Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulation

by

Jonathan William Babb

B.S. Electrical Engineering
Georgia Institute of Technology, 1991

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

Signature of Author ....
Department of Electrical Engineering and Computer Science
November 15, 1993

Certified by ....
Anant Agarwal
Associate Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by ....
Fred Morgenthaler
Chairman, Departmental Committee on Graduate Students

Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulation

by

Jonathan William Babb

Submitted to the Department of Electrical Engineering and Computer Science
on November 15, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Existing FPGA-based logic emulators are limited by inter-chip communication bandwidth, resulting in low gate utilization (10 to 20 percent of usable gates). This resource imbalance increases the number of chips needed to emulate a particular logic design and thereby decreases emulation speed, since signals must cross more chip boundaries. Current emulators only use a fraction of *potential* communication bandwidth because they dedicate each FPGA pin (physical wire) to a single emulated signal (logical wire). These logical wires are not active simultaneously and are only switched at emulation clocking speeds.

Virtual Wires overcome pin limitations by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA. A virtual wire connects a logical output of one FPGA to a logical input on another FPGA. Virtual Wires relax the absolute limits imposed on gate utilization. The resulting increase in bandwidth reduces the need for global interconnect, allowing effective use of low dimension inter-chip connections.

This thesis presents and analyzes the concept of Virtual Wires, and describes a Virtual Wires compiler which utilizes static routing and relies on minimal hardware support. Results from compiling netlists for the 18K gate Sparcle microprocessor and the 86K gate Alewife Communications and Memory Management Unit indicate that Virtual Wires can increase FPGA gate utilization beyond 80 percent without a significant slowdown in emulation speed.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Electrical Engineering

Acknowledgments

The intellectual contribution of both Russ Tessier and Anant Agarwal have been essential to the Virtual Wires project. Greatly magnifying our initial hacking efforts has been Matt Dahl, essential in the implementation of the Virtual Wires compiler system – most critically in the back-end interfaces. His work in conjunction with Russ's expertise in the detailed design and implementation of an initial wire-wrapped prototype board, and more recently of a full scale pc board, will give us the joy of seeing these raw concepts turned into a real working system. Alongside Matt and Russ, Silvina Hanono has implemented the initial debugging interfaces which will greatly enhance the usability of this system.

I also acknowledge the contributions of InCA, ltd. By allowing our research group to use their partitioning programs at maintenance costs, they have both saved us the effort of writing a partitioner and added credibility to our initial measurements of pin limitations.

I have most of the Alewife team to thank for their initial scepticism of our ideas - forcing us to wring them into something actually doable. Initial reactions such as "Why don't you go work for Quickturn..." eventually turned into stimulating discussions on issues of hardware design (Donald Yeung and Ken Mackenzie), large scale design verification (John Kubiatowitz), and FPGA Computing (David Chaiken and Kirk Johnson). John has also been helpful in supplying the two Alewife benchmark designs.

My officemate Gino Maa has been essential in infinitely long discussions on a range of issues that needed to be explored (but no thanks for his similarly instigated discussions). Comments and suggestions from those who reviewed the initial versions of this thesis are greatly appreciated. Thanks to mom and dad for their never failing love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Design Verification

Designers of complex digital logic systems are constantly faced with the problem of design verification. They must perform extensive tests to verify the correctness of their design before the fabrication, manufacture, and end use of any hardware systems. For many such systems this verification effort requires more time and money than that invested in the initial design. In particular, the high non-recurring expense (NRE) and long fabrication cycle time of VLSI chips makes such testing essential. For these chips, the need for thorough testing heavily impacts the design methodology itself.

Continuing growth in the scale of circuit integration and the resulting increase in the number of components in current designs [24] are compounding verification costs. To make matters worse, the computational power required to verify systems increases at a greater than linear rate with system size and circuit complexity. Thus even if computation speeds increase at the same rate as typical design size, the need for effective design verification will continue to increase.

Verification techniques fall into two categories: formal methods and empirical methods. Formal verification methods attempt to prove that an implementation satisfies a given specification. By using algorithmic shortcuts such as binary decision diagrams [32] and symbolic simulation [21], formal methods are able to reduce the computational requirements of system testing. Formal methods have also succeeded in the areas of automatic test pattern generation [27], synthesis for testability [18], and static timing analysis. However, the use of formal verification is quite limited if high-level specifications are not necessarily consistent and correct.

For the majority of their verification needs, system designers must then rely on empirical methods. These methods consist of building and testing software and hardware *prototypes* of the final system. They must test individual components and test how these components will interact in the complete system. In verifying functional logic, these prototypes can take the form of logic simulations, hardware models, and logic emulations.

Figure 1-1 shows the tradeoff between compilation speed and execution speed for these design prototypes in relation to the final system. The y-axis measures relative speeds for compiling or constructing a hypothetical design. The x-axis measures relative speeds for executing one set of test vectors on this design. For example, consider a final system which takes months to construct and runs a set of vectors in less than one minute. The same design and vector set could be compiled for a logic simulator on the order of minutes, but would take years to execute.

The following sections discuss these prototypes in more detail.

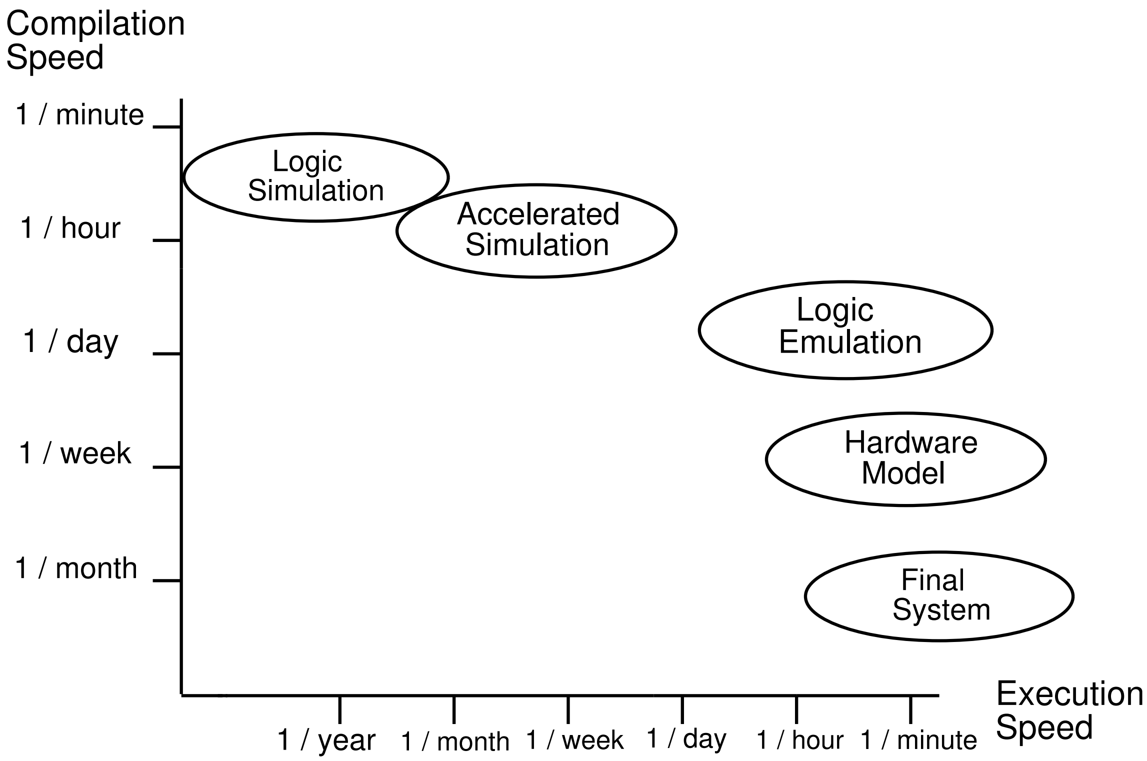

Figure 1-1: Verification Speed Tradeoffs

Logic Simulation

Most logic simulators are software only and are extremely time consuming. As a consequence, only partial testing can be done on all but the smallest of designs. For larger designs, testers must carefully orchestrate their limited set of input test vectors to efficiently exercise the design's major functions. On the other hand, no special hardware is required for software only simulations. Such simulations can run on a standard workstation or personal computer. Of the three classifications of prototypes, simulations are the most flexible in terms of making incremental design changes, and in allowing detailed timing analysis and debugging.

To improve simulation time, engineers have used both software and hardware techniques. Software based accelerations include compiled-code simulation, parallel simulation, and high level simulation. In compiled-code simulation [8], digital logic that is to be simulated is compiled directly into machine instructions rather than being interpreted. Parallel simulation techniques [40] have been reported to achieve speed-ups for digital logic simulation as well. In conjunction with these methods, researchers are speeding up simulation by directly simulating high-level description languages (HDLs) such as VHDL and Verilog [15].

Hardware acceleration ranges from co-processors for software simulators, to special purpose logic processors [10]. Co-processors allow a standard computer to be accelerated for logic simulation much like a floating point unit speeds up numerical calculations. However, in co-processors, the I/O transfer between the coupled host and hardware accelerator can easily become a bottleneck. In comparison, special purpose logic processors are computers in their own rights, often resembling a mainframe or supercomputer rather than a simple workstation (in both complexity and price). They permit even faster logic simulation, but are complicated to use and do not easily track today's fast moving technology curve. Section 2.1 discusses two such simulation engines in greater detail.

Hardware Models

At the other end of the prototyping spectrum, in terms of flexibility and cost versus speed, is hardware modeling. A hardware model, or hardware prototype, is usually designed to function as close as possible to the actual system being verified. For example, a wire-wrapped protoboard may be used as a hardware model of a small VLSI chip under design. Models can often be directly integrated into a target system. In contrast to simulation, these models can usually run at near real time speeds; however, costs, in both money and time to build, are often prohibitively high. Furthermore, there is usually very little flexibility in making more than small design changes. There is also the added difficulty of verifying the custom model itself! In the case of a chip, this difficulty may result in modeling costs that are higher than the original design costs of the final system [43].

Recent improvements in hardware models include the use of programmable logic and programmable PC boards [4]. The added complexity of reconfigurability can be offset by the ability to make design changes more rapidly. As software improves for such models, the distinction between hardware models, and the next category of prototypes, logic emulation, is becoming blurred.

Logic Emulation

Logic emulation offers a compromise between the flexiblity of software simulation and the speed of a hardware model. Logic emulation is distinguished from logic simulation in that the circuits being simulated are compiled directly into hardware. However, unlike the hardware model, a logic emulator can be quickly re-programmed, *via software only*, to emulate a new circuit. Besides the cost saving of re-use, the ability to quickly reconfigure minimizes time between design iterations. Like the hardware model, a logic emulator can be integrated directly into a target system.

In comparison with simulation, emulation compile time overheads are much higher, and analysis detail is lower. However, with the correct interfaces, a logic emulator can be connected directly into a simulation environment, allowing a simulation and an emulation to complement one another.

