

putting the onus on each configuration.

As alluded to in Section 2.5.3, OneChip-98 also puts severe restrictions on the sizes and alignments of its memory streams in order to more easily implement a novel scheme for memory interlocking. Streams must be a power-of-two in size and aligned on a corresponding power-of-two address, generally impeding the utility of the OneChip-98 reconfigurable unit. Moreover, with only one input and one output stream, OneChip-98 is unable to execute even a simple vector addition or to mix two audio signals. The other designs all support at least three arbitrary streams (apparently all in either direction), which in Garp can be further supplemented without limit by additional demand memory accesses. PipeRench can operate on streams with non-unit stride, but cannot perform arbitrary memory accesses on demand as Garp can.

With *data-dependent loop exits*, the number of loop iterations (or lengths of streams) does not have to be known in advance but can depend on the data read. On this point the four systems are evenly split, two for, two against. However, in their experience with PRISM-II, Agarwal et al. found data-dependent exits to be valuable [2], and Garp supports them with the ability of the array to zero the array clock counter itself.

Besides the three systems in the table, Garp's array has superficial similarities to the earlier Dynamic Instruction Set Computer (DISC and DISC-II) [78, 79]. The division of the array into rows to simplify array management is a technique that was first reported for DISC. Garp also resembles DISC in the way that multi-bit operations are naturally oriented across rows, and that global buses run orthogonally through the rows for bringing

values into and out of the array. DISC can further be said to have a configuration cache, although only one plane deep. Just like Garp, DISC can hold multiple configurations (less than full size) in its one plane. If a configuration is needed that is not in the cache, a fault occurs that is serviced in software by DISC's host processor, whereas Garp services such misses in hardware.

Although apparently not supported by the other systems in Table 4.4, Garp is not entirely unique in being preemptable. The Xilinx 6200 (first mentioned in Section 3.1.3) was capable of it, albeit with some complications. Setting aside the question of in-progress memory accesses, the necessary support consisted mainly of the ability to suspend the FPGA clock and to read/write all of the data state from outside the chip. Haug and Rosenstiel have reported on a system using the 6200 as an external accelerator that is time-shared along with the rest of the system [33, 34].

4.3 Implementation study

As the Garp architecture has never been implemented and tested physically, it is necessary to question how effectively it could be realized in standard VLSI. Our main interest is to assign estimates for chip area, speed, and power consumption, so that the Garp architecture can be more faithfully measured against a contemporary processor in Chapter 5.

To make the matter more concrete, a specific VLSI process technology has been assumed: 0.65 μm transistor gate length, three layers of metal, operating at 3.3 V, and with layout conforming to standard MOSIS scalable CMOS design rules (submicron "tight metal"). Although not state-of-the-art, this process technology was the best available through MOSIS when the project began, corresponding approximately with an early Intel Pentium Pro, a MIPS R4200, or an original Alpha 21064. In Chapter 5, the results from this study are extrapolated to the slightly better 0.5 μm process of an UltraSPARC 1. The implementation of Garp has been restricted to a realistic die area and power budget, taking into account this intended "shrink" from 0.65 μm to 0.5 μm.

4.3.1 Overall functional organization

Since the Garp architecture connects the reconfigurable array to the traditional MIPS processor through a simple coprocessor interface, there are relatively few points of

contact between the processor and the array. The following features are needed:

- A connection to the processor's register read and write ports is necessary to support the movement of individual 32-bit words between registers and the array by Garp's `mtga` and `mfga` instructions. Traditional MIPS coprocessor interfaces have always supported equivalent move-to-coprocessor and move-from-coprocessor instructions, so this feature is nothing new.

- The array must have access to the primary (L1) data cache. Assuming the cache is not made dual-ported, the only real complication is the need to arbitrate between the processor and the array if both attempt to access the cache at the same time. However, a simple policy should suffice, such as always giving preference to the processor when there is a conflict. (Note that if it were the reverse and the array always had preference, a steady stream of array memory accesses might ultimately prevent the processor from exercising control over the array.)

- Configuration loading should bypass the primary data cache, or the cache would be swamped whenever a new configuration was loaded into the array. Like most processors, MIPS loads instructions into a separate instruction cache, bypassing the L1 data cache, so routing configuration loads around the L1 data cache in the same way should not pose a significant burden.

- The array needs the option to prefetch from main memory into the data cache, and also to perform a memory access without cache allocation should the access miss in the cache. These options are not strictly necessary, but have been defined in the Garp architecture for performance reasons. Many commercially successful processors give software the same control over the caching of memory accesses.

- Some of Garp's new processor instructions must interlock (stall) processor execution if the array clock counter is nonzero. Since an instruction could be stalled on the clock counter indefinitely, the stall must be interruptible. However, the stall condition does not need to be remembered on a context switch so long as the stalled instruction gets reexecuted when the context is resumed.

None of these features requires any truly novel mechanisms, and together they would not be expected to add much to the cost of the processor. This leaves questions about the feasibility of implementing Garp focused squarely on the array itself.

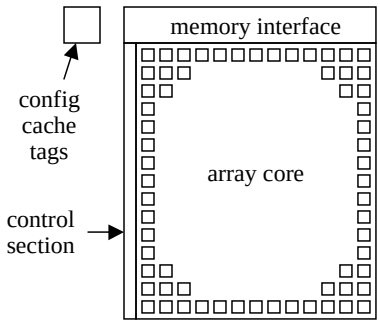


Figure 4.12: The four main parts needed to implement the complete Garp array. (Scale is only approximate.)

As Figure 4.12 illustrates, the hardware of the array can be divided into four main parts: (1) the core array of logic blocks, the largest part, (2) a control section incorporating the control blocks at the left edge of the array, (3) a memory interface, presumably adjacent to one of the edges either above or below the array, and (4) the tags for managing the configuration cache distributed within the array core. The control section is naturally the nexus for control signals entering and exiting the array core and also the place to coordinate control among the control blocks. The cache tags not only should keep track of which configurations are where in the cache but should also be responsible for choosing a configuration to evict when space is needed.

The most complex part outside of the array core is the memory interface, which among other things includes the memory queues, an alignment network, and control circuitry for tracking multiple memory accesses in progress. The memory interface is directly responsible for the memory buses that run through the array. In addition to the memory accesses initiated by the array itself, the memory interface must be involved in data transfers initiated by the processor (`mtga` and `mfga`) and in the loading of configurations from memory. The memory interface must also be responsible for checking the validity of a configuration while it is being loaded. If a configuration has more than one driver for any wire, it must be rejected. Since configurations cannot be edited in the cache, a configuration made active directly from the cache does not need to be checked again.

The local horizontal wires (H wires) on each row do not require checking because by design they always have exactly one driver in any configuration. The global horizontal wires (G wires) must be checked, but these can be tested in full as each row's configuration is brought in. However, in the case of the vertical wires, the information needed is spread

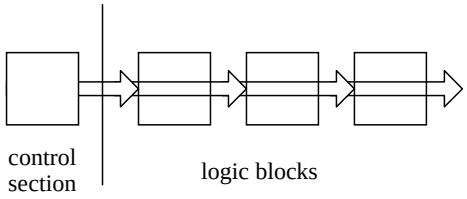


Figure 4.13: Broadcasting control signals across a row from the control block at the end.

across multiple configuration rows, one block per row. Assuming configurations are read from memory in contiguous order, one full row at a time, some status for each vertical wire must be maintained in the memory interface and updated as each row of the configuration is read. The status required is not great, amounting to only a single bit per wire to indicate whether a driver for that wire has yet been seen. Not all vertical wires span the entire height of the array; once a shorter vertical wire has been entirely validated, its status bit can be reclaimed for checking the next wire below it. The regular pattern of wire breaks, seen in Figure 4.10 and explained further in the architecture manual in Appendix A, makes it straightforward to construct a state machine to handle the breaks.

Unlike the network wires, the memory buses are used dynamically and so cannot simply be checked for driver conflicts at load time. Array memory accesses are initiated by the control blocks in the array; to guard against multiple rows driving a memory bus at the same time, the control section must check at run-time for such cases and abort array execution of the offending configuration. The main processor can be notified via an ordinary processor interrupt.

Control signals affecting the array core almost always apply to rows of logic blocks as units, not individual logic blocks or columns of blocks. Thus it makes sense to broadcast most control signals across each row from the row's control block at the end of the row, as suggested in Figure 4.13. Examples of control signals for a row would include: an enable for whether the row is currently active; wires that manipulate the configuration cache and select an active configuration from the cache; and signals for loading logic block registers from one of the memory buses or for driving a set of register values onto one of the buses. As far as these control features are concerned, Garp's logic blocks are almost completely passive, merely responding to the direct manipulation of the control blocks. Very little clocking, in fact, is needed within the array core itself beyond the architecturally visible logic block registers, which are latched every clock cycle when the array is executing.

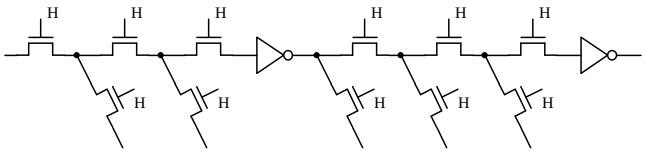


Figure 4.14: Routing a signal with pass transistors. Because the maximum voltage that can traverse a pass transistor is a threshold voltage drop below the voltage on the transistor gate, the transmitted signal has a lower voltage swing than the control signals on the transistor gates. The *H*'s in this and subsequent figures indicate a higher voltage on the gates than the signal passing through the transistor. For best efficiency, the signal is regularly regenerated by inverting buffers operating entirely at the lower data voltage.

4.3.2 Using pass transistors for switching

Much of a reconfigurable device is concerned with simply routing data through configuration-controlled switches, both between logic blocks and within them. The switches support the large number of possible signal paths that give the reconfigurable device its flexibility, but they also slow down signal propagation without performing any obvious computation. In VLSI, the very simplest type of switch is a single pass transistor, where the voltage on the gate controls whether the path between the source and drain is open or closed (Figure 4.14). In some case, the gate might be controlled directly by a configuration bit, resulting in a highly compact implementation.

Unfortunately, while a single N-type transistor can pass a zero-volt signal cleanly, it cannot pass a signal at the full source voltage, only a reduced voltage (specifically, a transistor threshold-voltage drop from the source voltage). This means that the signal on the downstream side of a pass transistor has a reduced voltage swing, ranging between zero volts and something rather less than the full source voltage. The reduced voltage is weaker—and thus slower—at driving subsequent transistor gates than the higher source voltage would be. Nevertheless, when a few switches are combined in sequence, the weakness of the reduced-swing signal at the output is made up for by the speed at which the signal traverses the simple pass transistors, and by the smaller area of the switches compared to the alternatives.

The advantage of pass transistors dies out after about four or five in series, due to their combined impedance (RC delay). For best results, the signal has to be restored with a buffer (preferably an inverter) about every four pass transistors, as shown in Figure 4.14. Since the input to the buffer has reduced swing, and since there is no harm in making

the output reduced-swing too, the buffer is most efficient if operated entirely at the lower voltage of the reduced-swing signal. This leads eventually to a dual-voltage design, where the high (full) voltage is generally associated with the configuration, and a reduced voltage is used for the active data signals.

This dual-voltage design might not be viable at the lower operating voltages expected in the future; that depends on whether transistor threshold voltages will be effectively scaled down along with operating voltages, a question still being researched. However, the technique does work for the 3.3-V technology being assumed; and considering that commercial FPGAs have been built this way, it has seemed fair to adopt the method for the hypothetical Garp implementation. In circuit diagrams where both voltages appear, full-swing signals will usually be distinguished with an *H* for their higher voltage; other signals in the same figure are low-swing by default.

4.3.3 Array wires

As with other reconfigurable devices, the wire network connecting the logic blocks is a large part of the Garp array. The Garp architecture stipulates that network wires are broadcasting: every logic block adjacent to a wire can sense the signal on that wire. The V (vertical) wires and the G (global horizontal) wires can also be driven by any of the logic blocks adjacent to or above the wire. Thus, a single V wire is defined logically as illustrated in Figure 4.15, with each adjacent logic block able to drive and/or sense the wire. The actual physical implementation of the wire does not have to mimic this logical view. Nevertheless, for shorter wires spanning only a few logic blocks, the simple circuit with the wire as a common node is already fairly efficient both in area and speed. The complications of longer wires will be discussed shortly.

Figure 4.15 shows a single wire; the wire channels between blocks contain many such wires, of course. A single logic block's view of the vertical wire channel is that of Figure 4.16. The Garp logic block has four inputs, *A*, *B*, *C*, and *D*, each of which can connect to any wire in the channel. Each input thus has an associated multiplexor to choose one of the wires for that input. (This is just for the vertical wires in the vertical wire channel. A similar story applies to the horizontal wires above and below the logic block, as well.) To reduce the combined loading of multiple logic blocks on the wires, each block should buffer the wires through small inverters before passing the signals to its four input

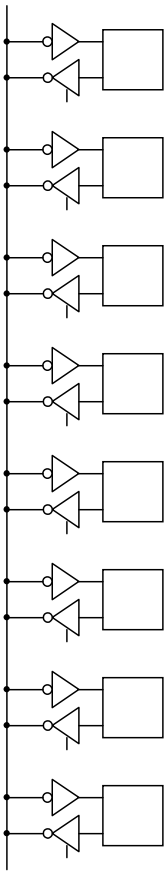


Figure 4.15: Physical implementation of a wire matching the Garp architecture's logical definition.

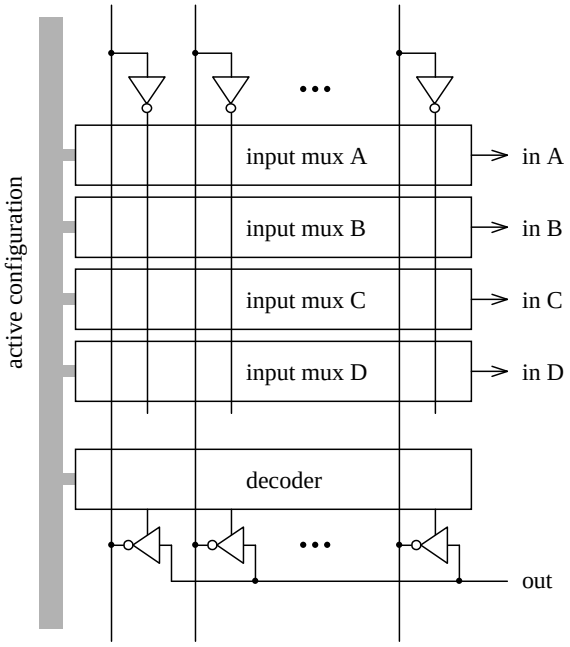


Figure 4.16: The input multiplexors and output drivers underneath the vertical wire channel at a logic block. To minimize loading on the network wires, inputs are multiplexed from a locally buffered copy of the wires. All data values are actually 2 bits in size though shown as single lines.

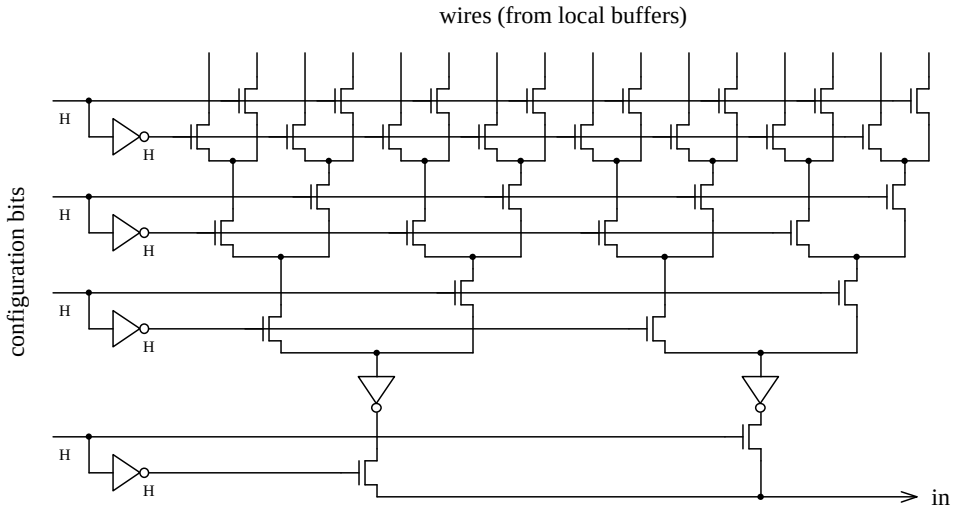


Figure 4.17: A single input multiplexor implemented as a binary tree of pass transistors. Again, the data values being multiplexed are really 2 bits wide.

multiplexors.

For the best speed in the smallest area, the input multiplexors can be constructed as a binary tree of pass transistors as shown in Figure 4.17. There are other ways to build multiplexors, but the pass transistor tree is actually one of the fastest. Buffers are needed within the tree to refresh the signal, but by then the tree has been reduced to only a few branches and the buffers are a small proportion of the total area.

At the output side, the logic block may drive one of the 2-bit wire pairs in the channel with an output value. To accomplish this, a battery of tri-state drivers can be connected to the wires in the channel, and a small decoder used to select one of them (or none) according to the destination encoded in the configuration (Figure 4.16). Assuming a little settling time can be tolerated during configuration changes, the output decoder does not need to be fast and consequently can be made quite small. As will be seen later, the area taken up by the decoder is actually dwarfed by the tri-state drivers themselves, which must be sizable to overcome the loading and resistance of the wires.

A fundamental reality that must be contended with is that each wire is connected to the tri-state drivers of multiple logic blocks, only one of which actually drives the wire. The active driver thus has to overcome the capacitance of all its kin. On the one hand, the drivers could be made stronger by making them larger; but on the other, that multiplies the loading along the entire wire, making the drivers' jobs harder. One advantage the tri-state drivers have in this context is that a typical tri-state driver is designed to respond quickly to changes in both the data input and the enable control signal. For the logic block output drivers, however, the enable signal only changes when the configuration changes, so enable response time can be sacrificed to improve other parameters.

Figures 4.18 and 4.19 show two common styles of tri-state drivers, both of which are troublesome for this application. In both circuits, components that affect only *select* response time have been drawn smaller than the other elements to indicate they can be built with transistors of minimal size. The trouble with the driver in Figure 4.18 is that it puts two transistors in series on both the pull-up and pull-down sides, forcing the transistors to be doubly large to achieve the same power, therefore unnecessarily compounding the wire capacitance problem. The other, in Figure 4.19, is better in that it has only a single transistor for the final pull-up and pull-down, but it also needs ten transistors, six of which cannot be a trivial size.

Figure 4.20(a) has a third alternative that is a better choice than the other two.

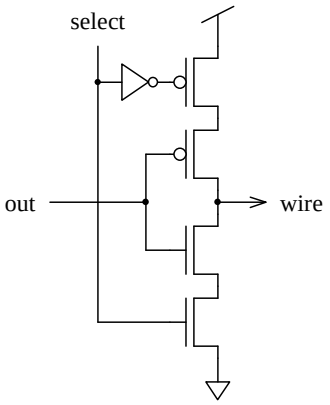


Figure 4.18: A common style of tri-state driver with four transistors in series. The pairing of pull-up and pull-down transistors means the transistors have to be twice as large as those of a simple inverter to achieve the same speed.

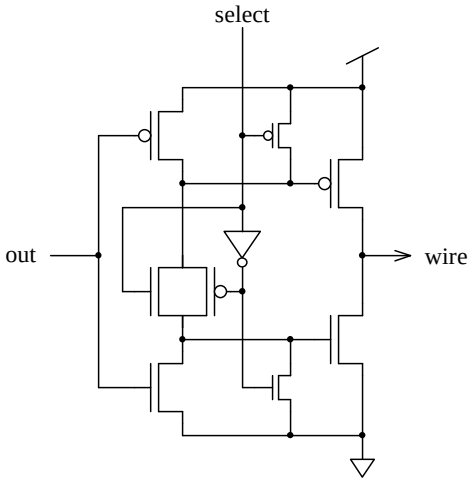


Figure 4.19: Another common tri-state driver circuit with only two final transistors in series. The two final transistors do not need to be as large as those of Figure 4.18, but now there are six non-minimal transistors and the signal is inverted twice.

This circuit depends on the higher voltage of the *select* configuration line, using pass transistors again to achieve the fewest number of transistors between *out* and the wire. Since Garp always drives pairs of bits in tandem, the small inverter can be shared between the pair of drivers, as seen in Figure 4.20(b). This circuit can be layed out neatly underneath the vertical wire channel as illustrated in Figure 4.21, dedicating nearly half the area just to the crucial pull-up and pull-down transistors. With the *out* lines for all the drivers in the logic block tied together (Figure 4.16), this circuit is not the fastest, but it is an excellent compromise between speed and area for the purpose.

Garp's longest wires run the length of the array, which at 32 rows is too long for the simple circuit of Figure 4.15 to work effectively. The combined capacitance of 32 output drivers on a wire is too much for any driver of reasonable size to overcome. The solution is to install configurable buffers between shorter segments, as illustrated in Figure 4.22. If each segment is eight blocks long, the longest wires contain three such buffers. The buffers need to be fairly hefty, but they ultimately save power by sharpening the rise and fall times of signals on the wire. The global horizontal wires (G wires) in the other dimension are 24 blocks long and thus have two buffers. With the insertion of the inter-wire buffers, the

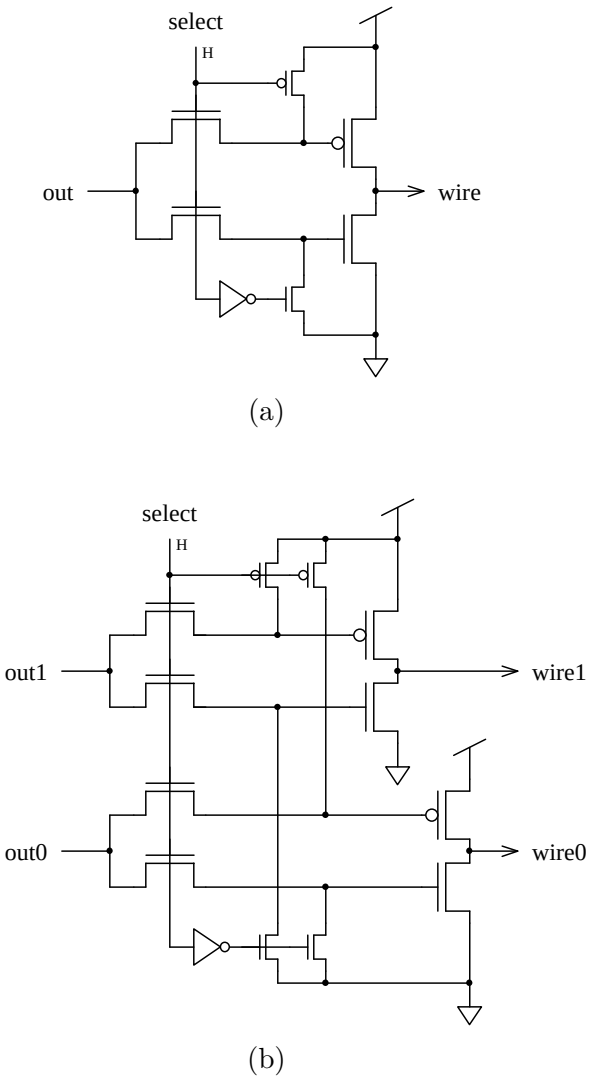


Figure 4.20: (a) A tri-state driver circuit with only four nonminimal transistors and only two final transistors in series, making use of a higher control voltage for the *select* line. (Compare with Figures 4.18 and 4.19.) (b) The same driver doubled for transmitting two bits in parallel.

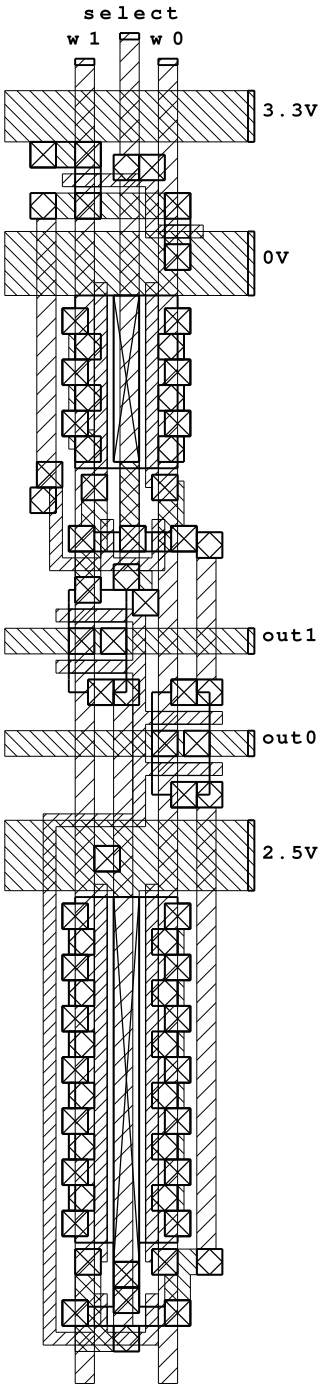


Figure 4.21: Layout underneath the vertical wires for driver circuit in Figure 4.20(b). The small *select* inverter is at the top. About half the area is consumed by the final pull-up and pull-down transistors.

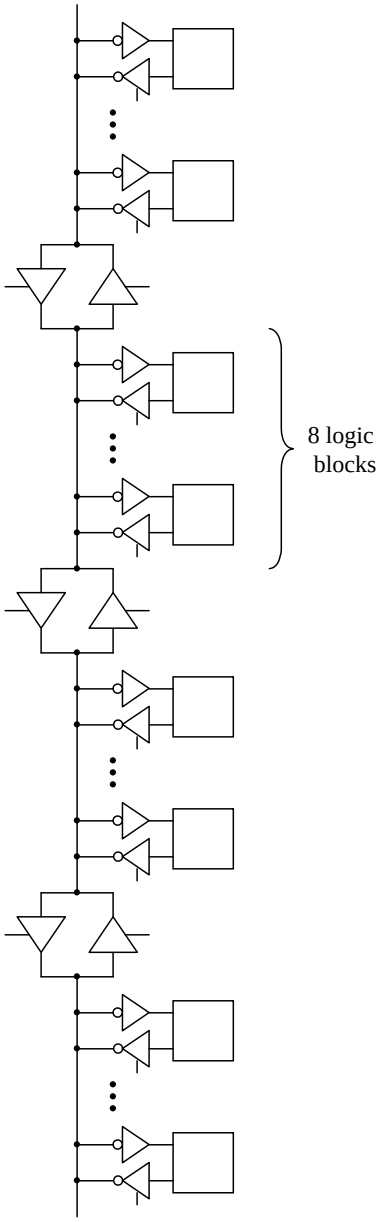


Figure 4.22: Breaking the longer wires into pieces eight logic blocks long joined by configurable buffers.

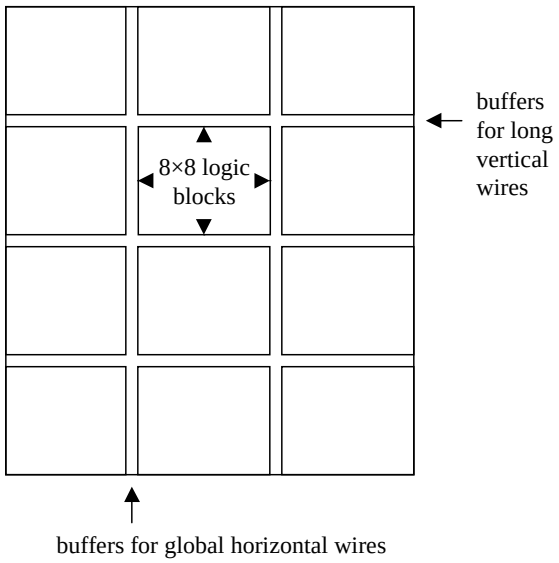


Figure 4.23: The array core as a 4×3 quilt of patches 8×8 logic blocks each. Between the patches reside the buffers for the longer wires.

array core layout becomes a 4×3 quilt of 8×8 logic-block patches, with the space between patches allocated to buffers, as in Figure 4.23.

The direction of each inter-wire buffer must be set to match the current configuration, even though the actual encoding of configurations defined by the architecture knows nothing of the buffers. Fortunately, since the array memory interface already must collect information about wire drivers to guard against multiple drivers on the same wire, it can easily use the same information to set the inter-wire buffers. As a configuration is loaded from memory, the array memory interface expands the configuration to include the wire buffer settings at the same time that it checks for driver violations. The interpolated configuration bits are then loaded into the physical array and managed the same as the others. The existence and operation of the inter-wire buffers can thus be made completely transparent to all Garp software.

4.3.4 Configuration cache management

The Garp design supports a configuration cache distributed within the core array. Until the need for more has been proved, it can be supposed a small number of cache planes will be enough to start with, something in the range of four to eight. For the Garp implementation, four cache planes have been assumed, with the understanding that this number could be doubled to eight without much difficulty if needed.

As mentioned in Section 3.1.4, the physical wire pattern in the array limits the locations at which an arbitrary configuration can be blindly loaded and executed. The pattern of vertical wires adopted for Garp (Figure 4.10) has a power-of-two recursion that permits a configuration of, for example, eight rows to work at any offset that is a multiple of eight. Configurations of five, six, or seven rows must be rounded up to the next power of two and located on a multiple-of-eight boundary. This does not mean, however, that a non-power-of-two configuration consumes the extra rows. Although an 11-row configuration must be placed starting at a 16-row boundary, rows after the 11th are free to hold a smaller configuration that will fit, such as a 4-row configuration beginning at the 12th row.

The power-of-two hierarchy suggests a version of the buddy system, a technique for allocating storage described by numerous sources including Knuth [43]. To reduce the work associated with allocating array space, the smallest unit of allocation will be four rows. The maximum number of one-row configurations a cache plane can store is therefore

$32 \div 4 = 8$, even though it would be possible to fit 32 one-row configurations into a single cache plane. It is expected very few useful configurations will be as small as four rows (let alone one row), so there is little need to carry the buddy system hierarchy all the way down to individual rows. Limiting the minimum allocation to four rows cuts the maximum number of loaded configurations, and thus the size of the cache management hardware, by a factor of four.

4.3.5 Logic block layout

To estimate the VLSI area needed to implement Garp, a promising logic block organization was floor-planned and then layout was completed for the most critical components, those thought to have the greatest influence on logic block size. Although the lookup tables can be considered the logical heart of each logic block, the hardware to perform the lookups represents only a small portion of a logic block's total area. To see why, consider that the two lookup tables in a Garp logic block each use 8 configuration inputs to reduce 6 input data bits to 2 output bits. Together that makes 16 configuration bits and 6 input bits, generating 4 output bits. The input multiplexors, in contrast, require 24 configuration bits to reduce 86 adjacent wires down to 8 bits of inputs to the logic block datapath. The differences in the numbers illustrate clearly how the input multiplexors will dwarf the lookup table hardware in size. On the output side, 26 bits worth of drivers are needed on the vertical wires, each fairly large as seen in Figure 4.21. Moreover, the logic block must find a place to store not only the 64 bits of *active* configuration but also four planes of configuration cache, for a grand total of 256 bits. Any missteps that prevent these three parts—the input multiplexors, output drivers, and configuration storage—from being ultra-dense will have visible impact on the size of the entire array core.

Figure 4.24 shows how an individual logic block might be organized. The logic block is framed by the horizontal wires above and below it and by the memory buses on each side (not shown on the left side). Underneath the network wires are the twelve input multiplexors leading to the four principal 2-bit inputs, and also the output drivers for the vertical wires and the horizontal wires below the block. The inner logic block datapath sits to one side, handing its two 2-bit outputs back to the output drivers and/or into the box below containing the data registers, which also have a connection to the memory buses for external reading and writing. Straddling everything on both sides is the configuration

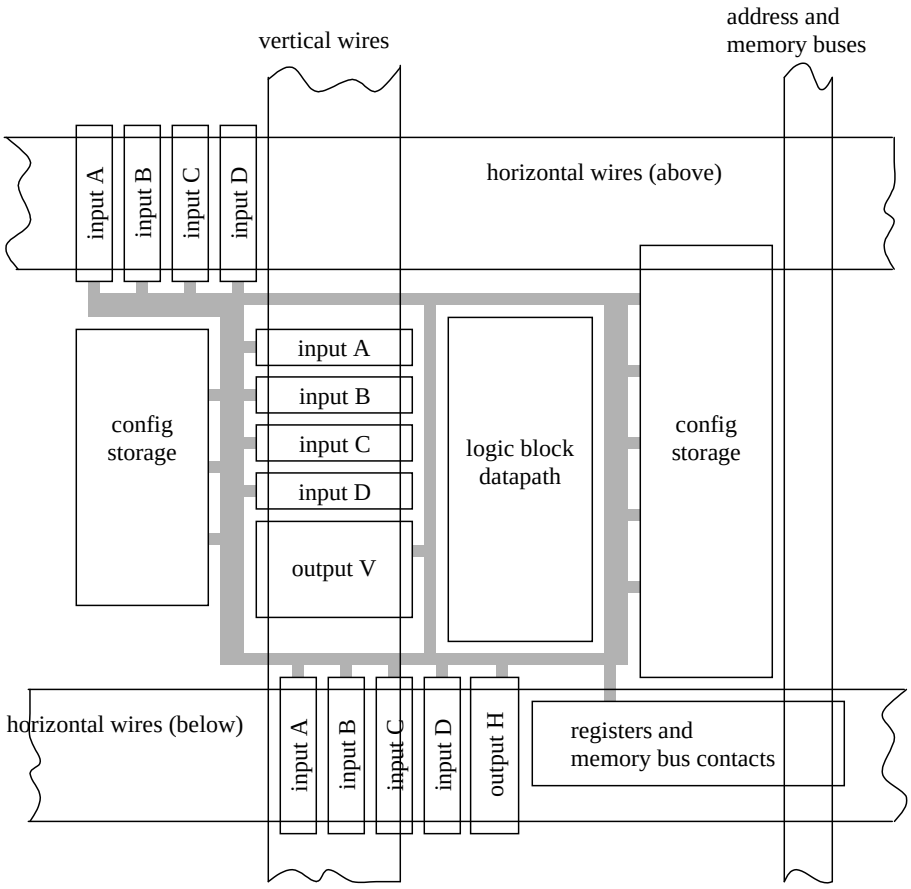
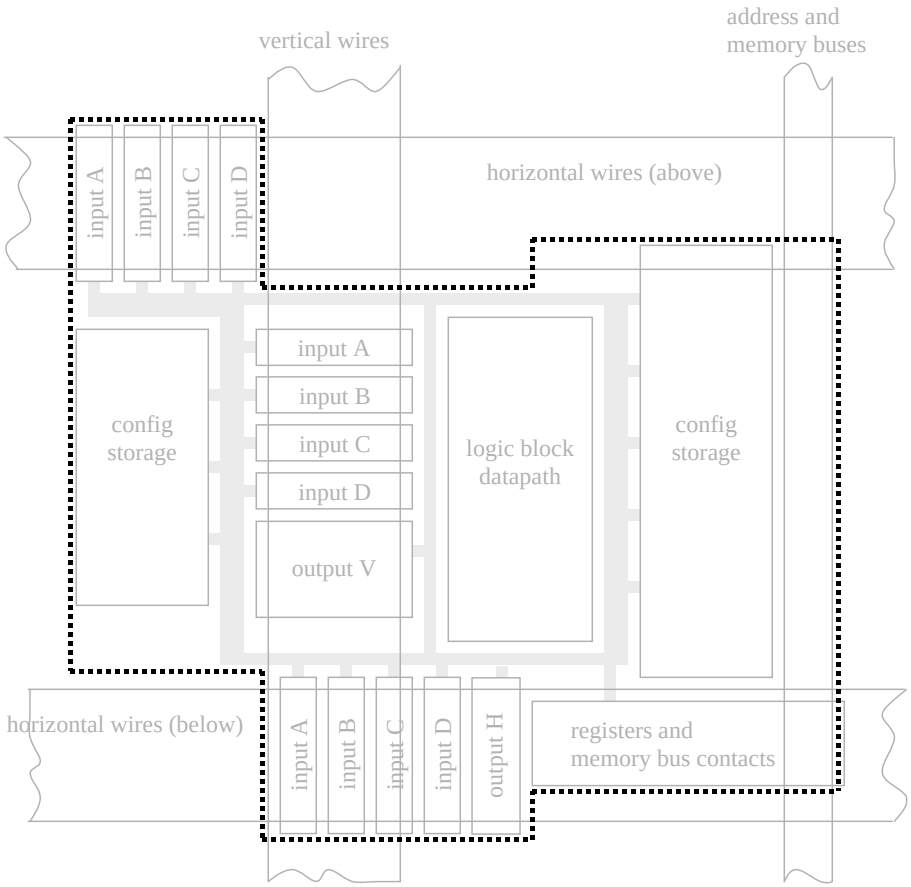
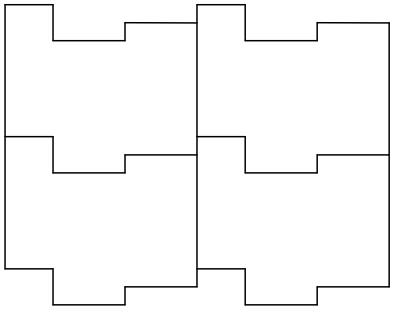


Figure 4.24: Proposed layout organization for a logic block (not exactly to scale.) The grey paths carry configuration control bits from the configuration storage to the rest of the logic block.



(a)



(b)

Figure 4.25: (a) Outline of a single logic block tile. (b) Illustration of how the tile tessellates in two-dimensional space.

storage, which accomodates the configuration cache and conveys the active configuration onto a web encircling the other parts (represented in grey in the figures).

Figure 4.24 is not exactly to scale but is intended to be approximate. The configuration storage is as small as it is only because dynamic memory is assumed for the cache. More about the storage and distribution of configuration bits will be covered in the next section.

Most logic blocks have identical neighbors on all four sides, so the logic block layout must tile in two dimensions. Figure 4.25 gives the outline of the tile and shows how the pattern repeats. Note that logic blocks above and below a horizontal wire channel must share the channel between them, whereas there is very little sharing contact between neighbors side-to-side since a vertical wire channel is confined to a single column of logic blocks.

In addition to the network wires and memory buses, the logic blocks have to be reached by power and ground, a clock signal, control signals from the row's control block, and the carry chain and shifts that connect to neighboring logic blocks on a row. Figure 4.26 shows how everything is designed to be weaved through in three layers of metal. The extra connections are all brought in across the row in the third metal layer (M3) between the two horizontal wire channels. Meanwhile, the network wires are made to bob up and down between the second and third metal layers. Unfortunately, this gives the network wires two metal layer transitions to traverse per logic block. The wires would be faster without the extra resistance of the vias, but it cannot be helped. On the other hand, the fact that the memory buses are not burdened with these vias will make them a little faster than the vertical wires, which is a plus.

As said earlier, key parts of the logic block design were actually layed out—principally the input multiplexors, vertical output drivers, table lookups, and the dynamic storage and active registers in the configuration storage blocks. All of these have been brought together into an image of a logic block tile in Figure 4.27, which also shows the space reserved for the remaining parts. The correlation between this image and previous diagrams should be clear. The third metal layer has been removed from the figure to prevent it from obscuring everything.

One thing to note in the figure is how the table lookups are indeed dwarfed by the input multiplexors and output drivers as expected. The rest of the large unfilled area for the logic block datapath is reserved for implementing all the extra functionality within Garp's

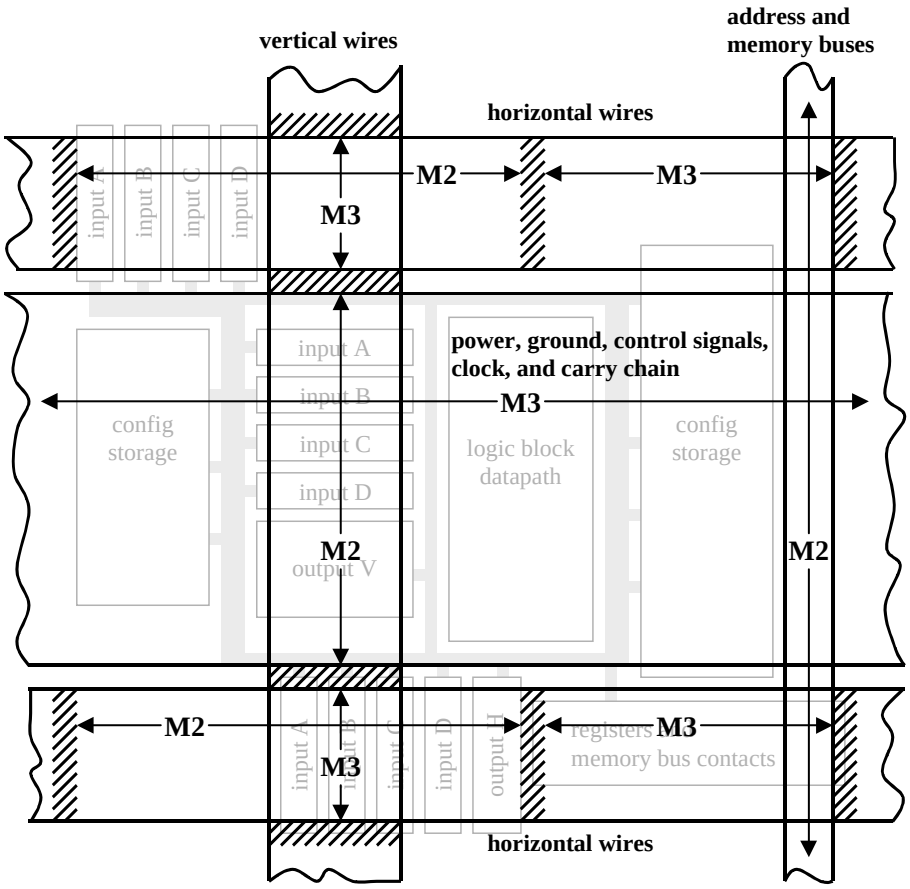


Figure 4.26: Allocation of metal layers 2 and 3 over an individual logic block. Where it's not shown to be allocated in this diagram, the metal-2 layer is available for use locally along with the metal-1 and polysilicon layers.

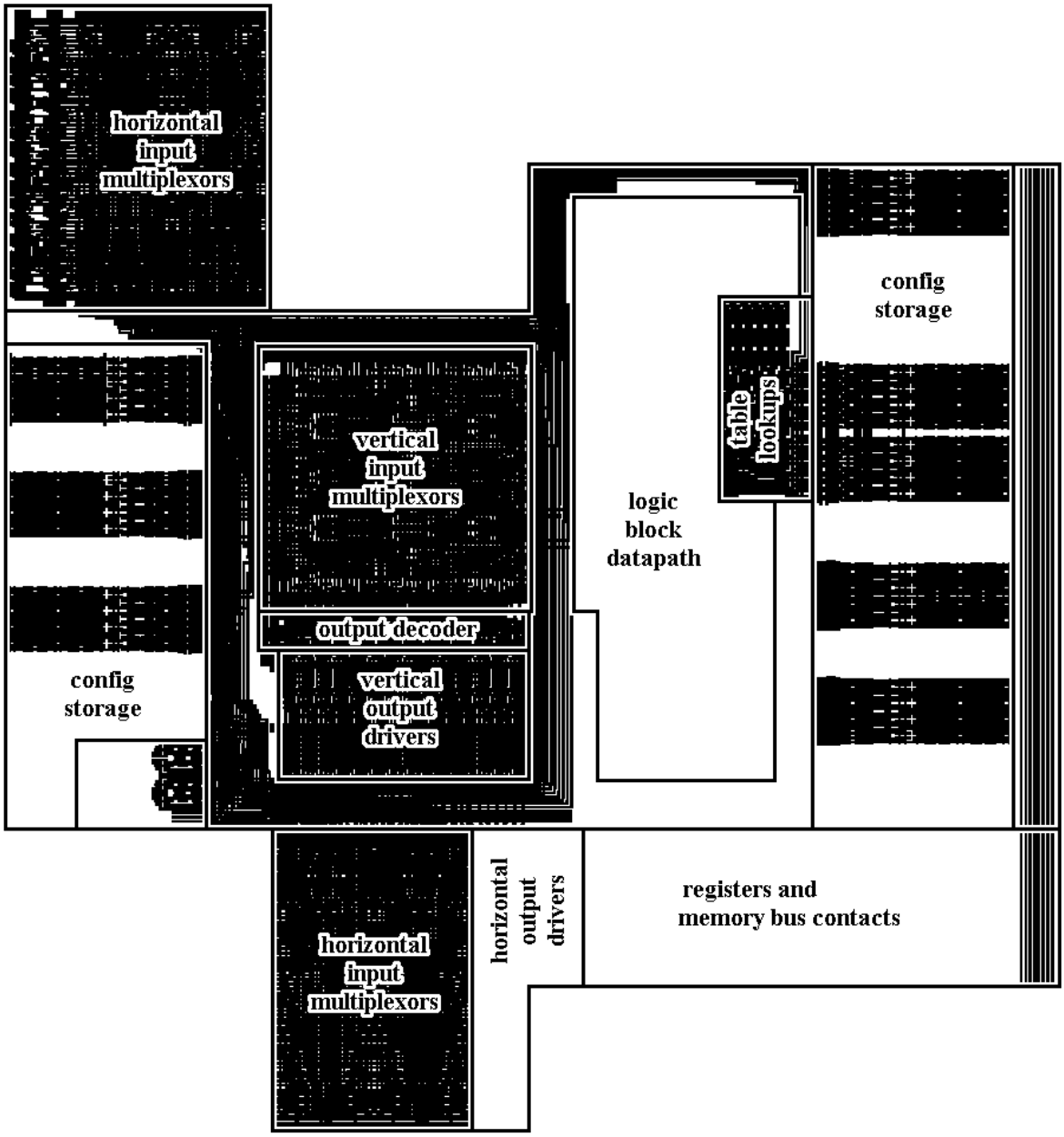


Figure 4.27: The relative space assumed for various logic block parts, based on detailed layout for the main density-sensitive components.

logic blocks such as the carry chain. The empty space within the configuration storage boxes are for controlling reading and writing of the cache; only the raw storage cells are mapped out in the figure. Lastly, the reason the horizontal output drivers have been given less space than the vertical ones is that there are fewer options for selecting a horizontal wire to drive. Only one of three H wire pairs can be driven by a logic block, and there are only four G wire pairs to choose from. Details of the workings of the horizontal wires are in the architecture manual in Appendix A.

According to the layout in Figure 4.27, the area corresponding to one logic block is 1225 λ vertically and 1565 λ horizontally, where 1 λ is half the drawn transistor gate length. For the 0.65 μm process, that equates to 0.40 mm by 0.51 mm, or a little more than 0.20 mm^2 per logic block, not counting additional components such as the memory interface.

4.3.6 Configuration storage and distribution

Configurations are loaded over the memory buses, using the full bandwidth available. Connections to the memory bus are made inside the box labeled "registers and memory bus contacts" in Figure 4.24 and lead in one direction to the data registers and in another to configuration storage. Figure 4.28 illustrates the path between the memory bus contacts and configuration storage. Configuration storage boxes of adjacent logic blocks sit back-to-back, and the memory bus channel between them is used to load part of the configuration for the block on the left and part for the block on the right.

Each memory bus channel is 8 bits wide (two bits for each of four memory buses), so 8 bits are loaded at a time over the channel. With four configuration contexts to store, the configuration cache is divided into pieces with four 8-bit bytes for the four contexts. As a complete configuration for a logic block is 64 bits, eight of these byte-wide pieces make up a complete cache for one logic block. Like any instruction cache, the configuration cache blocks are unidirectional: configuration bits enter from one side (from the memory bus) and exit out the other (toward the logic block).

The Garp implementation uses a standard 3-transistor dynamic cell for each cache bit. Figure 4.29 shows the design of a single bit-line from the cache, which is optimized primarily for size since quick access to a configuration is not a priority. At the output side is the active configuration register that drives the configuration bit into the logic block. The

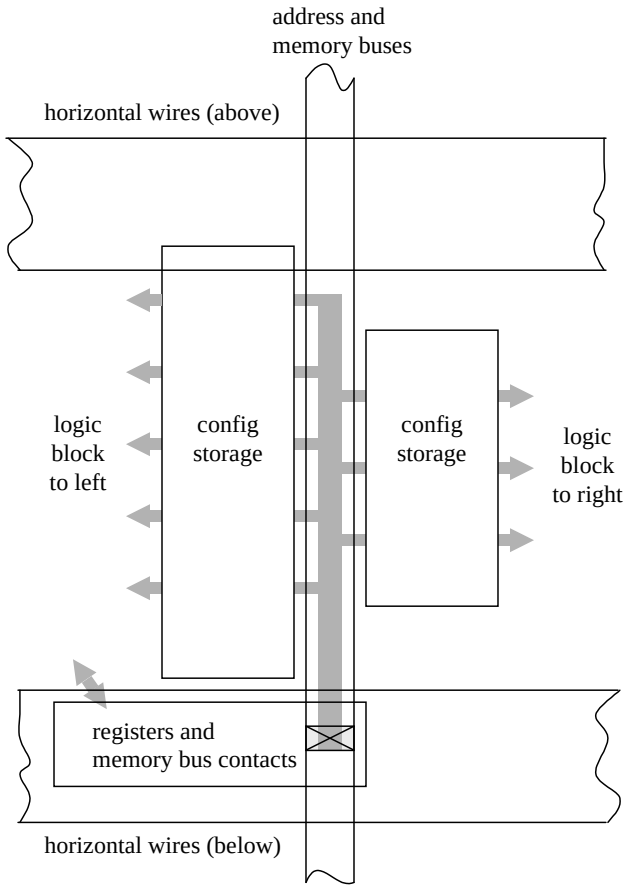


Figure 4.28: Contacts to memory buses, at the boundary between adjacent logic block tiles.

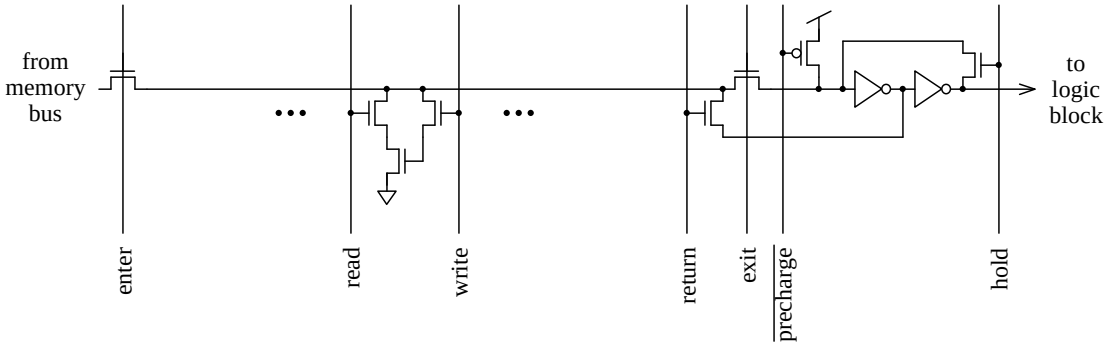


Figure 4.29: One bit-line in the configuration storage. With only four dynamic storage cells and no need for speed, sophisticated sense amp circuitry would be overkill.

cache control lines in the figure are intended to be governed by a minimalist, passive state machine that is manipulated by the control block at the end of the row. To simplify matters, the cache is read and written according to only a few set scripts. When configurations are read from outside memory, for instance, one configuration byte at a time is stored in sequence from over the memory buses. When a configuration is retrieved from the cache, all 64 bits are read simultaneously. In every case, all the logic blocks along a row participate in concert. No mechanism needs to exist for manipulating a single logic block's cache in isolation.

The demon of dynamic storage, of course, is retention. At the very least, the cache will have to be refreshed periodically. Cache refresh involves the following sequence:

1. suspend array execution by disabling register latching and all wire output drivers;
2. write the active configuration back to the cache (can be overlapped with step 1);
3. for each of the three other cached contexts, precharge the bit lines, read into the active configuration register, and write back to the cache;
4. precharge the bit lines and recall the suspended configuration into the active configuration register again;
5. wait for the configuration control lines to settle within the logic block; and finally
6. resume array exection.

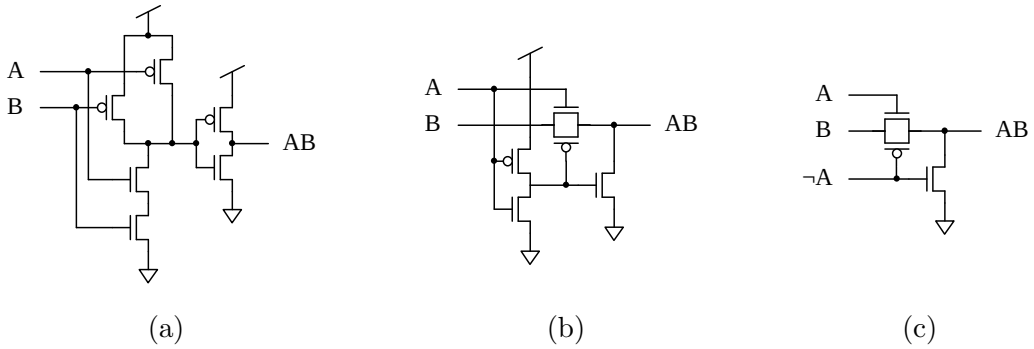
Each of these operations can occur in parallel for all 64 configuration bits in all of the logic blocks in the array. It should be pointed out that this refresh procedure depends once again on the ability to suspend execution of the Garp array for an arbitrary number of clock cycles, a feat not possible with ordinary FPGAs. The cache hardware could have been built without this requirement, but then another register separate from the active configuration register would have been needed, adding area to the configuration cache.

There is no problem refreshing the cache often enough to prevent loss from decay. Assuming as much as 1% of execution time is set aside for refreshes, a full refresh could easily be done every 50 μs, compared to milliseconds for a standard DRAM. The fact that there are so few stored cache bits per active configuration register bit (only 4:1) gives the configuration cache far more refresh bandwidth than is typical for DRAM.

Refresh can protect the dynamic storage from charge leakage, but it will still be susceptible to rare upsets from stray subatomic particles. The only way to detect such events is with redundancy. Flipped bits are certainly a rare occurrence; after all, mass market DRAM is regularly sold today without any redundancy protection and hardly anyone seems to notice the difference. But remember that configurations loaded into the Garp array are checked to ensure that no wire has more than one driver; if one of these bits were to change in the configuration, the results could be disastrous. Redundancy could therefore be used for the configuration bits that choose output wires to drive. The logic block would need to verify the consistency of the redundant coding and abort array execution, interrupting the processor, if an error is detected. The goal is merely to protect the hardware, not to allow the computation to complete as though nothing had happened. An alternative is to store the few bits that matter with static RAM instead of DRAM. It is likely this would require more area, but both methods could be tried and the smaller adopted.

As seen in the figures of logic block layout and especially in Figure 4.27, a logic block's active configuration is distributed through the block in dense rings around the logic block datapath and the vertical wire contacts. These bus-like tracks are optimized strictly for density, not speed; configuration bits are distributed in these tracks in polysilicon and the second metal layer, with contacts from either side made through the first metal layer as needed. In some instances, polysilicon leads straight to the gates of transistors being controlled by the configuration bit. Because so much of the configuration is distributed around the logic block in polysilicon wires, it has been assumed a configuration latched into the active configuration registers will need a couple of clock cycles to fully dissipate.

Some decoding of the Garp configuration is necessary within the logic block. Other than the input multiplexors and output drivers already covered, the amount of decoding required is minimal, and thus is most easily done within each logic block component as needed. As usual, the logic circuitry for decoding the configuration should be optimized for area at the expense of speed. Figure 4.30 presents, for example, alternative circuits for an AND gate created with a transmission gate that could be used in place of the more common full-complementary version. When made with the smallest possible transistors, such circuits can accomplish the necessary decoding fairly inobtrusively.



(a) (b) (c)

Figure 4.30: (a) A full complementary AND gate. (b) An AND gate made with a transmission gate. (c) A smaller version of (b) when the complement of one of the inputs is already available.

4.3.7 Logic block functions

The tentative layout in Figure 4.27 leaves space for the internal logic block functions previously summarized in Section 4.1.2 and documented in detail in Appendix A. Technically speaking, six different functions are supported, but by design the six functions can overlap some in their implementations. Figure 4.31 has a combined circuit with everything that must be fit into the reserved datapath portion of the logic block. Most of the boxes in the figure are fairly simple operations; many are nothing but multiplexors, in fact. (The purpose of all the pieces can be understood from the details in Appendix A.) At the top are the permutation boxes alluded to in Section 4.1.2. On the left is a four-way multiplexor implementing the select function and the "partial select" variant used for multiplication. Of the remainder, the largest pieces are the three-input table lookups and the carry chain.

To help gauge the area needed for the datapath, the table lookups have been layed out and included in Figure 4.27. Sixteen configuration bits provide the tables directly and are closely bound to the lookup hardware. Because the table lookups are essentially multiplexors, they are implemented here as binary trees of pass transistors just as the input multiplexors are. Beside the lookup tables, the main other complex part in the datapath is the carry chain, which can be expected to need an area a little less than that of the table lookups.

Aside from the sixteen table bits, eleven configuration bits control the operation of the datapath in a Garp logic block. Six of these bits are only needed for the permutation

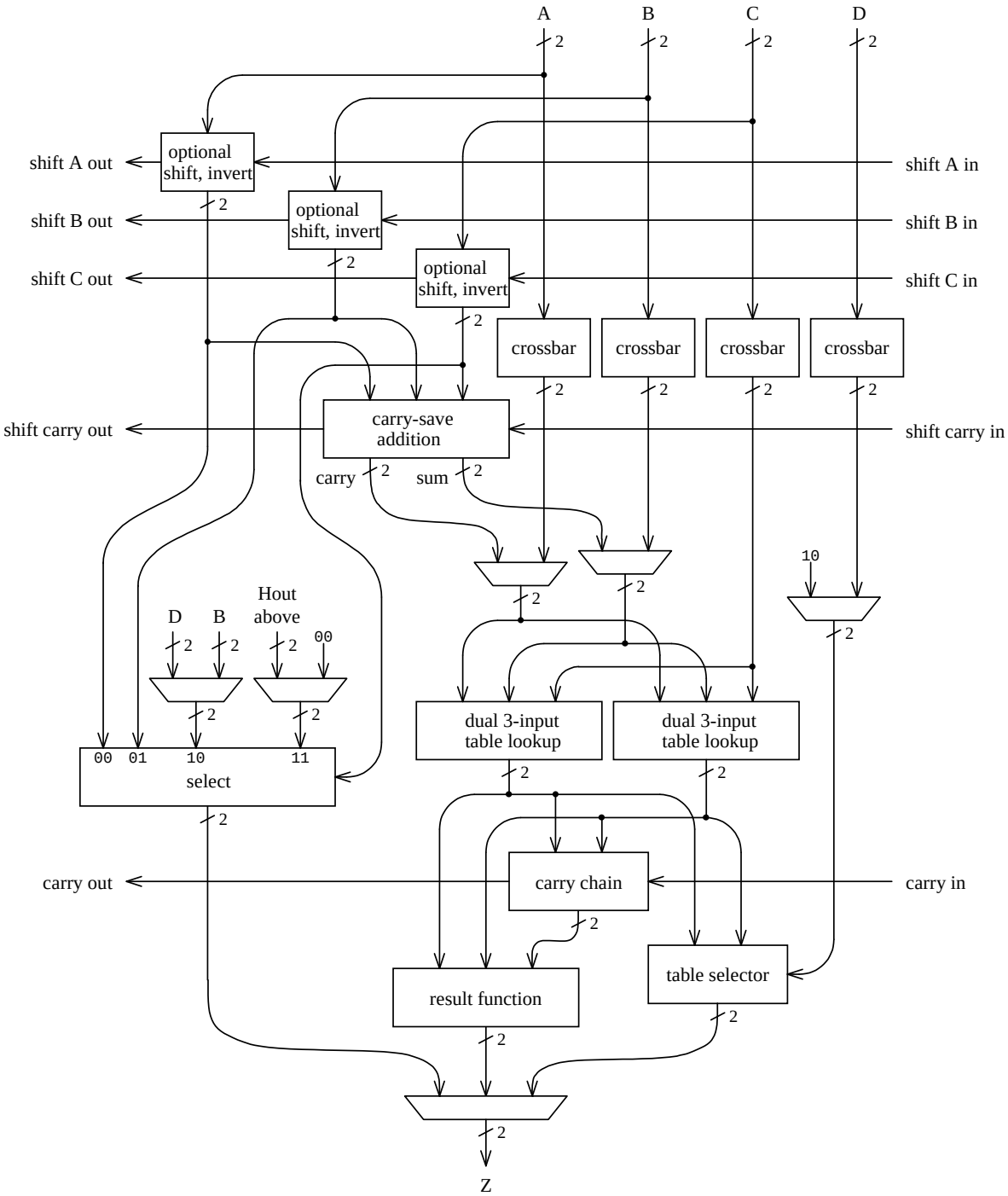


Figure 4.31: Complete contents of the logic block datapath. Of these boxes, the table lookups and carry chain are the most complex.

boxes; these have been brought to the top of the logic block datapath along with the inputs themselves in the layout of Figure 4.27. Routing and decoding of the remaining five control bits presents no real difficulty.

4.3.8 Speed, power, and area

In addition to the layout experiment, a circuit model for the critical path from one logic block to another was created and simulated to arrive at an estimate of the maximum clock speed for the array. As usual, the circuit model was also used to determine transistor sizes in the layout. The path modeled was: (1) from a register in one logic block, (2) onto a network wire, (3) over to a logic block the farthest distance allowed, (4) into an input multiplexor, (5) through the slowest function path, and (6) to a register in the destination logic block. Considering that most delay in FPGAs is through the wire network, the model was careful to include all parasitic capacitances for the network wires in addition to all transistors attached to every node along the path. Three separate critical paths had to be considered, corresponding to the three different sequences guaranteed to fit within one clock cycle in Section 4.1.4.

When the carry chain is used, it can be in the critical path. The carry chain itself was not modeled, but reliance was placed on the speed of carry chains in other devices created in similar process technologies. Interestingly, Hauck et al. did a detailed SPICE simulation of a 32-bit carry chain for reconfigurable hardware implemented in a 0.6 μm process and found the delay through the chain to be 6.1 ns [30]. However, their carry chain can invert the carry during propagation, even though that is never needed for the basic arithmetic operations and will slow down the carry chain noticeably. Since Garp does not allow the carry to be inverted, its carry chain is expected to take less than 5 ns to cover 23 logic blocks.

With these assumptions, simulation of the critical path circuit model has indicated that a 10 ns clock cycle, or 100 MHz clock, should not be too fast for the given 0.65 μm process.

From the same simulations, a rough estimate of power consumption can also be extrapolated for the array. This has been done by multiplying the worst-case power consumed by each circuit component by the number of such components in the entire array. The intention is not to get a highly accurate estimate but rather to disqualify entirely

unrealistic implementation techniques. The resulting estimate for power consumed by the array, not counting clock distribution, comes to about 4 W. Additional power would of course be needed for Garp's main processor and other parts, but the reconfigurable array is expected to be almost half the die area and perform much of the heaviest computing. Even considering the 4 W might be an underestimate, this seems within the right ballpark for the array compared to a typical high-performance desktop processor pulling power in the range of 15–20 W or higher.

Section 4.3.5 gave the area of a single logic block in the 0.65 μm process as 0.40 mm $\times$ 0.51 mm. Assuming 5% more in each dimension for the long wire buffers (Figure 4.23), and another 15% (the equivalent of almost five logic blocks rows) in the vertical dimension for the array memory interface, the total size of an array of 32×24 logic blocks comes to 15.5 mm $\times$ 12.8 mm or approximately 200 mm^2.

Chapter 5

Benchmarks and Statistics

This chapter evaluates the Garp architecture by benchmarking it against a standard Sun UltraSPARC 1/170 for a variety of applications. First, a hypothetical Garp implementation is defined that combines a Garp processor with a memory system identical to the UltraSPARC's. The software tools are then introduced that permit programs to be written and simulated for this hypothetical Garp. The suite of benchmarks is presented along with the performance of each system. The chapter ends with a collection of statistics for the array configurations used by the benchmarks, in case the information may be useful to future designs.

5.1 Hypothetical Garp

In an attempt to evaluate the Garp architecture, a hypothetical Garp has been compared against a Sun UltraSPARC 1/170. The UltraSPARC is a 4-way superscalar 64-bit processor running at 167 MHz, with 16 kB each of on-chip instruction and data caches. In addition to a floating-point unit, the processor supports Sun's VIS "graphics" instructions for small SIMD operations. The UltraSPARC is implemented at 3.3 V in a 0.5 μm process with four layers of metal, in a die size of 17.5×17.8 mm^2.

The hypothetical Garp has been constructed by removing the SPARC's superscalar integer and floating-point processing units from the UltraSPARC die and replacing them with a MIPS processor extended with Garp's reconfigurable array. Figure 5.1 shows die floorplans of the actual UltraSPARC and the proposed Garp derived from it. This surgery essentially puts a Garp on top of an UltraSPARC memory system. The new main

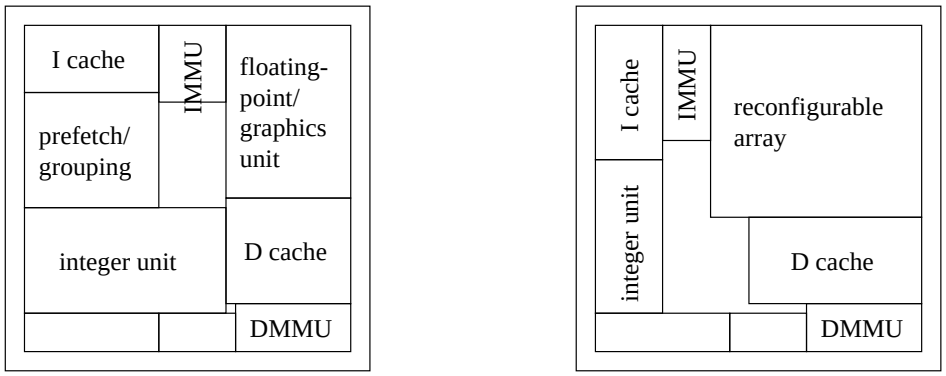


Figure 5.1: (a) Floorplan of the UltraSPARC die. (b) The hypothetical Garp die constructed in the same technology.

processor is a single-issue 32-bit MIPS-II, which is rather smaller and less powerful than the UltraSPARC's processing unit.

The previous chapter put the size of the Garp array in a 0.65 μm process as 15.5 mm $\times$ 12.8 mm, and its clock speed at approximately 100 MHz. Adjusting for the UltraSPARC's 0.5 μm process, it is believed the Garp array would scale to less than 12 mm$\times$ 10 mm and could be made to run at around 133 MHz. The additional fourth layer of metal in the UltraSPARC process provides some obvious room for error in the extrapolated array size. Another factor in our favor is the fact that layout in the UltraSPARC process is not bound by the "least common denominator" of the MOSIS scalable CMOS design rules, allowing improvements that also provide some room for error. The space allocated to the reconfigurable array in Figure 5.1 corresponds to this scaled-down size.

Removing the floating-point unit of course means that software that needs floating-point will be at a disadvantage on Garp. The situation is not as bad as it might seem at first, because the reconfigurable array can be configured to help with floating-point operations, as is testified for FPGAs by several sources [36, 49, 51, 68, 75]. Nevertheless, floating-point arithmetic is not as fast in the reconfigurable hardware, particularly for double-precision which is rather cramped in the small Garp array being considered here. The real intention for a future processor is not to eliminate the floating-point unit but to keep it as part of the main processor along with the reconfigurable array.

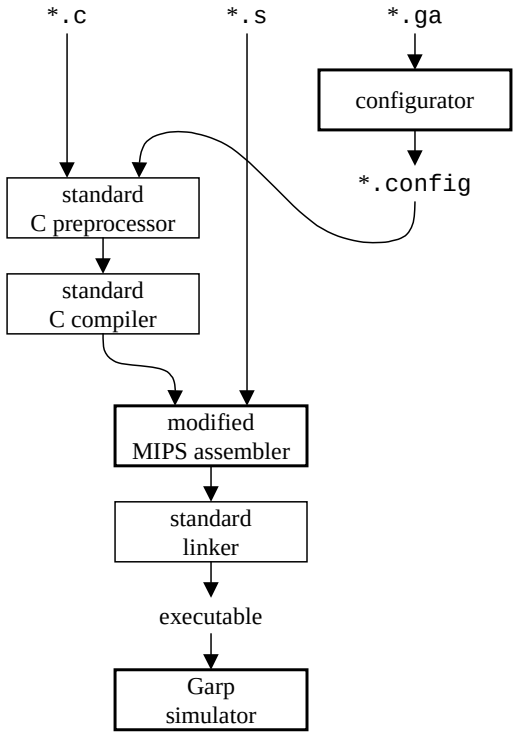


Figure 5.2: The Garp programming environment. New or modified tools are highlighted.

5.2 Software tools

Software tools have been created that make it possible to write programs for the hypothetical Garp and then simulate them with clock-cycle accuracy. The software path is summarized in Figure 5.2. Only two tools are substantially new: the *configurator* and the Garp simulator. The Garp assembler is merely a modified MIPS assembler.

An array configuration is coded in a "`.ga`" file in a simple textual language. This source is fed through a program called the *configurator* to generate a representation of the configuration as a collection of bits. For simplicity, the configurator creates a text file that can be used as an initializer for an integer array in a C program.

The only need for assembly language programming is to invoke the Garp instructions that interface with the reconfigurable array. Since the compiler being used is the GNU C Compiler (`gcc`), the same could be accomplished with inline '`asm`' statements.

5.2.1 The configurator

The configurator accepts a human-readable description of a configuration and converts it to the binary representation accepted by the reconfigurable array. The input language to the configurator is more akin to an assembly language than to either a high-level language or the typical FPGA netlist. Data and operations must be placed explicitly within rows and columns by the programmer. A configuration is defined as a collection of rows, with each row containing within it logic blocks in specific columns. The basic syntax is

```
row optional-row-name:
{
  column-number(s): logic-block-settings;
  ...
}
...
```

A feel for the permissible *logic-block-settings* is probably easiest to impart by example. The following specifies a complete configuration for adding three 32-bit values in columns 4–19 (the middle 16 columns of the array):

```
row .a: --Row 0
{
--Send Z registers onto vertical wires.
 4-19:  A(Zreg),function(A),bufferZ,Vout(Z);
--Send D registers onto horizontal wires below.
 4-19:  D(Dreg),bufferD,Hout(D);
}

row :   --Row 1
{
 4-19:  D(Dreg),bufferD;
--Add D registers and values from row 0; latch result in Z registers.
 4:     shiftzeroin;
 4-19:  A(.a),B(above),C(Dreg),
         add3,U(carry^sum),V(sum),result(U^K),bufferZ;
}
```

In this configuration, the values in the Z and D registers of row 0 and in the D registers of row 1 are added together and their sum stored in the Z registers of row 1. Column 4 is the least significant (rightmost) of the 16 columns. Row names (such as `.a`) must begin with a period to distinguish them syntactically.

The '`A(.a)`' field in the second row specifies that the A input for those logic blocks is to come from the row labeled '`.a`'—in this case, the first row. To obtain a connection through vertical wires, the programmer merely names the source needed for a logic block input. It is the responsibility of the configurator to choose specific vertical wires for making the connections. The A inputs in row 1 of the example are thus taken over vertical wires from row 0. The rather different syntax '`B(above)`', on the other hand, indicates that the B inputs are to be read from row 0 over the *horizontal* wires between the two rows. Each logic block can drive one output onto vertical wires and one onto horizontal wires.

For the example given, the output from the configurator is the text

```
{
  0x00000002,
  0x00000000, 0x00000008, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x0A00000E, 0xAAAA1C1E, 0x0A00000E, 0xAAAA1C1E,
  0x0A00000E, 0xAAAA1C1E, 0x0A00000E, 0xAAAA1C1E, 0x0A00000E, 0xAAAA1C1E,
   ... Eleven lines elided ...
  0x7C940C0E, 0x66CCF800, 0x7C940C0E, 0x66CCD800, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000,
}
```

which is suitable for initializing an array of 32-bit integers in C.

5.2.2 Linking a configuration into a C program

Garp's reconfigurable array is only used within the time consuming parts of a program where it can be usefully employed. The remainder of the program is written in C, is compiled with an ordinary C compiler, and is executed on the main processor without reference to the reconfigurable array. A configuration thus has to be linked into an ordinary C program.

Continuing with the example above, if the configurator output is in a file called '`add3.config`', the C code

```
uint32_t config_add3[] =
#include "add3.config"
;
```

suffices to initialize a C array `config_add3` with the desired configuration bits. This makes the configuration accessible to the program; however, it will still have to be loaded and activated in the array to actually do something. Since a configuration can only be invoked

with the new Garp-specific instructions that are unknown to the compiler, some assembly language programming is required.

The following Garp assembly code loads and executes the same example (refer back to Table 4.3):

```
add3:   la v0,config_add3  # Load v0 with pointer to config_add3 array.
        gaconf v0          # Load configuration.
        mtga a0,$z0        # Copy three operands to array, ...
        mtga a1,$d0
        mtga a2,$d1,2      # And step array 2 clock cycles.
        mfga v0,$z1        # Copy result back from array.
        j ra               # Return from subroutine.
```

The names `v0`, `a0`, `a1`, `a2`, and `ra` refer to ordinary MIPS registers; `la` is the MIPS "load address" instruction. The symbols `$z0` and `$d0` denote the Z and D registers of array row 0, while `$z1` and `$d1` are the same for row 1. The MIPS subroutine calling convention passes the first three subroutine arguments in registers `a0`, `a1`, and `a2`, with the subroutine return value being passed back in register `v0`.

With this assembly language stub, a program can add any three integers a, b, and c using the reconfigurable array by executing the ordinary subroutine call `add3(`a`,`b`,`c`)`. The `add3` subroutine first loads the proper configuration into the array (or switches to it, if it is already in the array's configuration cache). It then copies its three arguments into array registers, steps the array for 2 cycles to perform the addition, reads the sum back into `v0`, and returns.

Of course this example involves too much overhead. In practice, the array would be used for something substantial that could not just as easily be done in the main processor.

5.2.3 The simulator

A hardware implementation of Garp does not exist, so Garp programs must be executed on a simulator. The simulator loads and executes standard MIPS executables. Operating system calls are forwarded to the environment in which the simulator is running.

Outside of operating system calls, the simulator does its best to count true clock cycles. The main processor is assumed to be only a simple single-issue MIPS. Interlocks that stall instructions are observed and stall cycles counted. Memory caches are also modeled, so that cache miss stalls can be added in. Simulation of the configuration cache and

memory queues has been made to mimic real protocols at essentially a register-transfer level. Although the simulator is unlikely to be cycle-for-cycle identical with an actual implementation, its cycle counts should be realistic.

As far as practical, the full behavior of the UltraSPARC memory system has been duplicated, from the perspectives of both the main Garp processor and the reconfigurable array. The UltraSPARC 1 has the usual separate first-level instruction and data caches, each 16 kB, and a 512-kB unified second-level cache. Further details about the UltraSPARC memory system can be gleaned from Gwennap [26, 27].

5.3 Hand-coded benchmarks

Several benchmark applications have been coded by hand for Garp and their performance compared against the same applications on the UltraSPARC. Table 5.1 lists the benchmarks and summarizes the speed difference between Garp and the UltraSPARC for selected input sizes. The applications chosen are intended to cover a range of behavior and are not just those for which large speedups could be easily predicted. The first set, DES, MD5, and SHA, are cryptography-related applications: DES is a standard encryption algorithm, while MD5 and SHA are hash functions used for digital signatures. The other benchmarks are two common operations on images, a typical sorting problem, and a couple of string functions from the standard C library. All of the benchmark applications are covered in more detail in subsequent sections.

Cryptography applications have been chosen precisely because cryptography is claimed to be the sort of application that gives fine-grained reconfigurable hardware a chance to excel over clunky, instruction-fed processors. Three of the four cryptography benchmarks (all but ECB-mode DES) have inherent feedback loops that make latency critical. As mentioned later, such feedback is a frequent characteristic of cryptographic algorithms that can limit the parallelism available.

In contrast, DES encryption in ECB mode is an example of an application with tremendous data parallelism. The two image operations also fall into this category; all should be opportunities to showcase the reconfigurable array's ability to sustain many operations simultaneously.

Sorting is a traditional challenge problem. In theory highly parallel, its need to shuffle data in unpredictable patterns puts great stress on the memory system. By cleverly

benchmark	input size	167 MHz SPARC	133 MHz Garp	ratio
DES encrypt, CBC mode	1 MB	350 ms	91 ms	3.8
DES encrypt, ECB mode	1 MB	350 ms	18.7 ms	19
MD5 hash	1 MB	99 ms	55 ms	1.8
SHA hash	1 MB	103 ms	37 ms	2.8
Floyd-Steinberg dither of color image	900 kB	167 ms	9.8 ms	17
median filter of grey-scale image	300 kB	108 ms	2.5 ms	43
sort of $\langle \text{key}, \text{value} \rangle$ pairs	512 kB	63 ms	27 ms	2.4
strlen	1 kB	9.5 μs	0.94 μs	10
strcpy	1 kB	9.7 μs	1.19 μs	5.7
strlen	16 bytes	0.36 μs	0.23 μs	1.5
strcpy	16 bytes	0.46 μs	0.24 μs	1.7

Table 5.1: Synopsis of hand-coded benchmarks. The times for Garp are obtained from program simulation. Except for `strlen` and `strcpy`, Garp execution times include the cost of loading configurations into the cache from external DRAM.

complicating the work of the processor, the number of uncorrelated memory accesses to slow memory can be reduced.

The standard C string functions are examples of workhorse routines that see use over a wide range of data sizes—from strings only a few characters long to some over a kilobyte. In a production Garp system, one would like to use the reconfigurable hardware to accelerate standard library functions, fully transparently to calling programs. For small data sizes, however, overheads could swamp any potential speedups.

In the benchmarks, small data is generally assumed to be in the cache when that is a reasonable conjecture. Since the primary (L1) data cache is 16 kB, it is reasonable to suppose that strings less than 1 kB are still in this cache. Likewise, the `strlen` and `strcpy` functions themselves are assumed to still be in the processor's first-level instruction cache when the functions are called. By the same token, Garp configurations for these functions are assumed to be in the configuration cache when the functions are called on Garp. No such assumptions have applied to the other benchmarks, however. Aside from `strlen` and `strcpy`, quoted Garp execution times include the cost of loading configurations entirely from external DRAM.

The process of choosing the parts of the benchmarks to implement in the Garp reconfigurable hardware, and the job of designing the configurations, has all been done by hand for these examples. In no case has any attempt been made to do anything productive

with the main processor while the Garp array is operating. Speedup numbers are thus free of any multiprocessing effects.

5.3.1 Data Encryption Standard (DES)

One of the most important encryption algorithms over the last 20 years has been the Data Encryption Standard, or DES [67]. DES is a good application for reconfigurable hardware because normal processors have trouble implementing it efficiently. Implementations of DES in FPGAs have been reported by Tse et al. [72], by Miyamori and Olukotum for their REMARC board [57], and by Kean and Duncan [42].

DES encrypts 64 bits of data at a time, using a 56-bit key. Each group of 64 bits is run through an "obfuscation loop" sixteen times, and it is in this loop that DES spends most of its time. The 64 bits are first divided into two 32-bit quantities R_{-1} and R_0, and then the following steps are repeated for $i = 1$ up to 16 (see Figure 5.3):

1. Extract eight 6-bit subsequences from R_{i-1}, and XOR these with 48 bits from the encryption key.
2. Apply each of the resulting 6-bit values as an index into an "S-box" table of 4-bit values. (Each S-box is unique and approximates a random function.)
3. Perform a permutation on the 32 bits of S-box results. (This permutation is always the same.)
4. XOR the permuted result with the older R_{i-2} to form the new R_i.

After sixteen iterations, the encrypted 64-bit output is taken from R_{15} and R_{16}.

Software implementations of DES invariably implement the S-boxes as table lookups requiring a read from memory for each S-box evaluation. All told, $16 \times 8 = 128$ table-lookup memory reads are needed for each 64 bits encrypted. On the other hand, good software implementations can avoid the final 32-bit permutation by pre-permuting the S-box table entries. This makes the table entries a full 32 bits in size, but the eight S-box outputs need only be ORed together before being combined with R_{i-2}.

Unlike software, any sufficiently large reconfigurable hardware can implement this algorithm directly. The S-box table lookups and all the bit permutations can be done quickly and in parallel, without reference to external memory. A configuration in Garp's array needs only five cycles per inner loop iteration.