Before continuing the discussion of logic emulation in Section 1.3, we examine one of the most promising devices for implementing logic emulation: the Field Programmable Gate Array.

1.2 FPGAs and FPGA Computing

Since the birth of programmable logic devices in 1974, and the Field Programmable Gate Array (FPGA) around 1986 [11], programmable logic has traditionally been used to replace fixed discrete logic and gate array logic. In particular, the numerous advantages of FPGAs - reduced manufacturing time, the ability to accommodate design changes easier, and uniform part replacement, have made the FPGA a very attractive building block for logic design.

As the use of FPGAs has become more refined, another powerful property has become apparent - the ability for a system containing an FPGA to reconfigure that device to perform different functions at different times. In particular, static random access memory (SRAM) based FPGAs, such as the Xilinx 4000 Series [47], can be reconfigured. Figure 1-2 shows the high level architecture of an SRAM-based FPGA. The SRAM, shown to the side in a separate box for clarity, is actually interspersed throughout the programmable logic, interconnect, and I/O. By downloading memory configurations into the SRAM, the functionality of the FPGA can be completely defined, subject to the constraints of logic capacity, interconnect capacity, and I/O capacity.

If we take advantage of reconfigurability, we can build FPGA-based computing

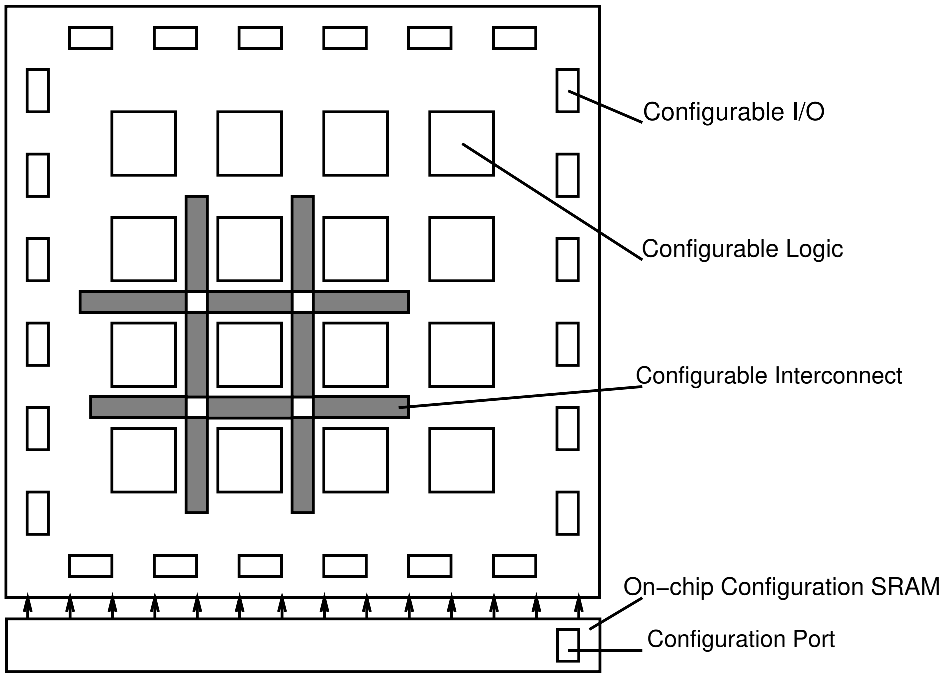

Figure 1-2: Architecture of an SRAM-based FPGA

machines. Users can specialize such a machine for the needs of their particular application by re-programming the FPGAs. In these machines, the binding of functions occurs at an even more primitive level than a reduced instruction set computer - at the circuit level. While this late binding can allow an application to run at hardware speeds, with the inherent parallelism of circuits, there is a price to be paid. The price is in terms of longer compilation time (we must synthesize circuits!), and a non-trivial configuration time.

Because of slow configuration time, re-use of FPGA logic, called *virtual logic*, is computationally expensive. More efficient virtual logic has been introduced to the FPGA world, in the Plessey ERA [23], with the *hardware subroutine* - "a circuit or part circuit optimized for the particular situation encountered, performing a specific task at a specific time, and then being replaced by another and another as the application demands." However, currently available high density FPGAs are not targeted at computing, and thus their manufactures have not optimized configuration time. Configuration time can be more than five orders of magnitude slower than circuit execution speeds. Without virtual logic, current FPGAs are too small to be useful singly for most computing applications. Furthermore, when multiple FPGAs are used in an array, an application's *computational working set*, defined as the set of active circuits within a given configuration period, must completely fit within the bounds of the array.

1.3 FPGA-based Logic Emulation

Logic emulation is a particularly well suited application for FPGA computing. We can use existing high level design language (HDL) compilers, or directly execute gate-level netlists. Although typical application size is larger than one FPGA, an array of FPGAs can emulate a circuit if we partition the circuit. With a large enough array, the computational working set of the algorithm is small enough to avoid the configuration overhead of using virtual logic. And finally, the application can be compiled with purely static communication patterns.

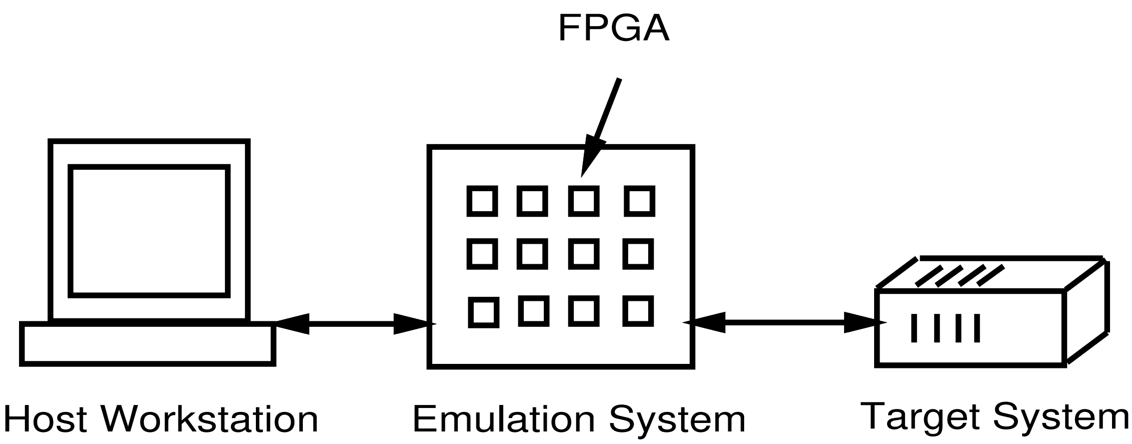

Figure 1-3: Typical Logic Emulation System

FPGA-based logic emulators are capable of emulating complex logic designs at clock speeds six orders of magnitude faster than a software simulator [44]. This performance is achieved by partitioning a logic design, described by a *netlist*, across an interconnected array of FPGAs (Figure 1-3). This array is connected to a host workstation which is capable of downloading design configurations, and is directly wired into the target system for the logic design. The netlist *partition* on each FPGA (termed FPGA partition throughout this thesis), configured directly into logic circuitry, can then be executed at hardware speeds.

Once configured, an FPGA-based emulator is a heterogeneous network of special purpose processors. Each FPGA processor is specifically designed to cooperatively execute its embedded circuit partition. As parallel processors, logic emulators are characterized by their interconnection topology (network), target FPGA (processor), and supporting software (compiler). The interconnection topology describes the arrangement of FPGA devices and routing resources (e.g. full crossbar, two dimension mesh, etc.). Important target FPGA parameters include gate count (computational resources), pin count (communication resources), and mapping efficiency. Supporting software is extensive, combining netlist translators, logic optimizers, technology mappers, global and FPGA-specific partitioners, placers, and routers.

1.4 Contributions of Thesis

In existing FPGA-based logic emulation architectures, both the logic configuration and the network connectivity remain fixed for the duration of the emulation. Thus the computational working set consists of the entire netlist to be emulated. Furthermore, each inter-FPGA partition signal consumes dedicated FPGA I/O resources – FPGA pins. For mapping typical circuits onto available FPGA devices, FPGA partitions are predominately pin limited; all available gates can not be utilized due to lack of pin resources to support them. Low utilization of gate resources increases both the number of FPGAs needed for emulation and the time required to emulate a particular design. Pin limits set a hard upper bound on the maximum usable gate count any FPGA gate size can provide. In other words, for a pin limited design, if we increase the number of available FPGA gates without increasing the number of available pins, we cannot use the additional gates. Trends indicate that this discrepancy will only get worse as technology scales - available gate counts are increasing faster than available pin counts.

To overcome pin limitations in FPGA-based logic emulators [5], [1] this thesis proposes the use of a compilation technique based on *Virtual Wires* [6]. This method can be applied to any topology and FPGA device, although some benefit substantially more than others. A virtual wire represents a connection between a logical output on one FPGA and a logical input on another FPGA. Established via a pipelined, statically routed [30] communication network, these virtual wires increase available off-chip communication bandwidth by multiplexing the use of FPGA pin resources (physical wires) among multiple emulation signals (logical wires).

We show that Virtual Wires effectively relax pin limitations. While low pin counts may decrease emulation speed, there is no longer a hard pin constraint which must be enforced. Since pin limits no longer restrict the amount of logic per FPGA, fewer FPGAs will be needed to emulate a given design. If this reduction in system size is large, the use of Virtual Wires can potentially increase emulation speed. We demonstrate that the gate overhead of using Virtual Wires is low, comprising gates which could not have been utilized anyway in the purely hardwired implementation. Furthermore, the flexibility of Virtual Wires allows the emulation architecture to be balanced for each logic design application.

[1] Although this paper focuses on logic emulators, Virtual Wires technology can be employed in any system comprising multiple interconnected FPGAs.

In our first implementation, we support Virtual Wires with a Virtual Wires compiler. We augment available off-the-shelf technology, with special tools for Virtual Wires. These additional software tools are used after the translation, mapping, and partitioning programs, and before the FPGA specific software. In particular, we have developed a *Global Placer*, *Dependency Analyzer*, *Virtual Router*, and *Route Embedder* (Chapter 5).

The Global Placer assigns FPGA partition to specific FPGAs in the emulator hardware. After global placement, the Dependency Analyzer is run to analyze the logic signal dependencies among the FPGA partitions. With this dependency information, the Virtual Router then statically schedules and routes inter-FPGA communication. The Route Embedder uses the resulting schedule and route to construct, *in the FPGA technology*, a statically routed network. This constructed FPGA hardware consists of a sequencer and shift loops (Chapter 6). The sequencer is a distributed finite state machine. It establishes virtual connections between FPGAs by strobing logical wires into special shift registers, the shift loops. Shift loops are then alternately connected to physical wires according to a predetermined schedule. These structures are embedded directly into the netlist for each FPGA.

In our results from compiling two complex designs, the 18K gate Sparcle microprocessor [3] and the 86K gate Alewife Communications and Memory Management Unit (A-1000) [29] (to be presented in Chapter 7), we show that the use of Virtual Wires can decrease FPGA chip count by a factor of 3 for Sparcle and 10 for the A-1000, assuming a crossbar interconnect. With Virtual Wires, a two dimensional torus interconnect can be used for only a small increase in chip count (17 percent for the A-1000 and 0 percent for Sparcle). Without Virtual Wires, the cost of replacing the full crossbar with a torus interconnect is over 300 percent for Sparcle, and practically impossible for the A-1000. Estimated emulation speeds are comparable to speeds without using Virtual Wire, ranging from 1 MHZ for the A-1000 to 2 MHZ

for Sparcle. Neither design compilation is bandwidth limited, but rather constrained by its critical path. With Virtual Wires, use of a low dimension network reduces emulation speed proportional to the network diameter: a factor of 2 for Sparcle and 6 for the A-1000. Based on the implications of these results we have fabricated a scalable, low cost emulation system [42].

Our final contribution is an analysis of the overhead of Virtual Wires and hard wires. We introduce *Rent ratios* and a *topological factor* for predicting pin limitation overhead without Virtual Wires. For Virtual Wires, we derive the optimal partition size. To conclude, we show how Virtual Wires gate utilization can scale with increasing FPGA device size, while hardwired utilization may even decline with increasing FPGA device size.

1.5 Organization of Thesis

This thesis continues with Chapter 2 detailing background information in several related fields. Previous work on hardware accelerated logic simulation provides much of the insight for today's logic emulators. Details of recent FPGA-based logic emulation systems, in industry and academia, support the pin limitation claims to be presented in Chapter 3. We also contrast Virtual Wires with related work in multiprocessor computing and data networks. Chapter 3 discusses the extent of the pin limitation problem in FPGA emulators and presents both a theoretical and an experimental formulation of the partitioning problem.

We present the basic concept behind Virtual Wires in Chapter 4, and give some details of a complete software system based on Virtual Wires in Chapter 5. In Chapter 6 we continue by describing the internal FPGA logic necessary for Virtual Wires. This chapter also describes a scalable, low cost emulation board which has been fabricated to verify this research. In Chapter 7 we use the front end of this software system to determine the benefit of using Virtual Wires. We perform these experiments by compiling two current benchmark designs for various interconnect topologies and FPGA device sizes. Following this empirical study, Chapter 8 presents a theoretical comparison of emulation with and without Virtual Wires. Finally, Chapter 9 concludes the thesis, summarizing the contributions made and presenting an extensive outline of directions for future research.

Chapter 2

Background

2.1 Hardware Accelerated Logic Simulation

At a high level, hardware accelerated logic simulation is very similar to FPGA-based logic emulation. The common approach to making these simulations go fast is to use large amounts of hardware. This section briefly discusses two such systems which do not use FPGAs, but are related to the ideas presented in this thesis. Both systems are parallel and require circuit partitioning. The first also has an attribute which isn't normally considered a simulator function - the ability to directly interface with real hardware (logic emulators use special "pods" which plug into the target system for interfacing). The second uses a communication technique analogous to static routing, the communication technique used by Virtual Wires.

Designed at the IBM T. J. Watson Research Center, the *Yorktown Simulation Engine* (YSE) [38] is a special-purpose highly-parallel programmable logic machine for the gate-level simulation of logic. It can simulate up to one million gates at a speed of over two billion gate simulations per second. It is composed of an array of logic processors, each capable of simulating 4096 gates. An inter-processor switch provides communication among up to 256 logic array processors. Applications range from design verification and fast simulation, to logic analysis and "partial hardware bring-up" (*i.e.* logic emulation). This engine is a follow-on to the previous Logic Simulation Machine [12], both of which are based on the concepts of John Cocke.

The *Very Large Simulation Subsystem* (VLSS) [45] is another massively parallel simulation engine, capable of performing hundreds of billions of gate evaluations per second. It is a compiled simulator with each processor being a special purpose full-custom chip containing 64 two-input one-output programmable gates. A full system, with 128,000 such chips, is required to achieve the rated performance. Communication is done with a hierarchy of buses using a time-division multiplexing (TDM) technique. In this technique, data is strobed from/to the bus by comparing a stored time-slot value for each pin with an on-board time-slot counter.

See [10] for a more comprehensive survey of hardware accelerators used in computer aided design.

2.2 FPGA-based Logic Emulation

Even before FPGAs existed, logic emulation based on *cellular arrays* [39] was being explored in academia. Cellular arrays can be traced back to work on cellular automata in the 1960's [37]. However this work could not be applied to logic emulation without the enabling technology of large scale integration, rapidly progressing in the 1970's. In Frank Manning's 1975 thesis [34], the use of rectangular arrays of programmable logic cells, in which information stored in a cell tells that cell how to behave, was proposed as an efficient means of logic simulation. Parts of his work explicitly show how such an "embedded machine" can be used in the place of another machine - conceptually very similar to FPGA-based logic emulation.

Since this work, FPGA-based logic emulation systems have been developed for design complexities ranging from several thousand to several million gates. The software for these systems is considered the most complex component and comprises a major portion of system price. In 1992, the price for a system was in the range of $2 to $3 per emulated gate. Thus such a system can easily merit a million dollar price tag.

Quickturn Inc. has developed emulation systems which interconnect FPGAs in a two-dimensional mesh [44] and, more recently, in a partial crossbar topology [33]. In both systems, they used the Xilinx 3090 series FPGA [46]. In the first system, they used FPGA for logic as well as inter-FPGA routing. In the second system, the Enterprise system, they added a custom crossbar chip alongside each FPGA. In this second system, nearly fifty FPGAs and fifty crossbar chips are contained on a single board capable of emulating up to 30,000 gates. These boards communicate via a sophisticated backplane structure (containing more custom crossbar chips). The system hierarchy can be further expanded by cabling together multiple boxes (each with a backplane) for a maximum capacity of over 6 million gates. Up to 64 Mbytes of memory can be emulated with the addition of special memory emulation cards.

Quickturn also sells MARS emulators, originally developed by Pie Design Systems, Inc. This system is very similar to Quickturn's Enterprise system. The MARS systems use the more advanced Xilinx 4000 series [47] and do not have special crossbar chips. The MARS system takes advantage of the Xilinx 4000's ability to emulate memory directly in the FPGA.

The Virtual ASIC system by InCA, ltd. [25] uses a combination of nearest neighbor and crossbar interconnect. This system is targeted at the low-end of the market and is not scalable like the Quickturn system. InCA also provides a logic partitioning tool which can be used for designers who build their own hardware.

AnyBoard, developed at North Carolina State University, [17] is an FPGA-based reconfigurable system. Among other uses, Anyboard can be used as a logic emulator for designs of a few thousand gates. The Anyboard project is focused on providing a low cost, rapid prototyping solution for universities.

For an overview of other available prototyping and emulation solutions, see [14].

2.3 Network Communication

Related network communication concepts for improving usable bandwidth include static routing, virtual circuit routing, and virtual channels.

The set of inter-FPGA communications established via Virtual Wires makes up a statically routed network. Static routing can be used whenever communication can be predetermined. *Static* refers to the fact that all data movement can be determined and optimized at compile-time. This mechanism has been used in scheduling real-time communication in a multiprocessor environment [30]. More recently, simulated annealing and linear programming techniques have been used to statically schedule communication patterns for the Numesh multicomputer [35]. Static routing has also been used in FPGA-based systolic arrays, such as Splash [20].

Virtual circuit routing [9], found in connection-oriented networks, is store-and-forward switching in which a particular path is set up when a communication session is initiated, and is maintained for the duration of that session. A session in this context is like a call in a telephone network - it represents a node pair that are sending messages to each other. The virtual aspect of such a circuit is that the physical connections are not dedicated to a single session, but can be multiplexed, like Virtual Wires, among several sessions. This allows transmission bandwidth to be used on an as-needed basis.

Using Virtual Wires for static routing for logic emulation is also similar to the use of virtual channels [16] in dynamically-routed networks. These channels decouple resource allocation, allowing active packets to pass blocked packets to use bandwidth that would otherwise be left idle.

Chapter 3

The Pin Limitation Problem

One of the high level design issues faced in FPGA-based logic emulation is the pin limitation problem. This problem is not unique to logic emulation - any attempt to split a large design onto multiple modules, as in the construction of mainframes in the 1960's, can potentially evoke this problem. If each module had an unlimited number of pins, then this problem would not exist. We could simply pack the modules randomly, connect up the pins, and have a functional system.

3.1 Pin Limitations in Logic Emulation

In existing FPGA-based emulator architectures, both the logic configuration and the network connectivity remain fixed for the duration of the emulation. Each emulated gate is mapped to one FPGA equivalent gate and each FPGA partition signal is allocated to one FPGA pin. For a partition to be feasible, the partition's gate and pin requirements must be no greater than the available FPGA resources. This constraint yields the four scenarios shown in Table 3.1.

For mapping typical circuits onto available FPGA devices, partitions are predominately pin limited; all available gates can not be utilized due to a lack of pin resources

Not Limited (unused FPGA pins and gates)	**Gate limited** (no unused gates, some unused pins)
Pin Limited (no unused pins, some unused gates)	**Balanced** (no unused pins or gates)

Table 3.1: Possible Partition Limitation Scenarios

to support them. For example Figure 7-2, in Chapter 7, shows that for equal gate counts in the FPGA partitions and FPGA devices, the required pin counts for FPGA partition sizes of our sample designs are much greater than the available FPGA device pin counts. Low utilization of gate resources increases both the number of FPGAs needed for emulation and the time required to emulate a particular design. Pin limits set a hard upper bound on the maximum usable gate count any FPGA gate size can provide. This discrepancy will only get worse as technology scales; trends (and geometry) indicate that available gate counts are increasing faster than available pin counts.

Besides the primary effects of pin limitation just outlined, logic emulators are also limited by several secondary effects. For example, when emulating VLSI chips, we would like to emulate on-chip SRAM memory. Current emulators either use the internal FPGA SRAM to emulate this memory, or provide reconfigurable cards which consume valuable pins. But most on-chip memories are much smaller than a single off-the-shelf SRAM memory chip, which is very cheap. Pin limitations restrict us from placing these cheap memories where they can be most useful - directly inside the FPGA array.

Another problem is observability. Since the purpose of logic emulation systems is to help debug and verify a design, we would ideally like access to all internal signals. But any signals we observe must use pins!

3.2 Theoretical Formulation: Graph Partitioning

As defined in set theory, a *partition* of A is a collection of pairwise disjoint nonempty sets, $\{A_i\}$, whose union is A:

- $A_i, A_j \in A$ and $i \neq j$ implies $A_i \cap A_j = \emptyset$, and
- $A = \bigcup_{A_i \in A} A_i$.