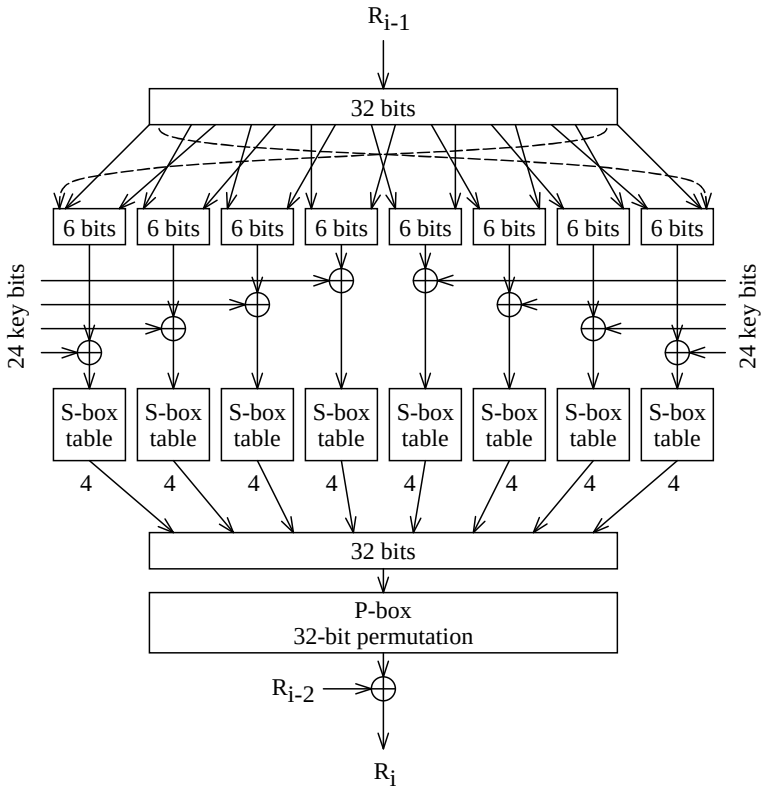


Figure 5.3: One iteration of the inner loop of DES. The $\oplus$ symbols indicate XOR operations.

Each iteration of the DES loop uses a different 48 bits from the key in the XOR in step 1 above. Thus in addition to the main loop, the 56 key bits have to be constantly permuted. By design, the permutation sequence is the same for every 64-bit group run through the obfuscation loop, so software implementations precompute the key permutations once, off-line, and subsequently read them from an array as each 64-bit group is encrypted. For the Garp configuration, it is easy enough to dedicate part of the array for permuting the key on the fly.

There are two common modes in which DES is employed: ECB (electronic codebook) and CBC (cipher block chaining). ECB mode encrypts each 64 bits of a message separately, whereas CBC mode uses the results from encrypting all the previous 64-bit groups to help encrypt the next 64 bits. The *chaining* property of CBC mode makes the encryption more secure, but introduces a strict data dependency from one 64-bit encryption to the next. With ECB mode, the encryption of every 64-bit piece of a message can all be done in parallel.

Simulation indicates that Garp would be 3.8 times faster than the UltraSPARC in CBC mode, where each 64-bit piece is encrypted separately in sequence. For ECB mode, the DES configuration's pipeline can with very little modification work on five encryptions simultaneously, making the total speedup $3.8 \times 5 = 19$ times faster than the UltraSPARC.

5.3.2 MD5 and SHA hashes

Another important type of cryptographic operation is the *one-way hash function*. An ordinary hash function maps an arbitrarily long string of bits to a small, fixed-size *hash value*. What distinguishes a *one-way* hash function is the difficulty of finding two different source strings that the function maps to the same hash result. One-way hash functions are used for generating digital signatures of documents. Two major one-way hash functions have been tried on Garp, MD5 and SHA [67]. MD5 has also been done for the NAPA1000 by Arnold [3].

MD5 treats a source string as a sequence of 512-bit blocks, updating the overall hash value as each block is processed in turn. The 128-bit hash is accumulated in four 32-bit variables named a, b, c, and d. Updating the hash for a single block requires 64 steps of the form depicted in Figure 5.4. Each individual step incorporates 32 bits from the current 512-bit block, but these 32-bit pieces are not all read sequentially from the block. As there

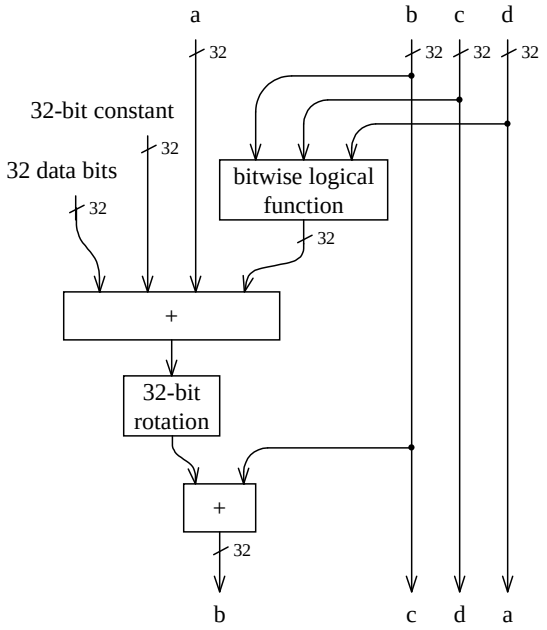


Figure 5.4: One iteration of the inner loop of the MD5 hash. A different 32-bit constant and a different rotation distance are applied each iteration. The bitwise logical function changes every 16 iterations.

are only sixteen 32-bit words in the entire block, each source word gets used exactly four times.

Each of the 64 steps also requires a unique 32-bit constant; and a rotation operation changes from step to step as well. Every sixteen steps, the bitwise logical function being used changes. Four different logical functions are thus employed in all.

To implement MD5 in Garp, four configurations have been created, corresponding to the four logical functions. Each configuration physically contains four steps, to take advantage of the fact that the rotation distances repeat every four steps within each group of sixteen steps. The rotations can therefore be "hard-wired" for the proper distances. The outputs of the fourth step are looped back to the inputs of the first, so that four iterations through the configuration equal sixteen algorithm steps. After four iterations, the next configuration is loaded from the cache, for another sixteen steps with the next bitwise logical function.

Because the 32-bit source pieces are not all read sequentially (and because there is no room to keep them in the array), demand memory accesses are used to load the

correct word at the proper time. For their part, the 64 constants are supplied sequentially in memory and read through one of the memory queues.

The latency through the four steps in one of these configurations is 17 clock cycles, averaging to $4\frac{1}{4}$ cycles per algorithm step. However, repeatedly changing configurations and resetting the memory queue (every 64 steps) consumes additional clock cycles. Simulation shows that, for large blocks, Garp is only about 1.8 times as fast as the UltraSPARC on the MD5 hash.

The SHA hash function is similar in style to MD5, differing only in all the details. SHA generates a 160-bit hash value using 80 algorithm steps for each source block. Like MD5, four different configurations are used in a cycle on Garp, corresponding to four different bitwise logical functions. For SHA, Garp is about 2.8 times faster than the UltraSPARC over large inputs.

5.3.3 Image dithering

Two image processing applications have also been implemented on Garp, the first being the dithering of a full-color 480×640 image to a fixed palette of fewer than 256 colors. The input image stores three bytes per pixel, for a total of 256 levels each of red, green, and blue for each pixel. The target palette in this case is the so-called "Web palette" used by Web browsers such as Netscape Navigator. This palette contains $216 = 6^3$ colors in an orthogonal arrangement with six levels each of red, green, and blue. The dithering algorithm employed is Floyd-Steinberg error diffusion, which is essentially the standard algorithm for this task [73].

The dithering of an image proceeds from top to bottom in scan-line order. Dithering each pixel involves the following two steps:

1. Find the color in the target palette closest to the given pixel color.

2. Find the color error introduced by using a not-quite-correct color, and distribute this error to neighboring pixels by adjusting the neighbors' colors.

Figure 5.5 shows how a pixel's color error is distributed (diffused) to its neighbors in the Floyd-Steinberg algorithm.

Finding the closest target color is a matter of reducing the source image's 256 levels each of red, green, and blue to the six levels each in the target palette. This is accomplished

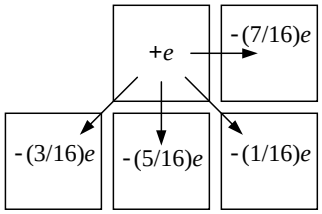


Figure 5.5: Floyd-Steinberg error diffusion. An image is dithered from top to bottom in scan order. Replacing a pixel's original color with the closest available color results in a color error e. This error gets pushed to four as-yet-uncommitted neighboring pixels by adjusting the original colors at those pixels. The process repeats with the next pixel to the right.

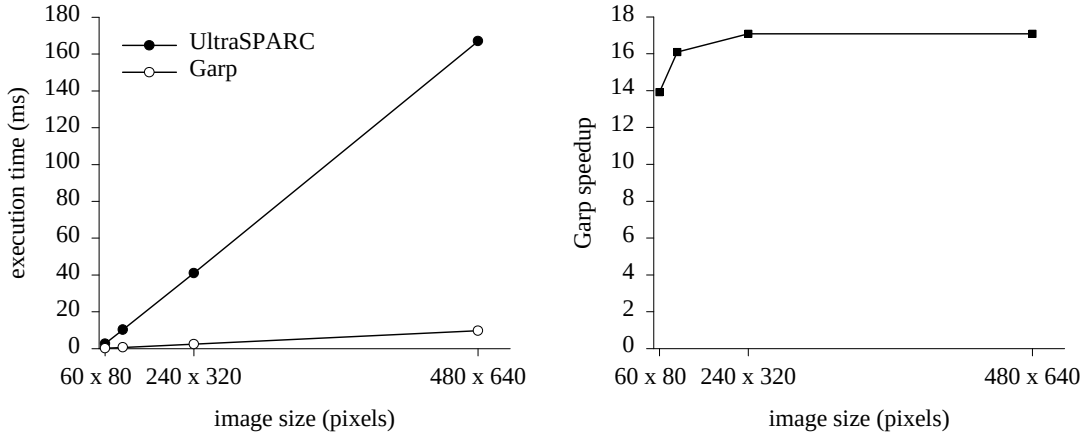


Figure 5.6: Execution times and speedups for image dithering. The two smallest image sizes tested are 60×80 and 120×160 pixels.

by dividing each color component by $(256-1)/(6-1) = 51$ and rounding. Calculating the error requires multiplying the result back by 51 and subtracting. Distributing the error involves four scales and additions to neighboring pixels as seen in the figure. To save some work, errors diffused to a single pixel by multiple of its neighbors are added together before being added into the destination pixel.

For this application, Garp has been found to be as much as 17 times faster than the UltraSPARC on large images. Figure 5.6 graphs the Garp speedups over the UltraSPARC for a range of image sizes. Garp's advantage comes from its ability to manipulate 8-bit quantities more adeptly. On both Garp and the UltraSPARC, the division by 51 is done by multiplying by an approximation to 1/51. Multiplies are implemented on both in terms of shifts and adds, which Garp can do fairly efficiently.

5.3.4 Image median filter

The other image operation implemented is a median filter, which can be used to correct outlying "spots" of noise in an image. Variants on the median filter have been done by Abbott et al. [1] for the Splash 2 board and by Box [8] for a custom FPGA board. For this benchmark, the images are grey-scale instead of color, although median filters can be done on color images, too.

The median filter replaces each pixel from the original image with the median of the nine pixels in the 3×3 neighborhood around that pixel. The median of a set of values is the value in the middle when the set is sorted into increasing order. Sometimes the median of the nine pixels is already the one in the center, but usually it will be a neighboring pixel with nearly the same value. A single pixel with an extreme white or black value inconsistent with its neighbors will never be chosen for the final image, which is how the filter eliminates such spots.

An important point is that, although it is easiest to think of the median operation in terms of sorting, sorting the nine pixel values to find the median is more work than strictly necessary. Knowing the median is less information than knowing the full sort of the nine values; therefore, it should not be shocking that the median value can be obtained with less work than doing a full sort. An algorithm for finding the median of exactly nine values is presented in Figure 5.7. The algorithm is based on two primitive operations: (1) sorting three values, and (2) finding the median of three values. Three values a, b, and c can be

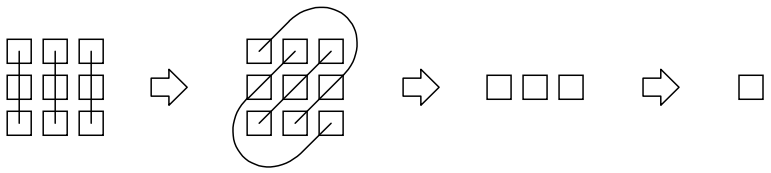


Figure 5.7: An algorithm for finding the median of nine values arranged in a 3×3 grid. First, individually sort the three columns from least to greatest value. Then find the medians of each of the three diagonals, wrapping around as shown. The median of those three medians is the median of the original nine.

sorted with the following short sequence of compare-and-exchanges:

```
if ( c < a ) Exchange a and c;
if ( b < a ) Exchange a and b;
if ( c < b ) Exchange b and c;
```

Finding the median of a, b, and c is even easier (assuming C notation):

```
if ( c < a ) Exchange a and c;
median = ( b < a ) ? a : ( c < b ) ? c : b;
```

As the figure explains, three of the 3-element sorting operations followed by four of the 3-element median operations suffice to extract the median of the original nine values.

A reasonable implementation of the median filter processes the image in scan-line order, so that after finding the median around one pixel, it next works on finding the median for its neighbor one to the right. Observe in Figure 5.7 that much of the work of the first step—sorting three pixels in three columns—will already have been done for the previous pixel on the left. Only the rightmost column is new. Therefore, it pays to save the sort of two of the columns from one pixel to the next. This reduces the work for each output pixel to just *one* 3-element sorting operation followed by four 3-element median operations.

A Garp implementation of the median filter algorithm turns out to be more than 40 times faster than the UltraSPARC on large images. Speedup numbers for different sizes of images are graphed in Figure 5.8. Compare-and-exchange operations are not something the typical processor does especially well, and that is the bulk of the median filter. The Garp configuration, in contrast, has a pipeline fed by three memory queues reading three scan lines simultaneously, one pixel per cycle each. At the other end of the pipeline, output pixels are written at the same rate of one per cycle (only with some finessing each time the production wraps around the edge of the image).

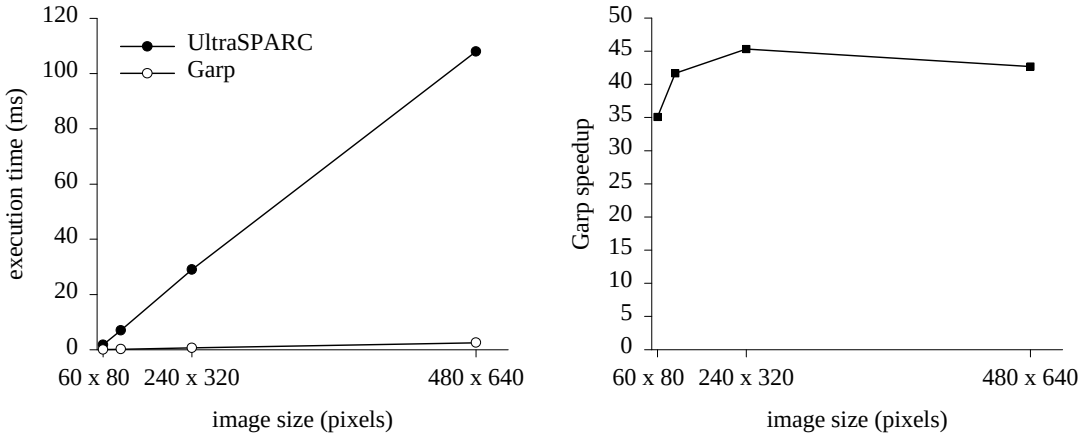


Figure 5.8: Execution times and speedups for the image median filter benchmark. Again, the two smallest image sizes tested are 60×80 and 120×160 pixels.

5.3.5 Sorting

Sorting is yet another of the Garp benchmarks. This benchmark orders a sequence of $\langle \text{key}, \text{value} \rangle$ pairs according to their 32-bit keys. The corresponding 32-bit values are not interpreted but must be correctly permuted with the keys.

The best implementation of sorting on Garp varies depending on the number of elements to sort. For less than 10,000 elements, a radix-sorting algorithm is used involving two buffers. One pass of the radix sort reads elements from one buffer and writes to the other; the next pass then works in the opposite direction for the next radix digit. With a radix size of four bits, eight passes are needed to fully sort the 32-bit keys.

A single radix sort pass is implemented with two Garp configurations. The first reads the entire source buffer and counts the number of elements with each digit value (one of $0, \ldots, 15$) at the digit position for that pass. Once it is known exactly how many elements there are for each digit value, the second configuration reads the elements a second time and writes them to the output buffer in their proper places. Using eight such passes, the radix sort algorithm can sort any number of elements in strictly linear time. The first configuration reads two elements every clock cycle, and the second processes one element per cycle, for a total of $1\frac{1}{2}$ cycles per element, per pass (not counting some inevitable memory stalls). With eight passes, that comes to only 12 clock cycles per element for the entire radix sort.

However, if the two radix sort buffers do not fit within the second-level (L2) cache,

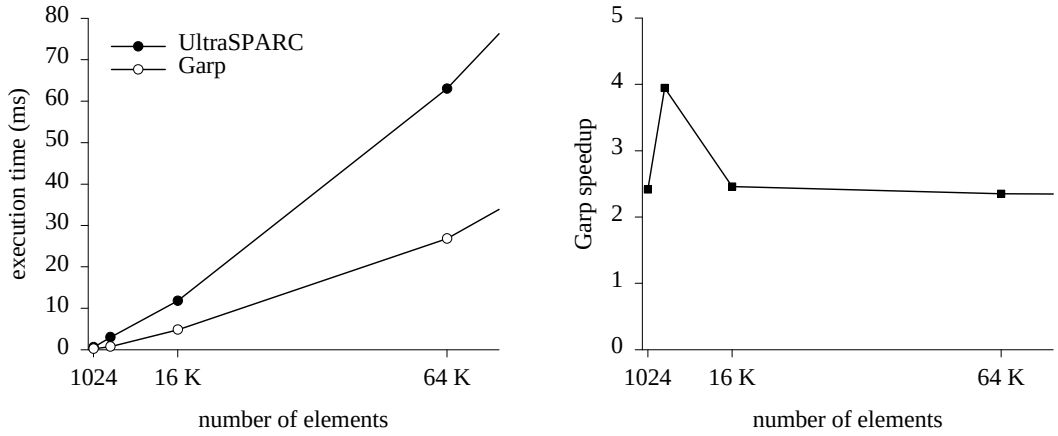


Figure 5.9: Execution times and speedups for sorting. Five tests have been run, with 1024, 4096, 16,384, 65,536, and 1,048,576 (2^{20}) $\langle \text{key}, \text{value} \rangle$ elements. At a million elements, which is off the scale of both graphs, the Garp speedup is 1.89 over the UltraSPARC.

cache misses will add tremendously to the total time. The radix sort's mandatory eight passes—each reading all of the elements twice—then become a serious liability. To avoid this cost, the Garp implementation uses radix sort on only 10,000 elements (80 kB in size) at a time. The input set is divided into groups of 10,000, and each group is sorted using the radix sort algorithm. The sorted groups are then merged together with mergesort-style passes. Each merge takes eight separate sorted streams as input and outputs a single sorted stream eight times as long. Unlike the radix sort passes, the merge passes are expected to overflow even the 512 kB second-level cache, thus forcing reads and writes ultimately to go to external DRAM. Up to 80,000 elements can be sorted this way with only a single expensive merge pass, and up to 640,000 elements with two merge passes through all the elements.

The merge pass is performed by an array configuration that reads irregularly from eight streams and writes sequentially to one merged stream. After each output element is written to the merged stream, the next element from the same input stream must be read before another output element can be chosen. The full latency between writing output elements is nine clock cycles, not including the time to service any cache misses. Cache misses occur frequently but not on every read. Because a cache line is larger than one element, each cache miss brings more than one element into the cache, and thus a miss does not occur for the subsequent elements in the same cache line. For the output stream,

Garp's memory queue hardware is used to buffer the merged stream into fewer, larger memory stores.

The radix sort and mergesort techniques are generally too complex for the UltraSPARC, which does better with a simple quicksort. Attempts to do something more sophisticated almost invariably lose more time executing the additional instructions than is saved by some clever algorithm. As with previous benchmarks, Figure 5.9 shows the difference in execution times between Garp and the UltraSPARC for different numbers of elements.

The bump in the speedup graph is due to cache limitations. With an unbounded L2 cache, speedups would continue to rise beyond a factor of four for sorts of more than 4096 elements. However, by 16,384 elements, the effects of cache misses are being felt; and execution times for larger sizes are dominated by DRAM access latencies.

5.3.6 Library functions `strlen` and `strcpy`

The last two benchmarks are the `strlen` and `strcpy` functions from the standard C library. The `strlen` function is fairly easy to implement. The Garp configuration for this function has a pipeline that, at one end, accepts sixteen new characters every clock cycle, and at the other accumulates a count of the string's length. The configuration must search for a null character in each bundle of sixteen and cut short the count as soon as one is found. The pipeline for doing this is four cycles long (plus four cycles of memory latency to feed the pipeline).

The `strcpy` function is a little more complex, since `strcpy` must write out the string at the same time—up to, but not one byte beyond, the terminating null character. For the middle part of a string, the `strcpy` configuration has a three-cycle pipeline that reads and writes sixteen characters on alternating clock cycles. After the end of the string is found, additional parts of the configuration write the remainder of the string, four characters at a time at first, then one character at a time if necessary.

Figures 5.10 and 5.11 have the usual graphs of execution times and speedups. Note that Garp's performance on small-length strings is not substantially better than that of the UltraSPARC. For strings of length 4, Garp is only 10% faster than the UltraSPARC for `strlen`, and 22% for `strcpy`. This increases to factors of 12 and 6, respectively, for longer strings.

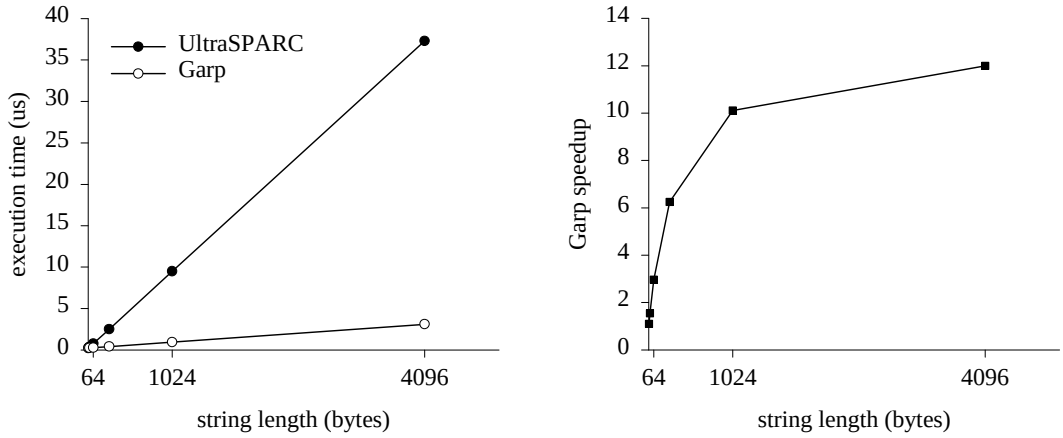


Figure 5.10: Execution times and speedups for the `strlen` function. Strings of length 4, 16, 64, 256, 1024, and 4096 characters have been tested.

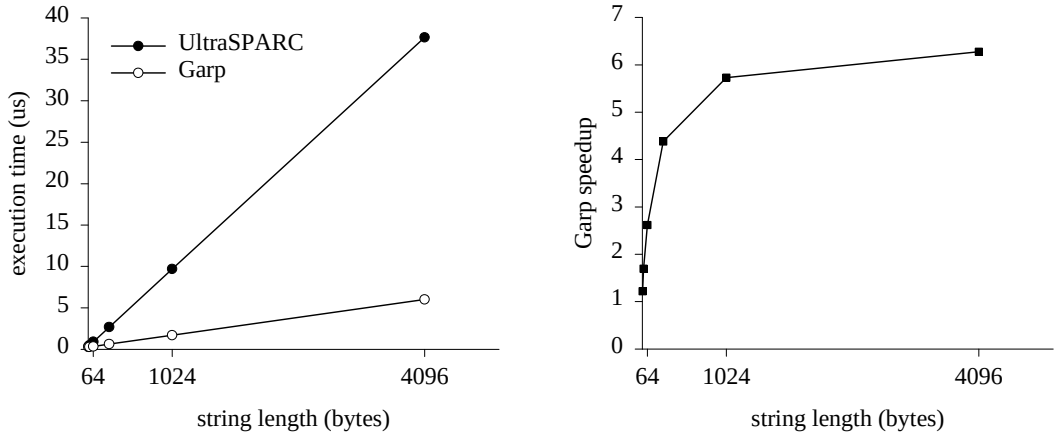


Figure 5.11: Execution times and speedups for the `strcpy` function. The same string lengths have been tested as for `strlen` in Figure 5.10.

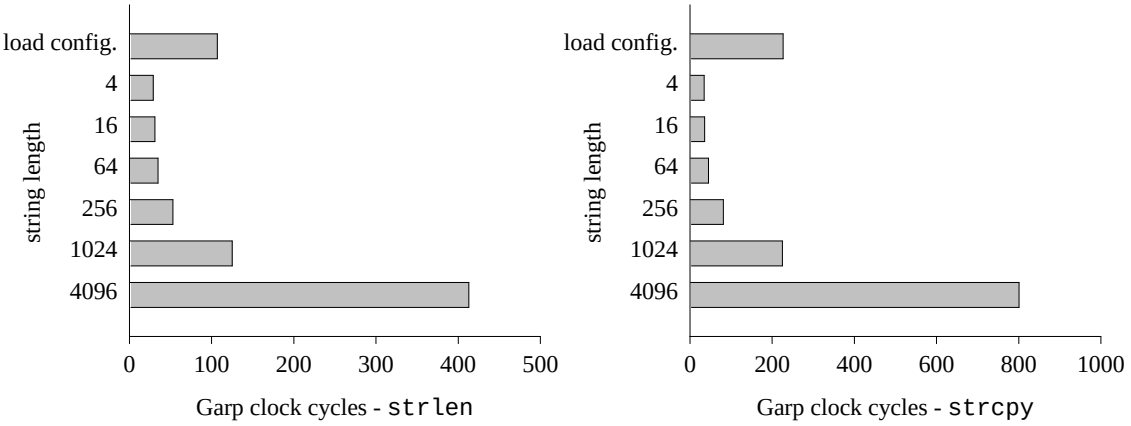


Figure 5.12: The time to bring the `strlen` or `strcpy` configuration in from DRAM compared to the time to execute the function on strings of various lengths.

The Garp hardware supports reads and writes of sixteen bytes at a time without requiring that the memory address be aligned on any special boundary. However, if a 2^k-byte access is not aligned on a 2^k-byte boundary, Garp breaks the access into two parts, introducing an extra stall cycle as necessary. The Garp execution times in Figure 5.10 and 5.11 assume no particular string alignments, and thus already include these extra cycles. If the source and destination strings happen to be aligned on sixteen-byte boundaries, Garp speedups on long strings exceed a factor of 17 for `strlen` and a factor of 9 for `strcpy`.

As noted earlier, for these functions (and these functions only), all data and function code is assumed to be resident in the first-level caches when the string functions are called. That applies not only to the instruction and data L1 caches, but also to the configuration cache in the reconfigurable array. In the case that the `strlen` or `strcpy` configuration is not already cached, the time to load the configuration from external DRAM (or from the L2 cache) could negate any advantage from using the reconfigurable array, depending in part on how many times the function is called after the configuration is brought into the cache. Figure 5.12 graphically compares the time required to load the configuration versus the time to execute the function for strings of various lengths. As the graphs show, configuration loading time is not negligible; but the surprising thing might be that it is not actually worse. The efforts to minimize configuration encoding size and maximize the bandwidth from memory in Garp keep the configuration cache miss penalty to a manageable size.

While Figure 5.12 is instructive, the more relevant question is how much config-

		number of calls to break even	
		`strlen`	`strcpy`
string length	4	37	30
	16	7	10
	64	2	4
	256	1	1

Table 5.2: The number of calls to `strlen` or `strcpy` needed to cover the initial configuration loading time and achieve parity with the UltraSPARC.

uration loading undermines Garp's speed compared to the UltraSPARC. Table 5.2 looks at this question from the point of view of the number of function calls that must be made before Garp fully pays down the cost of loading the configuration. The table suggests that for a typical average string length of eight characters, 20 calls or so should be enough to cover the cost of loading the configuration from memory and thus break even relative to the UltraSPARC.

5.3.7 Benchmark review

A sampling of the benchmark test cases are brought together in Table 5.3, sorted approximately by speedup. The table also identifies the immediate obstacle to achieving greater speedup in each cases. For the top three entries, the size of the Garp array is currently the limiting factor, meaning that a larger array would make it possible to push the speedup numbers higher. For long string lengths, the `strlen` and `strcpy` functions are limited instead by Garp's memory bandwidth. Doubling the available memory bandwidth would approximately double the speedup that could be achieved for these functions. As made clear earlier, the need to arbitrarily permute the contents of memory is primarily what limits the sorting benchmark. The remaining cases are constrained by the latencies of loop-carried dependencies, or by the greater significance of function overhead when operating on only small amounts of data.

A line in Table 5.3 divides the benchmark cases into two groups, with the ones above the line being examples of applications with abundant data parallelism that is easily exploited, and the ones below the line not having this property. The tentative conclusion to draw from these results is that, while there can be an advantage to using custom circuits to better implement non-parallelizeable applications, the real goldmine lies in applications with accessible parallelism. In fact, the majority of FPGA applications quoted—genome

benchmark	speedup	limiting factor
image median filter, 480 × 640	43	array size
image dither, 480 × 640	17	array size
DES encrypt, ECB mode, 1 MB	19	array size
strlen, 4 kB	12	memory bandwidth
strcpy, 4 kB	6.3	memory bandwidth
DES encrypt, CBC mode, 1 MB	3.8	latency
sort, 4096 elements	3.9	irregular memory accesses
sort, over 1 million elements	1.9	irregular memory accesses
SHA hash, 1 MB	2.8	latency
MD5 hash, 1 MB	1.8	latency
strlen, 16 bytes	1.5	overhead
strcpy, 16 bytes	1.7	overhead

Table 5.3: A representative set of benchmark test cases, sorted approximately by speedup, with the factors limiting further improvements.

matching, image filtering, military target recognition, graphics rendering, neural networks, etc.—are excessively data-parallel. Although there has not been time to try them all, many of these applications should presumably work well on Garp, too. In contrast, the non-parallel applications seem to be limited to speedups in the low single digits, sometimes reaching as high as 4.

5.4 Configuration statistics

This section presents numerous statistics on the Garp array configurations used in the benchmarks. Table 5.4 names the ten configurations covered and also gives their sizes in array rows and numbers of logic blocks. The four MD5 configurations (Section 5.3.2) are nearly identical, and so only one of them has been analyzed here. Likewise for the SHA hash. The configuration for DES in ECB mode is not exactly the same as the one for CBC mode, but, again, there is enough similarity that only the CBC-mode configuration is considered. The other benchmarks have only one configuration apiece, with the exception of the sorting benchmark which, as explained previously, uses three: two for the radix sort pass, and one for the mergesort pass.

benchmark	configuration name		number of array rows	number of logic blocks
DES encrypt, CBC mode	DES-CBC		24	552
MD5 hash	MD5	(1 of 4)	28	644
SHA hash	SHA	(1 of 4)	24	552
image dither	Dither		19	437
image median filter	Med-filter		12	276
sort	Radix-1	(counting digits)	24	552
	Radix-2	(copying elements)	21	483
	Merge		32	736
`strlen`	`strlen`		6	138
`strcpy`	`strcpy`		16	368

Table 5.4: The names and sizes of ten configurations from the benchmarks.

5.4.1 Functional density

For each configuration, Table 5.5 gives the fraction of logic blocks configured for any purpose, and also the percentage use of major logic block subsections. (It may be helpful to refer back to Figures 4.4 and 4.31 for the structure of a Garp logic block.) Most of the categories in the table (*inputs*, *functions*, and so on) are subdivided further in later tables.

Every logic block has four physical inputs, and each contributes to the *inputs* utilization percentage in the table depending on whether that input has an effect on the outputs for that logic block. If a block is configured in such a way that the value of an input has no impact on the logic block's results, then that input is considered unused. The D input is also considered active if the D output is used—either latched in the D register or driven onto an output wire (Figure 4.4). Active inputs do not necessarily come from other logic blocks across the wire network but might be registers or constants within the logic block. Table 5.6 breaks out the utilization of each of the four logic block inputs, A, B, C, and D, separately.

The next two categories, *crossbars* and *shift-inverts*, are the permutation boxes described in Section 4.1.2 and visible in Figure 4.31. There are four physical crossbar boxes and three shift/invert boxes. Because Garp does not permit the crossbars and shift/invert boxes to both be in use at the same time in a logic block, the average of these two categories can be no more than 50%. Which is operative depends on the logic block's function mode. A permutation box is considered used (or active) if the input connected to it is active and

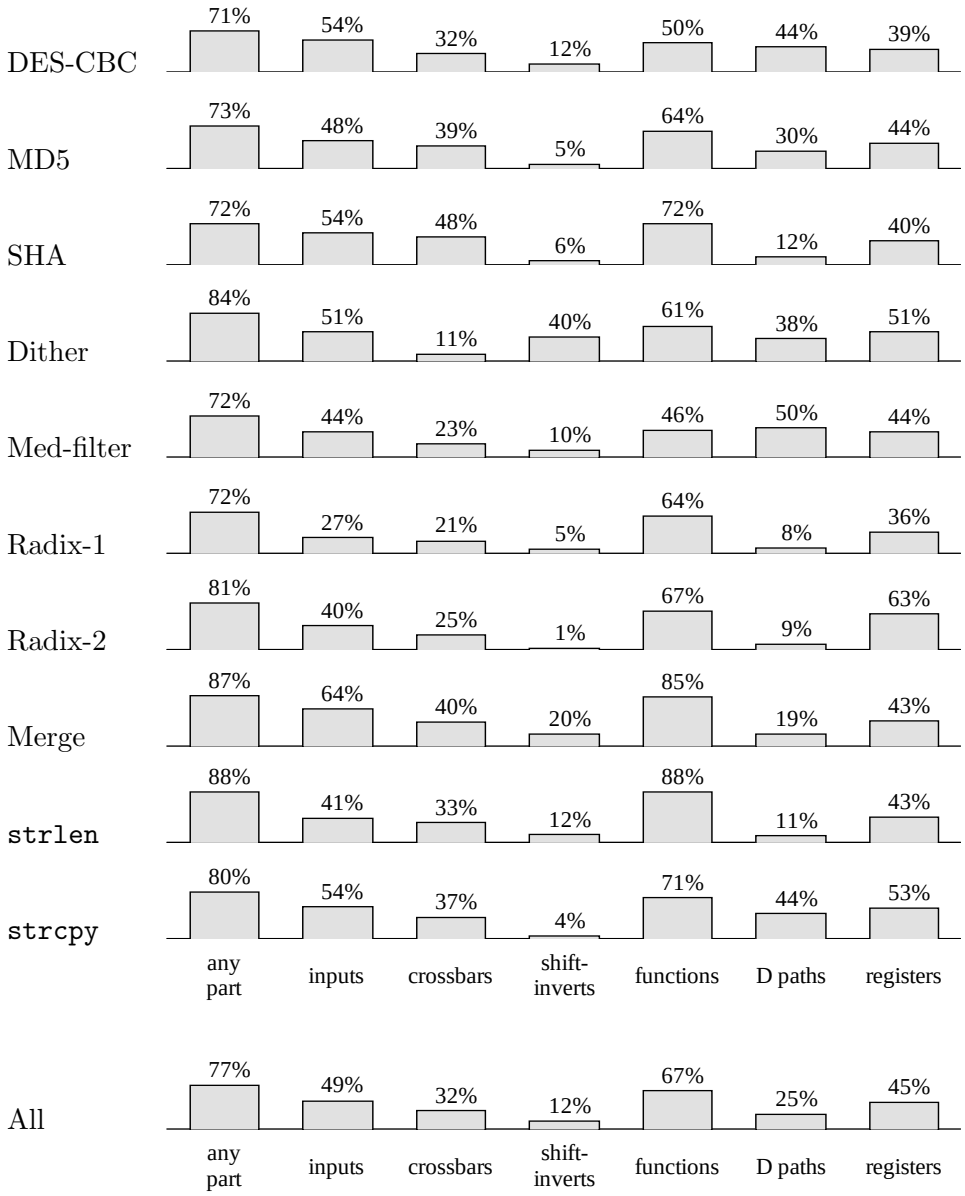


Table 5.5: Utilization of the major logic blocks parts.

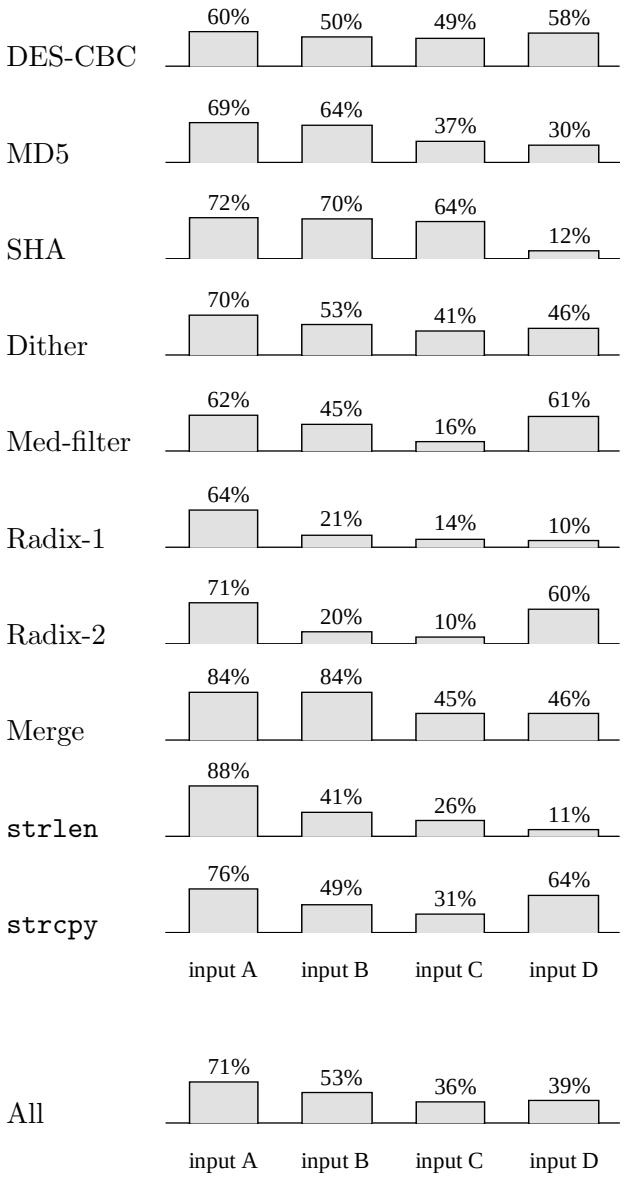


Table 5.6: Utilization of each of the four logic block inputs tabulated separately.