In particular, we are interested in network partitions, where each set member is a *node* of a directed hypergraph network. In electrical circuits, these nodes represent the primitive cells of a digital design (usually defined in a *library*) and are interconnected by *hyperedges* which represent the digital wiring. "Hyper" refers to the possibility of *fanout* in a digital network. The graph partitioning problem belongs to the class of NP-hard problems [36] - given n nodes to partition into m subsets, the number of possible different partitionings, P, is:

$$P = \frac{n!}{(n/m)!^m m!}$$

In FPGAs, differing internal resources, such as registers and lookup tables, complicate the partitioning problem. However, for simplicity we assume that all FPGA resources are either *gates* or *pins*. The number of *feasible* partitionings is then reduced only by the finite pin and gate restrictions on each subset, or partition.

Given partitions with maximum gate capacity G and fixed pin capacity P, and individual gate and pin requirements of g_i and p_i, we can then define an *optimal* *partitioning*, with respect to circuit size, as one which partitions a set of n nodes into the minimal number of subsets m:

$$min(m)$$

$$g_i < G, i \in m$$

$$p_i < P, i \in m$$

Pin requirements for a given partitioning can be determined from a two dimensional *adjacency matrix* derived from the edges in the unpartitioned network. In adjacency matrix M, each term M_{ij}, stores how many connections go from node i to node j. When partitioning a network, we collapse this matrix into a new adjacency matrix, M', in which M'_{ij} stores the number of connections from partition subset i to partition subset j. M' can be derived from M by summing the rows or columns of all the nodes collapsed into each partition row or column.

Because the partitioning problem can not be solved in polynomial time, programmers must use efficient heuristic partitioning algorithms to optimize partitions with a reasonable amount of computational effort. Well-known algorithms include the Kernighan and Lin (K&L) [26] mincut algorithm which has complexity in the number of components of $O(n_2 \log n)$ for an optimization pass, and an efficient bisection heuristic by Fiduccia and Mattheyses (F&M) [19] which exhibits linear complexity in the number of pins, running in $O(p)$.

3.3 Empirical Observation: Rent's Rule

In 1960, E.F. Rent of IBM prepared two internal memoranda containing the log plots of pins versus gates (termed blocks in the literature of that time) for portions of the IBM series 1400 computers. In [31], Landman and Russo present a careful analysis of his results based on a computer algorithm used to partition several designs for various pin limits. The basic result is the following equation:

$$\textbf{Rent's Rule}: \qquad N_p = K_p N_g^B \tag{3.1}$$

where N_p is the number of pins, N_g is the number of gates, K_p is a proportionality constant, and B is Rent's constant. The values of K_p and B reported for the IBM computers were 2.5 and 0.6, respectively. This rule only applies if $N_g \gg 1$. As with most rules, it has limitations. See [31] and [7].

Equation 3.1 can be used to measure the communication parameters of a given implementation technology as well as the parameters of a design specification. A design's architecture and organization greatly affect its parameters. For example, pipelining increases communication requirements due to dependencies between pipeline stages. In high-speed systems, reliance on parallel I/O and avoidance of multiplexed or bidirectional I/O places high demands on communication requirements.

Figure 3-1 shows design parititioning data points fitted to Rent's rule on a log-log scale.[1] When Rent's curve is applied to different size partitions of the same circuit, then we can assign a special meaning to the constants K_p and B (since this is a log-log curve, B is the slope of the curve and K_p is the intercept). K_p gives us a rough indication of the magnitude of the communication requirements of the circuit, while B provides information about the inherent locality of the partitioned circuit. For a circuit, the larger B is, the less communication locality there is. Typical values of B for circuits are in the range of $0.4 - 0.7$. For a technology, the larger B is the more long-distance interconnect is available. For example, $B = 0.5$ for a 2-d mesh and $B = 1$ for a crossbar. Rent's constant can tell us how well a category of designs will scale - if Rent's constant for the technology is smaller than that of the design, then we know that at some point we cannot continue to scale efficiently.

[1] these points are from the Sparcle design to be discussed in Chapter 7.

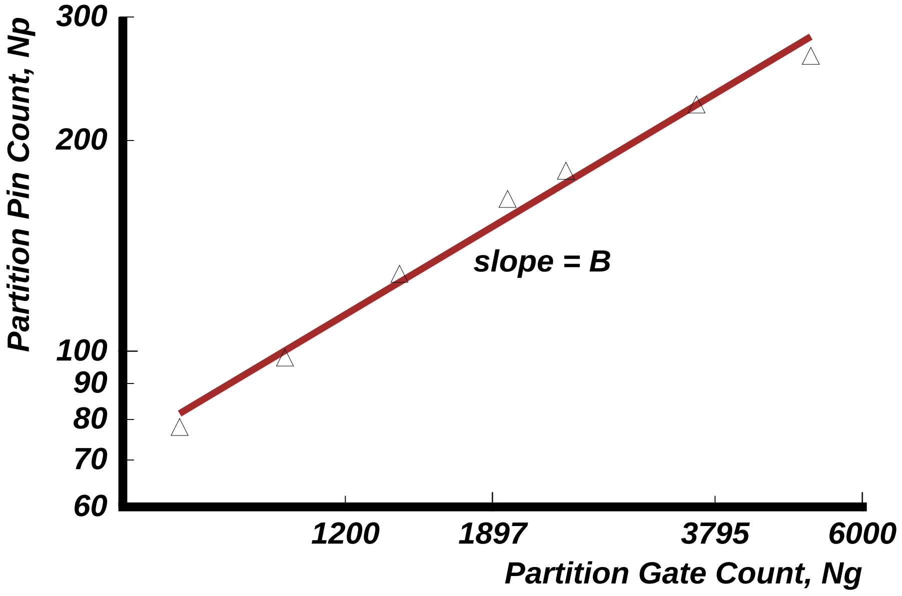

Figure 3-1: Example Rent's Rule Plot

Assuming we are in the pin-limited region, values of B near 0.5 mean that a reduction in total pin requirements can result in a roughly quadratic increase in the usable gates per module. Thus an optimizing partitioner can be quite valuable. In fact any mechanism which reduces the requirements on FPGA pin resources, such as the use of a crossbar network, can have the same quadratic increase. However, when Virtual Wires are employed, this quadratic effect will be mitigated to a linear tradeoff of pins for internal gate resources. See Sections 7.6, Section 7.8, and Chapter 8 for more detail on these tradeoffs.

Chapter 4

Virtual Wires

To overcome pin limitations in FPGA-based logic emulators, we propose the use of a compilation technique which utilizes *Virtual Wires* [6]. This chapter describes an implementation of Virtual Wires in the context of a complete emulation software system, independent of target FPGA device and interconnect topology. Chapter 5 will continue with focus on the implemented software framework itself: the Virtual Wires Compiler.

4.1 Conceptual Overview

One to one allocation of emulation signals (logical wires) to FPGA pins (physical wires) does not exploit available off-chip bandwidth because:

- emulation clock frequencies are one or two orders of magnitude lower than the potential clocking frequency of the FPGA, technology.
- all logical wires are not active simultaneously.

The emulation clocking frequency is slower than the potential FPGA clocking frequency because of long internal FPGA routing delays and because of long inter-

FPGA routing delays. The internal FPGA routing delays are long because the signals must pass through the programmable switches which determine how the FPGA is routed. Inter-FPGA delays are slow because of chip-to-chip crossings. In comparison with a non-programmable circuit, logical functions are also slower. However, internal registers can toggle just as fast as in a non-programmable circuit.

In end-product FPGA circuits, higher clocking speeds are achieved by pipelining the design more deeply. In other words, the maximum circuit depth is reduced so that less delay is incurred in the critical paths. But when we map a VLSI chip design for a fast technology onto FPGAs, this extra pipelining is not automatically added.

Logical wires are not active simultaneously because of differing computational delays, and signal dependencies. While glitches (changes in a signal during the early part of the clock cycle, before the signal settles to a final value) can propagate throughout the circuit, their effects are irrelevant in a synchronous design. Thus each signal only needs enough bandwidth to transfer its value once per emulation clock. Furthermore, these windows of transfer are not restricted to happen at any particular time during the emulation clock period, so long as signal dependencies are observed.

Thus although the FPGA partitions are *pin limited*, each pin is severely under-utilized. For an example, if an FPGA's pins could be clocked at 50 MHz, but are only clocked at an emulation clock of 1 MHz, a system which utilizes 100 percent of available pins can only exploit 2 percent of the theoretically available bandwidth! We would like to make better use of available bandwidth, especially since I/O limitations are causing low gate efficiency.

By pipelining and multiplexing physical wires, we can create Virtual Wires to increase usable bandwidth. A virtual wire represents a single connection between a logical output on one FPGA partition and a logical input on another FPGA partition. However, this connection is composed of one or more intermediate physical wires, and furthermore, each of these physical wires may be shared, in a time multiplexed fashion, with other virtual connections. Figure 4-1 shows an example of six logical wires allocated to six physical wires. In comparison, Figure 4-2 shows the same example with the six logical wires sharing a single physical wire. In this example, the physical wire is multiplexed, or *virtualized*, between two pipelined *shift loops* (see Section 6.1.1). By clocking the physical wires at the maximum frequency of the FPGA technology, several logical connections can share the same physical resources without a significant slowdown in the emulation clock speed.

4.2 Resource Scheduling

Multiplexing and pipelining pin resources introduces the issues of resource contention and data dependency. In a hardwired system, where FPGA pins are dedicated, there can be no resource contention: pins are not shared. Also, data dependencies are automatically met when pins are locked for the entire emulation clock. In our Virtual Wires system we must rely on resource scheduling to avoid contention and to meet any data dependencies.

Systems based on Virtual Wires exploit several properties of digital circuits to allow efficient resource scheduling. In a logic design, evaluation flows from system inputs to system outputs. In a synchronous design with no combinational loops, this flow can be represented as a directed acyclic graph. Thus, through intelligent dependency analysis of the underlying logic circuit, logical values between FPGA partitions only need to be transmitted once per emulation clock period (see Section 5.6). Furthermore, since circuit communication is inherently static (the wiring does not change from one clock period to the next), communication patterns will repeat in a predictable fashion. By exploiting this predictability, communications can be *statically* scheduled to increase the utilization of pin bandwidth. Section 5.7 discusses our resource scheduling solution in detail.

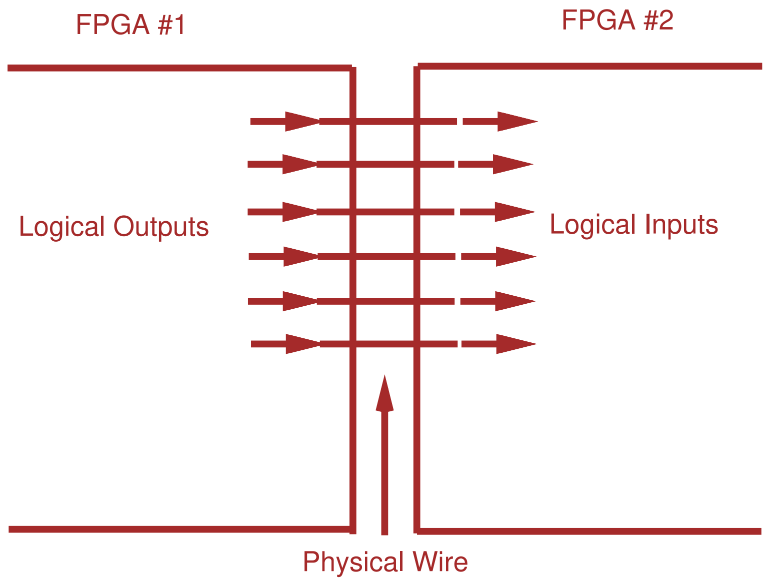

Figure 4-1: Hard Wire Interconnect

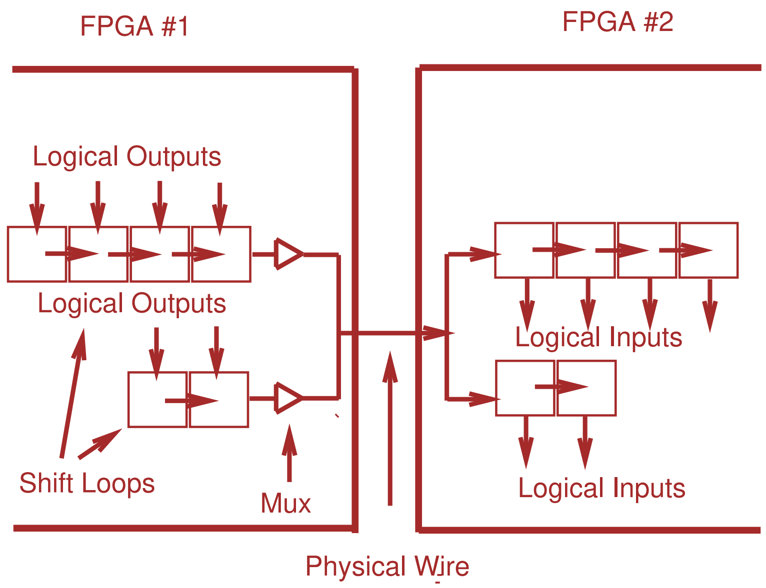

Figure 4-2: Virtual Wire Interconnect

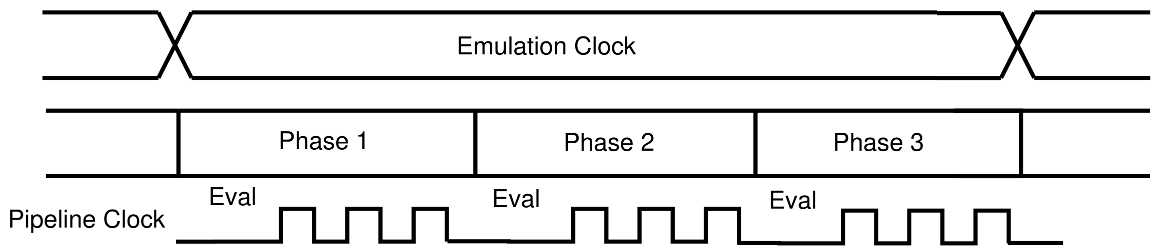

Figure 4-3: Emulation Phase Clocking Scheme

4.3 The Emulation Clocking Framework

The various clocks used in the Virtual Wires system define a framework for system-level design with Virtual Wires. Let us first describe this framework based on multiple clocks (see Figure 4-3).

The *emulation clock* period is the clock period of the logic design being emulated. We break this clock into evaluation *phases*. We use multiple phases to evaluate the multiple FPGA partitions across which the combinational logic between flip-flops in the emulated design may be split. In other words, evaluation within each FPGA partition, followed by the communication of results to other FPGA partitions is accomplished within a phase.

A phase is divided into two parts: an evaluation portion and a communication portion. Evaluation takes place at the beginning of a phase, and logical outputs of each FPGA partition are determined by the logical inputs in the input shift loops. At the end of the phase, outputs are sent to other FPGA partitions with the pipelined shift loops and intermediate hop stages (see Chapter 6). These pipelines are clocked with a *pipeline clock* (Figure 4-3) at the maximum frequency of the FPGA. After all phases within an emulation clock period are complete, the emulation clock is ticked.

In contrast, hardwired systems dedicate a physical pin to a distinct wire in the circuit and let the evaluation "flow" through multiple partitions within the emulation clock period until the entire system settles. Phases in Virtual Wires systems allow a physical pin that is unused during some portion of the emulation clock period to be utilized by other signals.

4.4 Limitations and Assumptions

The use of Virtual Wires is limited to the synchronous logic portion of a digital design. Any asynchronous signals must still be "hardwired" to dedicated FPGA pins. This limitation is imposed by the inability to statically determine dependencies in asynchronous cycles. Furthermore, we assume that each *combinational loop* (such as a flip-flop) in a synchronous design is completely contained in a single FPGA partition. This last assumption is taken care of if combinational loops are contained in the base library primitives of the input technology.

For simplicity, this model assumes that the emulated logic uses a single global clock. We also assume that the circuit contains no internal busses or tri-state devices. There is no fundamental reason why these limitations cannot be overcome with a more sophisticated implementation of Virtual Wires. However, such improvements are beyond the scope of this thesis.

Chapter 5

Software Framework

In our first implementation, we support Virtual Wires with a *Softwire Compiler*. The Virtual Wires portion of this compiler analyzes logic signal dependencies and statically schedules and routes FPGA communication. These results are then used to embed, in the FPGA technology, a statically routed network.

5.1 Overview

The input to the Softwire compiler consists of a netlist of the logic design to be emulated, target FPGA device characteristics, and FPGA interconnect topology. The compiler then produces a configuration bitstream which can be downloaded onto the emulator. Figure 5-1 outlines the compilation steps involved. Briefly, these steps include translation and mapping of the netlist to the target FPGA technology, partitioning the netlist, placing the partitions into an interconnect topology, routing the inter-node communication paths, and finally FPGA-specific automated placement and routing (APR). The shaded tools in Figure 5-1 are the tools which we designed for implementing Virtual Wires. These tools include the *Global Placer*, *Dependency Analyzer*, *Virtual Router*, and *Route Embedder*. These tools take the place of the

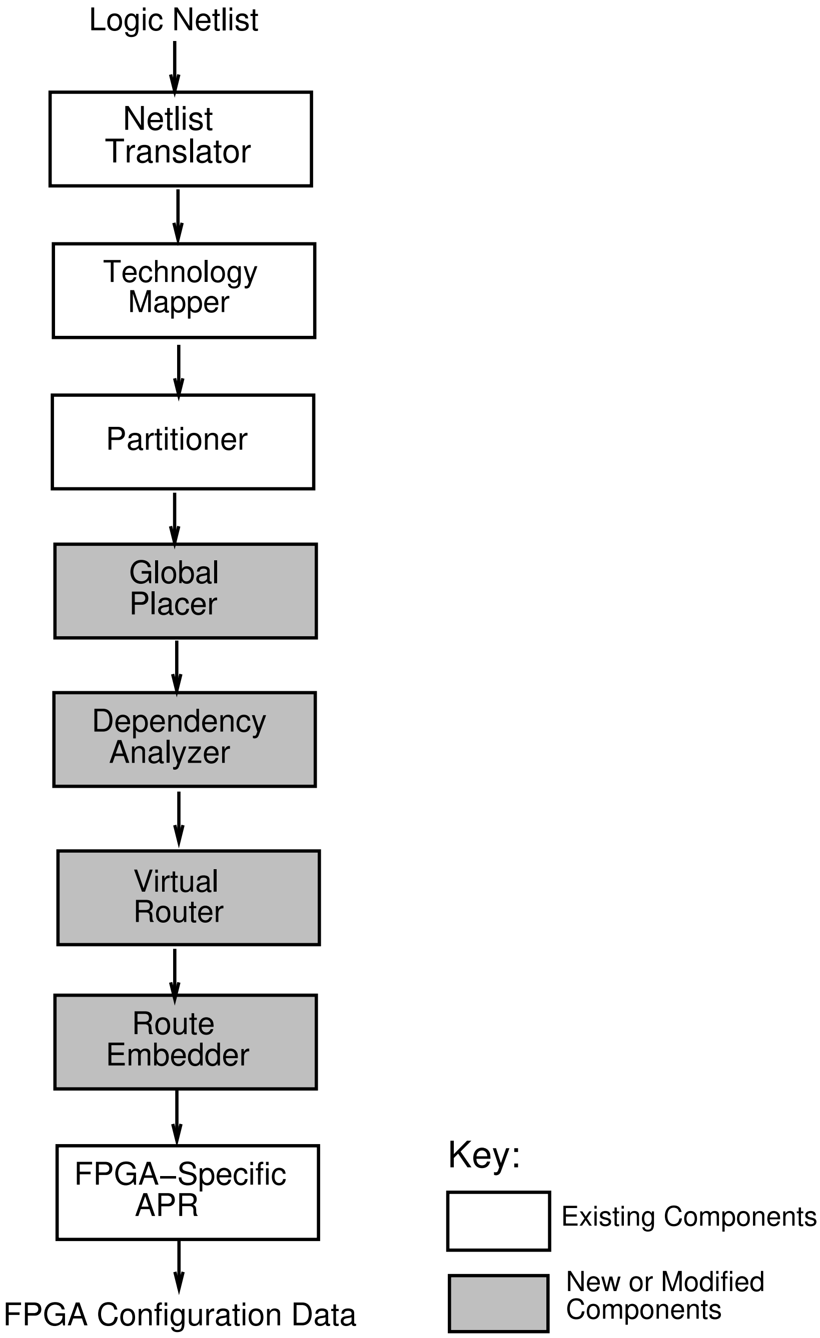

Figure 5-1: Softwire Tool Flowchart

hardwired global place and route tools of hardwired logic emulators. We implemented these new tools with less than 10,000 lines of C and Unix scripts.

5.2 Netlist Translation

The input netlist to be emulated is usually generated with a hardware description language or schematic capture program. This netlist must be syntactically translated into a format readable by our Softwire tools. Commercial tools are available for generic translation.

5.3 Technology Mapping

The translated netlist is specified in terms of the source technology library - for example LSI's LCA100K technology. This netlist must next be mapped to a target library of FPGA primitives. The technology mapping problem can be defined as follows:

> **Technology Mapping Problem:** Given a netlist N_s and a source library L_s, which defines the functionality of the primitives in N_s, produce a netlist N_t which is defined in terms of a given target library L_t.

It is important to perform this operation before partitioning so that partition gate counts accurately reflect the characteristics of the target FPGAs. Otherwise, differences in technology efficiencies will adversely affect the partitioning step.

A very simple technology mapping involves defining each primitive in the source library in terms of primitives in the target library. The inefficiency of this kind of mapping can be largely made up if followed by a logic optimization pass. If more sophisticated mapping is required, commercially available tools can be purchased which will map from one technology to another.