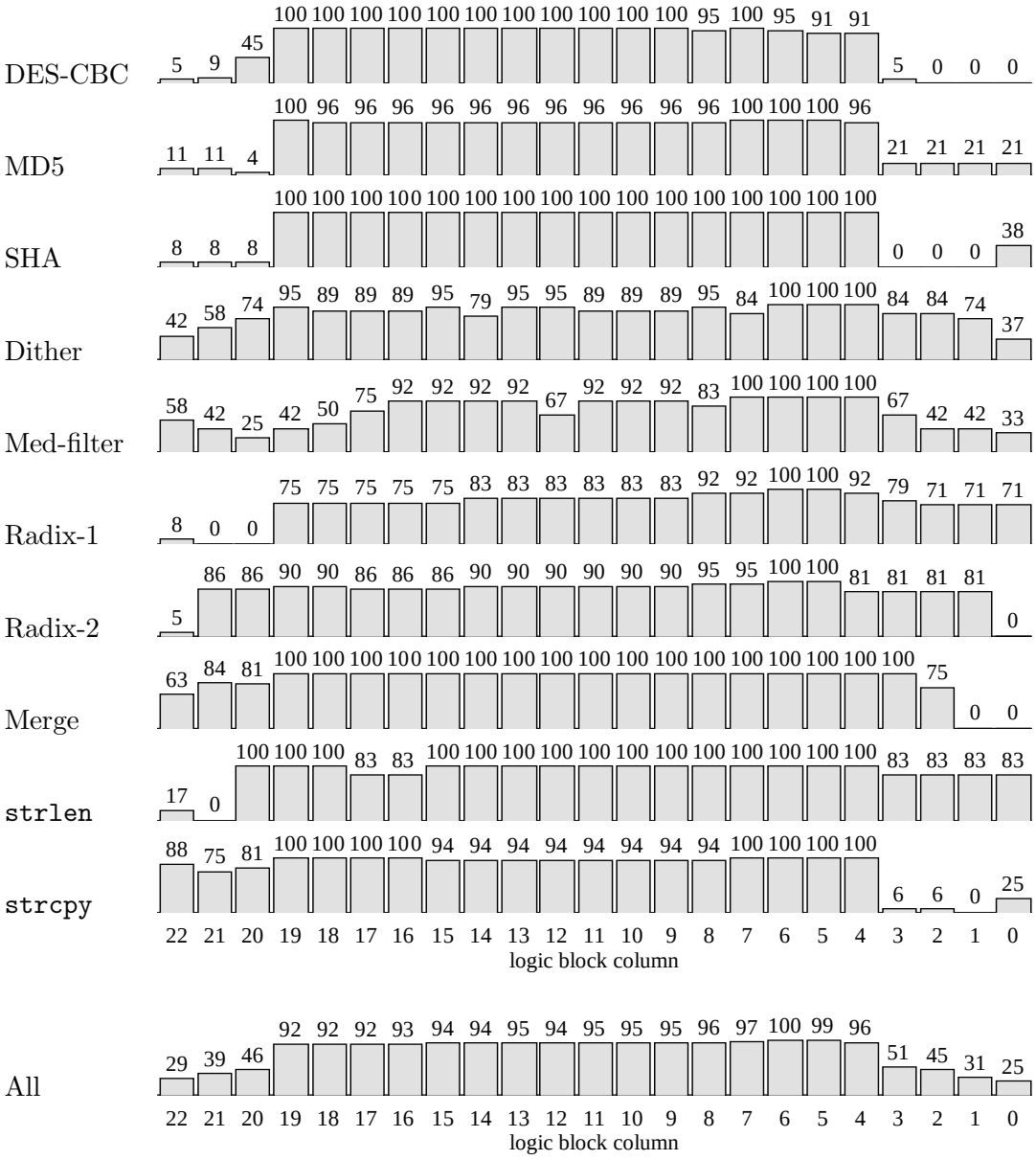


Table 5.7: Logic block utilization by array column. Columns 4 through 19 are the middle 16 logic blocks of the array, aligned with the memory bus.

the input passes through that permutation box as part of the selected function mode for that logic block. Active permutation boxes do not necessarily perturb the input but may pass the value through as given and still be counted. All that matters here is that they had an *opportunity* to perturb the input value.

In the *functions* column, a logic block is generally counted if the function Z output is used somewhere. However, a logic block configured as a simple table lookup that merely passes one of the inputs unchanged (thus ignoring the other inputs) is not considered to have a useful function and is not counted.

The D *paths* entry counts only those cases for which the D output is driven onto an output wire. In most instances, the D path is used to make a connection from one wire to another, since the Garp wire network has no way to connect wires otherwise.

Lastly, the *registers* category reports the utilization of the Z and D registers, which may participate in the circuit or be bypassed (Figure 4.4).

Table 5.7 divides out overall logic block utilization (the *any part* category of Table 5.5) for the 23 columns of blocks in the reconfigurable array. The center section of the array in columns 4 through 19 is aligned with the middle 32 bits of the memory bus and thus is usually the core of any datapath configured in the array. The extra logic blocks—three on the left and four on the right—are typically used for control signals; although in configurations layed out by hand such as these are, they sometimes participate in the computation, too. The table clearly shows that the side logic blocks are not employed as consistently as those in the middle sixteen columns. Nevertheless, it is safe to say their existence contributes to so many of the numbers reaching up to 100% in the middle columns. From experience, not having the side blocks would make it impossible to pack the main datapath as tightly, almost surely resulting in a decrease in functional density in the array overall.

5.4.2 Logic block inputs

Among the inputs actively in use, Tables 5.8 and 5.9 break down the sources of the inputs by type. The previous section defined what counts as an active input. Besides the obvious option of reading from a nearby network wire, any of the four inputs (A, B, C, or D) can be taken directly from one of the two registers in the logic block, or can be configured as a constant. Note that a third of the inputs (33%) come from one of these

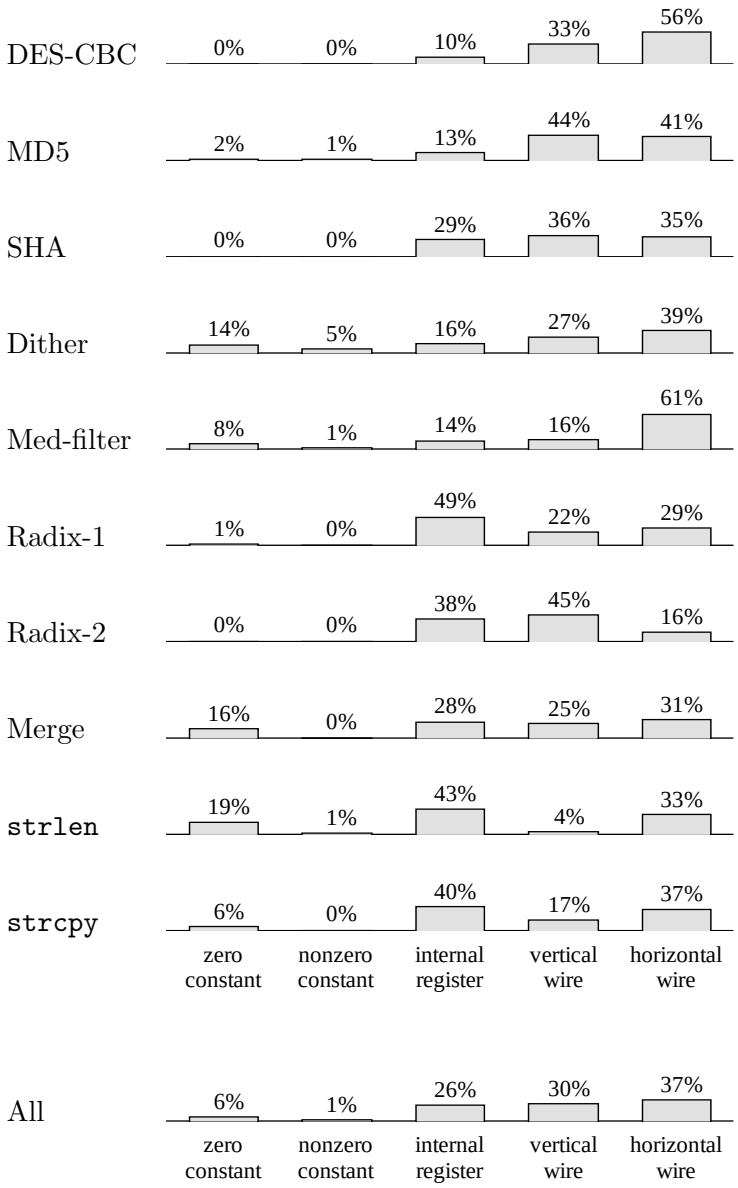


Table 5.8: Distribution of input sources among active inputs. All rows add up to 100%.

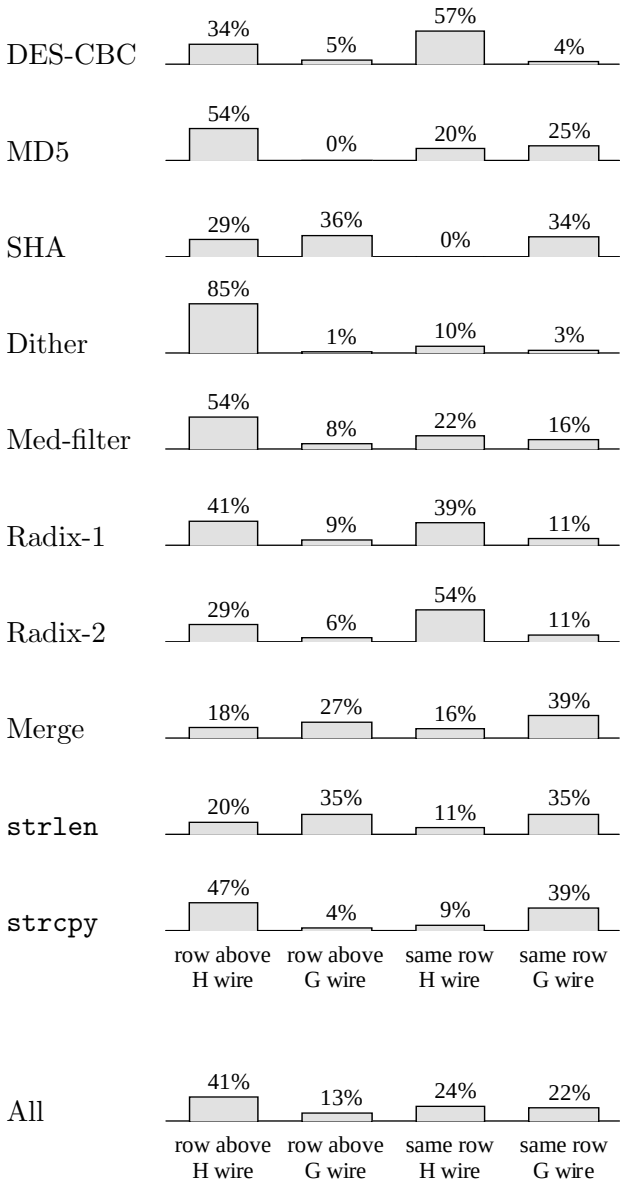


Table 5.9: Further subdivision of the horizontal wire inputs from the previous table.

sources and not from a wire.

For inputs read over a horizontal wire, Table 5.9 further distinguishes whether the wire is a local (H) or a global (G) wire, and whether it is from the row above or along the same row. (The different types of horizontal wires were explained in Section 4.1.3.) Interesting to note is that almost two-thirds (64%) of the signals read from a horizontal wire come from the row above and only a third are from the same row.

5.4.3 Logic block functions

For logic blocks with a useful function, Table 5.10 breaks down the percentage use of each function mode (recall Section 4.1.2 and Table 4.2). Partial select mode is never used because none of the chosen benchmarks performs a multiply of two variable values. The only benchmark requiring any multiplication at all is image dithering, which explains why its configuration is the only one to make much use of triple-add mode. Perhaps not surprisingly, the cryptography applications make more use of the table-lookup functions than the other benchmarks.

Table 5.11 shows how the permutation boxes have been configured for the different applications. Section 5.4.1 specified what permutation boxes are considered active and are thus included in the table. Although it is almost never used, the reverse crossbar function comes for free in the hardware given the other three options (straight, high-bit duplicated, low-bit duplicated). This should be even more obvious for the combined shift-and-invert operation. Actually, all of the shift/invert variants are important for multiplications by various constants, despite the fact that the combined shift-and-invert case seems not to have been needed for the multiplies done by the image dithering benchmark.

5.4.4 Granularity

Table 5.12 gives the results of an experiment to find the apparent bit widths of operations encoded in the configurations. Configurations have been searched (mechanically) for sequences of similarly configured logic blocks that could be considered multi-bit operations, each layed out along a row. The table expresses the results in terms of the percentages of logic blocks within the perceived multi-bit operations of each size, ranging from 2 to 36 bits wide. Only even widths are possible because the Garp array itself has a granularity of 2 bits.

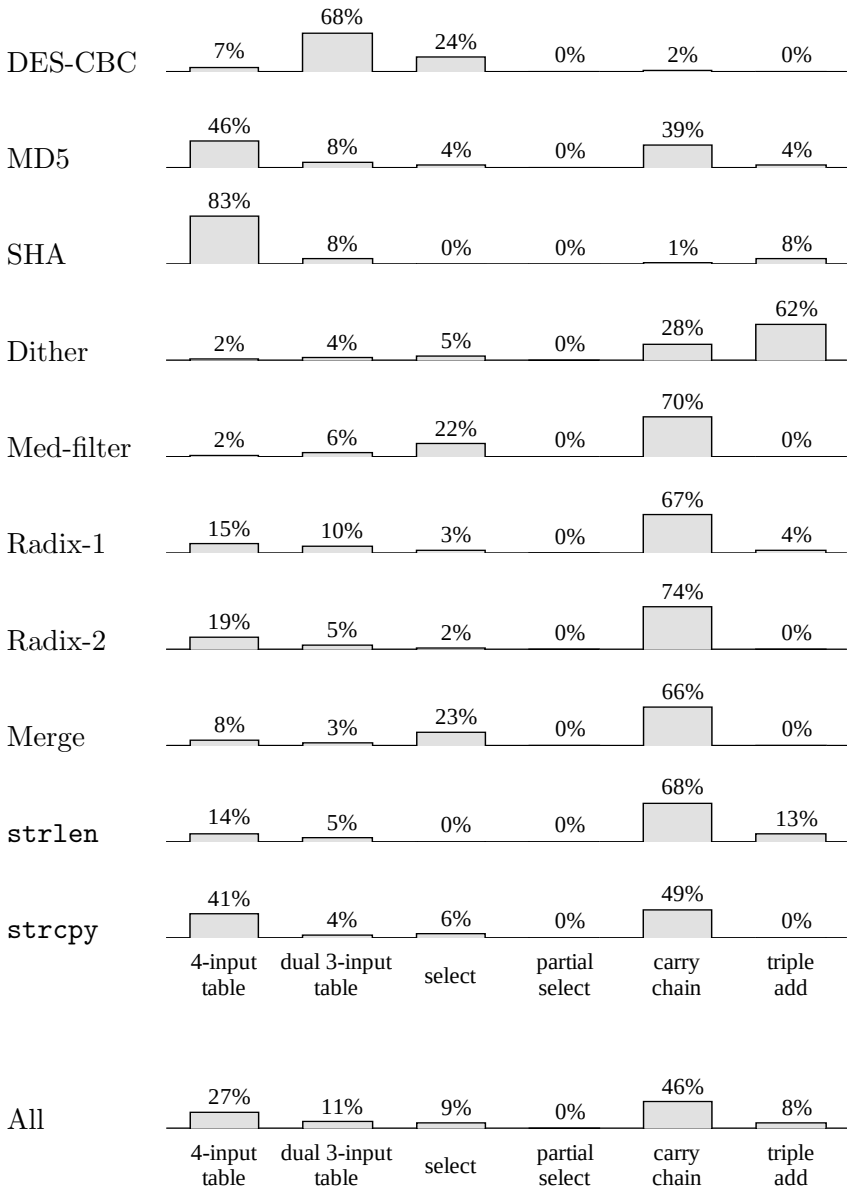


Table 5.10: Distribution of logic block function modes. Only logic blocks with a useful function are counted, so all rows add up to 100%.

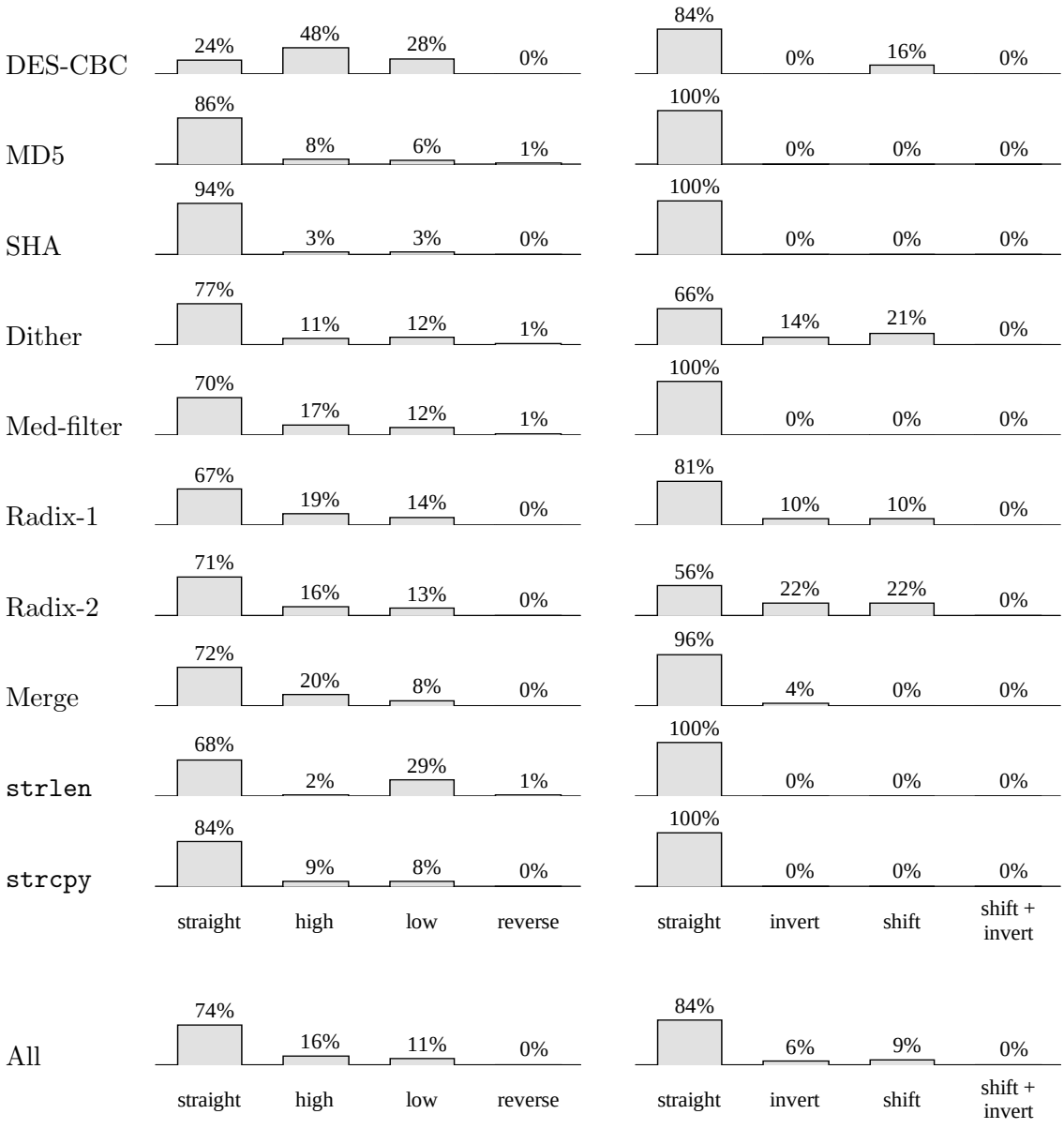


Table 5.11: Distribution of permutation box functions. Only active crossbar and shift/invert boxes are counted, as specified in Section 5.4.1.

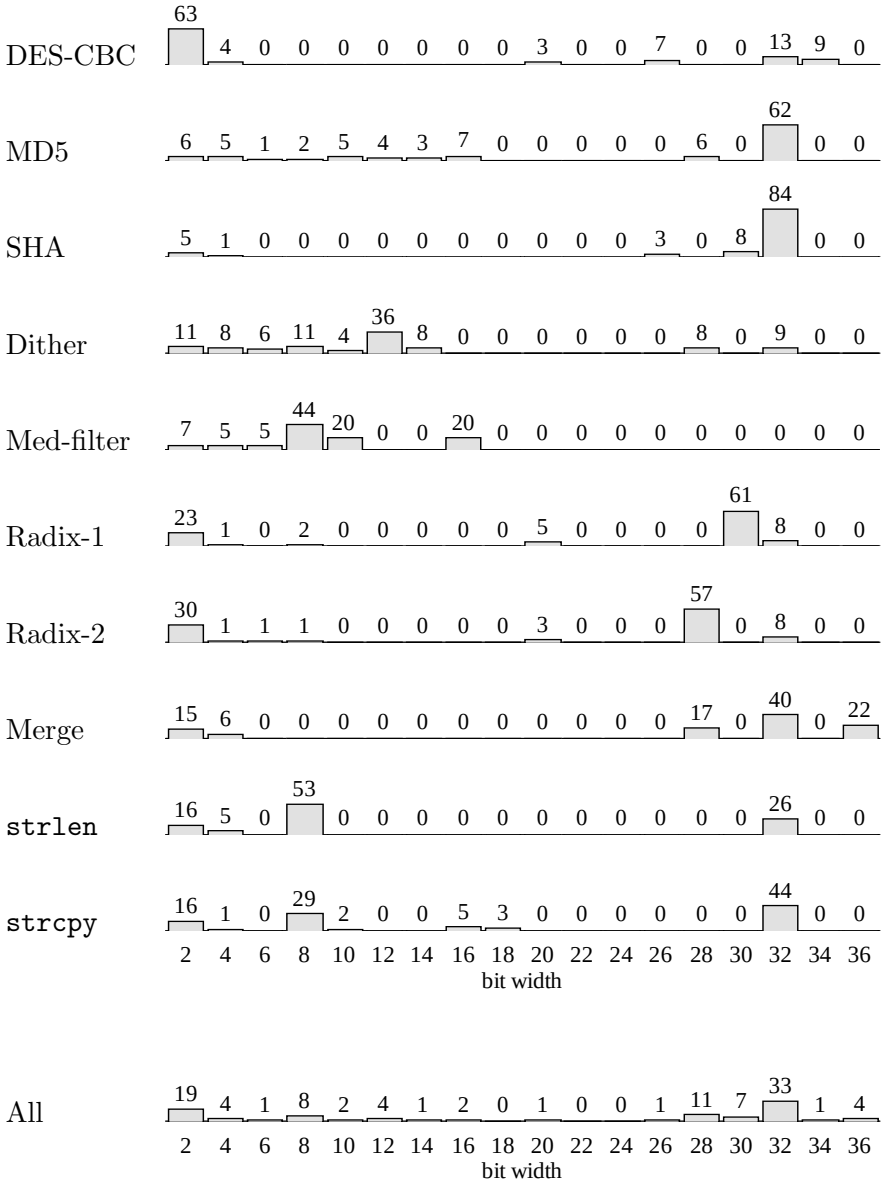


Table 5.12: The percentage of logic blocks within operations of bit width ranging from 2 to 36. Each row adds to 100% of all logic blocks.

bit width	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
percentage	67	7	1	7	1	2	1	1	0	0	0	0	0	3	2	7	0	1

Table 5.13: Percentage of operations of each bit width ranging from 2 to 36. Interpreted from Table 5.12.

The image dithering configuration has a peak at 12 bits because the multiplications by 1/51 are done 12 bits wide. The other peaks are generally at 2, 8, 16, and (nearly) 32 bits. Note that there are even some widths greater than 32 bits (DES-CBC, Merge). Oddly, the cryptography configurations are the most unimodal, with DES preferring 2-bit (or maybe 1-bit) operations, and the one-way hash functions at the other extreme preferring 32-bit operations.

The table is a bit deceiving, since at first blush it seems to recommend a granularity of 32 bits (having the highest average, at 33%). However, the table reports percentages of *logic blocks* within each size, not percentages of *operations* of each size. To find the latter requires dividing by the number of logic blocks for each width and normalizing for the total number of multi-bit operations. This exercise has been done in Table 5.13, which dramatically puts operations of more than 2 bits in the minority as a percentage of all operations. The reason 32-bit operations consume a large fraction of logic blocks overall is because they use many logic blocks individually.

Of course, the results of Table 5.13 are biased by the fact that the configurations were hand-coded specifically for reconfigurable hardware with a granularity of 2 bits. Grouping operations into multi-bit runs was not a priority; and it is not possible to say what the distribution might be if it had been.

5.4.5 Wire connections

Logical connections between Garp logic blocks are ideally made through a single wire, but in order for that to happen, the two blocks must be in the same column (using a vertical wire) or in the same or neighboring rows (using a horizontal wire). Some logic block connections will need more than one hop to get from source to destination, where a *hop* is defined as a traversal across a single straight piece of wire. In Garp, each additional hop requires entering and exiting a logic block, if only through the logic block's D path.

Table 5.14 gives the percentage distribution of number of hops for logical connections between blocks in the configurations. A logic block is considered to facilitate a hop

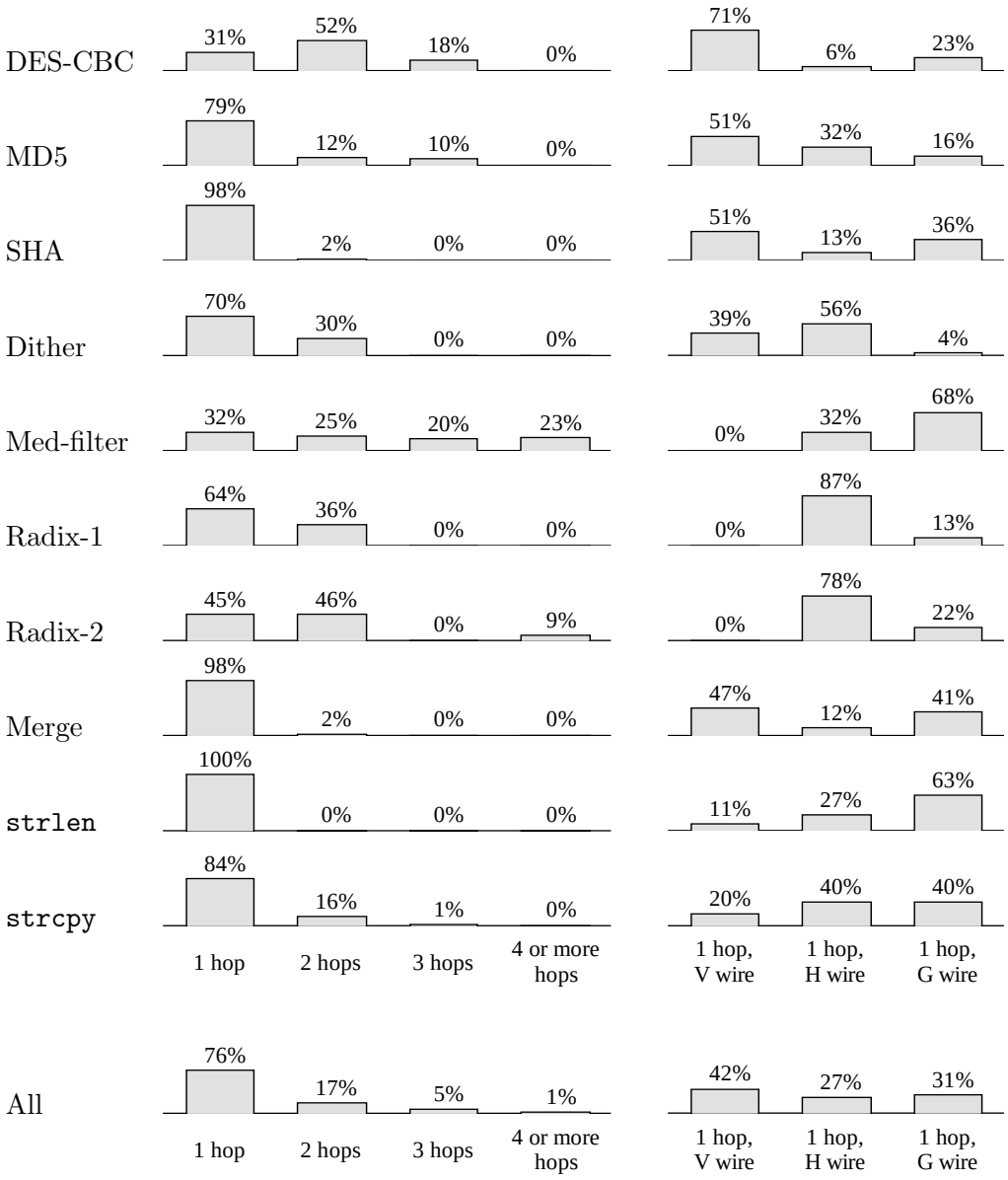


Table 5.14: Distribution of wire hops for each logical connection between logic blocks. The three columns on the right break down the 1-hop case by wire class.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
21–16	0	0	0	0	0	0	0	0	0	0	0
15–8	0	0	0	8	44	64	28	0	0	0	0
7–4	0	4	5	13	32	82	27	3	2	0	0
3–2	0	1	1	4	14	75	13	1	8	3	0
1	0	9	9	8	2	1	1	7	4	6	0
0	0	8	5	5	4		2	1	0	0	0
1	0	0	0	0	1	17	0	0	0	0	0
2–3	0	0	0	0	0	0	0	0	0	0	0
4–7	0	0	0	0	0	0	0	0	0	0	0
8–15	0	0	0	0	0	24	0	0	0	0	0
16–21	0	0	0	0	0	0	0	0	0	0	0

(a) DES-CBC.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
27–24	0	0	0	0	0	0	0	0	0	0	0
23–16	0	0	0	0	0	32	0	0	0	0	0
15–8	0	0	0	0	0	80	0	0	0	0	0
7–4	0	0	0	0	0	128	0	0	0	0	0
3–2	0	6	11	0	0	80	0	0	10	5	0
1	0	15	9	0	0	179	2	18	16	23	2
0	0	2	0	8	8		12	28	28	56	10
1	0	0	0	0	0	0	0	0	0	0	0
2–3	0	0	0	0	0	32	0	0	0	0	0
4–7	0	0	0	0	0	0	0	0	0	0	0
8–15	0	0	0	0	0	32	0	0	0	0	0
16–23	0	0	0	0	0	32	0	0	0	0	0
24–27	0	0	0	0	0	0	0	0	0	0	0

(b) MD5.

Figure 5.13: Connection vector plots for each of the benchmark configurations. The destination logic block is always in the middle of the plot; the numbers give the count of connections from sources at each relative position (or set of positions, usually). The source positions within the strips marked by the lines are those from which a connection can be made with a single hop through a vertical or horizontal wire.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
23–16	0	0	0	0	0	0	0	0	0	0	0
15–8	0	0	0	0	0	32	0	0	0	0	0
7–4	0	0	0	0	0	16	0	0	0	0	0
3–2	0	0	0	0	0	32	0	0	0	0	0
1	0	6	0	0	16	80	15	27	36	73	37
0	0	0	0	0	0		1	0	36	72	36
1	0	0	0	0	0	224	0	0	0	0	0
2–3	0	0	0	0	0	48	0	0	0	0	0
4–7	0	0	0	0	0	48	0	0	0	0	0
8–15	0	0	0	0	0	0	0	0	0	0	0
16–23	0	0	0	0	0	0	0	0	0	0	0

(c) SHA.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
18–16	0	0	0	0	0	0	0	0	0	0	0
15–8	0	0	8	2	0	0	0	0	0	0	0
7–4	0	0	3	0	0	42	0	0	1	0	0
3–2	0	0	0	15	24	98	0	0	0	0	0
1	0	12	21	36	40	23	0	4	4	0	0
0	0	7	4	1	0		8	4	6	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2–3	0	0	0	0	0	18	0	0	0	0	0
4–7	0	0	0	0	0	18	0	0	0	0	0
8–15	0	0	0	0	0	0	0	0	0	0	0
16–18	0	0	0	0	0	0	0	0	0	0	0

(d) Dither.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
11–8	0	0	0	0	0	0	0	0	0	0	0
7–4	0	0	0	0	0	0	0	0	1	0	0
3–2	0	4	20	0	8	12	8	0	28	8	0
1	1	3	5	2	1	5	3	2	10	7	0
0	3	5	10	4	1		2	2	6	22	0
1	0	0	0	0	0	0	0	1	0	0	0
2–3	0	0	0	0	0	18	0	0	0	0	0
4–7	0	0	0	0	0	18	0	0	0	0	0
8–11	0	0	0	0	0	0	0	0	0	0	0

(e) Med-filter.

Figure 5.13, part 2.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
23–16	0	0	10	2	0	0	0	0	0	0	0
15–8	0	0	16	16	0	0	0	0	0	0	0
7–4	0	0	6	8	2	2	2	4	4	1	0
3–2	0	0	0	4	0	0	0	0	0	0	0
1	0	0	32	6	5	1	21	6	0	0	0
0	0	0	0	0	1		49	18	4	1	0
1	0	0	0	0	0	0	0	0	0	0	0
2–3	0	0	0	0	0	0	0	0	0	0	0
4–7	0	0	0	0	0	0	0	0	0	0	0
8–15	0	0	0	0	0	0	0	0	0	0	0
16–23	0	0	0	0	0	0	0	0	0	0	0

(f) Radix-1.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
20–16	0	0	4	0	0	0	0	3	0	4	0
15–8	0	0	16	0	0	0	0	8	0	16	0
7–4	0	0	8	0	0	0	0	4	0	8	0
3–2	0	0	4	0	0	0	0	1	0	4	0
1	0	0	16	2	3	1	11	4	0	0	0
0	0	0	0	0	1		33	18	4	1	0
1	0	0	0	0	0	0	0	0	0	0	0
2–3	0	0	0	0	0	0	0	0	0	0	0
4–7	0	0	0	0	0	0	0	0	0	0	0
8–15	0	0	0	0	0	0	0	0	0	0	0
16–20	0	0	0	0	0	0	0	0	0	0	0

(g) Radix-2.

Figure 5.13, part 3.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
31–24	0	0	0	0	0	0	0	0	0	0	0
23–16	0	0	0	0	0	0	0	0	0	0	0
15–8	0	0	0	0	3	60	0	0	0	0	0
7–4	0	0	0	0	2	38	0	0	0	0	0
3–2	0	0	0	0	1	145	7	0	0	0	0
1	40	64	32	24	16	18	10	10	0	0	0
0	0	64	32	26	11		34	34	32	56	0
1	0	0	0	0	0	144	0	0	0	0	0
2–3	0	0	0	0	1	19	0	0	0	0	0
4–7	0	0	0	0	0	4	0	0	0	0	0
8–15	0	0	0	0	0	8	0	0	0	0	0
16–23	0	0	0	0	0	0	0	0	0	0	0
24–31	0	0	0	0	0	0	0	0	0	0	0

(h) Merge.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
5–4	0	0	0	0	0	1	0	0	0	0	0
3–2	0	0	0	0	0	8	0	0	0	0	0
1	7	14	5	3	5	6	0	1	0	0	0
0	3	6	1	3	1		5	2	4	8	0
1	0	0	0	0	0	0	0	0	0	0	0
2–3	0	0	0	0	0	0	0	0	0	0	0
4–5	0	0	0	0	0	0	0	0	0	0	0

(i) `strlen`.

	22–16	15–8	7–4	3–2	1	0	1	2–3	4–7	8–15	16–22
15–8	0	0	0	1	1	18	0	0	0	0	0
7–4	0	0	0	1	1	49	1	0	0	0	0
3–2	1	1	4	2	1	5	0	0	0	0	0
1	0	9	0	2	2	84	5	0	3	5	0
0	4	15	9	5	4		7	7	20	37	16
1	0	1	0	1	0	4	0	0	0	0	0
2–3	1	0	0	0	0	2	0	0	0	0	0
4–7	1	0	0	0	1	1	0	0	0	0	0
8–15	0	0	0	0	0	0	0	0	0	0	0

(j) `strcpy`.

Figure 5.13, part 4.

if it passes a value unaffected from one wire to another. Registers are ignored; values may or may not be delayed by registers. Any sharing of wires when a signal fans out from one logic block source to multiple destinations is also not accounted for in the numbers.

As the table shows, the great majority of logical connections actually meet the ideal of only a single wire hop, and 94% of connections are made in no more than two hops. For what it is worth, the single-hop cases seem to be fairly evenly distributed among the different classes of veritical and horizontal wires.

Figure 5.13 provides connection vector plots for all of the benchmark configurations. These plots give an indication of the physical lengths of connections made between logic blocks. They also provide visual verification of the claim that most connections can be satisfied with only a single hop.

5.4.6 Memory accesses

The inherent memory access patterns of the various benchmarks are indicated in Table 5.15. Nearly all the benchmarks operate on one or more streams. A couple of them use *circular buffers*, which are contiguous blocks of memory that are read/written in stream-like fashion, but cyclically. The MD5 configurations (Section 5.3.2) read 64 constants from what is essentially a circular buffer, while the image dithering configuration needs a circular buffer to hold the errors to be diffused from one scan line to the next (Section 5.3.3).

Only MD5 is forced to perform truly nonsequential accesses. As explained earlier, the MD5 algorithm reads the next sixteen 32-bit words in a nonsequential pattern as it updates its hash value. One can imagine loading these sixteen words from the head of a sequential stream into the Garp array and then accessing them internally in whatever order is needed, but there is no room in the array to do this for MD5. Consequently, these reads are counted here as inherently non-sequential accesses.

Table 5.15 also tallies the actual Garp resources each configuration uses to access memory: specifically, the number of hardware memory queues allocated and whether the configuration makes independent (demand) memory accesses. Despite the fact that most of the benchmark data is organized as streams, the majority of configurations do actually make demand memory accesses. The most common reason is the need to read and write more than three streams, which is the maximum possible via Garp's memory queues. The Radix-2 configuration (part of the radix sort pass) operates on seventeen streams total, one

configuration	streams in	streams out	circular buffers in	circular buffers out	other memory accesses	memory queues used	demand accesses used
DES-CBC	1	1	–	–	–	2	no
MD5	–	–	1	–	yes	1	yes
SHA	1	–	–	–	–	1	no
Dither	1	1	1	1	–	2	yes
Med-filter	1 (×3)	1	–	–	–	3	yes
Radix-1	1	–	–	–	–	1	no
Radix-2	1	16	–	–	–	1	yes
Merge	8	1	–	–	–	1	yes
`strlen`	1	–	–	–	–	0	yes
`strcpy`	1	1	–	–	–	0	yes

Table 5.15: Inherent memory access patterns for the benchmark configurations, and the memory access resources actually used.

	(bytes / array clock cycles)			
configuration	memory queue reads	demand memory reads	memory queue writes	demand memory writes
DES-CBC	8/80	–	8/80	–
MD5	4/4	4/4	–	–
SHA	4/2	–	–	–
Dither	4/4	8/4	1/4	8/4
Med-filter	2/1	1/1	1/1	–
Radix-1	16/1	–	–	–
Radix-2	8/1	–	–	8/1
Merge	–	8/9	8/9	–
`strlen`	–	16/1	–	–
`strcpy`	–	16/2	–	16/2

Table 5.16: Peak memory bandwidth requirements of each benchmark configuration. The 8/80 entries for DES-CBC, for example, mean that every 80 array clock cycles the configuration reads/writes eight bytes simultaneously in one cycle. Cache misses or other stalls may delay the array clock.

input stream and sixteen output streams. While the input stream is easily delegated to the memory queue hardware, the sixteen output streams are written irregularly, making the writes almost indistinguishable from arbitrary demand writes. The reverse is true of the mergesort configuration, which reads irregularly from eight input streams and writes its output to a single memory queue.

The memory queues are not used for `strlen` and `strcpy` simply to avoid the extra overhead cost. Similarly, the remaining third memory queue is not employed by the image dithering configuration to avoid having to reset the queue for each scan line.

The rate at which the different configurations attempt to access memory is recorded in Table 5.16. The four memory buses total to 128 bits, so the maximum memory bandwidth available is 16 bytes per cycle. The radix sort and the string functions are notable for saturating the memory buses every array clock cycle. The numbers in the table represent attempted rates, however, not the bandwidth actually sustained by the memory system. Although the Radix-2 configuration attempts an 8-byte write every array cycle, the write often misses in the cache resulting in a slower actual rate. As noted earlier, all three sorting configurations are particularly susceptible to cache misses.

Chapter 6

Garp Retrospective

This chapter collects lessons learned and other observations about Garp with the benefit of hindsight. Some positive aspects are considered first, followed later by weaknesses that deserve better attention in the future. In between is a review of some of the changes that were made to the Garp design to correct earlier flaws.

6.1 Noteworthy features

First, on the positive side, some Garp features seem deserving of special emphasis that may not be reflected elsewhere in this thesis.

6.1.1 Processor handling of start-up, shut-down, and other particulars

One feature that turned out to be more valuable than expected is the main processor's ability to stop the reconfigurable array at will and access all of its data registers. Of course, this has an obvious application in debugging, since it is then easy for a program to single-step the array and report what is happening. What was unexpected is the extent to which this feature can also be used during normal execution to help applications run faster.