After completion of this step, the netlist has been mapped as if the target system were one large FPGA. If desired, we can use logic optimization tools on this netlist to further optimize for the target technology.

5.4 Partitioning

After mapping the netlist to the target technology, it must be partitioned into logic blocks which can fit into the target FPGAs. With only hardwires, each partition must have both fewer gates and fewer pins than the target device. With Virtual Wires, the total gate count (logic gates and virtual wiring overhead) must be no greater than the target FPGA gate count. In our current implementation, we use the Concept Silicon partitioner by InCA [25]. This partitioner performs K-way partitioning with min-cut and clustering techniques to minimize partition pin counts.

The output of the partitioning tool consists of a set of FPGA partitions, and a top level netlist which contains the information for wiring the partitions together to form the original netlist. It is important to note that partitioning in no way affects the underlying digital circuit - it only involves a re-arrangement of the circuit *hierarchy* to produce top level modules which will each fit into an individual FPGA, and which have minimum interconnection costs. Figures 5-2 and 5-3 show an example circuit before and after partitioning. The elements in each dotted circle will be grouped together in the same FPGA partition.

5.5 Global Placement

Following logic partitioning, individual FPGA partitions must be assigned to specific FPGAs. In a crossbar topology, placement is not important, since each FPGA is a distance of one from each other FPGA. However, in a mesh, placement is more

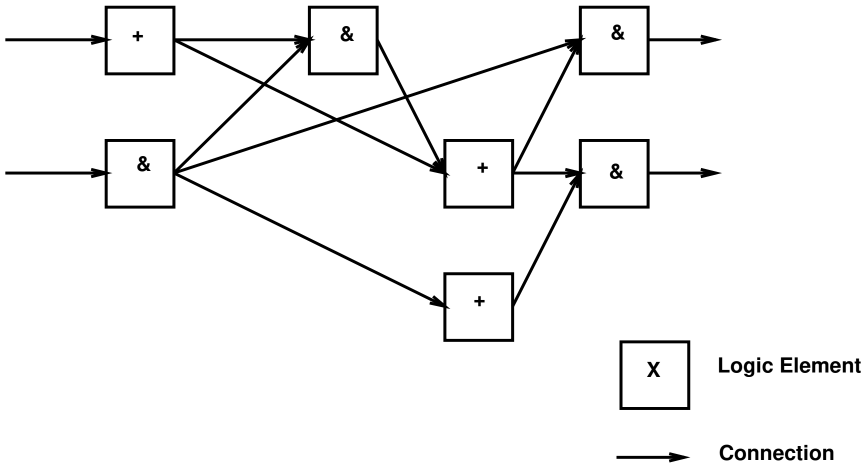

Figure 5-2: Unpartitioned Circuit

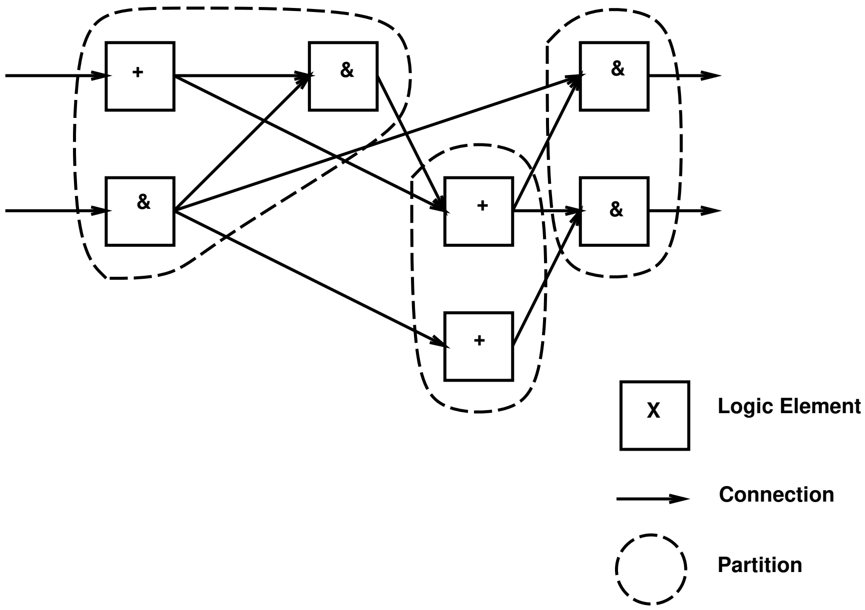

Figure 5-3: Partitioned Circuit

important. Initially, we have implemented a placer which minimizes total system communication for a given topology, thus reducing the required bandwidth for transferring the digital information. In the future, we would like to place the FPGA's to minimize the distances of signals along the critical path.

Total system cost T, is determined by multiplying the adjacency matrix M' (defined in Chapter 3) for the input partitioning by a Manhattan distance matrix D determined by the emulation system topology. All terms in this resulting product matrix are then summed to yield total system costs:

$$T = \sum_{ij} M'_{ij} * D_{ij}$$

By allowing any positive integer distance matrix, our compiler can map and optimize for arbitrary topology. Tables 5.1 and 5.2 show example distance matrices for the 3x3 mesh and crossbar topologies shown in Figures 5-4 and Figure 5-5.

We formally define this placement problem as follows:

> **Global Placement Problem:** Given a set of FPGA partition with its associated inter-partition connection matrix, M' and a set of FPGAs with its associated inter-FPGA distance matrix D, assign each FPGA partition to exactly one FPGA such that total system cost T is minimized.

By our definition of total system cost, an optimal global placement will minimize the total number of chip-to-chip virtual wires, including through wires. The disadvantage of this cost metric is that it will not optimize for potential *hot spots* in the network interconnection. However, since there is no limit to the number of logical wires than can be multiplexed on a given physical wire, the net effect of any network congestion will be a slow down in emulation speed, not an increase in the number of FPGA devices required.

As defined, global placement is an NP-complete integer assignment problem. However, simple algorithms to improve a given placement do exist. Our implemented

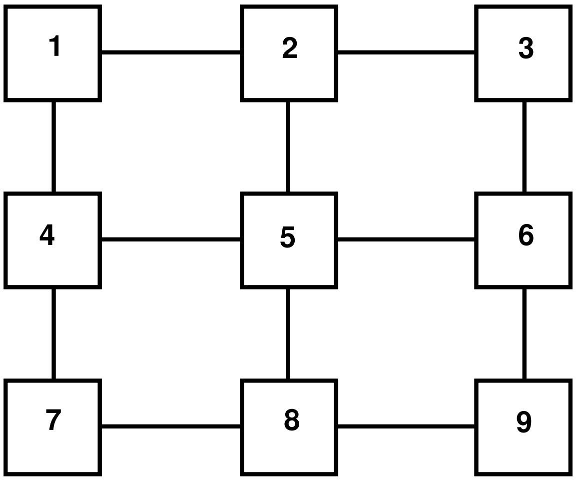

Figure 5-4: Mesh Topology

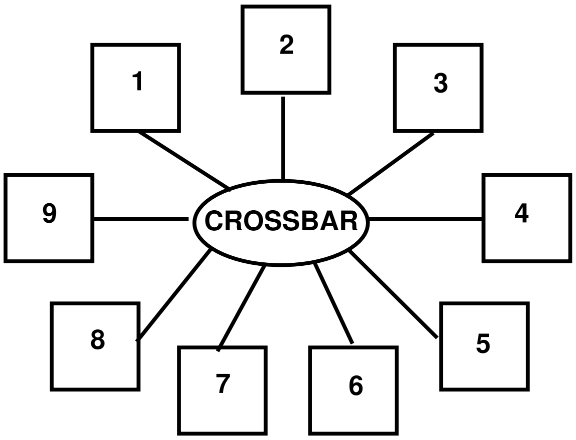

Figure 5-5: Crossbar Topology

$$\begin{vmatrix} 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 & 4 \\ 1 & 0 & 1 & 2 & 1 & 2 & 3 & 2 & 3 \\ 2 & 1 & 0 & 3 & 2 & 1 & 4 & 3 & 2 \\ 1 & 2 & 3 & 0 & 1 & 2 & 1 & 2 & 3 \\ 2 & 1 & 2 & 1 & 0 & 1 & 2 & 1 & 2 \\ 3 & 2 & 1 & 2 & 1 & 0 & 3 & 2 & 1 \\ 2 & 3 & 4 & 1 & 2 & 3 & 0 & 1 & 2 \\ 3 & 2 & 3 & 2 & 1 & 2 & 1 & 0 & 1 \\ 4 & 3 & 2 & 3 & 2 & 1 & 2 & 1 & 0 \end{vmatrix}$$

Table 5.1: Distance Matrix for a 3x3 Mesh Topology

$$\begin{vmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{vmatrix}$$

Table 5.2: Distance Matrix for a 3x3 Crossbar Topology

placement optimizer first makes a random placement. This is followed by a local search for cost-reducing swaps. The swaps are discovered by looping through all possible $\begin{pmatrix} m \\ 2 \end{pmatrix}$ pairwise swaps, and choosing either the best possible swaps, or choosing good swaps when they are first found. For further optimization, we have implemented a simulated annealing [28] algorithm. This algorithm is iteratively used to improve the resulting placement. The advantage of using simulated annealing is that local minima can be avoided. However, there is no useful upper bound on the time it takes to find an optimal solution.

5.6 Dependency Analysis

The circuits in the FPGA partition for each FPGA form a *sequential machine* (Figure 5-6). Since each FPGA partition is to be completely contained in a single FPGA, the execution of the internal circuitry may not seem relevant to forming multi-FPGA circuits. This is not the case - we must know which outputs depend on which inputs for each FPGA partition in order to correctly meet circuit dependencies.

We define signal dependence as follows:

> **Signal Dependence:** An output depends on an input if a change in that input can combinationally change the output.

In the spirit of finite state machines (FSMs), we define a Moore output to be a partition output which only depends on the internal next state. A Mealy output can depend on the next state, as well as inputs to the machine. If all FPGA partitions are strictly Moore machines (with only Moore outputs), then there is no need for dependency analysis. However, this is not the case - *combinational paths* may pass through multiple FPGA partitions.

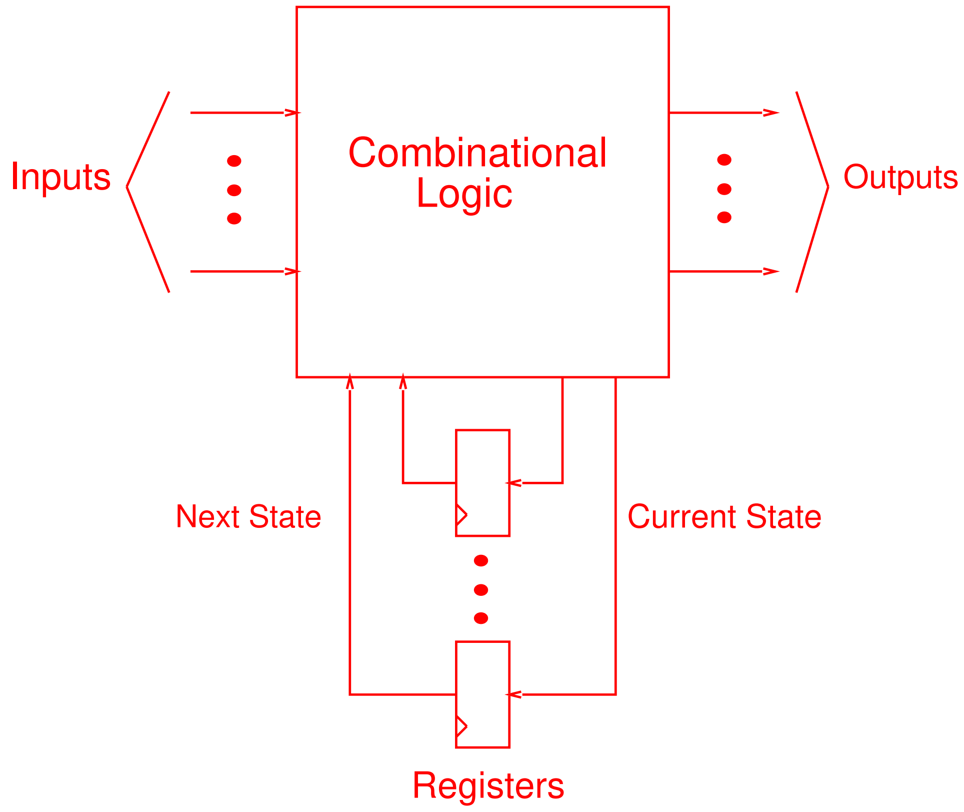

Figure 5-6: Sequential Machine

Primitive	Dependence
Combinational Element	All Inputs
Register Element	No Inputs

Table 5.3: Dependence Rules for Library Primitives

The combinational logic in each FPGA partition forms a directed acyclic graph (DAG), which we refer to as the signal flow graph (SFG). Given a netlist, we calculate a dependence matrix P, by backtracing all combinational paths from Mealy outputs towards the inputs. In backtracing we assume that all outputs depend on all inputs for combinational library primitives, and that no outputs depend on any inputs for register library primitives (Table 5.3). In hierarchical netlists, the procedure is simplified with recursion - we can calculate a dependence matrix for each submodule, and then use the submodule matrices when determining the parent's matrix.

In our initial implementation, P consists entirely of 1s and 0s. $P_{ij} = 1$ if output i depends on input j, and 0 otherwise. In the future, we plan to augment this analysis with static timing information. In this case we will replace the values in P with the *propagation delay* from each input to each output. If there is no dependence, the propagation delay is $-\infty$.

Once we have determined P for each FPGA partition, we then determine the SFG for the toplevel netlist which interconnects the FPGA partitions. Figure 5-7 shows an example of such a graph. Each node in this SFG represents a partition output (or external I/O). Analysis of this graph reveals the critical paths. These will affect the execution speed of our emulation. We define the longest critical path to be the path which passes through the most FPGA partitions combinationally. This path may originate at an external input or internal register, and will terminate at an external output or internal register. We cannot assume that each FPGA partition is a combinational primitive - this could potentially produce combinational cycles.

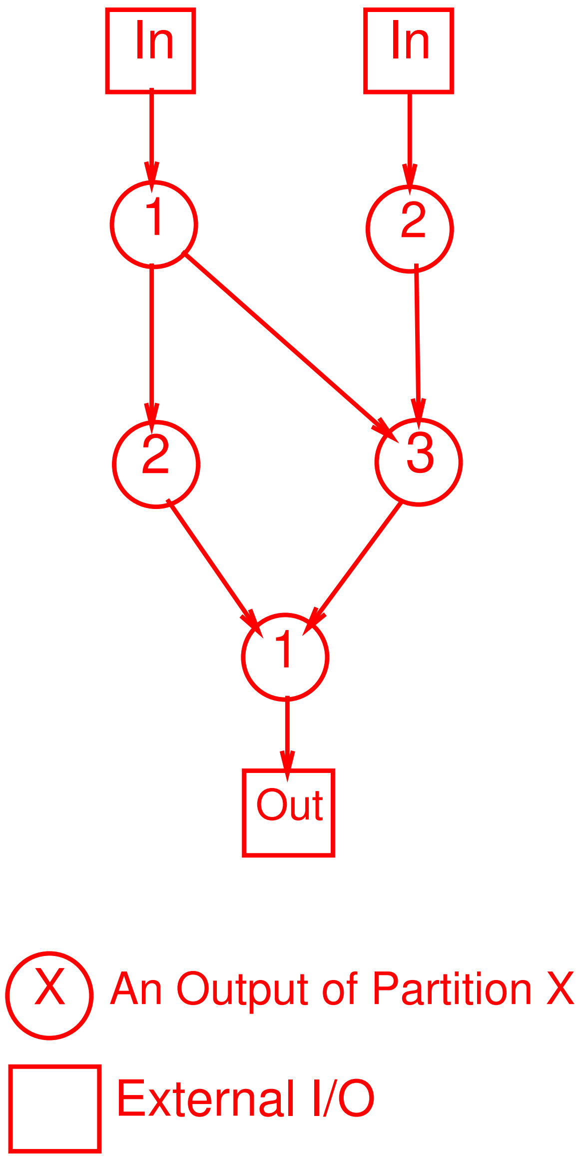

Figure 5-7: A Signal Flow Graph

Note that in Figure 5-7 there is a path that passes through partition 1 twice. While a combinational path may pass through a partition more than once, it cannot pass through a particular partition output multiple times - this would imply that the original network is not acyclic. Thus the worst case path length is at most the sum of all FPGA partition outputs. In fact, given an arbitrarily partitioned circuit, the tightest upper bound on combinational path length is equal to the maximum circuit depth of the unpartitioned netlist[1]. The maximum circuit depth is defined as the maximum levels of logic from any input (or register) to any output (or register).

5.7 Virtual Routing

The Virtual Router combines the operations of phase assignment and global routing. Inputs consists of a set of FPGA partitions that have been assigned to FPGA devices, the SFG from the dependency analysis, the topology of the emulation architecture, and the pin assignment of the external pod connectors to the target system. We route in both space and time simultaneously. This is achieved by scheduling each logical wire to a particular phase and assigning pipeline time and space slots on physical *ports* to form a virtual path. A port represents all direct connections from one FPGA to another FPGA. Actual pin assignment for each port is reserved until the embed phase.

Before phase assignment, we determine the *criticality* of each logical wire. Criticality is determined by the length of the longest dependency path (in the SFG) which originates at a logical wire. We want to schedule critical signals as early as possible for the fastest execution.

Phase assignment uses the following methodology. In each phase, the router first

[1]Given at least two partitions, this worst case can be constructed by walking the critical path and placing every element at an even level in partition 1 and every element at an odd level in partition 2. The remaining elements can go anywhere.

determines the schedulable wires. A wire is schedulable if all wires it depends upon have been scheduled in previous phases. The router then uses weighted shortest path analysis, with a cost function based on pin utilization to route as many schedulable signals as possible, routing the most critical signals first. Any schedulable signals which cannot be routed are delayed to the next phase.

When signals must traverse multiple hops, global paths must be routed. All wires in a given shift loop are pipelined along the same global path during the phase assigned to that shift loop. These global paths are *not* restricted to dimension-order routes, where signals must travel the complete distance in one dimension before turning to travel in the next dimension. Such routing is often used to simplify dynamic routing in multiprocessors. However, we do restrict these global paths to travel along a shortest distance path in FPGA hops. For a global crossbar, this restriction allows only one path. However, for a 2d-mesh, this restriction still allows available paths of

$$\frac{(X+Y)!}{X!\,Y!}$$

where X and Y are the 2-D manhattan distances between the neighboring nodes. Our routing algorithm is allowed to choose among these alternate paths to avoid congestion.

5.8 Route Embedding

The Route Embedder transforms the virtual routes assigned in the previous phase to appropriately-sized shift loops and associated logic. This logic is in the form of a netlist, and is added to each FPGA partition to complete the internal FPGA hardware description. The details of this constructed FPGA hardware are described in the next chapter.

In this implementation, the embed phase also assigns signals to particular pins (the router only assigns them to an abstract port going to a neighbor FPGA). Further optimization to permute this pin assignment are performed at this stage to improve intra-FPGA routing and timing.

After embedding is complete, there is one complete netlist for each FPGA. In order to verify that all the transformations to get to these netlists worked correctly, we insert all FPGA netlists into the top level netlist description of the emulator topology. We simulate this system level netlist (it's just another circuit) in software to verify that the virtual wires were constructed correctly. If our simulation vectors pass, then we have successfully *virtualized* the original netlist. (This step is not necessary once we trust that the virtualization is correct by construction.)

5.9 Vendor Specific APR

The netlists for each FPGA is then processed with vendor-specific FPGA automatic place and route (APR) software to produce configuration bitstreams. This software step is usually somewhat time consuming - each FPGA must be separately place and routed. However, this step is also very easy to parallelize, for example on a network of workstations.

The resulting configuration bitstream can then be downloaded to an FPGA array, such as the system described in Section 6.2, with the vendor download software. Once this last step is successful, we have an emulated machine which should be able to completely mimic all external functionality of the original system.

Chapter 6

Hardware Implementation

6.1 Hardware Support for Virtual Wires

The hardware for Virtual Wires is very simple. In fact, Virtual Wires requires no custom hardware support:[1] the software compiler synthesizes the required components directly into the FPGAs netlists. This "hardware" is downloaded into the configuration for the FPGA device. Thus, any existing multi-FPGA system can take advantage of virtual wiring.

There are many possible ways to implement hardware support for Virtual Wires. This section describes a simple and efficient implementation. The additional logic to support Virtual Wires can be composed entirely of *shift loops* and a small amount of phase control logic.

6.1.1 Shift Loops

A shift loop (Figure 6-1) is a circular, loadable shift register with enabled shift-in and shift-out ports. Each shift register is capable of performing one or more of the

[1] unless one considers re-designing an FPGA optimized for Virtual Wires.