Assume the reconfigurable array is used to execute critical program loops. Most loops require a certain amount of start-up and shut-down activity that cannot be ignored; however, accomodating this overhead in a confirguration means dedicating scarce array resources to circuits that sit idle most of the time. Because this overhead circuitry takes up space, some loops that otherwise would fit within the array will be crowded out and have to

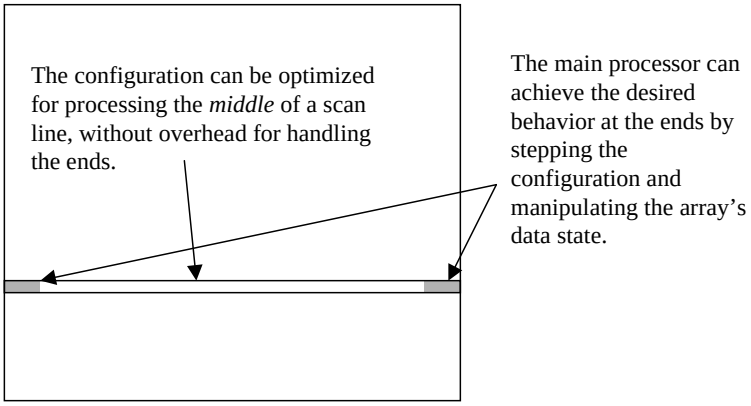


Figure 6.1: Typical processing of an image, with a configuration of the reconfigurable hardware capable of handling only the middle of each scan line and the edges being corrected by the main processor.

be executed back on the main processor. Worse, when everything does fit, the start-up and shut-down circuitry inserts multiplexors into the main part of the loop, often increasing the length of the critical path and thus slowing execution speed.

It is usually better to build the configuration without this start-up and shut-down overhead and rely on the main processor to handle it instead. After loading a new configuration into the array, the processor can single-step array execution for a few cycles and adjust the array pipeline state for each step to ensure that the loop is properly started. In the simplest cases, all that has to be done is to initialize a few registers before loop execution is started; however, many times a loop must be stepped and adjusted for several clock cycles to prime the pipeline. A reverse process can occur at the end of loop execution to extract the results.

In the same way, the processor can be used to smooth over many other special cases. A good example is an image processing application that progresses through an image in scan-line order, as illustrated in Figure 6.1. Because the edges of an image often require special handling, some amount of special-case circuitry is usually needed if the array is to process an entire image from top to bottom on its own. But for anything other than the smallest images, the great majority of time is clearly spent in the middle of the image, where the circuitry for the edge cases is useless. The configuration will usually be more efficient without the extra circuitry, leaving it to the main processor to take care of the edges.

On Garp, the easiest way to do that is not for the main processor to perform

the calculation at the edges entirely itself—although that can be done—but rather for the processor simply to tweak the configuration's calculation by single-stepping the array near the edges and manipulating the array's register contents. This trick is used for the dithering benchmark presented in Section 5.3.3, but would not be possible if the processor did not have arbitrary access to the array registers.

6.1.2 Support for extended functions in logic blocks

More than traditional FPGAs, Garp's logic blocks are designed with basic arithmetic operations in mind. Although Garp supports standard table lookups, the table-lookup modes have not been the most popular. From Table 5.10 in the previous chapter, the two table-lookup modes are seen to account for only $27 + 11 = 38\%$ of all logic block functions, with the remaining 62% configured for one of the "higher-level" functions. The single most frequent function mode (46%) is the carry chain mode which supports additions, subtractions, and comparisons of two values (sometimes with a third control input). In the image dithering application, two-thirds of the logic blocks are configured in triple-add mode, reflecting the multiplications and divisions performed by that benchmark. Only the cryptography applications and to some extent `strcpy` make major use of the table-lookup modes.

Of course, 38% is not an insignificant fraction, and the table modes do play a role in the control circuitry of every benchmark. The point is rather that, even on a fine-grained array such as Garp, the arithmetic operations seem to dominate as a rule, justifying the extra support for them in the Garp design.

6.1.3 Limited configuration turnaround

The Garp architecture includes a configuration cache but expressly denies that a configuration can be loaded from the cache in only one or two clock cycles. It is always anticipated that switching to a cached configuration might take several clock cycles, possible as many as eight or ten cycles. The architecture does not *enforce* a delay when loading from the cache, but the *license* for a delay is valuable to minimizing the area of the Garp array.

Recalling the implementation survey in Section 4.3 and particularly the logic block layout in Figure 4.27 (page 97), much of the design is predicated on the freedom to ignore

configuration delays. Without that freedom, the tight tracks for distributing configuration bits around the logic block in polysilicon would be totally unacceptable, as would the many small pieces of circuitry involved in decoding the configuration, including notably the slow-but-narrow decoder for the vertical output drivers visible in Figure 4.27. Very likely, parts of the configuration storage and input multiplexors would have to be redesigned as well. All told, it might be conservative to suppose an increase of 30 to 50% in array area. The benefit of being able to change configurations more quickly could not justify such a cost.

6.1.4 Array access to memory

In Section 3.1.6, a case was made for giving the reconfigurable array direct access to memory, without intervention by the main processor. The data in Section 5.4.6 confirmed that some of the benchmark kernels depend on having maximal bandwidth to memory, the radix sort and string functions specifically. These applications would not run nearly as fast if memory accesses had to be routed through the main processor. The image dithering benchmark, too, although not saturating the array's memory buses like the others, still makes more accesses to memory than can be sustained by the Garp main processor. Thus it also would be somewhat hobbled if the processor were exclusively responsible for memory accesses.

It must be emphasized that memory bandwidth alone is not the reason for any of the reported Garp speedups. For all of the chosen benchmarks, the UltraSPARC implementation is not primarily limited by bandwidth to memory. However, when the reconfigurable array performs a computation an order of magnitude faster than the UltraSPARC (as it does for dithering and `strlen`, for example), it is only natural for memory bandwidth needs to increase. The decision to give the Garp array wide, direct access to memory is consistent with the expectation that it will deliver speedups.

6.1.5 Array clocking and context switches

A final important aspect of Garp is the separation of the notion of *logical* array clock cycles from real time. Any reconfigurable hardware that interacts with a processor memory system through standard caches must be prepared to deal with unpredictable delays due to cache misses. By automatically suspending array register updates in hardware, the reconfigurable array can be stalled and cache misses serviced completely transparently to

the executing configuration. This frees configurations to concentrate on the computation itself and not on synchronization issues. Array clock cycles in Garp are the logical units of array execution, just as instructions are units of execution in standard processors.

Context switches are a far more extreme example of decoupling array clock cycles from real time. A preemptive context switch must be able to freeze array execution at any stage and preserve the array's state until the corresponding process is swapped back in again. The array clock cycles in Garp provide clean boundaries at which array execution can be suspended and resumed.

6.2 Corrected mistakes

Although the design of Garp went through several variations early on, a few flaws were not realized until late in the game, precipitating some late changes to the architecture. Three corrections are perhaps worth mentioning:

- The control blocks, which provide the reconfigurable array the ability to perform memory accesses and stop array execution, were originally placed at the right edge of the array instead of the left where they are now. Because the carry chains in the array operate from right to left, the right side of the array is closest to the least significant bits of multi-bit values. Putting the control blocks on that side was intended to leave more room at the left for overflow handling. It should have been obvious, however, that decisions are often made on the outcome of comparisons, and that comparison results using the carry chain would be available only on the left side. Rather than continue to transmit these signals all the way from one side of the array to the other, the control blocks were moved to the left side to shorten the distance.

- At one time, the main processor did not have access to the data registers in the extra logic blocks at the sides of the array; only the middle 16 logic blocks could be copied to and from the processor, in the same way that only the middle 16 blocks can be the source or destination of a memory access (Section 4.1.1). This made it inconvenient, bordering on impossible, for the processor to perform certain start-up and shut-down duties as discussed earlier. It also meant that context switches needed additional special handling to save and restore those bits. The instruction set was eventually extended to give the main processor access to the full array.

- Currently the Garp array's short horizontal wires (H wires) span eleven blocks and can send a value a distance of nine blocks away, corresponding to a shift of 18 bits (Section 4.1.3 and Section A.2.1 in the appendix). Previously, the H wires were a little shorter and could only support a shift of up to 16 bits. It is easy to see the reason for choosing a distance of 16 bits, since that is a common data size. What only became clear from experience, however, is the need for the wires to be just a little longer to provide a margin of flexibility when laying out 16-bit operations in the array.

Two other changes have come about since the first publications about Garp:

- In addition to the six logic block functions covered in Section 4.1.2, the original architecture included another for variable shifts. In this mode, 17 bits from the horizontal wires above the logic block were treated as a 17-bit value, which was then shifted right according to the values from the usual logic block inputs. The two least-significant bits of the shifted value were returned as the function result. When used in concert across a row, this mode created a one-row variable shifter.

  The variable shift mode had several strikes against it, however. To start with, it was rarely needed, and in fact never used by any of the benchmarks. Second, the hardware for this mode did not overlap much with the other logic block functions, so it was not cheap to include. Worse, it was not even immediately suitable for all common shift operations. C-style right shifts, for example, required extra logic blocks and delay to work correctly. For all these reasons, the mode was ultimately discarded. A minor adjustment was made to the definition of select mode at the same time so that variable shifts could be implemented in a few Garp rows, as witnessed back in Table 4.1.

- Section 4.1.5 discusses how the basic arithmetic operations can usually be done in just the natural number of logic blocks in Garp—for example, an 8-bit comparison can be done in exactly four logic blocks (at two bits per block). This feature makes it easier to configure the Garp array for small-SIMD, *segmented* operations, such as four adjacent 8-bit comparisons on a pair of 32-bit words. Originally, certain inequality comparisons could not be segmented in this way. To fix the shortcoming, the operation of the carry chain and triple-add modes had to be modified. In general, ensuring the segmentability of arithmetic operations requires some care in the definition of the logic block functionality connected with the carry chain.

6.3 Weaknesses

Even with corrections, the Garp architecture is far from perfect. Some flaws in the basic design, and the challenge of programming the device, bear scrutiny.

6.3.1 Wire network

The Garp array was intended to be expandable to larger sizes as technology improves by increasing the number of rows. Previous configurations would be able to run without modification in the same number of rows as before, merely being a smaller fraction of the new array. Configurations filling the new, larger array would not run on older Garps, just as software requiring Intel's MMX extensions cannot be run on older Pentiums without that feature.

Unfortunately, the Garp architecture's insistence on having wires that stretch the full height of the array makes this a difficult proposition in practice. In Section 4.3.3, a scheme for hiding buffers within the longest wires was proposed that works for the 32-row Garp but is not ideal. It was originally hoped that similar schemes could be used for longer wires in larger arrays, but this was largely naive. With 64 rows it might be possible to implement the longest Garp wires with tricks involving multiple physical wires; but at some point it has to be recognized that signals simply cannot be sent arbitrary distances in a single clock cycle. Wire networks in larger reconfigurable arrays need to be able to take several clock cycles to transmit a value, presumably with registers embedded in the network to support pipelining. As designed, Garp's wire network is not really adequate for arrays larger than the 32-row Garp assumed in this thesis.

Another problem with the Garp network is the shortage of resources for connecting between horizontal and vertical wires. A logic block can often make one such connection (using the D path) independent of the normal logic block function. In some case—for instance, if it is not otherwise occupied—it can make two connections, although these are generally in the minority. A connection between a horizontal and a vertical wire in either direction can be called a *turn*.

One turn per logic block is often sufficient for datapaths created in a configuration because of a tendency in datapaths toward mostly orthogonal connections without turns. When two values are needed as inputs to an addition, for example, the operands can usually be brought straight over vertical wires without any horizontal shifting being needed across

the columns. For control circuitry, however, and for some irregular datapaths, one turn per logic block is not always enough. Arranging and routing the control circuitry in the benchmark configurations has sometimes been a kind of puzzle, requiring that logic blocks with connections between them be placed in vertical and/or horizontal relationships to conserve the available turn resources. Automatic place-and-route tools would be hard pressed to duplicate this effort; and, fittingly, no commercial FPGAs are so constrained. Again, the need for a better wire network in Garp is apparent.

6.3.2 Memory bottleneck

The Garp reconfigurable array accesses memory over its four memory buses, which stretch across all the array rows. With arrays of more than 32 rows, access to memory becomes more problematic, for two reasons: First, a larger array implies a need for more bandwidth to memory to avoid having memory bandwidth be too much of a bottleneck by contrast. This entails a multiplying of the memory buses and potentially the memory queues as well. Currently, the Garp architecture permits any memory bus to be matched with any memory queue, with a crossbar bridging the two groups. Whether this would continue to be an attractive arrangement is uncertain.

A bigger concern, though, is the same one that arose for the Garp wire network. Because the memory buses span the entire array, if the array is made much larger, it becomes impossible to transmit a value over a memory bus in only one clock cycle. If the memory buses are pipelined as was proposed for the network wires, they then cease to be buses and become something else less intuitive and perhaps more difficult to work with. A solution to the dilemma is not provided here, other than to suggest that the discussion in Section 3.1.6 concerning the array's interface to memory may need to be revisited.

6.3.3 Programming experience

The basic Garp development tools presented in Section 5.2 were used to create all of the benchmark configurations. These tools have been enough to get the job done but do not provide an especially attractive programming environment for someone used to writing software in a high-level language. Describing a configuration at the primitive level of the Garp configuration assembler (the *configurator*) is actually a little more laborious than coding in normal assembly language, already considered a labor of last resort by most.

The simplest of the Garp configurations took about a day to create and debug, while some took several days. Whether better Garp tools could be constructed that would make this task easier is a nonobvious question; it may be that they could. Better tools for creating configurations by hand were never high on the list of priorities for this project.

To investigate compiling from a high-level language like C to a hybrid reconfigurable machine, Callahan has created a C compiler for Garp [11, 12, 13]. His `garpcc` automatically chooses kernels within a program to implement in the reconfigurable array, then creates confirgurations for those kernels and integrates them with the rest of the program, which is compiled for Garp's main MIPS processor. Callahan's compiler translates its chosen kernels into configuration descriptions that are fed to the previous low-level tools, in the same way that many C compilers invoke an assembler as the final compilation stage before linking.

The compiler is still in development, and few complete programs have been tested with it so far. One of these is a program for wavelet image compression; Table 6.1 summarizes the performance of this program when compiled for Garp and for the UltraSPARC-1/170 that was used for comparison purposes in Chapter 5. As the table shows, `garpcc` is able to eke out a 27% speedup for Garp relative to the UltraSPARC.

There is no expectation that the Garp C compiler will match the performance possible with hand-coded configurations. Nevertheless, the compiler has the distinct advantage of being able to consider many more program loops than can reasonably be done by hand, and in the space of only a few minutes, not days. The full extent of the performance achievable with the compiler remains to be seen. However, unless the results turn out to be far better than expected, the difficulty of programming Garp will continue to be a weakness of the system.

loop number	number of executions	percentage of total time	167 MHz SPARC	133 MHz Garp	ratio
1	1	22.1	4.6 ms	0.50 ms	9.2
2	320	8.5	1.77	1.00	1.77
3	774	3.7	0.76	0.45	1.69
4	1	5.7	1.18	0.50	2.4
5	3262	5.7	1.18	1.38	0.86
6	448	10.1	2.1	1.35	1.56
7	448	5.5	1.15	0.94	1.22
8	448	14.9	3.1	4.4	0.70
9	448	6.7	1.39	1.04	1.34
total of 9 loops		82%	17.2 ms	11.6 ms	1.49
other code		17	3.6 ms	4.8 ms	0.75
total		100	20.8 ms	16.4 ms	1.27

Table 6.1: Results from compiling a wavelet image compression program to both the UltraSPARC and to Garp using Callahan's `garpcc`. The Garp compiler extracts nine loops for implementation on the reconfigurable array. Execution times for a loop combine all invocations of the loop together. Time percentages are with respect to the UltraSPARC, which is taken as the default system.

Chapter 7

Conclusions

7.1 Summary of contributions

The main contributions presented in this thesis are:

- An investigation of the basic issues for integrating a reconfigurable unit into a traditional, general-purpose processor. It is recommended that a reconfigurable unit be attached as an on-chip coprocessor outside the immediate processor pipeline, and that it be geared to executing whole program loops, with its own independent path to memory. The implications of cache miss stalls, context switching, and virtual memory page faults on the design are also addressed.

- An evaluation of how certain reconfigurable hardware parameters such as granularity and preferred arithmetic style (bit-serial, bit-pipelined, or bit-parallel) impact the reconfigurable unit's efficiency. It is found that, although bit-pipelined arithmetic is faster for some operations such as additions and multiplications, bit-parallel forms are more robust at handling a wider range of operations and require fewer register bits for data pipelining. Support for building multipliers in the reconfigurable hardware is also considered, with the conclusion that—assuming fast carry chains are already embedded in the reconfigurable hardware—additional hardware support for three-input adders allows denser multipliers without a major sacrifice in delay or chip area.

- The detailed specification of Garp, a prototype combined processor/reconfigurable architecture responding to the previous issues. Garp is the first design to fully adapt a reconfigurable computational unit to the conventions of standard general-purpose

systems. Unlike other reconfigurable hardware, Garp fixes the clock within its reconfigurable array and supports a notion of *logical* array clock steps separate from real time. Also fairly novel is the introduction of *control blocks* at one edge of the reconfigurable array to give the array control over memory accesses.

- A study of a potential VLSI implementation of Garp, including some ideas about the configuration cache and its effect on logic block layout. A proposal to verify the integrity of a configuration while loading it from external memory is believed to be an innovation. The study demonstrates the feasibility of implementing Garp, with estimates for area and clock speed.

- A test of the fitness of the Garp design through the benchmarking of an assortment of example applications. Execution times on Garp are measured against a comparable UltraSPARC, showing an impressive Garp advantage for at least certain problems. For three applications, Garp is determined to be faster than the UltraSPARC by factors of 17, 19, and 43, respectively. Based on the benchmark results, a brief analysis is made of Garp's effectiveness for different types of applications.

- Statistics extracted from the benchmark configurations that may help with the design of other Garp-like reconfigurable units. Among other things, the statistics show that arithmetic functions are more frequently used than table-lookup functions, and that straight connections through a single wire account for three-quarters of all communications between logic blocks.

- A retrospective of Garp's highlights and weaknesses, as well as some corrections that were made to the architecture over time.

7.2 Application niche

With speedup factors on some benchmarks in the high teens (19 times faster for DES encryption, 17 times for image dithering), and exceeding as much as 40 for the median filter benchmark, it is clear that a Garp-like reconfigurable unit would be a boon for certain applications. Overall experience with the benchmarks (summarized in Table 5.3, page 128) suggests that the applications most likely to be successful are those with ample parallelism

beyond simple ILP (instruction-level parallelism). Even then, applications limited to only ILP have sometimes run three to four times faster on Garp than on the UltraSPARC 1/170.

Reconfigurable hardware is often touted as being best at "bit-manipulation" functions, with cryptography and compression offered as examples. While there is no question that fine-grained reconfigurable hardware can be faster at bit-level operations than normal processors, Garp has not experienced the same order-of-magnitude speedups on cryptography functions as it has with some of the more arithmetic-oriented applications such as image dithering.

Encryption algorithms, by their nature, are designed to maximize the dependency between the original plaintext and its encrypted form, so that, ideally, changing one bit in the original causes a change in half the bits in the encrypted encoding. Called an *avalanche effect* by cryptographers, this maximizing of dependencies between operations has the side effect of minimizing the algorithm's parallelism. An encryption method can be modified to be more parallel only at the expense of being less secure, which is the difference between the CBC and ECB modes for the DES encryption algorithm (Section 5.3.1). A similar property applies to compression as well: a compression algorithm can be made more parallel only by sacrificing some compression quality.

In the absence of parallelism, reconfigurable hardware can only gain a speedup by improving the latency of operations. For pure bit-manipulation functions, this might lead to a factor of three or four improvement in speed, as shown for example by DES in CBC mode. Most real algorithms are not purely based on bit-manipulation, however. The one-way hash functions MD5 and SHA, for instance, include 32-bit additions which the Garp array cannot do any faster than the UltraSPARC. Consequently, the hash functions achieve a speedup of only a factor of two or three on Garp.

The greatest triumphs for reconfigurable computing have come from applications with both bit manipulation *and* tremendous parallelism. A leading example is genome (DNA) pattern matching, for which machines built with FPGAs have been compared favorably to Cray supercomputers. The fine granularity of the FPGAs allows them to pack in a large number of the small operators needed for genome matching, thus exploiting a large quantity of parallelism inherent in the problem. Without the parallelism, the reconfigurable machines could not be as successful.

In Section 2.1.2, program parallelism was divided into three categories: ILP, inter-iteration or data parallelism, and thread parallelism. Although reconfigurable hardware

is freely able to exploit all three forms, thread parallelism is not a factor in any of the benchmarks selected. The reasons for this have been: (1) Garp's array is often not large enough for *one* loop, much less more than one; (2) memory access conflicts between multiple threads would be difficult to arbitrate within the array; and finally, (3) finding good examples by hand can be tedious. With a larger reconfigurable unit, it would be a good topic of research to examine whether automatic compilation could be successful at finding simultaneous threads of execution to move into the reconfigurable array.

Another significant technique that has not been tested on Garp is run-time specialization for loop invariants. A *loop invariant* is a value that is constant for the duration of a loop. Run-time specialization generally entails creating a template configuration with parts that are changed based on values known during execution. Before the configuration is loaded into the reconfigurable unit, the template would be copied and edited in memory by the main processor for the appropriate constants. Run-time specialization is often useful for pattern matching problems, as demonstrated for genome pattern matching on the PeRLe-1 [48] and for text keyword search on another FPGA board [25]. Use of the technique on Garp has been left as an open exercise.

7.3 Architectural alternatives

Before rushing out to adopt a reconfigurable unit, it is important to recognize there may be better ways to use the same chip area to accelerate applications. This research has not tried to measure Garp's performance against other proposals, relying instead on a commercial superscalar processor as a common baseline for comparison. Where warranted, direct comparisons between reconfigurable units and other alternatives is left for future research.

Vector units are a promising idea being studied in the IRAM project at the University of California, Berkeley [45]. Although long relegated to the realm of supercomputers, vector processors are emerging as another option for using new quantities of transistors to accelerate software. Vector units achieve speedups by capitalizing exclusively on data parallelism. Since reconfigurable units often get their best speedups from the same sources, vector processors are a clear rival to reconfigurable computing. Commercial processors have already begun incorporating some vector capabilities, in the form of small-SIMD instructions such as VIS for SPARCs and MMX for Intel Pentiums.

One potential advantage reconfigurable hardware has over vector units is the ability to exploit thread parallelism. Reconfigurable hardware can also be better when independent loop iterations share significant common subexpressions, since vector processors often find it costly to share information across iterations. (This is true, for example, for the image median filter used as a benchmark in Chapter 5.) Again, exactly which types of software do better with each system will have to be deferred to future study.

Either way, building a faster compute engine may simply shift the bottleneck to the memory system. As already noted, Garp's reconfigurable array could not grow much larger without a reworking of the memory interface. Berkeley's IRAM project has been very attentive to the interface between their vector unit and memory, tailoring the vector unit expressly to take advantage of properties of the memory system. In contrast, the reconfigurable computing community does not have a coherent model for memory. Many reconfigurable systems have adopted ad hoc memory banks, requiring explicit partitioning of data into separate banks. While there have been attempts to automate this partitioning [5, 21], experience has shown it to be a hard problem for compilers to solve. For a general-purpose platform, a better model will be needed that is compatible with high-level languages and compilers.

7.4 Programming challenge

The challenge of programming reconfigurable devices is a serious obstacle. Creating configurations by hand is at least as difficult as programming in assembly language, although this might be due as much to the limitations of the development tools as to the actual hardware itself. To provide a better programming environment, many experimental compilers—too many to list here—have been developed for nonstandard variants of a high level language, usually a version of C. A few compilers actually accept pure standard C as input. Besides the Garp compiler created by Callahan (Section 6.3.3), standard C compilers have been described by Jantsch et al. [39] and Peterson et al. [62]. The work by Weinhardt and Luk to adapt vectorizing compiler techniques to reconfigurable hardware is probably also worth mentioning [77].

Even so, there is a large gap between what can be accomplished by hand and what has been demonstrated so far with automatic compilation. While it is clear that compilers need to be improved, part of the solution will have to involve making reconfigurable

hardware more amenable to compilation.

At this date, vector processors can leverage off of decades of experience with vectorizing compilers. Another advantage enjoyed by vector systems is the ability to scale up with more hardware without the need for software to be recompiled. A vector operation of length N can be executed by one functional unit in N steps, or by N functional units simultaneously in one step, or by some other number of functional units in between. As long as the vector lengths in applications are long enough, doubling the number of functional units in a vector processor cuts execution time approximately in half without any change to the software.

Recall that this valuable scaling property is exactly what is obtained by the *virtualizing* of streaming reconfigurable hardware described in Section 2.5.3 and exhibited by PipeRench. In fact, because PipeRench targets much the same data-parallelism as vector processors, it is not immediately obvious to what extent the differences between them really matter. A thorough contrasting of the two approaches would be another good research topic.

Normally, as with Garp, increasing the size of a reconfigurable unit will not lead to a performance increase for an existing program unless a new configuration is generated to utilize the expanded hardware. Assuming at best the program has been compiled from a high-level language with a good compiler, the original source code will have to be recompiled at a minimum. To maintain top performance, each processor implementation would require its own collection of compiled executables, an unattractive proposition for software suppliers. More likely, old software would be upgraded for newer processors only irregularly, causing the reconfigurable unit to be perpetual underutilized.

Other than PipeRench-style virtualization, the only solution to this problem would seem to be some kind of just-in-time compilation (as is frequently used for Java programs, for example). Even if this is possible, substantial work remains to be done on compilers for reconfigurable hardware before just-in-time compilation can be seriously contemplated.

7.5 Outlook

All told, the results presented in this thesis indicate considerable promise for the integration of a reconfigurable device into future microprocessors. If a reconfigurable unit is truly to become a familiar object in mainstream processors, the reconfigurable hardware

itself will need to be specifically designed for the task and not just a clone of current FPGA chips. It is believed the Garp architecture represents an advance in this direction. Nevertheless, serious shortcomings remain that may hinder the acceptance of such a device if creative remedies cannot be found. It can be hoped that further research will continue to address these issues and ultimately clarify whether reconfigurable computing is the best model for the future or if that title belongs to some other style of computing technology.

Bibliography

[1] A. Lynn Abbott, Peter M. Athanas, Luna Chen, and Robert L. Elliot. Finding lines and building pyramids with Splash 2. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–163, April 1994.

[2] Lalit Agarwal, Mike Wazlowski, and Sumit Ghosh. An asynchronous approach to efficient execution of programs on adaptive architectures utilizing FPGAs. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 101–110, April 1994.

[3] Jeffrey M. Arnold. Mapping the MD5 hash algorithm onto the NAPA architecture. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 267–268, April 1998.

[4] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, March 1993.

[5] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 70–80, April 1999.

[6] Patrice Bertin and Hervé Touati. PAM programming environments: Practice and experience. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, April 1994.

[7] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 59–68, February 1999.

[8] Brian Box. Field programmable gate array based reconfigurable preprocessor. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 40–48, April 1994.

[9] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[10] Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing pipeline-reconfigurable FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55–64, February 1998.

[11] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[12] Timothy J. Callahan and John Wawrzynek. Instruction-level parallelism for reconfigurable computing. In Reiner W. Hartenstein and Andres Keevallik, editors, *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 248–257, August 1998.

[13] Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 57–64, November 2000.

[14] Don Cherepacha and David Lewis. A datapath oriented architecture for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 1–11, February 1994.

[15] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and compiling applications for RaPiD. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 116–125, April 1998.

[16] André DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, April 1994.

[17] Apostolos Dollas, Euripides Sotiriades, and Apostolos Emmanouelides. Architecture and design of GE1, a FCCM for Golumb ruler derivation. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 48–56, April 1998.

[18] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD: Reconfigurable pipeline datapath. In Reiner W. Hartenstein and M. Glesner, editors, *Proceedings of the 6th International Workshop on Field-Programmable Logic and Compilers*, pages 126–135, August 1996.

[19] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 106–115, April 1997.

[20] Greg J. Gent, Scott R. Smith, and Regina L. Haviland. An FPGA-based custom coprocessor for automatic image segmentation applications. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–179, April 1994.

[21] Maya B. Gokhale and Janice M. Stone. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 63–69, April 1999.

[22] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, April 2000.

[23] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

[24] Paul Graham and Brent Nelson. Frequency-domain sonar processing in FPGAs and DSPs. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 304–305, April 1998.

[25] Bernard Gunther, George Milne, and Lakshmi Narasimhan. Assessing document relevance with run-time reconfigurable machines. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 10–17, April 1996.

[26] Linley Gwennap. UltraSparc unleashes SPARC performance. *Microprocessor Report*, 8(13):1–5, October 1994.

[27] Linley Gwennap. UltraSparc rolls out at target clock speed. *Microprocessor Report*, 9(7):1–3, May 1995.

[28] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 65–74, February 1998.

[29] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, April 1997.

[30] Scott Hauck, Matthew M. Hosler, and Thomas W. Fry. High-performance carry chains for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 223–233, February 1998.

[31] Scott Hauck, Zhiyuan Li, and Eric Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 138–146, April 1998.

[32] Scott Hauck and William D. Wilson. Runlength compression techniques for FPGA configurations. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 286–287, April 1999.

[33] G. Haug and W. Rosenstiel. Reconfigurable hardware as shared resource for parallel threads. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 320–321, April 1998.

[34] Gunter Haug and Wolfgang Rosenstiel. Reconfigurable hardware as shared resource in multipurpose computers. In Reiner W. Hartenstein and Andres Keevallik, editors, *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 149–158, August 1998.

[35] Simon D. Haynes and Peter Y. K. Cheung. A reconfigurable multiplier array for video image processing tasks, suitable for embedding in an FPGA structure. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–234, April 1998.

[36] Walter B. Ligon III, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, April 1998.

[37] Christian Iseli and Eduardo Sanchez. A C++ compiler for FPGA custom execution units synthesis. In Peter Athanas and Kenneth L. Pocek, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 173–179, April 1995.

[38] Jeffrey A. Jacob and Paul Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 145–154, February 1999.

[39] Axel Jantsch, Peeter Ellervee, Johnny Öberg, and Ahmed Hemani. A case study on hardware/software partitioning. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 111–118, April 1994.

[40] Jack Jean, Xuejun Liang, Brian Drozd, and Karen Tomko. Accelerating an IR automatic target recognition application with FPGAs. In Kenneth L. Pocek and Jef-

frey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 290–291, April 1999.

[41] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 92–101, April 1999.

[42] Tom Kean and Ann Duncan. DES key breaking, encryption and decryption on the XC6216. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 310–311, April 1998.

[43] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, second edition, 1973.

[44] P. Kollig and B. M. Al-Hashimi. Reduction of latency and resource usage in bit-level pipelined data paths for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 227–234, February 1999.

[45] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, September 1997.

[46] Ronald Laufer, R. Reed Taylor, and Herman Schmit. PCI-PipeRench and the SwordAPI: A system for stream-based reconfigurable computing. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 200–208, April 1999.

[47] Dominique Lavenier and Yannick Saouter. Computing Goldbach partitions using pseudo-random bit generator operators on an FPGA systolic array. In Reiner W. Hartenstein and Andres Keevallik, editors, *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 316–325, August 1998.

[48] Eric Lemoine and David Merceron. Run time reconfiguration of FPGA for scanning genomic databases. In Peter Athanas and Kenneth L. Pocek, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, April 1995.

[49] Yamin Li and Wanming Chu. Implementation of single precision floating point square root on FPGAs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 226–232, April 1997.

[50] Zhiyuan Li and Scott Hauck. Don't care discovery for FPGA configuration compression. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 91–98, February 1999.

[51] Loucas Louca, Todd A. Cook, and William H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116, April 1996.

[52] Wayne Luk, Teddy Wu, and Ian Page. Hardware-software codesign of multidimensional programs. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 82–90, April 1994.

[53] Donald MacVicar and Satnam Singh. Accelerating DTP with reconfigurable computing engines. In Reiner W. Hartenstein and Andres Keevallik, editors, *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 391–395, August 1998.

[54] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 135–143, February 1999.

[55] Sally A. McKee, Robert H. Klenke, Kenneth L. Wright, William A. Wulf, Maximo H. Salinas, James H. Aylor, and Alan P. Batson. Smarter memory: Improving bandwidth for streamed references. *Computer*, 31(7):54–63, July 1998.

[56] Ethan Mirsky and André DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, April 1996.

[57] Takashi Miyamori and Kunle Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–11, April 1998.

[58] Matthew Moe, Herman Schmit, and Seth Copen Goldstein. Characterization and parameterization of a pipeline reconfigurable FPGA. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 294–295, April 1998.

[59] Emeka Mosanya and Eduardo Sanchez. A FPGA-based hardware implementation of generalized profile search using online arithmetic. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 101–111, February 1999.

[60] Akihisa Ohta, Tsuyoshi Isshiki, and Hiroaki Kunieda. New FPGA architecture for bit-serial pipeline datapath. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 58–67, April 1998.

[61] S. R. Park and W. Burleson. Configuration cloning: Exploiting regularity in dynamic DSP architectures. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 81–89, February 1999.

[62] James B. Peterson, R. Brendan O'Connor, and Peter M. Athanas. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 178–187, April 1996.

[63] 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, November 1994.

[64] Michael Rencher and Brad L. Hutchings. Automated target recognition on SPLASH-2. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 192–200, April 1997.

[65] 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.

[66] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, July 1993.

[67] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley and Sons, second edition, 1996.

[68] Nabeel Shirazi, Al Walters, and Peter Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In Peter Athanas and Kenneth L. Pocek, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, April 1995.

[69] Satnam Singh and Robert Slous. Accelerating Adobe Photoshop with reconfigurable logic. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 236–244, April 1998.

[70] Alexandre F. Tenca and Miloš D. Ercegovac. A variable long-precision arithmetic unit design for reconfigurable coprocessor architectures. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 216–225, April 1998.

[71] Steve Trimberger, Khue Duong, and Bob Conn. Architecture issues and solutions for a high-capacity FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 3–9, February 1997.

[72] K. Tse, T. Yuk, and S. Chen. Implementation of Data Encryption Standard by programmable gate array. In *Proceedings of the 3rd International Workshop on Field-Programmable Logic and Applications*, pages 412–419, September 1993.

[73] Robert Ulichney. *Digital Halftoning.* MIT Press, Cambridge, Massachusetts, 1987.

[74] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[75] Al Walters and Peter Athanas. A scaleable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 333–334, April 1998.

[76] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, April 1993.

[77] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 52–62, April 1999.

[78] Michael J. Wirthlin and Brad L. Hutchings. A dynamic instruction set computer. In Peter Athanas and Kenneth L. Pocek, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, April 1995.

[79] Michael J. Wirthlin and Brad L. Hutchings. Sequencing run-time reconfigured hardware with software. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 122–128, February 1996.

[80] Ralph D. Wittig and Paul Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, April 1996.

[81] Xilinx. *The Programmable Logic Data Book*, 1994.

[82] Tsukasa Yamauchi, Shogo Nakaya, and Nobuki Kajihara. SOP: A reconfigurable massively parallel system and its control-data-flow based compiling method. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 148–156, April 1996.

[83] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C compiler for a processor with a reconfigurable functional unit. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 95–100, February 2000.

Appendix A

The Garp Architecture

A.1 Introduction

The Garp processor architecture combines an industry-standard MIPS processor with a new reconfigurable computing device that can be used to accelerate certain computations. Figure A.1 shows the organization of this architecture at the highest level. The core of Garp is an ordinary processor supporting the MIPS-II instruction set. Added to this is a device called a *reconfigurable array*, which is a two-dimensional array of small computing elements interconnected by a network of wires. Garp's reconfigurable array somewhat resembles *field-programmable gate arrays* (*FPGAs*) available from Xilinx, Altera, and other manufacturers.

Each computing element in the reconfigurable array can perform a simple logical or arithmetic operation on operands 2 bits in size. Larger computations are achieved by aggregating these small elements into larger computational circuits. The function of each array element and the connections between the elements are determined by a *configuration* of the array, which is loaded under the direction of the main processor. The array's configuration can be changed as often as desired, allowing the array to be applied to different pieces of a computation over time.

Use of the reconfigurable array is controlled exclusively by the program executing on the main processor. Although any program can execute entirely on the main processor without referencing the reconfigurable array at all, certain computations can be completed faster by the array than by the main processor. Thus it is expected that for certain loops or subroutines, programs will switch execution temporarily to the array to obtain a speed

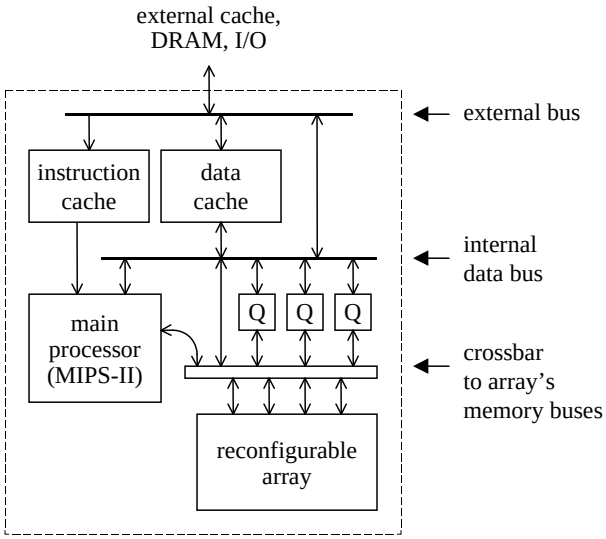


Figure A.1: Basic organization of Garp.

advantage.

This document defines the Garp architecture by detailing Garp's extensions to the MIPS-II architecture. Documentation for the MIPS-II architecture can be found elsewhere. The reconfigurable array itself is described first in Section A.2, after which Section A.3 covers the integration of the array with the main processor and memory system.

A.2 Reconfigurable array

The core of the reconfigurable array is a two-dimensional matrix of small processing elements called *blocks* (Figure A.2). One block on each row is known as a *control block*, and the rest of the blocks are *logic blocks*. The number of columns of blocks is fixed at 24. The number of rows is implementation-specific, but can be expected to be at least 32.

The basic "quantum" of data within the array is 2 bits. All wires are organized in pairs to transmit 2-bit quantities, and logic blocks operate on these values as 2-bit units. Operations on 32-bit quantities thus generally require 16 logic blocks.

As Figure A.1 shows, the array has access to the standard memory hierarchy of the main processor. Four *memory buses* run vertically through the rows for moving information into and out of the array (Figure A.2). During array execution, the memory buses are used for moving data to and from memory and/or the main processor. For memory accesses,

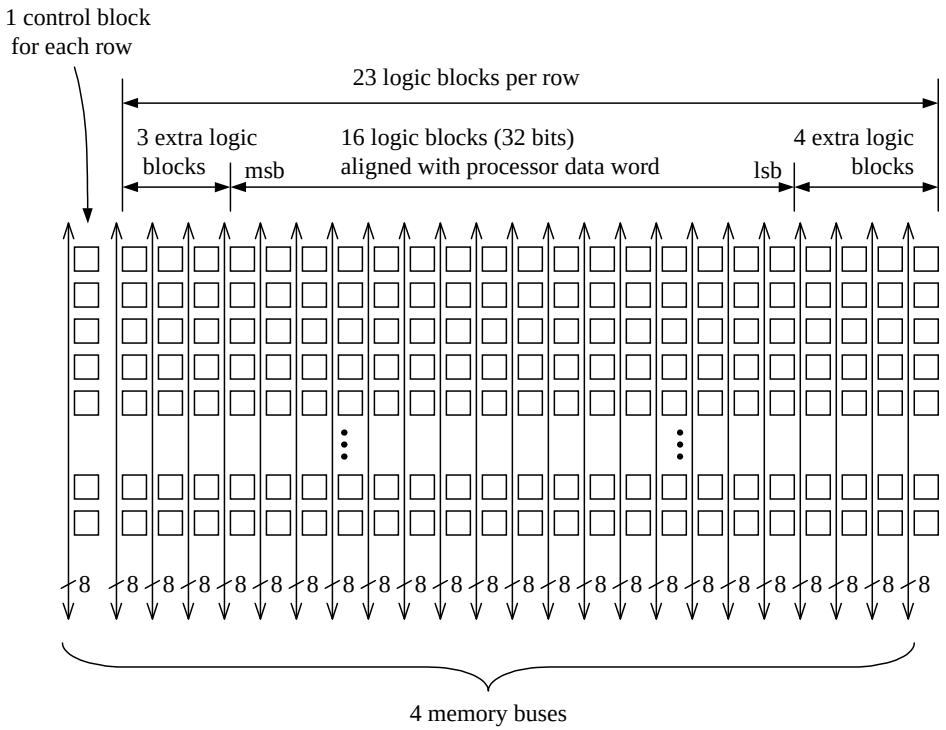


Figure A.2: Structure of the reconfigurable array. In addition to the memory buses, the array blocks are connected by an internal wire network (not shown).

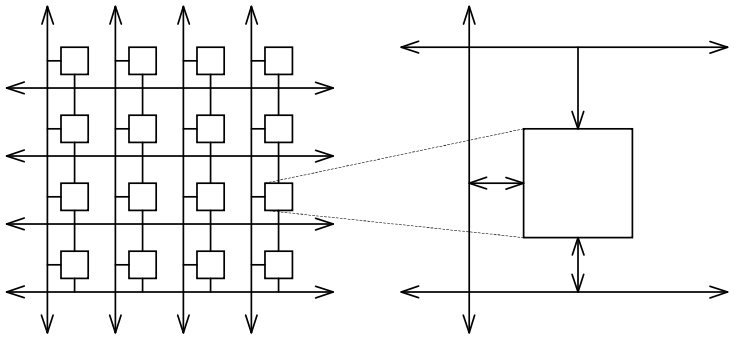


Figure A.3: Internal wiring within the array (independent of the memory buses). Here each arrow represents multiple physical wire paths.

transfers are limited to the central portion of each memory bus, corresponding to the middle 16 logic blocks of each row. For loading configurations and for saving and restoring array state, the entire bandwidth of the memory buses is used.

The memory buses are not available for moving data between array blocks; instead, an internal *wire network* provides connections between blocks. Wires of various lengths run orthogonally vertically and horizontally. Figure A.3 summarizes the available wire paths. Vertical wires can be used to communicate between blocks in the same column, while horizontal wires can connect a block to others in the same row or in the next row below. There are no connections from one wire to another except through a logic block. However, every logic block includes resources for potentially making one wire-to-wire connection independent of its other obligations.

In addition to performing a small computation, each logic block can hold a few bits of data in registers. These data registers are latched synchronously according to an *array clock*, the frequency of which is fixed by the implementation. No relationship between the array clock and the main processor clock is required, although it is intended that the two clocks be the same. Like in the main processor, the array's clock governs the progress of a computation in the array.

Each logic block can implement a function of up to four 2-bit inputs. Operations on data wider than 2 bits can be achieved by adjoining logic blocks along a row. Construction of multi-bit adders, shifters, and other major functions along a row is aided by hardware invoked through special logic block modes. In particular, a fast carry chain runs right-to-left across each row to facilitate large adders and comparators that execute in a single array clock cycle. Since there are 23 logic blocks per row (the leftmost block on each row being a control block), there is space on each row for an operation of 32 bits, plus a few logic blocks to the left and right for overflow checking, rounding, control functions, extended data widths, or whatever is needed.

Figure A.4 shows the main data paths through a logic block. Four 2-bit inputs (A, B, C, and D) are taken from adjacent wires and are used to derive two outputs. One output is calculated (Z), and the other is a direct copy of an input (D). Each output value can be optionally buffered in a register, after which the two 2-bit outputs can be driven onto as many as three pairs of wires leading to other logic blocks. The logic block registers can also be read or written over the memory buses.

The next few subsections cover the core array architecture in more detail: first

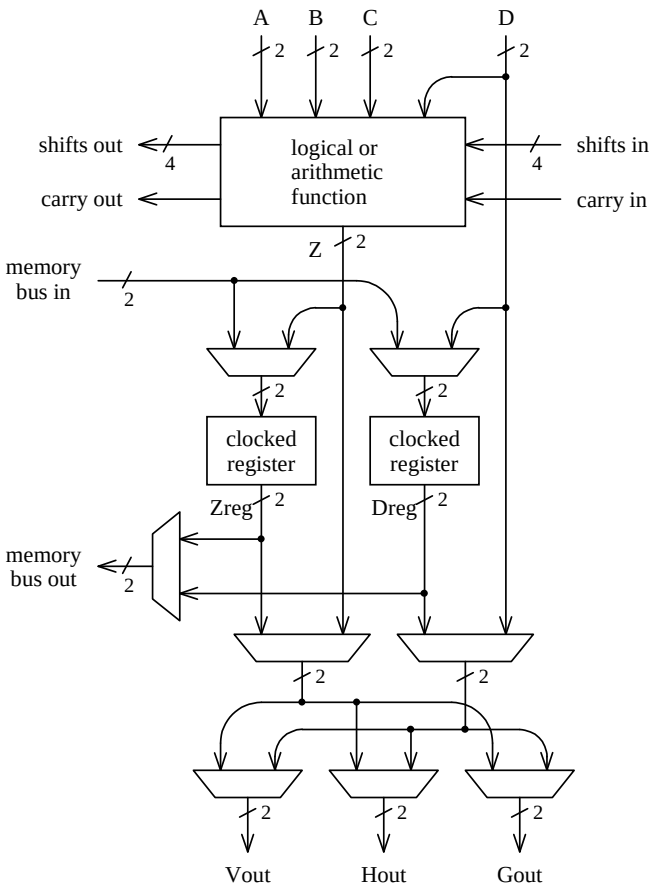


Figure A.4: Simplified logic block schematic.

the inter-block wire network, then the data paths within the logic blocks, and finally the available logic block functions (illustrated in Figure A.4 as a nondescript box). Discussion about the control blocks and the memory buses are deferred until the integration of the array with the main processor is covered in Section A.3.

A.2.1 Internal wire network

Internal wires run vertically and horizontally within the array for moving data between logic blocks. All wires in the network are grouped into pairs to carry 2-bit quantities. Each pair of wires can be driven by only a single logic block but can be read simultaneously by all the logic blocks spanned by the pair. The wire network is passive, in that a value cannot jump from one wire to another without passing through a logic block.

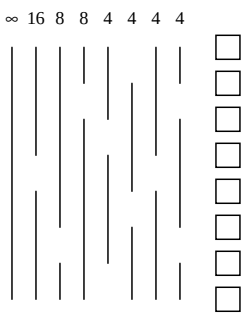
The internal wires are divided into three groups: the *vertical wires* (also called *V wires*), the *global horizontal wires* (*G wires*), and the *local horizontal wires* (*H wires*). Wires running horizontally between logic block rows are either global (G wires) or local (H wires). The G wires span the entire 24-block width of the array, while the H wires nominally span exactly 11 blocks. Only the V wires run vertically and come in a range of lengths.

The pattern of horizontal wires is not the same as that of the vertical wires, so the vertical and horizontal dimensions of the array are not symmetric. This asymmetry is due to the preference for aligning multi-bit operations across rows and not columns. The three categories of wires (V, G, and H) are described in turn below.

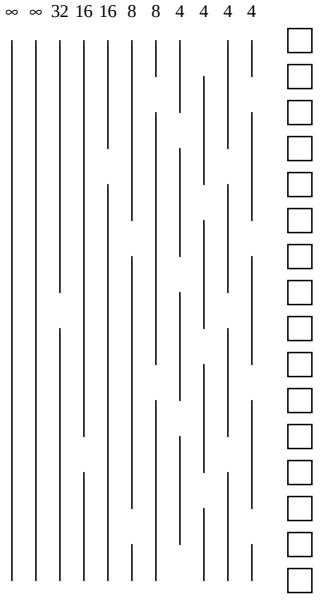
Vertical wires (V wires)

Each column of array blocks has a set of vertical wires (V wires) for making connections among the blocks in that column. Because the number of rows in the array is not strictly fixed, the amount of vertical wiring available depends on the number of rows a configuration has. Figure A.5 illustrates the patterns of vertical wires for configurations with 8, 16, and 32 rows. Each V wire spans a specific set of blocks, any one of which can be configured to drive the wire. All logic blocks spanned by a wire can read from that wire simultaneously. By configuring the vertical wires of several columns in concert, multi-bit values are easily moved among array rows.

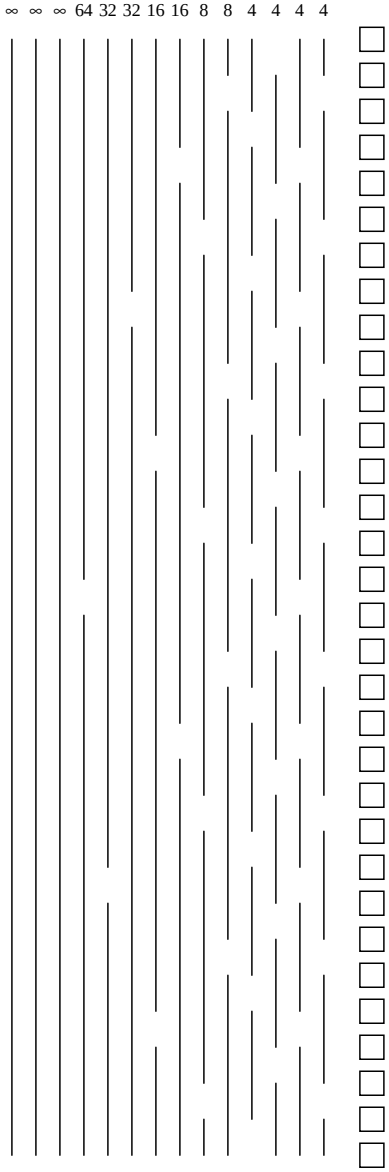
Each pair of wires has a nominal length, shown along the tops of the subfigures in



(a) Vertical wires for 8 rows.

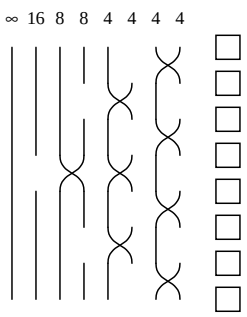


(b) Vertical wires for 16 rows.

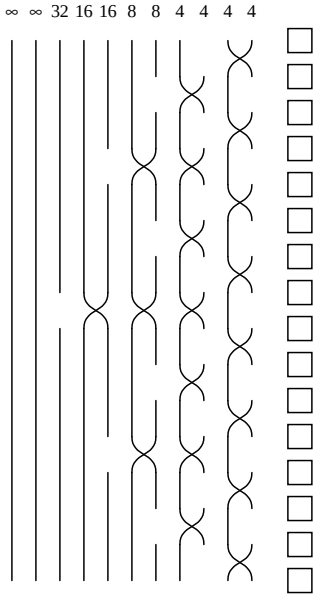


(c) Vertical wires for 32 rows.

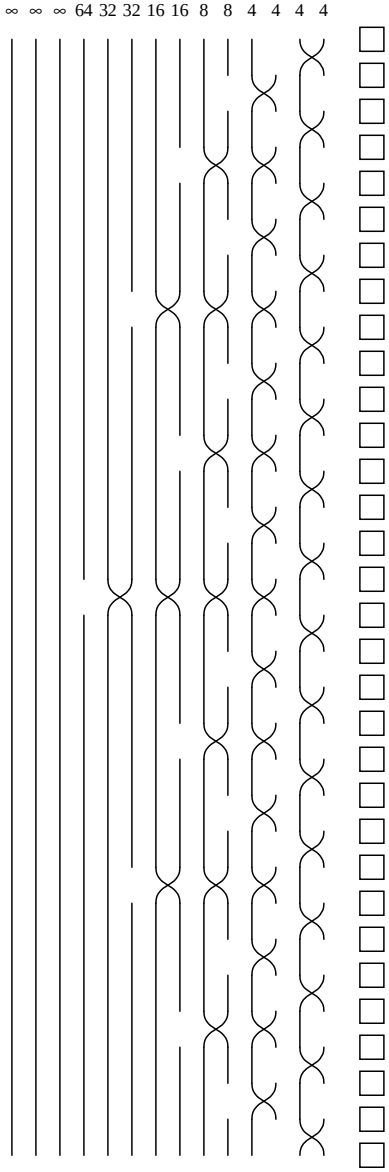
Figure A.5: The vertical wires (V wires) for arrays of various sizes. The boxes represent a single column of the array. Each line drawn actually represents a pair of wires (2 bits). Each wire pair can connect to all of the blocks it spans vertically. The numbers at the top give the nominal lengths of different wires.



(a) Vertical wires for 8 rows.



(b) Vertical wires for 16 rows.



(c) Vertical wires for 32 rows.

Figure A.6: Twisting of the vertical wires to obtain a recursive structure. Compare with Figure A.5. Note that the pattern of vertical wires for an 8-row array is repeated in the upper and lower 8 rows of a 16-row array, and the 16-row pattern is in turn repeated in the upper and lower halves of a 32-row array.

Figure A.5. Except for some wires with nominally infinite length (global vertical wires), the nominal lengths of wires are all powers of 2. The actual length of a wire can be shorter than its nominal length if the wire would extend above the topmost row or below the bottommost row (or both). Thus although a 32-row array includes wires with nominal length 64, no such wire is longer than 32 blocks in reality. The same obviously applies for the global wires labelled as having infinite length.

Each doubling in the number of rows merits an increase in the number of wire tracks, as seen in Figure A.5. An array of 8 rows has wires up to a nominal length of 16, and one global wire pair (wires of infinite length). An array of 16 rows adds wires with nominal length 32, and one more global wire pair. Each successive doubling adds three new wire tracks, one of which is a global wire pair. An array of 64 rows has wires with nominal lengths up to 128, as well as 4 global wire pairs.

At each logic block, every vertical wire to which the block could connect has assigned to it a unique index that identifies the wire from that logic block. A configuration uses these indices to specify the vertical wires to which a block connects. The assignment of indices to wires is based on a peculiar *twisting* of the wires illustrated in Figure A.6. (This twisting gives the vertical wires a recursive structure, a property which can also be exploited to improve the efficacy of the *configuration cache* introduced in Section A.3.1.) Numbers are assigned to wires according to how close the wire is to the logic block in Figure A.6. The closest wire is assigned index number 0, the next closest number 1, and so on. Note that, because of the twisting, a wire's number can change from one logic block to another. The assignment of indices is therefore different for each logic block.

There can be at most one driver for each V wire. Configurations are checked when they are first loaded to ensure that this requirement is met. Configurations failing this test cannot be loaded.

Global horizontal wires (G wires)

Unlike the vertical wires, which are always associated with only a single column of array blocks, the G and H wires exist between rows and are thus accessible by logic blocks in the rows above and below the wires (Figure A.7). A horizontal wire can be read from both above and below the wire, but it can be driven only by a logic block in the row above the wire. Thus, a horizontal wire can be used to communicate among the columns of a

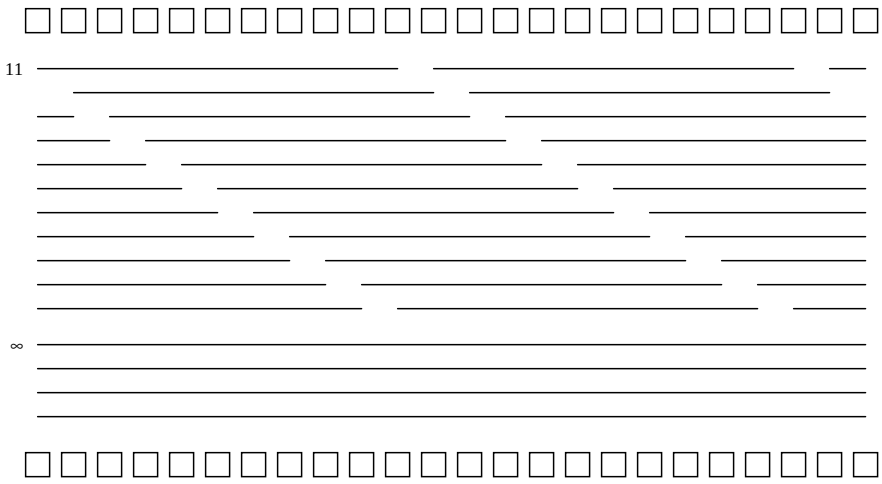


Figure A.7: The horizontal wires between two rows. Again, each line actually represents a pair of wires (2 bits). There is a full set of pairs of length 11 (H wires), and four 2-bit buses that span the entire width of the array (G wires). Each wire pair can be read by all of the blocks it spans horizontally, from logic blocks both above and below the wires.

single row, or from a logic block in one row to a different column in the row immediately below. This bias favors computations that proceed downward from one row to the next.

The G wires are the ones in Figure A.7 with nominally infinite length. A G wire can be driven by any logic block in the row above the wire. As with the V wires, a configuration that has more than one driver for a G wire cannot be loaded. Although Figure A.7 shows the G wires as spanning the control blocks in the leftmost column of the array, control blocks cannot examine or drive the G wires (Section A.3.2).

Local horizontal wires (H wires)

The remaining wires in Figure A.7 are H wires, all with nominal length 11. Like the G wires, each H wire can be driven by a logic block from above the wire and can be read by any block above or below the wire. Figure A.8 shows the logic blocks reachable via an H wire when the wire is driven by the logic block centered above the wire.

Unlike the other two wire categories (V and G), the H wires are unique in that there are limited options for choosing which logic block drives each H wire. For each row, a single choice is made that determines a unique driver for *all* the H wires immediately below that row. Figure A.9 illustrates the three options available to each row. The default is for every H wire below the row to be driven from the center as in Figure A.8. Alternatively, every

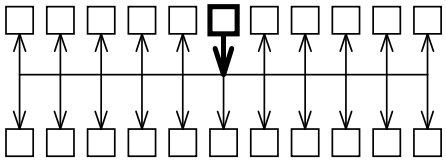
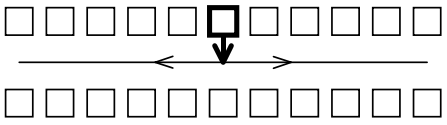
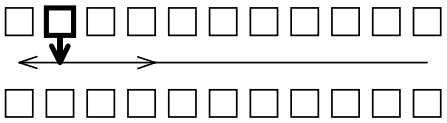


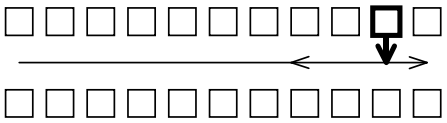
Figure A.8: The logic blocks reachable via an H wire driven from the center.



(a) Driven from the center.



(b) Driven from the left end (shift right).



(c) Driven from the right end (shift left).

Figure A.9: The three options for driving the H wires below a row.

H wire below the row can be driven from near the left end of the wire (Figure A.9(b)); or every H wire below the row can be driven from near the right end of the wire (Figure A.9(c)).

Which of the three options will be used for driving the H wires across an entire row is determined by the control block at the end of the row. Because the choice of driver is made for all wires along a row in concert, every H wire automatically has exactly one driver. It is not possible for a configuration to specify more than one driver for any H wire.

A.2.2 Logic block configurations

The principle data paths within a logic block are depicted in Figure A.4. A logic block selects up to four 2-bit inputs, A, B, C, and D, from among the wires at hand, and performs a logical or arithmetic function on these inputs to generate the output value Z. This value is optionally buffered in an internal register and then driven onto as many as three adjacent wire pairs leading to other logic blocks. At the same time, the original D input can also be optionally buffered and driven onto any of the same wire pairs.

A logic block can drive output values simultaneously onto exactly one of the V wire pairs, plus one of the G wire pairs, plus one of the H wire pairs. It is not possible for a

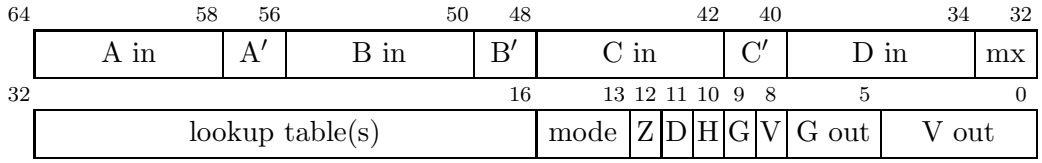


Figure A.10: Logic block configuration encoding. 64 bits of configuration state are needed for each active block. The A′, B′, C′, mx, lookup table, and mode fields together determine the logic block function (Section A.2.3).

[63..58] A in	
000000	$A = 00$ (binary)
000001	$A = 10$ (binary)
000010	$A =$ internal Z register
000011	$A =$ internal D register
010000	$A =$ V wire pair 15
⋮	⋮
011111	$A =$ V wire pair 0
100000	$A =$ leftmost H wire pair above
⋮	⋮
101010	$A =$ rightmost H wire pair above
101100	$A =$ G wire pair 3 above
⋮	⋮
101111	$A =$ G wire pair 0 above
110000	$A =$ leftmost H wire pair below
⋮	⋮
111010	$A =$ rightmost H wire pair below
111100	$A =$ G wire pair 3 below
⋮	⋮
111111	$A =$ G wire pair 0 below

Figure A.11: Configuration encoding for logic block inputs. The four inputs, A, B, C, and D, have identical encodings.

[12] Z	
0	suppress latching of Z register; output Z directly
1	latch Z register every cycle; output Z register

[11] D	
0	suppress latching of D register; output D directly
1	latch D register every cycle; output D register

[10] H	
0	$Hout = Z$
1	$Hout = D$

[9] G	
0	$Gout = Z$
1	$Gout = D$

[8] V	
0	$Vout = Z$
1	$Vout = D$

[7..5] G out	
000	no output to G wires below
100	output *Gout* to G wire pair 3 below
⋮	⋮
111	output *Gout* to G wire pair 0 below

[4..0] V out	
00000	no output to V wires
10000	output *Vout* to V wire pair 15
⋮	⋮
11111	output *Vout* to V wire pair 0

Figure A.12: Configuration encoding for logic block registers and outputs.

single logic block to drive more than one V wire pair, more than one G wire pair, or more than one H wire pair. A logic block can drive any one (or none) of the V wire pairs at hand, and can drive any one (or none) of the G wire pairs below the block (but not above). As stated in the previous section, every logic block drives one H wire pair below, in a pattern across each row that is selected by the control block at the end of the row. In each direction (V, G, and H), the output can be selected from either the Z or the D result.

For each logic block, 64 bits of internal *configuration state* determine the active configuration of that block. The configurable elements of a logic block include the sources of the inputs, the function performed on those inputs, the operation of the registers, and the destinations for the outputs. Figures A.10 through A.12 detail the encodings of a logic block's configuration state.

Figure A.4 shows that a logic block's registers can be read from or written to the memory buses. However, this path is not under the control of the logic block itself and so is not represented in the logic block configuration. Transfers over the memory bus are instigated by the main processor and/or by the control block at the end of each array row (Section A.3).

Note that if the logic block function does not require all four inputs, the D path can be used as a completely independent path—for example to route and/or buffer a value between wires (Figure A.13). Many of the available logic block functions ignore the D input, leaving it free for this purpose.

In addition to selecting among the internal wires, any of the A, B, C, or D inputs can be set to a constant. The supported constant values are binary 00 and 10. Binary values 01 and 11 are not provided because they can be synthesized from the other two in most cases.

The outputs of the internal registers can also be connected back as logic block inputs, as is illustrated for the Z register in Figure A.14. A notable application of this feature is to use the D path to buffer an input or the function output for an extra cycle. Figure A.15 shows how an input can be delayed by connecting the D register output to one of the function inputs. Conversely, in Figure A.16 the D path is tied to the Z register output to delay the Z result one cycle.

Each register can operate in either of two modes. If a register is used as a buffer, it automatically latches a new value every clock cycle. Alternatively, a register can be bypassed on output, in which case it never latches except when it is written to via a memory bus.

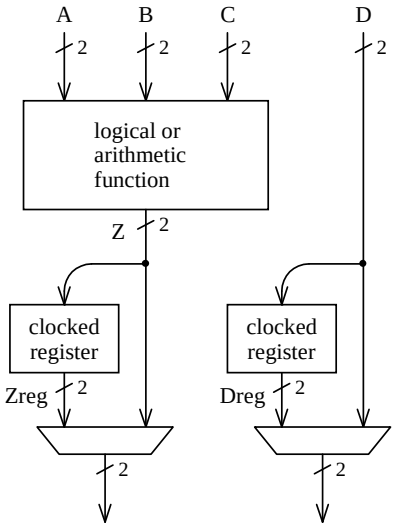


Figure A.13: Use of the D input as a completely separate path for routing or copying.

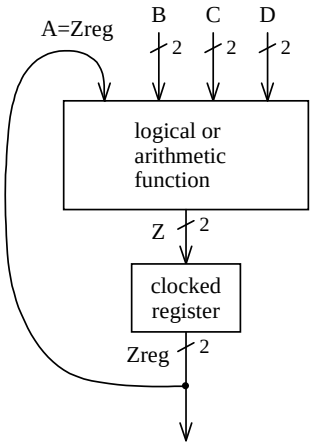


Figure A.14: Any of the A, B, C, or D inputs can be taken from the internal registers.

The input side of a bypassed register is thus effectively decoupled from the logic block. Note that this implies, for instance, that the registers in the examples of Figures A.14 and A.16 could not be bypassed on output, or else they would cease to act as buffers.

From the perspective of the memory buses, a bypassed register will hold a value written to it until it is updated again over a memory bus. By connecting the output of such a register to a logic block input, a value latched from a memory bus can be used immediately in the logic block function. Figure A.17 demonstrates the use of both internal registers for holding memory bus inputs in this way.

A register selected as a buffer can be written to over a memory bus too, in which case the memory bus value supercedes any internal value for that clock cycle. The value written from the memory bus will be subsequently overwritten in the next clock cycle.

Figure A.18 gives a more detailed view of the logic block internal paths.

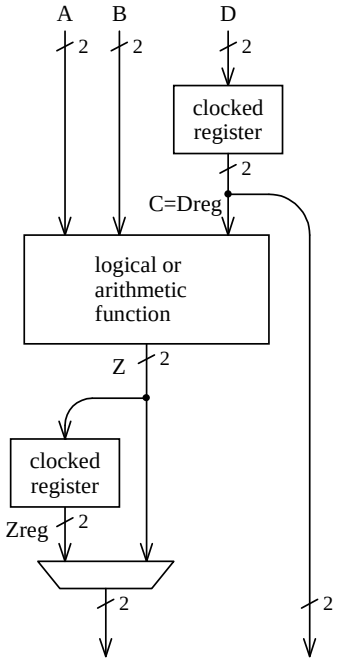


Figure A.15: Delaying one logic block input using the D path.

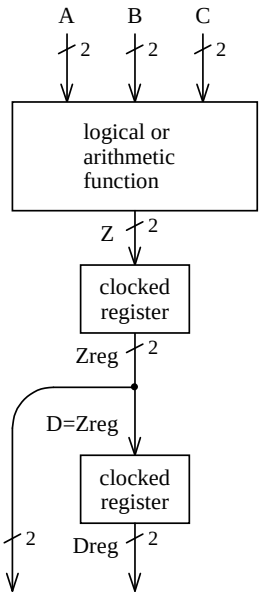


Figure A.16: Delaying the Z output using the D path.

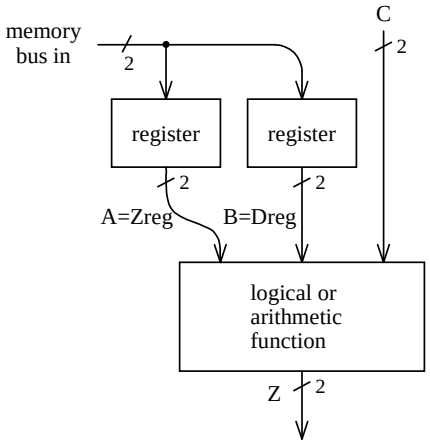


Figure A.17: Values read over the memory buses can be latched into either internal register and used immediately as function inputs within the logic block.

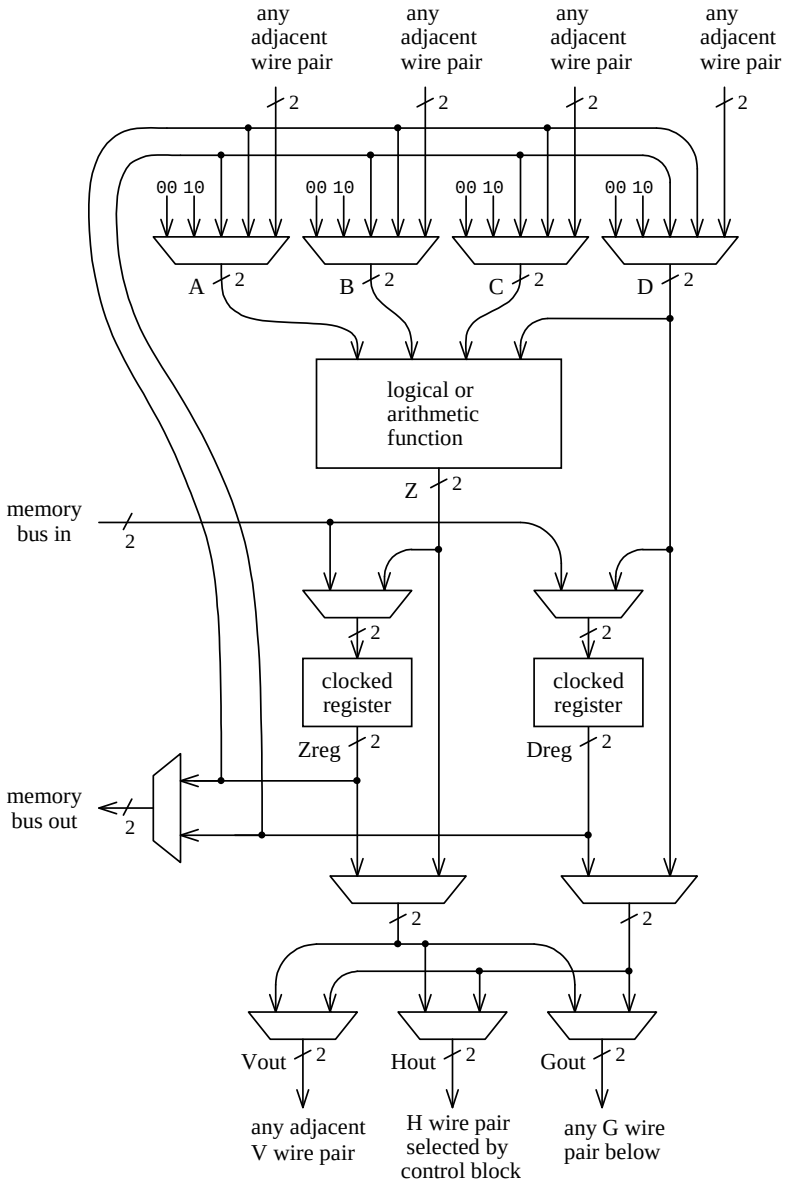


Figure A.18: A more complete logic block diagram.

mode [15..13]	mx [33..32]		
000	D′	table mode	
001	01	split table mode	
01k	00	select mode	(k = 0 suppresses shifts in)
01k	01	partial select mode	”
10k	result	carry chain mode	(k = 0 suppresses carry in)
11k	result	triple add mode	(k = 0 suppresses shifts, carries in)

Figure A.19: The function mode encodings. For many modes, bit k (bit 13) determines whether shifts and carries in are to be suppressed.

A.2.3 Logic block functions

This section details the computational functions that a logic block can perform. The main inputs to this function are the four values A, B, C, and D, each of which is 2 bits in size. The primary output is Z, also 2 bits.

In addition to the primary ones, several miscellaneous inputs and outputs are associated with specific logic block functions. Most of these extra connections are to a block's nearest leftmost and rightmost neighbors to support multi-bit functions built out of multiple logic blocks along a row. Details about these extra inputs and outputs are given below as each function mode is reviewed.

A logic block's function is determined by several fields in the active configuration—the A′, B′, C′, mx, lookup table, and mode fields (Figure A.10). The mode and mx fields together select among the six possible function modes (Figure A.19). Each mode is defined below. Most modes make some use of the lookup table, although some do not. In all modes, the A′, B′, and C′ fields choose some form of initial perturbation of the corresponding inputs.

Table mode

Table mode is the basic mode for performing simple logical functions, as shown in Figure A.20. After each input passes through a *crossbar* function, a table lookup implements an arbitrary bitwise logical operation on the four inputs to give the result. Table mode is selected when the configuration's mode field is 000 (binary).

The operation of the crossbars is illustrated in Figure A.21. As its name implies, each crossbar allows each of its 2 output bits to be selected independently from either of its input bits. There are four possibilities: pass the incoming bits through unperturbed,

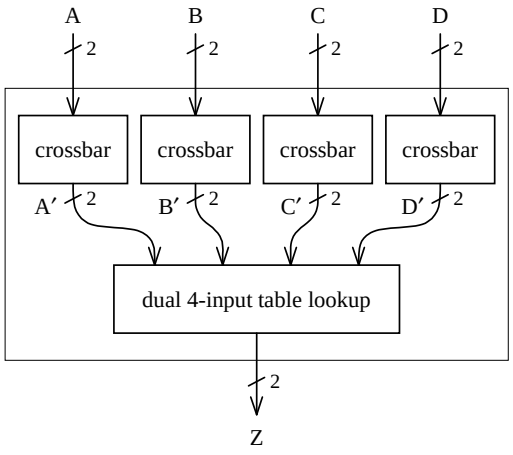
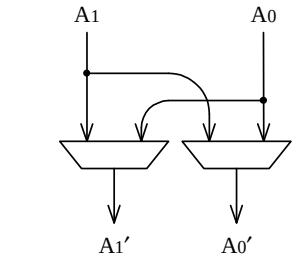


Figure A.20: Table mode (mode = 000). The mx field selects the crossbar function for D'.



[57..56] A′	
00	$A' = A_0A_0$
01	$A' = A_0A_1$
10	$A' = A_1A_0$
11	$A' = A_1A_1$

Figure A.21: The crossbar functions and encoding.

Figure A.22: Interpretation of the lookup table in table mode. The lookup table function f takes 4 input bits and returns a single output bit. Listed underneath each table entry is the pattern of input bits corresponding to that table output. The 2 bits of Z are calculated independently using the same function: $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$.

duplicate incoming bit A_0, duplicate incoming bit A_1, or swap the two bits. As there is no D′ configuration field, the mx field defines the D crossbar function in this mode.

The 16-bit lookup table specifies an arbitrary 4-input logical function f, as shown in Figure A.22. This function is independently applied to the high and low bits of the four inputs to generate the high and low bits of the result; that is, $Z_1 = f(A'_1, B'_1, C'_1, D'_1)$ and $Z_0 = f(A'_0, B'_0, C'_0, D'_0)$.

Split table mode

Split table mode (Figure A.23) is just like table mode except that D' is fixed at 10 (binary). This has the simple effect of allowing the two bits of Z to be calculated using

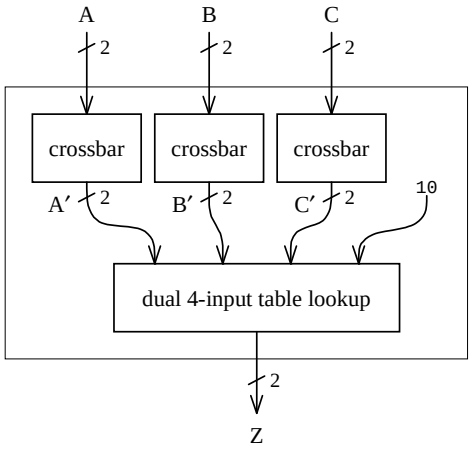


Figure A.23: Split table mode (mode = 001, mx = 01).

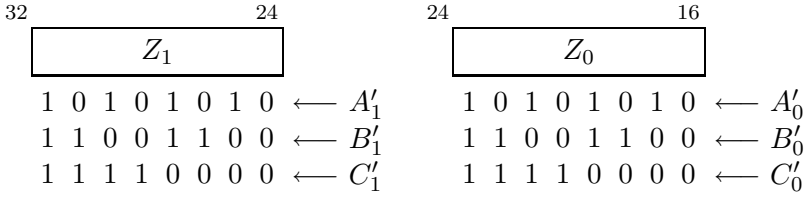


Figure A.24: Interpretation of the lookup table in split table mode. Forcing $D'_1 = 1$ and $D'_0 = 0$ causes the 2 bits of Z to be calculated based on separate 3-input functions. Compare with Figure A.22.

separate 3-input functions, as shown in Figure A.24. The usual D path through the logic block is not affected (Section A.2.3); the D input to the logic block function is simply ignored by the function box. Split table mode is chosen when mode = 001 and mx = 01.

Select mode

Select mode implements a multiplexor of four inputs, as illustrated in Figure A.25. In place of the crossbars, three of the inputs, A, B, and C, are optionally shifted and/or complemented (inverted) to form the perturbed values A', B', and C'. The resulting C' is then used to select one of the other four inputs as follows:

C'	Z
00	A'
01	B'
10	D
11	*Hout* from row above

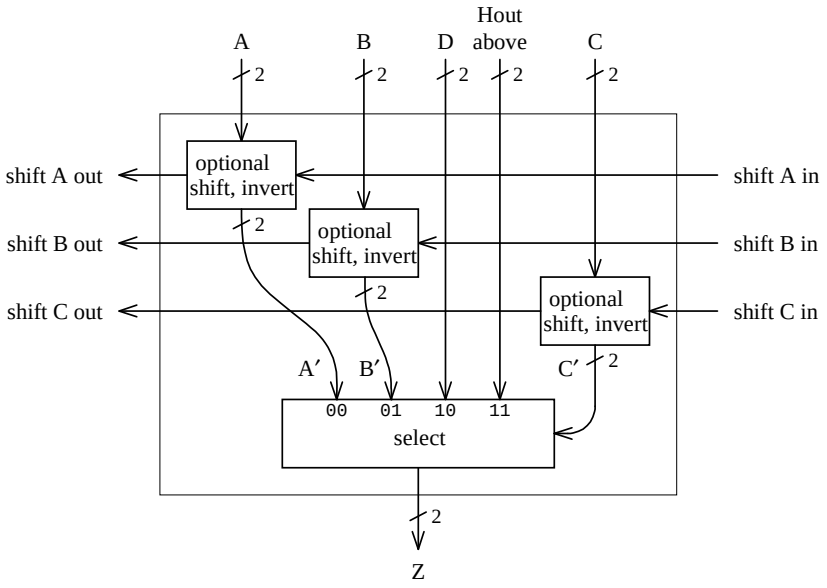
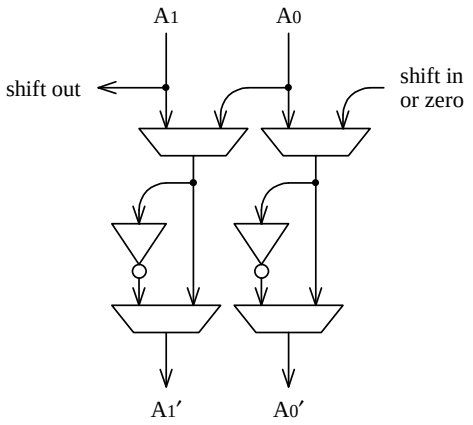


Figure A.25: Select mode (mode = 011, mx = 00). If mode bit 0 is set to 0 (i.e., mode = 010, mx = 00), the function is the same except that all shifts in are assumed to be 0.



[57..56] A′	
00	$A' = A_1 A_0$
01	$A' = \neg(A_1 A_0)$
10	$A' = A_0 A_{-1}$
11	$A' = \neg(A_0 A_{-1})$

Figure A.26: The shift-invert functions. The "¬" symbol represents logical complement. If the function mode suppresses shifts in, bit A_{-1} is 0. Otherwise, bit A_{-1} is taken from bit A_1 from the logic block on the right (regardless of the mode the logic block on the right is in).

One of the inputs is the value of *Hout* from the logic block in the same column in the row above. This is the value being driven on the H wires by the logic block immediately above. (There is always some such value, with the exception of the first row of a configuration.) It is improper for C' to be 11 (binary) if select mode is used on the topmost row of a configuration.

The functions of the shift-invert blocks are shown in Figure A.26. A 2-bit input is first optionally shifted left one bit, and then the resulting 2-bit value (shifted or not) is optionally complemented. When a shift is performed, a bit to shift in is taken from the high bit of the same input from the logic block to the immediate right (regardless of what mode the logic block on the right is in). It is improper to depend on the bit shifted into the rightmost logic block on a row.

The shifts into all three shift-invert blocks can together be forced to 0 by the configuration. This option is useful for the rightmost logic block of multi-block functions and also for the rightmost logic block on a row. Shifts into the individual shift-invert blocks cannot be independently suppressed.

Select mode is chosen with mode = 010 or 011, and mx = 00. The first case (mode = 010) suppresses shifts in, while the second (mode = 011) does not.

Select mode performs no table lookups. The configuration's lookup table field must be set to the constant

32																16
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	

Partial select mode

Partial select mode (Figure A.27) is like ordinary select mode, but with a different set of values from which to choose:

C'	Z
00	A'
01	B'
10	B (not shifted or inverted)
11	00

This mode is enabled when mode = 010 or 011, and mx = 01. Setting mode to 010 suppresses perturbation shifts in, while mode = 011 does not. Unlike ordinary select mode, in partial select mode there is no restriction on the value of C' on the topmost row of a configuration.

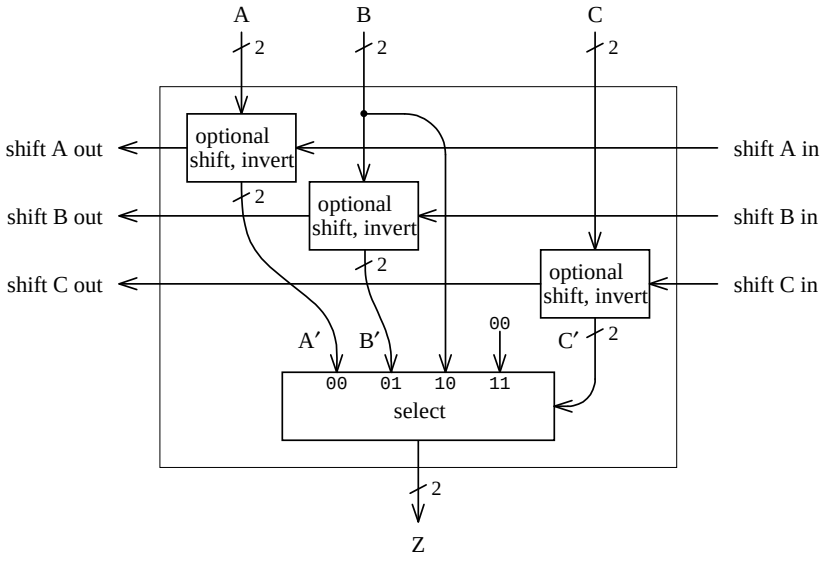


Figure A.27: Partial select mode (mode = 011, mx = 01). Again, if mode bit 0 is set to 0 (mode = 010, mx = 01), the function is the same except that all shifts in are assumed to be 0.

Carry chain mode

Carry chain mode performs a logical or arithmetic function involving the fast carry chain across a row. The mode is diagramed in Figure A.28. Only three inputs, A, B, and C, are used; the D input is ignored. (The D path still exists and can be employed separately; see Section A.2.2.) Like table mode, the three inputs are passed through crossbar functions before being applied to table lookups. The results from the table lookup are used to control the carry chain, and then these same values are logically combined with the carry chain output to obtain the final result Z.

The table lookups deliver a total of four bits that control the carry chain: a *propagate* and a *generate* signal are associated with the low bit of the result, and another pair of such signals are associated with the high bit of the result. Figure A.30 shows how these control bits affect the carry chain. If a *propagate* bit is 1, the carry into that position is propagated to the next higher bit position; otherwise, the corresponding *generate* value is used as the carry out to the next bit position. When *propagate* is 1, the *generate* value is ignored by the carry function (although it may still be used in the result function; recall Figure A.28). As Figure A.30 shows, the carry chain repeats the same operation at each

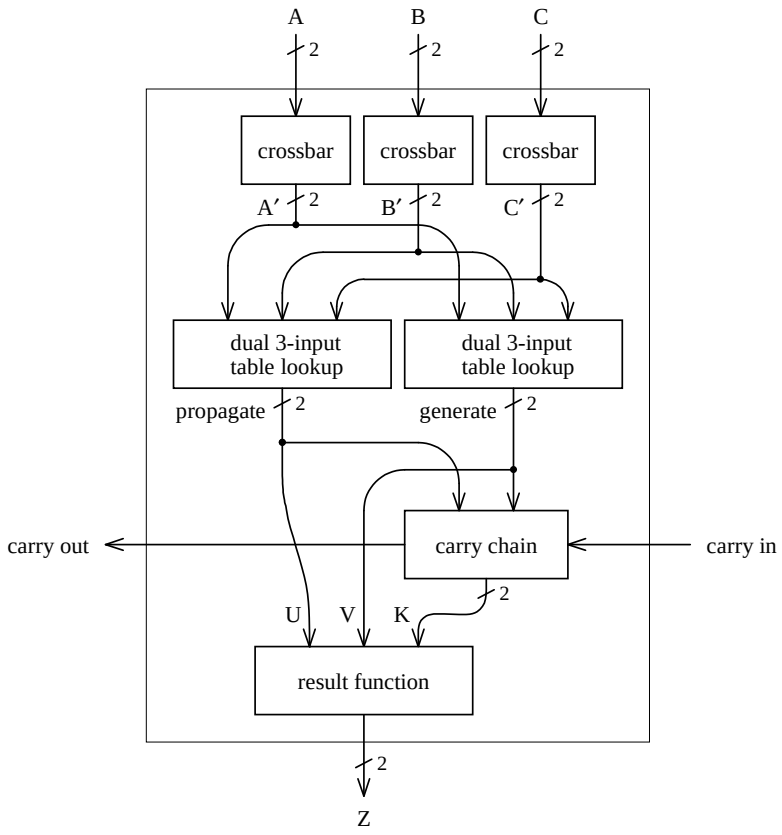


Figure A.28: Carry chain mode (mode = 101). The mx field selects the result function. If mode bit 0 is set to 0 (i.e., mode = 100), the function is the same except that the carry in is assumed to be 0.

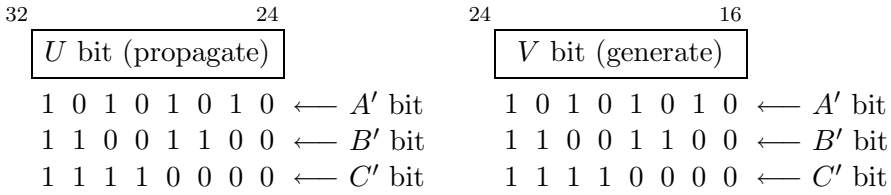


Figure A.29: Interpretation of the lookup table in carry chain mode.

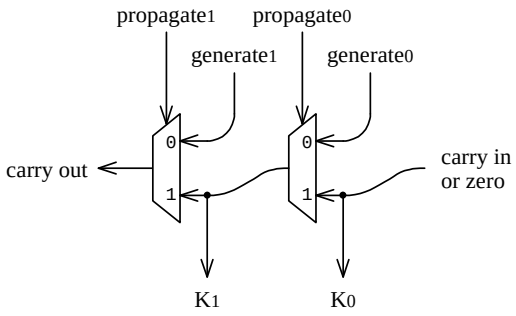


Figure A.30: Operation of the carry chain. The low-order carry in can be forced to 0 by the logic block mode.

[33..32] mx	
00	$Z = V$
01	$Z =$ carry out
10	$Z = U \oplus K$
11	$Z = \neg(U \oplus K)$

Figure A.31: The result functions for modes using the carry chain.

bit position.

The operation of the lookup tables is documented in Figure A.29. There are two 3-input tables: one is the *propagate* table, and the other the *generate* table. Each table is looked up twice, once for the low bit position and once for the high bit position.

The carry chain outputs a 2-bit value K, comprising the carry into each bit position (Figure A.30). This is fed into the result function, along with the original *propagate* and *generate* signals, which are renamed to U and V, respectively. The result function implements one of four bitwise logical functions given in Figure A.31, chosen by the mx field of the configuration.

A logic block is in carry chain mode when mode = 100 or mode = 101. The first case forces the carry into the low bit (K_0) to be 0. The second case accepts the carry out from the logic block on the right. It is improper to depend on the carry in when the logic block to the immediate right is not in carry chain mode. This applies in particular to the rightmost logic block on a row, which has no logic block to the immediate right.

Triple add mode

The most complex mode is triple add mode, which can perform a sum or difference of three inputs (Figure A.32). Each of the three inputs is first passed through a shift-invert function (Figure A.26), and then a carry-save addition is performed on the perturbed inputs. The two outputs of the carry-save addition are used to index two lookup tables to obtain the carry chain *propagate* and *generate* signals (Figure A.33). From this point forward, triple add mode is identical to the simpler carry chain mode.

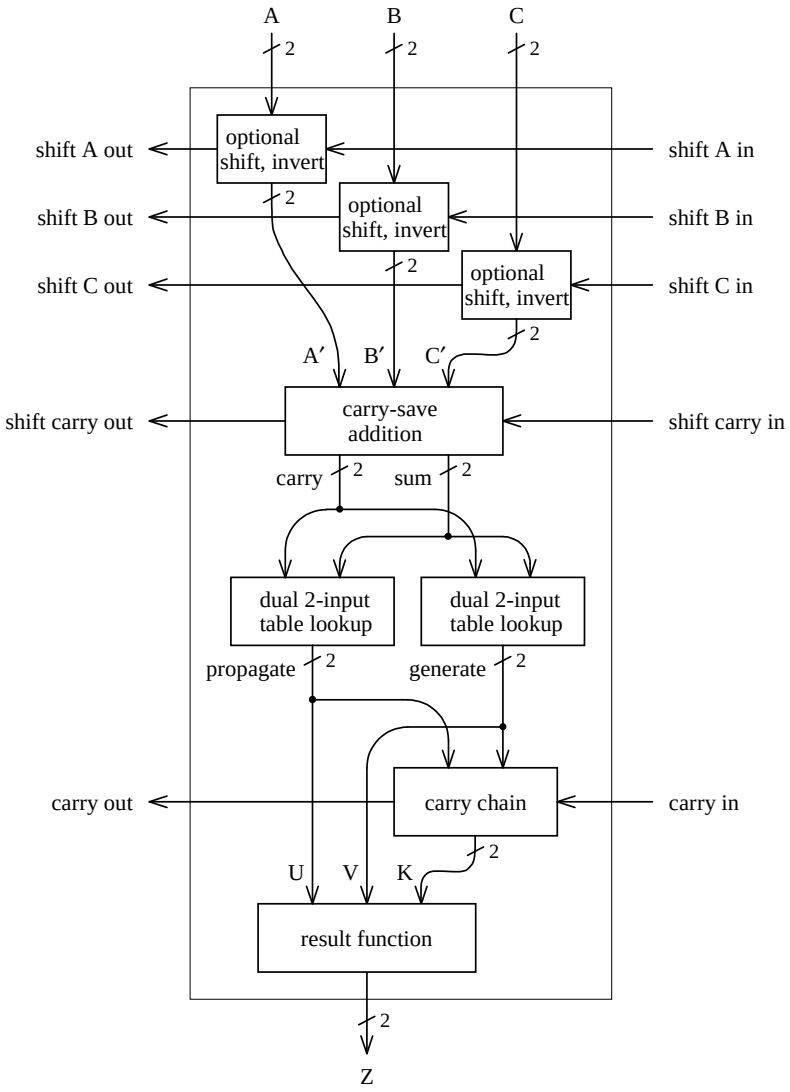


Figure A.32: Triple add mode (mode = 111). The mx field selects the result function. If mode bit 0 is set to 0 (i.e., mode = 110), the function is the same except that all shifts and carries in are assumed to be 0.

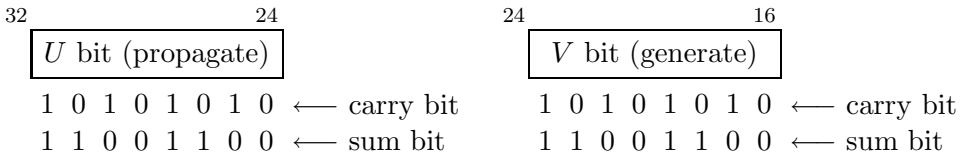


Figure A.33: Interpretation of the lookup table in triple add mode. Each table is represented by eight bits, even though four bits would be sufficient. The redundancy in the tables is required and must be consistent.

The carry-save addition performs the usual function, with a so-called "*sum*" output calculated bitwise as $A' \oplus B' \oplus C'$, and a *carry* output calculated as $(A' \wedge B') \vee (A' \wedge C') \vee (B' \wedge C')$ and shifted left by one bit position in the same manner as the shift-invert functions. As with the shift-invert functions, the shift carry in (ultimately $carry_0$) can be forced to 0 by the configuration mode field.

Triple add mode is selected when mode = 110 or 111. The first case forces the shifts in and the carry in to be 0, whereas the second case accepts these from the logic block on the right. The mx field specifies the result function. As for carry chain mode, it is improper to depend on the carry in or the shift carry in when the logic block to the immediate right is not in triple add mode. Likewise, it is improper to depend on any of the shifts or carries into the rightmost logic block on a row.

A.2.4 Internal timing

Delays within the array are defined in terms of the sequences that can fit within each array clock cycle. Only three sequences are permitted:

- short wire, simple function, short wire, simple function;
- long wire, any function not using the carry chain; or
- short wire, any function.

Any other sequence must be assumed to require multiple clock cycles. A *short wire* is a local horizontal wire (H wire) or a vertical wire of length 8 or less. A *simple function* is either a table mode function or a traversal of the independent "D path" in a logic block. At the end of a cycle, values can be latched in logic block registers without affecting these rules.

Within combinatoric circuits, it is not necessary to latch intermediate results in registers at the end of every clock cycle unless the latches are desired to achieve pipelining. There is, however, a maximum allowed path delay between registers of 8 array clock cycles.

A.3 Integration of array with main processor

The loading and execution of array configurations is under the control of the main processor. Several instructions have been added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the array and the processor's own registers. Configurations and data are transferred to/from the array over the memory buses that run through the entire array (Figure A.2).

During array execution, the array itself can initiate reads or writes to memory (via the memory buses) without intervention by the main processor. Such memory accesses are coordinated by the control blocks at the end of each array row. Array memory accesses go through the same memory hierarchy as the main processor, including the first-level data cache. The array thus has available to it a relatively large, fast memory store which is automatically kept consistent with memory accesses made by the processor.

In addition to on-demand "random" accesses to memory, three *array memory queues* provide enhanced support for sequential memory accesses.

A.3.1 Processor control of array

The main processor's instruction set has been extended with 20 new instructions for controlling the array. The full list of added instructions is documented in Table A.1 at the end of this section.

Array clock counter

Array execution is governed by a countdown counter called the array clock counter. While the clock counter is nonzero, it is decremented by 1 with each array clock cycle. When the array clock counter is zero, the latching of array registers is disabled, effectively stopping the array.

A configuration can be loaded into the array only when the clock counter is zero. After loading a configuration, the main processor can set the array clock counter to nonzero to start the array executing for a given number of clock cycles. The counter can be set using the `gabump` instruction. Various other processor instructions are also able to set the counter in addition to their other functions.

There is no defined relationship between the array clock and the rate at which the main processor executes instruction. To ensure proper synchronization, most processor

instructions that interact with the array first stall until the clock counter reaches zero before performing their function. The clock counter thus provides the mechanism by which array calculations are interlocked with subsequent dependent processor instructions.

Since the number of clock cycles needed for a calculation may not be known in advance, the array has the ability to halt itself whenever its function is complete, by forcibly zeroing the clock counter. How the array can be configured to do this is discussed along with the other functions of the control blocks in Section A.3.2. It is also possible for the processor to halt the array at any time by zeroing the clock counter. The `gastop` instruction which performs this function is covered in connection with context switches in Section A.3.1.

The array clock counter is a 32-bit register, of which only the least significant 31 bits actually count down. The most significant bit is a "sticky" bit: once set, it remains set until the entire counter is forcibly zeroed either by the array or by the processor (`gastop`). Since the array is halted only when the entire 32-bit counter is zero, the most significant bit acts as an "infinity" bit. If the latency of an array calculation is entirely data-dependent, the processor can set the most significant bit of the clock counter to start the array operating indefinitely. The array can then zero the clock when its computation completes. If the processor is ready to receive array results before the array is done, the first instruction attempting to retrieve data from the array will interlock as usual until the counter is zeroed by the array.

Loading configurations

The loading of array configurations is under the control of instructions executed by the main processor. Loading a configuration makes the configuration *active*, so that the configuration controls the behavior of the array. Only one configuration can be active in the array at a time. Loading a new configuration replaces the previous one.

Although logically only one configuration can be loaded at a time, in practice one can expect an implementation to incorporate within the array a *configuration cache* of recently loaded configurations, so that the process of "loading" a configuration does not necessarily involve transferring it from external memory every time. Only a few processor clock cycles should be needed to load a configuration from the configuration cache. If a configuration is not in the cache, it can be expected that close to the full aggregate bandwidth of the memory buses will be used to load it from external memory.

The smallest configuration is one row, and every configuration must fill exactly some number of contiguous rows. When a configuration is loaded that uses less than the entire array, the rows that are unused are automatically made inactive. The first, topmost row of a configuration is row number 0 by default, and subsequent rows are labelled with increasing integers.

The active configuration can be changed only when the array clock counter is zero (the array is halted). The instructions that load configurations will stall waiting for the clock counter to become zero before performing their function.

The simplest instruction for loading configurations is `gaconf`, which takes a single register operand giving the address of the configuration stored in memory. The first 4 bytes (32 bits) at this address are interpreted as a count of the number of rows of the configuration. Following this row count is 8 bytes for each block (control blocks and logic blocks) of the configuration, starting with $24 \times 8 = 192$ bytes for row 0, and so on for each row. The configuration for a row contains first the 8 bytes for the control block, followed by the logic block in the leftmost column 22, on down to the rightmost logic block in column 0. In addition to loading a configuration into the array and making it active, `gaconf` initializes the Z and D registers of all logic blocks to zero.

During the time a configuration is active, its copy in memory must not be changed because it may need to be reloaded at any time. (Reloads can be caused by context switches in a multitasking system, for example.) Furthermore, if an inactive configuration is modified in memory and explicitly reloaded, the changes may not take effect if an earlier unmodified version of the configuration is still in the cache. Before an attempt is made to load a modified configuration, the previous version must be definitely cleared from the cache. The `gacinv` instruction performs this function. It may be executed even while another configuration is running.

The `gaconf` instruction does not allow state to remain in the logic block registers from one configuration to another. A more complex pair of instructions supports configuration *overlays* for this purpose. The `gaalloc` instruction reserves a group of rows into which subsequent configurations will be overlayed. Like `gaconf`, `gaalloc` displaces any currently active configuration and zeros all of the Z and D registers in the array; but no configuration is yet loaded by the instruction. The `gaconfo` instruction loads a configuration into a previously allocated group of rows. An overlayed configuration may not extend beyond the rows allocated by `gaalloc`, but it need not fill the allocation, and it may be loaded starting

at a row other than the first allocated row. All of the register state within the allocated space is preserved from one overlay to another. Nevertheless, if an overlay is smaller than the allocated space, only the rows of the overlaying configuration are made active. Inactive rows maintain their data state until subsequently made active.

The `gaalloc` instruction takes as an operand a pointer to a 32-bit word in memory, the value of which is the number of contiguous rows to allocate. Although logically this indirection through a pointer is unnecessary (the register operand could just as easily have specified the number of rows directly), the pointer is intended to be used as an identifier internally by the cache. If a later execution of `gaalloc` precedes a repeat sequence of `gaconfo` overlays, the same pointer should be used as the operand to both `gaalloc`'s in order to maximize cache utilitization.

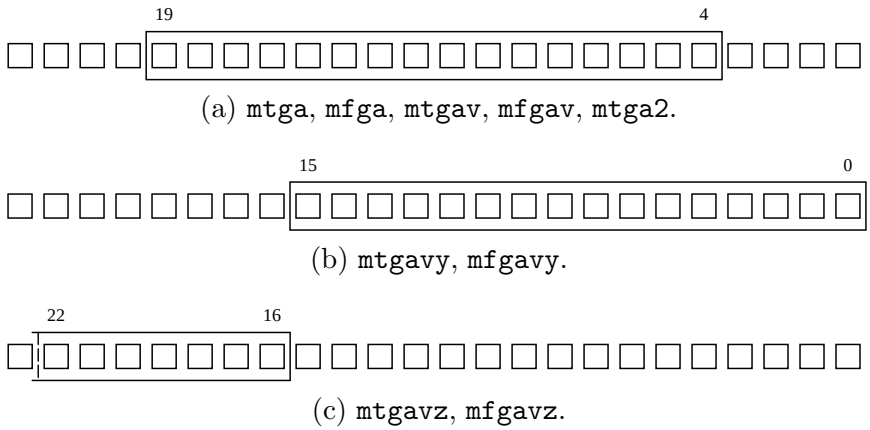
For completeness, `gareset` "unloads" any active configuration. Array activity is disabled until such time as another configuration is loaded.

Transfering data to/from array

The processor has a collection of instructions for copying data between the processor register file and the registers in array logic blocks. Since a single logic block's Z or D register is only 2 bits, most data transfers gang together 16 contiguous logic blocks on a row so that 32 bits of data are copied at a time. An individual transfer copies to or from the 16 combined Z registers or the 16 combined D registers of the 16 logic blocks. For each transfer operation, an array row number must be specified, along with whether the array source/target is to be the Z or D registers. These two parameters (row number and Z/D selection) are encoded as constants within some instructions, while other instructions obtain them from an additional register operand.

Each row has 23 logic blocks, but an individual transfer operation only touches at most 16 of them corresponding to a full 32-bit word. The set of logic blocks read or written is fixed for each particular instruction (Figure A.34). Most instructions copy to/from only the middle 16 logic blocks found in columns 4 through 19 inclusive. A few variant instructions allow access to the logic blocks at the extreme left or right ends of a row. The rightmost logic block is always associated with the least significant 2 bits transferred, and the leftmost logic block is associated with the most significant 2 bits.

The `mtga` (*move to Garp array*) instruction transfers a 32-bit word from a processor



(a) `mtga`, `mfga`, `mtgav`, `mfgav`, `mtga2`.

(b) `mtgavy`, `mfgavy`.

(c) `mtgavz`, `mfgavz`.

Figure A.34: The set of logic blocks read or written by various processor instructions. The `mtga` (*move to Garp array*) instructions copy a 32-bit word from a processor register to a contiguous set of logic block registers along an array row. The `mfga` (*move from Garp array*) instructions transfer a word in the opposite direction. Since logic block registers are 2 bits each, 16 logic blocks correspond to a 32-bit data word. The leftmost block on each row is a control block containing no visible data registers.

register to the Z or D registers of the middle 16 logic blocks of a fixed row. The row number and the choice of Z or D registers are encoded as constants in the `mtga` instruction. The `mfga` (*move from Garp array*) instruction is the same except it transfers in the opposite direction, from the array to a processor register. Instructions `mtgav` and `mfgav` are similar, but instead of hardcoding the array row number and Z/D register choice in the instruction, a second register operand supplies these parameters.

Access to the logic blocks in the leftmost and rightmost columns of a row is provided by variants of `mtgav` and `mfgav`. The `mtgavy` and `mfgavy` instructions access columns 0 through 15 but are otherwise identical to `mtgav` and `mfgav`. At the other end of a row, variants `mtgavz` and `mfgavz` read or write the 14 bits of columns 16 through 22. (Column 23 is left out because it contains the control blocks, which have no visible data registers.) These last two instructions are unusual in that they only transfer 14 bits. For `mtgavz`, the most significant 18 bits of the source processor register are ignored; while in the other direction, `mfgavz` zeros the most significant 18 bits of the destination processor register.

The processor can copy to or from the array only when the array clock counter is zero. If the clock counter is nonzero, a data transfer instruction will stall until the clock counter becomes zero. The instructions `mtga` and `mfga` can also set the clock counter to a small constant after performing their transfer.

Memory queue control

Two processor instructions (`galqc` and `gasqc`) are used to load and store the state of the array *memory queues*. The details of these instructions are deferred until Section A.3.3 when array memory queues are covered.

Saving and restoring array state

Information about the active configuration is stored in three read-only registers:

`$gacr3` – The pointer that was the argument to `gaalloc` when the current array allocation was made. The 32-bit word at this address gives the number of rows allocated. If the array allocation was made by `gaconf` (without a separate `gaalloc`), this is the pointer that was the argument to `gaconf`.

`$gacr4` – The pointer to the configuration in memory that was the argument to `gaconf` or `gaconfo`.

`$gacr5` – The row offset that was the argument to `gaconfo`. If the active configuration was loaded by `gaconf`, this value is zero.

The `cfga` instruction can be used to retrieve any one of these values into a processor register.

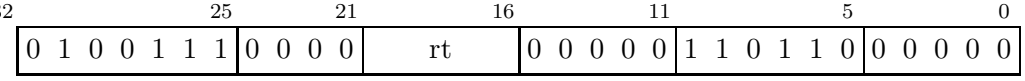
When a context switch occurs while the array is active, it must be possible to suspend the array and save its state so that the computation can be resumed at a later time. The first step toward suspending the array is to execute the `gastop` instruction, which in one step copies the clock counter to a processor register and zeros the counter. The current allocation and configuration can be obtained from the array control registers above, and the logic block registers can be read out using the `mfgav` instructions already described. The state of the array memory queues is saved using the `gasqc` instruction. The remaining internal state of the array, including the status of pending memory reads, can be written to memory using the special `gasave` instruction.

Resuming an array computation requires first that the array allocation be restored by executing `gaalloc` with the previously saved value from `$gacr3`. The active configuration is reloaded by executing `gaconfo` with the values that were saved from `$gacr4` and `$gacr5`. The logic block registers can be restored using simple `mtgav` instructions, while `galqc` reloads the state of the array memory queues. Once the logic block registers are restored,

`garestore` can be used to read back the internal state that had been saved by `gasave`. The final step for resuming the array is to use `gabump` to restore the array clock counter to the value originally returned by `gastop`.

Besides reading back the array's internal state, `garestore` also ensures that combinatorial propagations in the array are given time to complete, following the recent restoration of the logic block register values.

Load array configuration



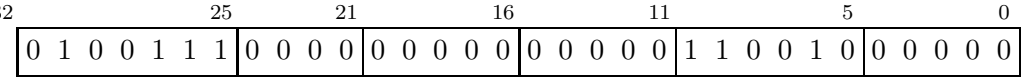
`gaconf` *rt*

Loads a configuration from memory and makes it active. Register *rt* gives the starting address of the configuration in memory.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released. Sufficient space is allocated within the array to hold the specified configuration, and the configuration is loaded and made active. The Z and D registers in the newly allocated space are zeroed. This instruction is equivalent to the sequence of a `gaalloc` instruction followed by `gaconfo`.

The copy of the configuration in memory must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Reset array

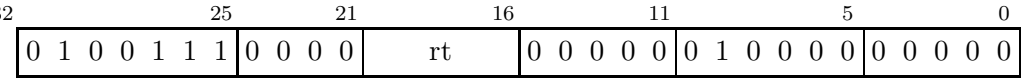


`gareset`

Resets the array, releasing the existing array allocation.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released.

Flush array configuration from cache

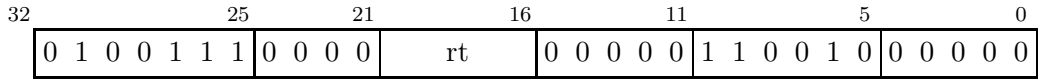


`gacinv` *rt*

Flushes from the configuration cache the configuration or array allocation at the address given by *rt*.

Table A.1: Added instructions, part 1.

Allocate array space



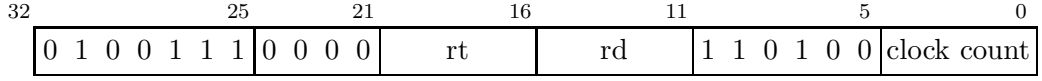
`gaalloc` *rt*

Allocates space within the array for a configuration, without actually loading a configuration. Register *rt* gives the address of a word in memory specifying the number of rows to allocate.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. Any array reads still in progress are then cancelled, and the existing array allocation, if any, is released. The new allocation is put into effect, with all array rows inactive. The *Z* and *D* registers in all of the newly allocated space are zeroed.

The memory word pointed to by register *rt* must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Load array configuration overlay



`gaconfo` *rt*,*rd*,*count*
`gaconfo` *rt*,*rd*

Loads a configuration from memory into the previously allocated array space, while preserving array data state. Register *rt* gives the starting address of the configuration in memory, and register *rd* gives the first allocated row at which to load the overlaying configuration.

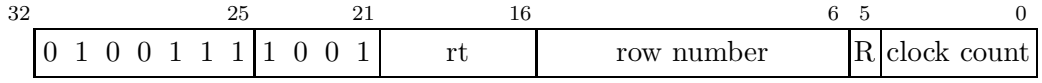
If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. The specified configuration is then loaded and made active, after which the array clock counter is set to the 5-bit unsigned integer constant encoded in the instruction.

The configuration being loaded cannot extend outside the current allocated array space. Although the existing array allocation remains in effect in its entirity, only the rows of the overlaying configuration are made active. The contents of the *Z* and *D* registers within the allocated array space are unaffected by this operation.

The copy of the configuration in memory must not change until a flush configuration instruction (`gacinv`) is executed for this address.

Table A.1, part 2.

Copy word to array



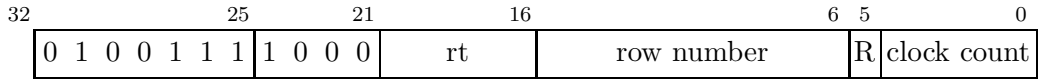
`mtga` *rt*,*reg*,*count*
`mtga` *rt*,*reg*

Copies the value in register *rt* to the middle 16 logic blocks of a fixed array row. The array row number is encoded as an unsigned integer constant in the instruction. If instruction bit R is 0, the concatenation of the sixteen 2-bit Z registers in columns 4 through 19 is the destination of the copy. If R is 1, the concatenation of the sixteen D registers in columns 4 through 19 is the destination. Column 4 receives the least significant 2 bits of the value copied and column 19 receives the most significant 2 bits.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero. The copy is then performed, after which the clock counter is set to the 5-bit unsigned integer constant encoded in the instruction.

For the *reg* argument to `mtga`, the assembler accepts the syntax `$z`n or `$d`n, where n is the array row number expressed as a decimal integer numeral. (For example, `$z19` denotes the Z registers of array row 19.) The *count* argument must be an integer constant. If *count* is not given it defaults to zero.

Copy word from array

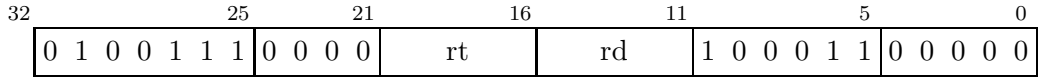


`mfga` *rt*,*reg*,*count*
`mfga` *rt*,*reg*

This instruction is identical to `mtga` except that the direction of the copy is reversed.

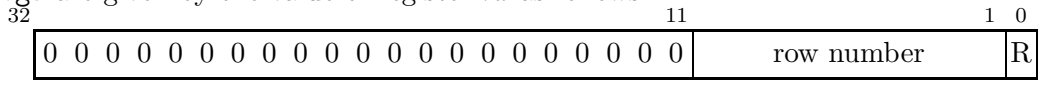
Table A.1, part 3.

Copy word to array, variable row



`mtgav` *rt,rd*

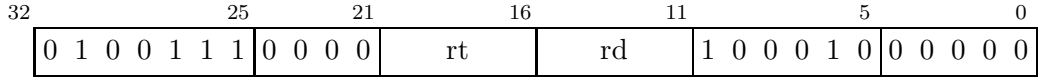
Copies the value in register *rt* to the middle 16 logic blocks of the array row specified by register *rd*. This instruction is similar to `mtga` except that the row number and the R field of `mtga` are given by the value of register *rd* as follows:



As with `mtga`, if the least significant bit of *rd* (bit R) is 0, the destination of the copy is the concatenation of the sixteen *Z* registers in columns 4 through 19 of the specified row; whereas if the least significant bit of *rd* is 1, the destination is the concatenation of the sixteen D registers in columns 4 through 19. Column 4 receives the least significant 2 bits of the value copied and column 19 receives the most significant 2 bits.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the copy is performed.

Copy word from array, variable row

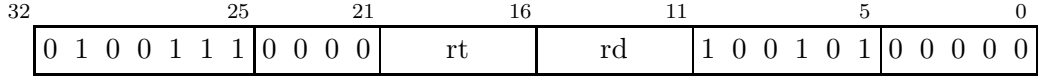


`mfgav` *rt,rd*

This instruction is identical to `mtgav` except that the direction of the copy is reversed.

Table A.1, part 4.

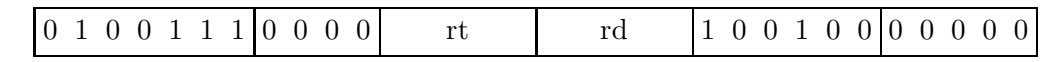
Copy word to array, variable row, low columns



`mtgavy` *rt,rd*

This instruction is identical to `mtgav` except that the destination of the copy is columns 0 through 15 of the specified row. Column 0 receives the least significant 2 bits of the value copied and column 15 receives the most significant 2 bits.

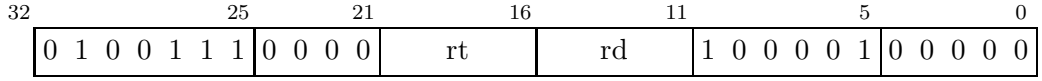
Copy word from array, variable row, low columns



`mfgavy` *rt,rd*

This instruction is identical to `mtgavy` except that the direction of the copy is reversed.

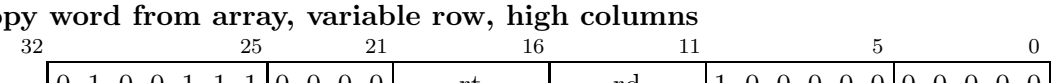
Copy word to array, variable row, high columns



`mtgavz` *rt,rd*

This instruction is identical to `mtgav` except that the destination of the copy is columns 16 through 22 of the specified row. Only the least significant 14 bits of source register *rt* are copied. The most significant 18 bits of *rt* are ignored. Column 16 receives the least significant 2 bits of the value copied and column 22 receives the most significant 2 bits.

Copy word from array, variable row, high columns

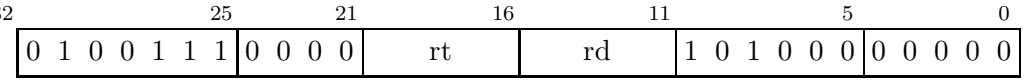


`mfgavz` *rt,rd*

This instruction is identical to `mtgavz` except that the direction of the copy is reversed. The most significant 18 bits of destination register *rt* are zeroed.

Table A.1, part 5.

Load array queue control

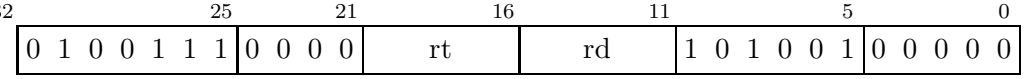


`galqc` *rt,rd*

Loads the control registers for the queue specified by register *rd* with the 20 bytes at the address given by register *rt*. The value of register *rd* must be an integer in the range of 0 to 2 inclusive, indicating one of the three array memory queues. Details about the memory queues and the queue control registers can be found in Section A.3.3 and Figure A.46.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the load is performed.

Store array queue control

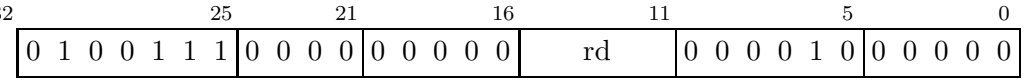


`gasqc` *rt,rd*

Stores the control registers for the queue specified by register *rd* into 20 bytes starting at the address given by register *rt*. The value of register *rd* must be an integer in the range of 0 to 2 inclusive, indicating one of the three array memory queues.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the store is performed.

Increase array clock counter

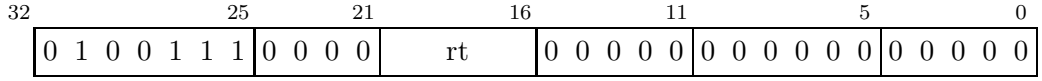


`gabump` *rd*

Adds the value in register *rd* to the array clock counter. The addition is performed modulo 2^{32}. If a carry out of the most significant bit occurs (unsigned overflow), the most significant bit of the clock counter is set.

Table A.1, part 6.

Stop array

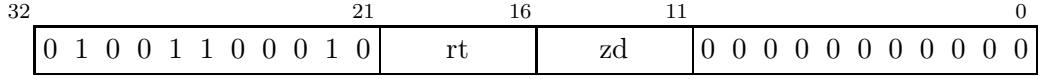


`gastop` *rt*
`gastop`

Zeros the clock counter, halting array execution. Register *rt* gets the value that the counter had before being zeroed.

The assembler allows the destination operand to be omitted, in which case *rt* is set to the MIPS pseudo-register `$0` (zero register).

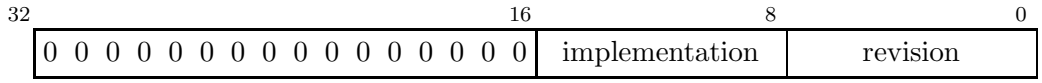
Copy word from array control register



`cfga` *rt*,*zd*

Copies the array control register *zd* to processor register *rt*. The 5-bit *zd* field must be one of the following integers:

0 The version register, which has the format



1 The number of bytes that `gasave` writes to memory. (This is constant for a given implementation/revision.)

3 The pointer that was the argument to `gaalloc` or `gaconf` when the current array allocation was made.

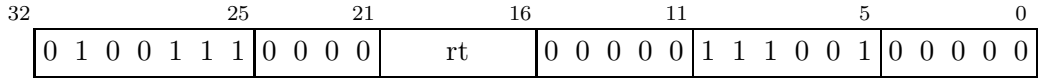
4 The pointer to the configuration in memory that was the argument to `gaconfo` or `gaconf`.

5 The row offset that was the argument to `gaconfo`, or zero if the active configuration was loaded by `gaconf`.

The assembler accepts for *zd* either the notation `$`*n* or `$gacr`*n*.

Table A.1, part 7.

Save internal array state

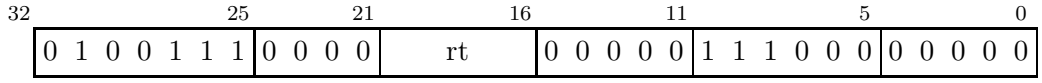


`gasave` *rt*

Saves the internal state of the array to memory at the address given by register *rt*. The internal state includes in particular the status of pending reads from memory. The amount of memory needed to store the saved state can be discovered by reading the `$gacr1` control register using the `cfga` instruction.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the store is performed.

Restore internal array state



`garestore` *rt*

Loads the internal state of the array from memory at the address given by register *rt*. This instruction also stalls long enough to ensure that conbinatorial signals in the array have had time to settle, assuming a maximum path delay of 8 array clock cycles.

If the array clock counter is nonzero, this instruction first waits for the clock counter to fall to zero before the load is performed.

Table A.1, part 8.

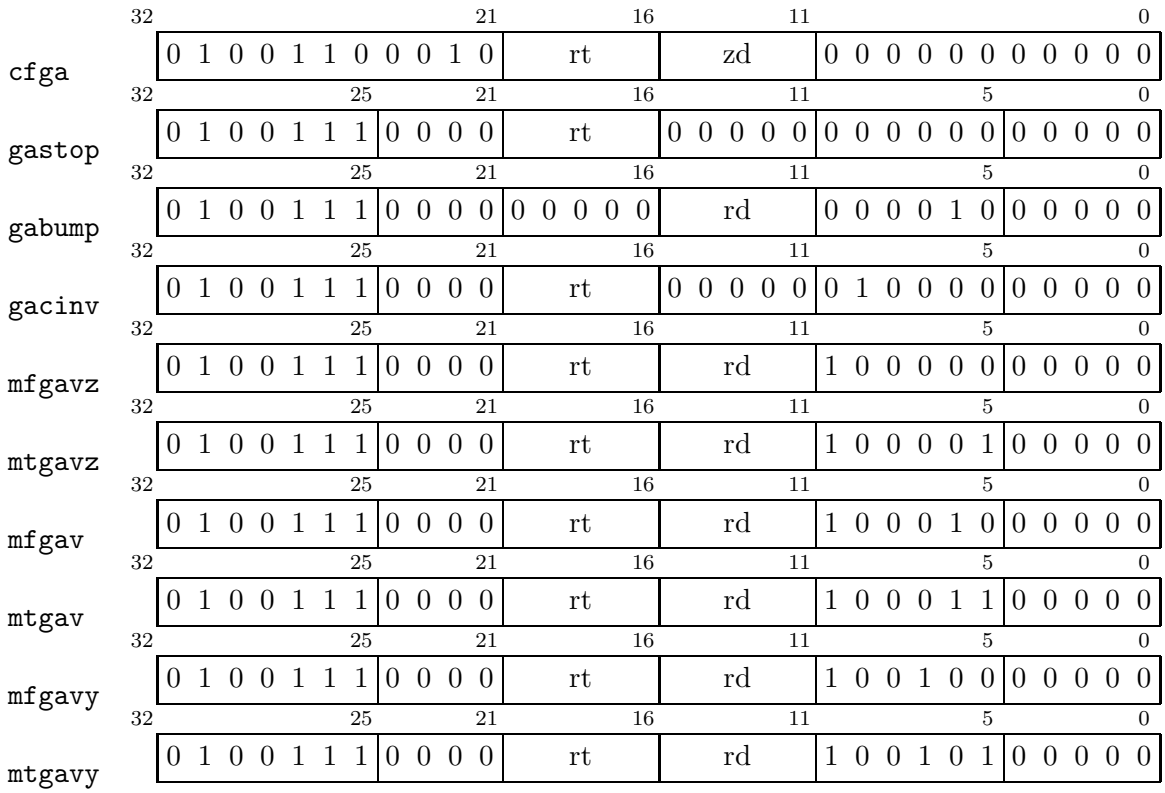


Table A.2: List of added instructions in encoding order.

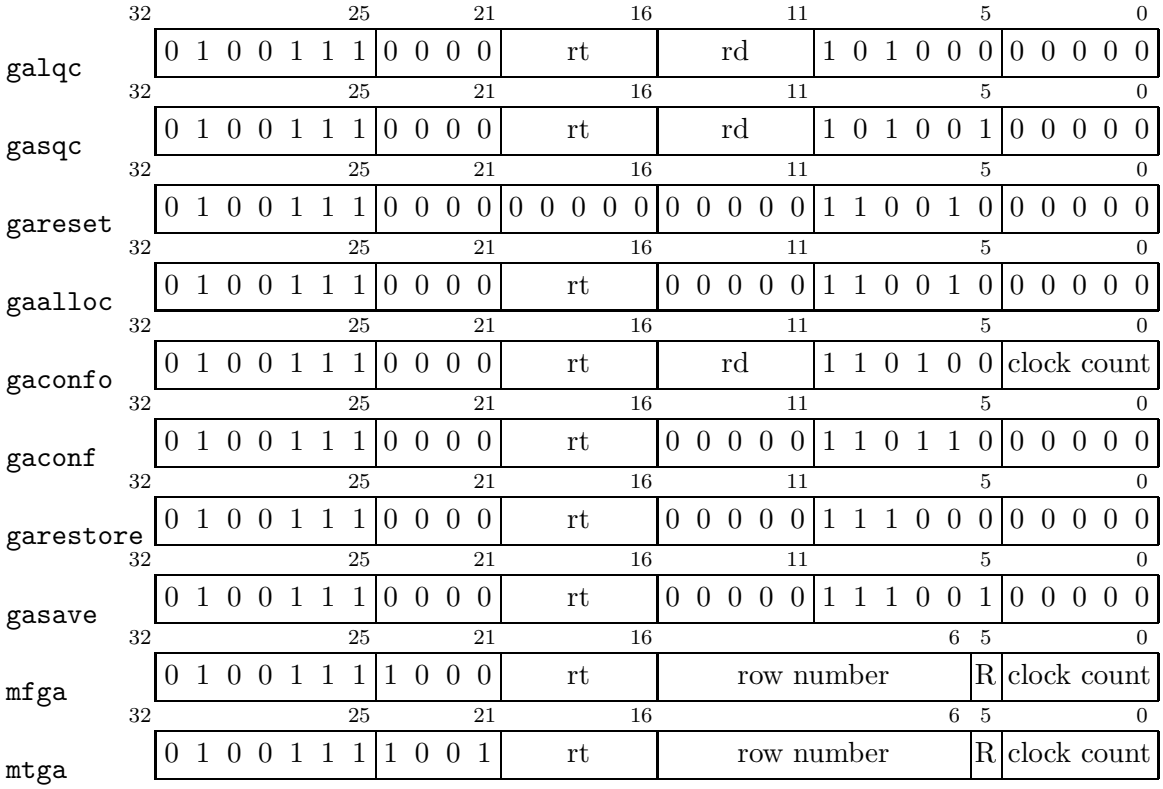


Table A.2, continued.

A.3.2 Array control blocks

The control blocks at the left end of each row help interface between the array on the one hand and the processor and memory on the other. The functions a control block can perform include:

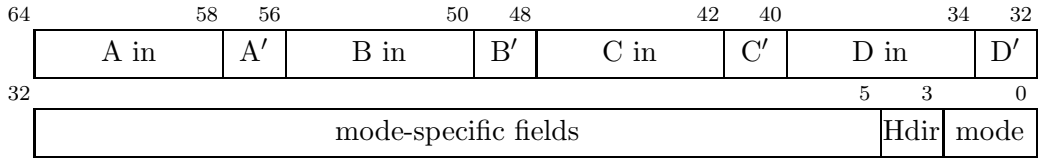
- zero the clock counter (thus halting array execution);
- interrupt the processor;
- initiate a memory access at an arbitrary address;
- initiate a read or write through an array memory queue; and
- load or store the data of a memory access to/from logic block registers.

As always, the active configuration determines which if any of these functions each control block might perform. Figure A.35 shows the general encoding of the configuration for a control block. Like a logic block, a control block's configuration is 64 bits, with four 2-bit inputs, A, B, C, and D, taken from adjacent wires. These inputs are used to control some subset of the functions listed above, according to the *mode* in which the control block is configured.

Regardless of mode, the 8 bits of input are always reduced down to three control signals as illustrated in Figures A.36 and A.37. First, each individual 2-bit input is reduced to a single bit, either by discarding one of the bits or by logically *or*-ing the two bits (Figure A.37). The resulting A' signal is then used to gate each of the corresponding B', C', and D' signals to construct the three control signals. The three control signals are thus generated directly from the B, C, and D inputs, except that A acts as an enable for all three signals.

Any of the four inputs can be fixed to binary constant 00 or 10, the same as for a logic block. Otherwise, a control block input can come from a local horizontal wire (H wire), either from above or below the control block. Control blocks have no outputs, so there are no vertical wires associated with the column of control blocks. Aside from the fewer options, the encoding of control block inputs is identical to that for logic blocks (Figure A.10).

For timing purposes, inputs that are not constant must come directly from a logic block register across the connecting H wire to the control block, as illustrated in Figure A.38.



[63..58] A in	
000000	$A = 00$ (binary)
000001	$A = 10$ (binary)
100010	$A =$ leftmost H wire pair above
⋮	⋮
101010	$A =$ rightmost H wire pair above
110010	$A =$ leftmost H wire pair below
⋮	⋮
111010	$A =$ rightmost H wire pair below

[57..56] A′	
00	$A' = A_0$
10	$A' = A_1 \vee A_0$
11	$A' = A_1$

[4..3] Hdir	
00	H wires driven from right end (shift left)
01	H wires driven from center
10	H wires driven from left end (shift right)

[2..0] mode	
000	no function
010	processor interface
110	memory interface

Figure A.35: Control block configuration encoding.

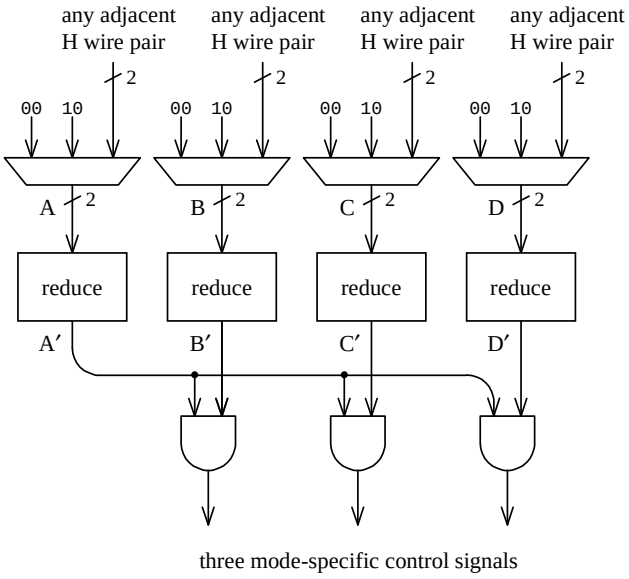


Figure A.36: Control block signals.

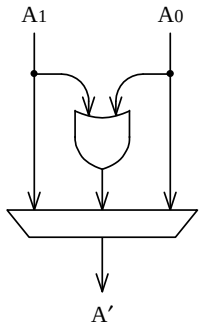


Figure A.37: The reduction functions.

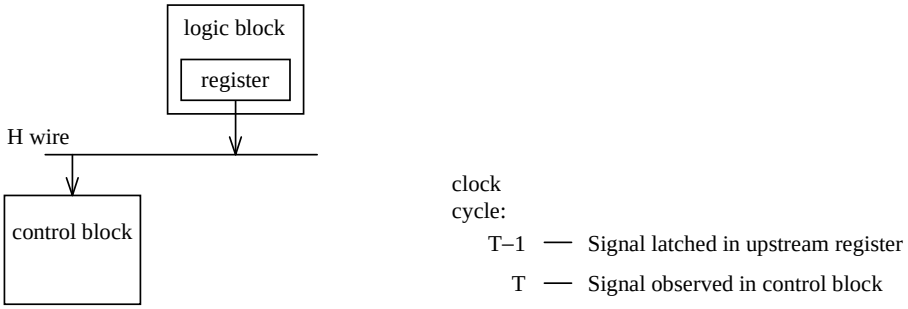
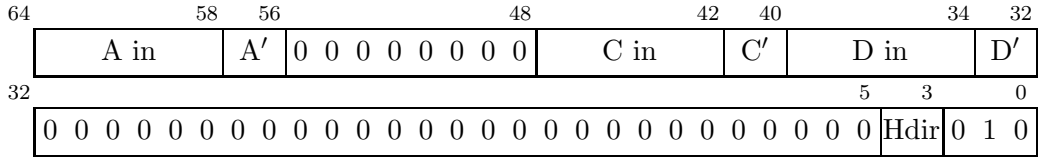


Figure A.38: A control block input must come directly over a local horizontal wire from a logic block register in the same row or the row above.



A' → enable
C' → zero clock counter
D' → interrupt processor

Figure A.39: Configuration encoding for a control block in processor interface mode.

A logic block register that supplies an input to a control block in this way is known as the *upstream register* for that control input.

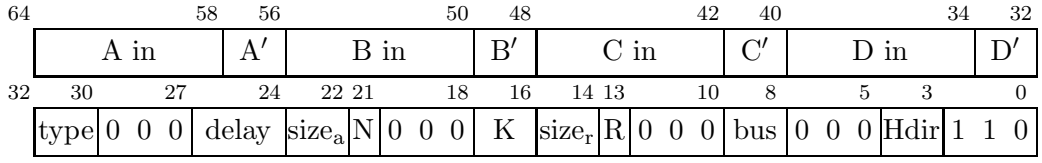
Processor interface blocks

In processor interface mode, a control block can perform two simple actions connected with the main processor. The format of a processor interface configuration is given by Figure A.39. The C' input allows the array to zero the clock counter, and the D' input makes it possible for the array to interrupt the processor. If the A' and C' inputs are both 1, the array clock counter is zeroed at the end of the current array clock cycle, thus halting array execution. If A' and D' are both 1, the main processor is forced to take an interrupt. Note that array execution is not directly affected by any processor interrupts.

Memory interface blocks

Memory accesses can be initiated from the reconfigurable array without direct processor intervention. A memory access proceeds in two steps: the *initiate step* starts the access by providing a memory address, and the *transfer step* transfers the data (Figure A.41). The address is read from the Z registers of a selected row, over a special *address bus* that runs parallel to the four memory buses already mentioned. Up to four contiguous words can be read or written in one memory access, where the word size is selectable as either 8, 16, or 32 bits. Each word is transferred over a separate memory bus.

For memory writes, the initiate and transfer steps must occur together in the same clock cycle. For reads, the initiate step necessarily precedes the transfer step. Only one demand access to memory can be initiated in each array clock cycle, although multiple memory accesses may be in different stages of progress at any one time.



A' → enable
B' → initiate memory access
C' → load/store registers over bus
D' → direction of transfer (load or store)

Figure A.40: Configuration encoding for a control block in memory interface mode. Details about the various fields are covered in Figures A.42, A.43, and A.47.

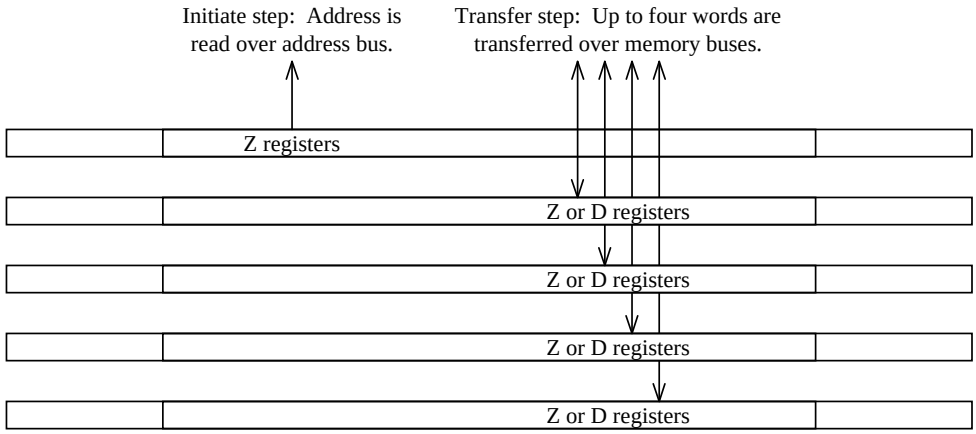
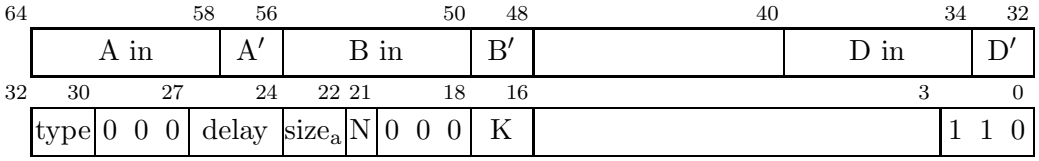


Figure A.41: The two steps of a memory access initiated by the array.

The array sees the same memory hierarchy as the main processor, including all data caches. Misses in the first level data cache may cause array execution to be stalled while the data is fetched from external memory. To reduce cache misses, the array can perform prefetching accesses that merely load the data cache. Array memory accesses may also generate page fault traps as discussed later.

In *memory interface* mode, a control block has the ability either to initiate a memory access or to participate in the transfer of data, or both. Figure A.40 shows the format of the memory interface configuration. Because the initiate and transfer phases are controlled independently, the configuration fields associated with the initiate step will be presented first, separately from those concerned with the transfer step.

Figure A.42 highlights the parts of a memory interface configuration that control the initiation of memory accesses. The actual instigation of a memory access is controlled



A' → enable
B' → initiate memory access
D' → 0 = read, 1 = write or prefetch

[31..30] type	
01	demand access, read/prefetch, cache allocate
10	demand access, read/write, cache allocate
11	demand access, read/write, no cache allocate

[26..24] delay	
000	1 cycle
⋮	⋮
111	8 cycles

[23..22] size_a	
00	8 bits
01	16 bits
10	32 bits

[21] N	
0	aligned address (ignore bottom bits)
1	possibly nonaligned address

[17..16] K	
00	demand access 1 word
01	demand access 2 words
10	demand access 4 words

Figure A.42: Memory interface configuration fields associated with the initiate step of a demand memory access.

by the B' signal, while the direction of access (read versus write) is determined by D'. A demand access to memory is initiated whenever A' and B' are both 1. If D' is 0 at that time, the access will be a read; otherwise it will be either a write or a prefetch, depending on the configuration. When a control block initiates a demand memory access, the contents of the Z registers in the logic blocks in columns 4 through 19 of the same row are sent over the address bus to provide a 32-bit address for the memory system.

Other configuration fields control various apects of the memory access. The size_a and K fields choose the word size and number of words to access, respectively. The largest possible access is to four contiguous 32-bit words, while the smallest is to a single 8-bit "word." The word size and the number of words must each be a power of two within these ranges.

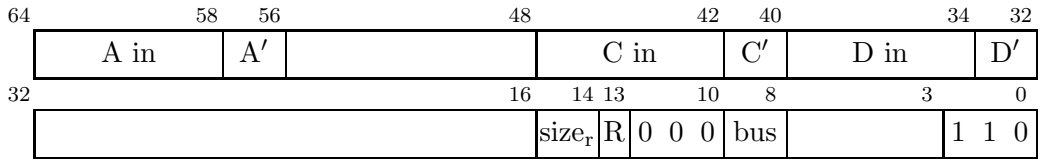
If an access is not a read (that is, if $D' = 1$), it is either a write or a prefetch. The type field of the configuration (Figure A.42) determines whether a non-read memory access is a write or a prefetch, and also whether a cache miss should cause data to be brought into the cache. Since prefetches are performed solely for the purpose of bringing data into the cache, it makes no sense not to do cache allocation on misses in this case. Normal reads and writes may be configured for cache allocation on misses or not.

When the access word size is larger than a byte, the given address may not be aligned on a natural word-size boundary. The configuration chooses one of two possibilities: either the least significant bits of the address are ignored, or a nonaligned memory access is performed at the specified address. The number of bits ignored is dependent on the word size: 1 bit if the word size is 16 bits, and 2 bits if the word size is 32 bits.

Finally, the delay field in the configuration determines the perceived delay for read accesses. This field is ignored for writes and prefetches. The timing details of memory accesses are covered later in this section.

Although any number of control blocks can be configured as memory interfaces, only one control block can initiate a memory access during any array clock cycle.

The transfer step performs the actual movement of data, either simultaneously with the initiate step in the case of writes, or after the data has been read from memory. Figure A.43 shows the memory interface configuration fields associated with the transfer step. The C' input to the control block decides, for each clock cycle, whether a transfer into or out of the row occurs on that cycle. The D' signal indicates the direction of transfer, the same as it does for the initiate step. Other fields of the configuration determine: (1) which



A' → enable
C' → load/store registers over bus
D' → 0 = load, 1 = store

[15..14] size$_r$	
00	8 bits
01	16 bits
10	32 bits

[13] R	
0	load/store Z registers
1	load/store D registers

[9..8] bus	
00	bus 0
01	bus 1
10	bus 2
11	bus 3

Figure A.43: Memory interface configuration fields associated with the transfer step of a memory access.

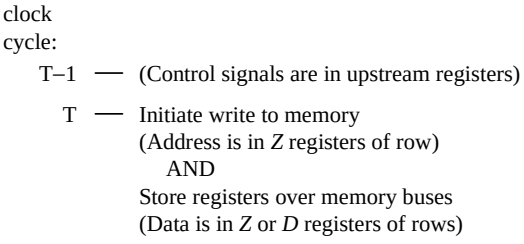


Figure A.44: Timing of a memory write executed by the array. The write "occurs" in the active cycle following the control signals being applied (see text).

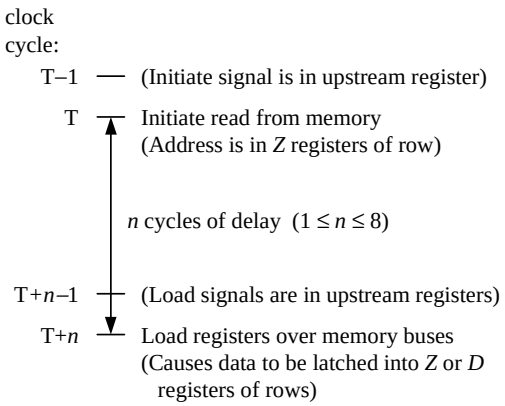


Figure A.45: Timing of a memory read executed by the array.

memory bus the data will be written to or loaded from, (2) which registers will be written out or loaded (Z or D), and (3) the number of bits to write or load (8, 16, or 32). If 32 bits are to be transferred, the Z or D registers in columns 4 through 19 of the row will either be written to or read from the selected bus. If only 16 bits are to be transferred, only columns 4 through 11 are affected, and the registers in columns 12 through 19 are not involved. Likewise, if only 8 bits are to be transferred, only columns 4 through 7 participate in the operation.

Each word of the as many as four words transferred has a memory bus dedicated to it during the transfer. The first word at the given memory address is copied over bus 0. If the access involves more than one word, subsequent words are copied over buses 1, 2, and 3, in that order. For example, a memory access of two words involves buses 0 and 1: bus 0 for the word at the given address in memory, and bus 1 for the next contiguous word in memory.

Following the initiation of a memory read of n words, and after a specific number of array clock cycles have elapsed (discussed below), buses 0 through $n-1$ will contain the values read from memory. At that time, the control blocks on the rows into which these values should be latched must signal the transfer step.

For writes, the initiate and transfer steps are signaled simultaneously. Exactly one word must be driven onto each of the n buses. Figure A.44 shows the timing for a write. Conceptually, the write occurs in the clock cycle following the control signals being applied. If the clock counter is zeroed in the same clock cycle that the write is signaled, the write will not occur until the clock counter is subsequently given a nonzero value, continuing execution of the same configuration. If the current configuration is never resumed, the write will never actually occur, despite having been initiated.

Figure A.45 shows the timing for a read. For reads, the delay field of the memory interface configuration specifies the number of array clock cycles by which the data will be delayed. If this is at least as great as the actual latency of a read operation, execution of the array will not be stalled waiting for the read to return. Otherwise, an implementation must stall the array sufficiently to give the appearence that the data was returned in the specified number of array clock cycles.

A.3.3 Array memory queues

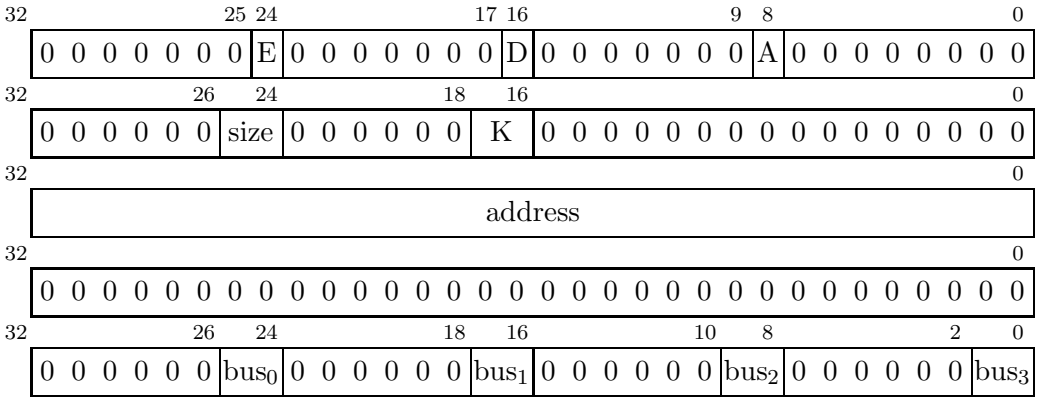
Besides being able to request memory accesses directly, the array has available to it three *memory queues* that can increase the performance of sequential accesses. The array reads or writes to/from a queue much as it does directly to/from memory, except that it does not supply an address or other information about the access. A memory queue is programmed by the main processor with this information in advance, using the `galqc` instruction. Figure A.46 shows the format of the 20 bytes of control information that are loaded from memory into a memory queue's controller by the `galqc` instruction. A corresponding `gasqc` instruction writes back this information to memory in the same format to facilitate context switches.

Like demand memory accesses, a memory queue can be programmed to transfer up to four words on each request. Unlike a demand access, the four words can be matched to buses arbitrarily, so that the first word is not necessarily transferred over bus 0, etc. The last four bytes loaded into the queue controller assign a bus to each word (Figure A.46).

The configuration of a control block that allows it to initiate a queue access is a variation of the one for demand memory accesses, as seen in Figure A.47. As before, an access is initiated whenever A' and B' are both 1. The only additional information encoded in the configuration is the queue number to access. The direction of transfer (read or write) is determined by the queue itself and is not decided by the D input to the control block as it is for demand accesses. The address bus is not used for queue accesses. The transfer step of a queue access is identical to that for a demand access, keeping in mind that the association between buses and words is not fixed but is set by the queue controller.

Ultimately, a queue access has the same affect as a demand memory access at the address maintained within the queue controller (recall Figure A.46). After each access to a memory queue, the stored address is incremented to the next contiguous byte following the last one read or written. A `gasqc` instruction writes out a queue control record with this updated address, so that `galqc` can properly restore the state that the queue had at the time `gasqc` was executed.

Figures A.48 and A.49 show the timing of a queue write and a queue read, respectively. The timing for a queue write is indistinguishable from that of a demand memory write, while a queue read appears the same as a demand read with the delay fixed at 1 clock cycle. Like a demand memory write, a queue write is not committed until one array clock



E	
0	queue disabled
1	queue enabled

D	
0	read
1	write

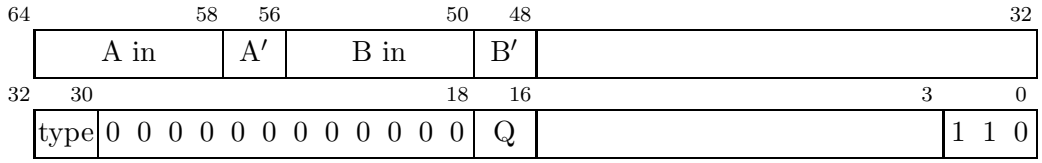
A	
0	no cache allocate
1	cache allocate

size	
00	8 bits
01	16 bits
10	32 bits

K	
00	1 word per access
01	2 words per access
10	4 words per access

bus_n	
00	word n on bus 0
01	word n on bus 1
10	word n on bus 2
11	word n on bus 3

Figure A.46: Format of a queue control record.



A' → enable
B' → initiate queue access

[31..30] type	
00	queue access

[17..16] Q	
00	queue 0
01	queue 1
10	queue 2

Figure A.47: Memory interface configuration fields associated with the initiate step of a memory queue access. (Compare with Figure A.42.)

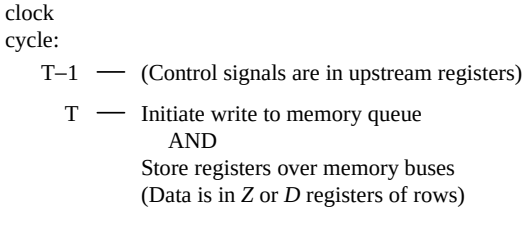


Figure A.48: Timing of a write to an array memory queue. The write "occurs" in the active cycle following the control signals being applied (see text).

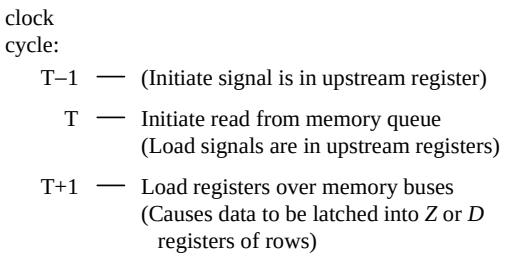


Figure A.49: Timing of a read from an array memory queue. This is equivalent to a demand memory read with a fixed delay of 1 clock cycle.

cycle following the initiation of the write by the control block. If the clock counter becomes zero in the same cycle that the queue write is initiated and if array execution is never resumed, the write will not occur.

Each of the three queues can be accessed on every array clock cycle, simultaneously with the initiation of a new demand memory access every cycle. This makes it possible to achieve four independent memory accesses per clock cycle to/from the array. However, each memory bus can transfer only one word during any given cycle, so in total a maximum of four 32-bit words can be accessed each cycle.

Appendix B

Garp Application Notes

This appendix contains some notes for how to program common operations using Garp's features.

B.1 Single-bit operations

Because of the Garp array's 2-bit granularity, values are passed between logic blocks as 2-bit pairs and are mostly operated on by logic block functions as 2-bit values. Simple *true*-or-*false* Boolean values are most conveniently encoded in the array as binary 00 or 11. Since the table mode logic block function operates on both bit positions identically, arbitrary four-input Boolean functions can be programmed with the output emerging in the same form as either 00 or 11. The crossbar permutation boxes permit inputs to be selected from individual source bits by duplicating either the high or low bit of an input into both bit positions before the input is passed to the lookup table.

B.2 Shifts

Unsigned shifts by a constant distance are easily accomplished using the horizontal wires between two rows. The upper row drives the value to be shifted onto the wires, and the lower row reads the shifted bits. Since values between blocks are transmitted as 2-bit pairs, this always suffices for unsigned shifts a distance of an even number of bits. The shifted value is immediately available for use as an operand within the lower row.

When a value must be shifted an odd number of bits over, the lower row is usually

responsible for performing the final shift by one bit. The shift/invert permutation boxes are partly designed for this job, but they are only accessible when the logic blocks in the second row are in certain modes (the select modes and triple-add mode). For other modes (the table modes and the carry chain mode), the crossbar boxes must be used to pick out the correct bits, with the logic block functions encoded independently for the high and low bit positions.

Signed left shifts are indistinguishable from unsigned left shifts. A signed right shift, on the other hand, must replicate the original sign bit into some number of bit positions. This cannot be done using the shift/invert permutation boxes unless there is a source somewhere that has the sign bit duplicated in both the high and low bits (like the Boolean values in the previous section). More often, signed right shifts depend on using the crossbar boxes to get everything into the correct places.

In most cases, the same value can be shifted different distances for more than one input to a row. For example, an addition operation can take the value a from the row above shifted left by two bits for one input, and shifted left by four bits for the other input, to compute the value $4a + 16a = 20a$. This technique for multiplication is covered in a later section.

Variable shifts are done in the Garp array as a series of multiplexors choosing among constant shifts. A complete variable left shift of a 32-bit value would first optionally shift by 16 bits in one row, then shift by either 0, 4, 8, or 12 bits in a second row, and finally shift by either 0, 1, 2, or 3 bits in a third row. Each of the three stages corresponds to up to two bits of the shift amount, and can be implemented in a single row using the logic block select mode and the horizontal wires. (The select control input must be inverted by the shift/invert box so that a value of binary 00 corresponds to accepting the value from the row above without any shift.)

B.3 Using the carry chain

The Garp array's carry chains can be used for any function that is expressible in terms of the carry chains' *propagate* and *generate* control signals. A simple example is a test for whether two integers a and b are equal. As the carry chain progresses from right to left, a 0 carry chain value can be defined to mean that the two integers are equal for all the least-significant bits up to that position. If any pair of corresponding bits in a and

a	b	$k_{\text{out}} = \text{result}$	propagate	generate
0	0	k_{in}	1	–
0	1	1	0	1
1	0	1	0	1
1	1	k_{in}	1	–

Table B.1: Configuration of the carry chain for the comparison $a \neq b$. The value of *result* is the same as k_{out}.

a	b	$k_{\text{out}} = \text{result}$	propagate	generate
0	0	k_{in}	1	–
0	1	1	0	1
1	0	0	0	0
1	1	k_{in}	1	–

Table B.2: Configuration of the carry chain for the comparison $a < b$.

b are not the same, the carry out from that position should be forced to 1 to indicate the two values are unequal. However, if two corresponding bits of a and b are the same, the carry out should duplicate the carry in, because whether the two integers are equal up to that point depends entirely on whether they were equal up to the previous bit position. If and only if *all* the corresponding bits in a and b are the same will the original 0 carry into the least-significant bit position propagate through to the other side unchanged. Table B.1 shows how the *propagate* and *generate* tables are programmed to get the desired effect with the logic blocks in carry chain mode. The final result is exactly the carry out from the most-significant bit position, and will be 0 (or *false*) if $a = b$ and 1 (or *true*) if $a \neq b$.

A small modification to the configuration, as illustrated in Table B.2, changes the comparison from $a \neq b$ to $a < b$. In this case, if two corresponding bits of a and b are not the same, the carry out is forced to either a 1 (*true*) or 0 (*false*) depending on which of a or b has the greater bit value. As before, if the two corresponding bits of a and b are the same, then whether $a < b$ up to that point depends on whether it was true for the bits to the right of that position. The final result of the comparison is again exactly the carry out from the most-significant bit position.

The *propagate* and *generate* settings for calculating an addition $a + b$ are given in Table B.3. The *result* output is not the same as k_{out} but can be represented instead as *propagate* $\oplus\, k_{\text{in}}$ (which appears in the Garp documentation as the result function $U \oplus K$).

Subtraction is implemented as a variant of addition, making use of the rule that

a	b	k_{out}	result	propagate	generate
0	0	0	k_{in}	0	0
0	1	k_{in}	$\neg k_{\text{in}}$	1	–
1	0	k_{in}	$\neg k_{\text{in}}$	1	–
1	1	1	k_{in}	0	1

Table B.3: Configuration of the carry chain for the addition $a + b$. The value of *result* is *propagate* $\oplus\, k_{\text{in}}$.

the bitwise logical complement of an integer a is the same as $-1 - a$. From this simple fact we can derive that

$$a - b = -1 - ((-1 - a) + b) = \neg(\neg a + b),$$

where the notation $\neg a$ denotes the bitwise logical complement of a. Thus the subtraction $a-b$ can be accomplished by complementing the input a, adding b, and lastly complementing the result. The complement of an input can be effected in the configuration of the *propagate* and *generate* lookup tables, while the result is complemented simply by choosing the result function $\neg(U \oplus K)$ instead of $U \oplus K$.

B.4 Adding or subtracting three terms

The triple-add logic block mode supports the direct addition of three terms, $a + b + c$, by first reducing the three inputs to two with a carry-save adder and then feeding the two results to the carry chain for summing. Up to two of the three terms can be negated using tricks similar to that just shown for subtraction. The simplest case is the one with two negated terms, because

$$a - b - c = -1 - ((-1 - a) + b + c) = \neg(\neg a + b + c).$$

This is clearly the same scheme used to get $a-b$ above. In triple-add mode, the complement of input a is achieved by programming the shift/invert permutation box to complement (invert) that input.

The case with just one term negated is more subtle, and requires the following theorem:

Theorem: *If* $m + n = a + b + \neg c$ *then* $m - \neg n = a + b - c$.

Proof: $m-\neg n = m-(-1-n) = m+n+1 = a+b+\neg c+1 = a+b-1-c+1 = a+b-c$.

This theorem gives us a way to compute $a+b-c$: First feed a, b, and $\neg c$ into the carry-save adder, so that the resulting *sum* and *carry* terms have the same sum as $a+b+\neg c$. Then instead of adding *sum* and *carry* as usual, program the rest of the logic block to calculate $sum - \neg carry$ (in exactly the same manner as done for $a-b$ in the previous section) to obtain the result $a+b-c$.

B.5 Multiplication

In practice, most multiplications are by constants, which is fortunate because multiplying by a constant is usually more efficient than a general multiplication of two variables. If the variable factor is small (only a few bits) and the constant factor large, the problem can be turned into a lookup table, with the variable factor supplying the index into the table. Otherwise, multiplying by a constant is best done as a tree of three-input adders as discussed in Section 3.2.4. To calculate $100 \times a$, for example, it suffices to sum $2^7a - 2^5a + 2^2a = (128 - 32 + 4) \times a = 100 \times a$. If the value a is being driven onto the horizontal wires below some row, the row immediately below can perform all the necessary shifts and adds by appropriate configurations of triple-add mode and selecting the proper bits off the horizontal wires above. Multiplications by larger constants just require more adders using more rows; all but the last row duplicates the variable factor onto horizontal wires via the logic block "D paths," so that each row can obtain the necessary shifted partial products to add.

Figure B.1 breaks up all the odd integers up to 85 into the fewest power-of-2 terms that sum to that integer. For the even integers, observe that an odd integer c has the same number of terms as every even integer of the form 2^kc for any k. Thus, for instance, $100 = 2^2 \times 25 = 2^2 \times (2^5 - 2^3 + 2^0) = 2^7 - 2^5 + 2^2$ as already noted.

The pattern visible in Figure B.1 makes it easy to determine the smallest multiplier constant that requires more terms than can be accomodated by some number of array rows:

array rows	number of addends that can be summed	smallest multiplier that does not fit
1	3	43
2	5	683
3	7	10923
4	9	174763

Stated the other way around, it is known that a multiplication by any constant less than

			$43 = 2^6 - 2^4 - 2^2 - 2^0$
			$45 = 2^6 - 2^4 - 2^2 + 2^0$
$1 = 2^0$			$47 = 2^6 - 2^4 - 2^0$
			$49 = 2^6 - 2^4 + 2^0$
$3 = 2^2 - 2^0$			$51 = 2^6 - 2^4 + 2^2 - 2^0$
$5 = 2^2 + 2^0$			$53 = 2^6 - 2^4 + 2^2 + 2^0$
		$23 = 2^5 - 2^3 - 2^0$	$55 = 2^6 - 2^3 - 2^0$
		$25 = 2^5 - 2^3 + 2^0$	$57 = 2^6 - 2^3 + 2^0$
	$11 = 2^4 - 2^2 - 2^0$	$27 = 2^5 - 2^2 - 2^0$	$59 = 2^6 - 2^2 - 2^0$
	$13 = 2^4 - 2^2 + 2^0$	$29 = 2^5 - 2^2 + 2^0$	$61 = 2^6 - 2^2 + 2^0$
$7 = 2^3 - 2^0$	$15 = 2^4 - 2^0$	$31 = 2^5 - 2^0$	$63 = 2^6 - 2^0$
$9 = 2^3 + 2^0$	$17 = 2^4 + 2^0$	$33 = 2^5 + 2^0$	$65 = 2^6 + 2^0$
	$19 = 2^4 + 2^2 - 2^0$	$35 = 2^5 + 2^2 - 2^0$	$67 = 2^6 + 2^2 - 2^0$
	$21 = 2^4 + 2^2 + 2^0$	$37 = 2^5 + 2^2 + 2^0$	$69 = 2^6 + 2^2 + 2^0$
		$39 = 2^5 + 2^3 - 2^0$	$71 = 2^6 + 2^3 - 2^0$
		$41 = 2^5 + 2^3 + 2^0$	$73 = 2^6 + 2^3 + 2^0$
			$75 = 2^6 + 2^4 - 2^2 - 2^0$
			$77 = 2^6 + 2^4 - 2^2 + 2^0$
			$79 = 2^6 + 2^4 - 2^0$
			$81 = 2^6 + 2^4 + 2^0$
			$83 = 2^6 + 2^4 + 2^2 - 2^0$
			$85 = 2^6 + 2^4 + 2^2 + 2^0$

etc.

Figure B.1: The fewest power-of-2 terms that sum to each odd integer up to 85. A pattern can be constructed centered on the exact powers of 2 (observe above: 8, 16, 32, 64).

683, for example, can be done in no more than two array rows. Moreover, as Figure B.1 shows, even though it is not possible to multiply by 43 in three terms, there are still many values larger than 43 that can be multiplied in just three terms. Multiplication by 1000, for instance, requires only one row because $1000 = 2^3 \times 125 = 2^3 \times (2^7 - 2^2 + 2^0)$.

When neither factor is a constant, multiplication takes a little more work. Figure B.2 shows the physical organization in the Garp array of a multiplier taking two unsigned 16-bit variables and generating a 32-bit product. This example multiplies by four multiplier bits each clock cycle, taking four iterations (and thus four clock cycles) to complete an entire 16-bit multiplication. An additional three cycles of latency pushes the total time for one multiplication to seven clock cycles.

The *multiplier shifter* shifts the multiplier by four bits each cycle. Each of the two *partial product selectors* determines the product of the multiplicand and two bits from the multiplier, the result of which is either 0, the multiplicand, $2 \times$ the multiplicand (trivial), or

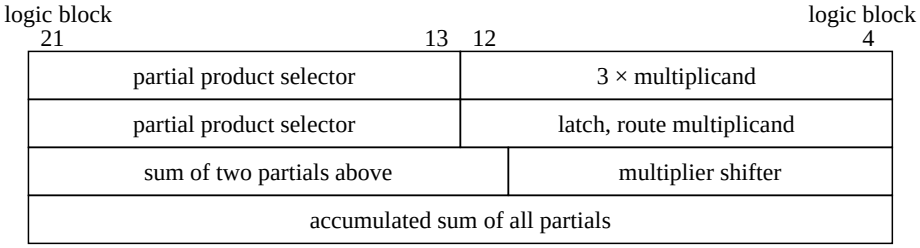


Figure B.2: Layout in the Garp array of a multiplier taking two unsigned 16-bit variables and calculating a 32-bit product.

$3 \times$ the multiplicand. A separate set of logic blocks calculates $3 \times$ the multiplicand for use by the partial product selectors. (This value does not change until the next pair of factors.) A *partial product selector* is primarily a multiplexor of four inputs, configured using the partial select mode provided for this purpose. The rest of the pieces merely add up the partial products. Other variable multipliers can be constructed along the same principles.