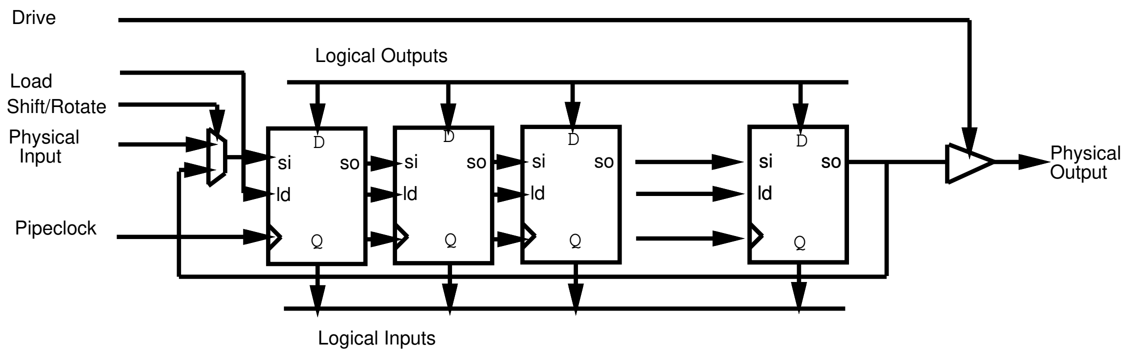

Figure 6-1: Shift Loop Architecture

- **Load** — Strobes logical outputs into shift loop.
- **Store** — Drives logical inputs from shift loop.
- **Shift** — Shifts data from a physical input into shift loop.
- **Drive** — Drives a physical output with last bit of shift loop.
- **Rotate** —Rotates bits in shift loop.

Table 6.1: Shift Loop Operations

following operations: *load*, *store*, *shift*, *drive*, and *rotate* (Figure 6.1). In our current design, for simplicity, all outputs loaded into a shift loop must have the same final destination FPGA. As described in Section 5.6, a logical output can be strobed once all its corresponding depend inputs have been stored. The purpose of rotation is to preserve inputs which have reached their final destination, and to eliminate the need for empty gaps in the pipeline when shift loop lengths do not exactly match phase cycle counts. Note that in this implementation store cannot be disabled.

Shift loops can be re-scheduled to perform multiple output operations. However, since the outputs of internal latches being emulated will depend on the logical inputs, inputs will need to be stored until the tick of the emulation clock.

6.1.2 Intermediate Hops

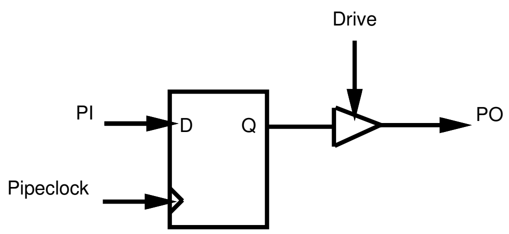

Figure 6-2: Intermediate Hop Pipeline Stage

For networks where multiple hops are required (*i.e.* a mesh), one bit shift loops which always shift and sometimes drive are used for intermediate stages (Figure 6-2). These stages are chained together, one per FPGA hop to build a pipeline connecting the output shift loop on the source FPGA with the input shift loop on the destination FPGA.

6.1.3 Phase Control Logic

The phase control logic directs execution in our simple implementation. This logic is *distributed* across the FPGA array to avoid bottlenecks and maximize speed. It functions as a sequencer which controls the phase enable (denoted *drive* in Figure 6-1) and strobe lines (denoted *load* in Figure 6-1), the pipeline clock, and the emulation clock. The phase enable lines control the the shift loop to FPGA pin connections. Recall that multiple shift loops (including single-bit shift stages for intermediate hop pipelining) can connect to a single physical pin through tri-state drivers as depicted in Figure 4-2. The phase strobe lines load the shift loops on the correct phases. The control logic is generated with a state machine specifically synthesized and optimized for a given phase specification and FPGA partitioning.

6.2 Prototype Emulation System

While this paper focuses primarily on software, the ultimate goal of this research is a low-cost, reconfigurable emulation system. Figure 6-3 shows a high level diagram of a Virtual Wires prototype board. This board contains a 2-D mesh of 16 Xilinx 4005 FPGAs. Connecting each FPGA with its neighbors are 8 bit point-to-point connections. The edges of the mesh are routed to connectors, allowing multiple board systems to be built by plugging boards together.

To allow for low cost memory emulation, each FPGA has an optional 64Kx4 SRAM. The cost of these memories is not significant compared to the FPGA costs. Communication with the host can be established either through a low bandwidth serial interface (via an HC11 microcontroller), or through a high bandwidth SBUS interface. The SBUS interface is programmed into a 17th Xilinx chip. This high bandwidth interface is important when running in a tightly coupled mode with a software simulator (Section 6.2.1). External I/O to the target systems is connected to unused edge connectors. In the case of VLSI chip emulation, this I/O, in the form of an emulation *pod*, is inserted in the chip socket of the target system.

6.2.1 Simulation Acceleration

Prior to performing in-circuit emulation in a target system, it is desirable to simulate the logical behavior of the design either individually or as a part of a target system simulation. Current simulation technology limits comprehensive testing of this type due to the slow gate evaluation speed of contemporary software simulators. Our prototype system addresses this limitation by allowing the logical behavior of the design under test to be *emulated* while the rest of the system is *simulated*. This tight coupling of different type of logic simulators is referred to as *cosimulation*.

Simulation is accelerated by replacing the design simulation model in a standard

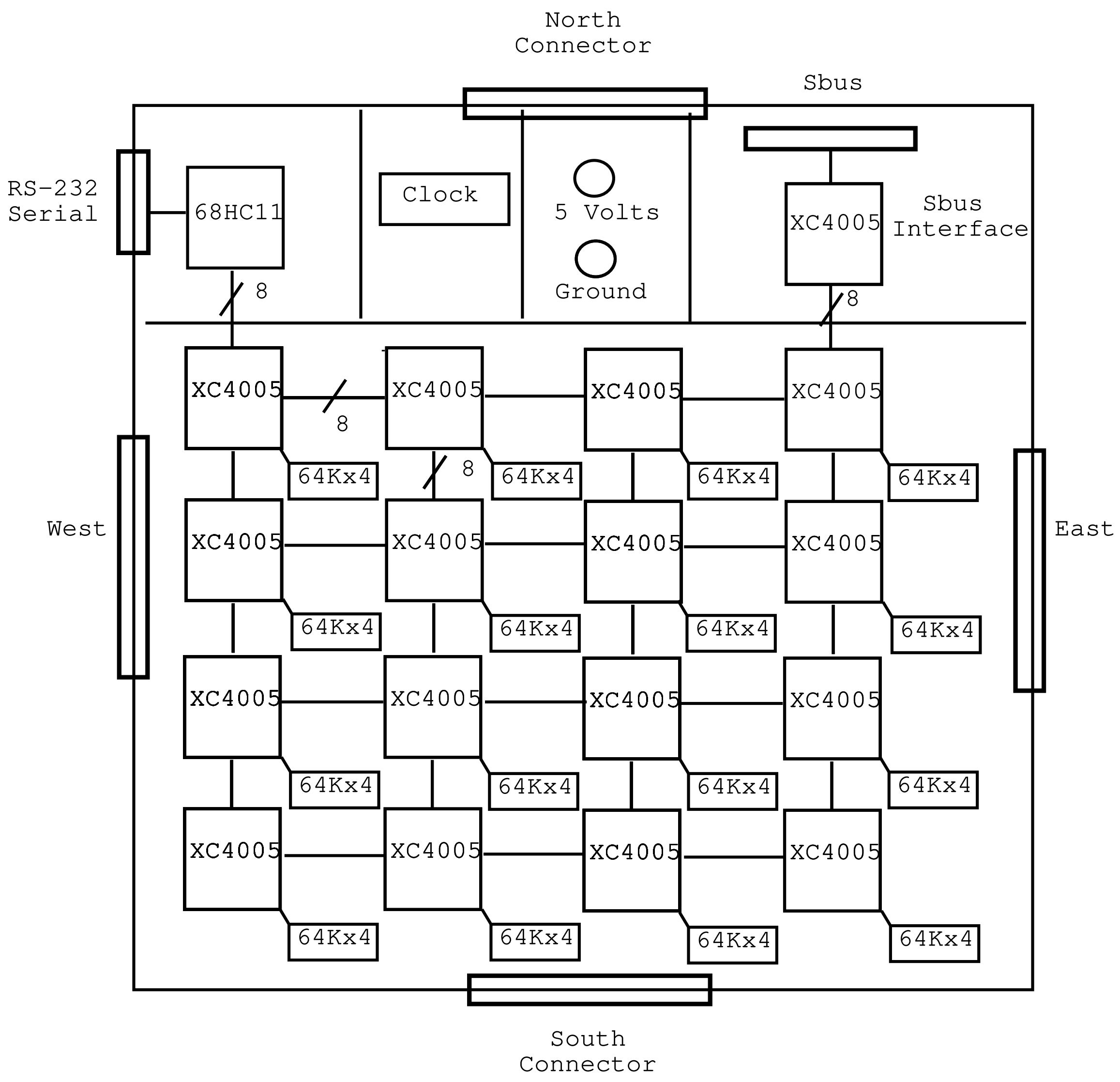

Figure 6-3: Scalable, Low-Cost Emulation Board

simulator with a remote call to the emulation system. Data inputs to the design which otherwise would be evaluated by the simulator are instead passed to the FPGA-based hardware through a host workstation. Results of evaluation in the emulator are subsequently returned to the simulation for graphical display or for use by other portions of the system simulation. This technique of remote access to the emulation system works particularly well for synchronous logic, since evaluation times may be limited to transitions of the system clock. Figure 6-4 shows our current simulation accelerator interface. See [42] and [22] for more detail on the simulation accelerator.

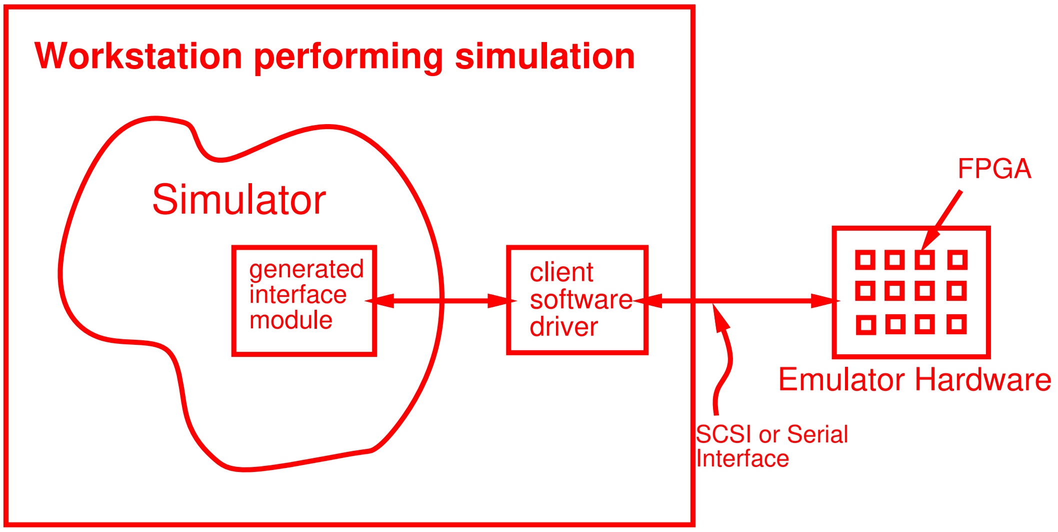

Figure 6-4: Simulation Accelerator Interfaces

Chapter 7

Results

7.1 Overview

We have implemented the entire Virtual Wires software system described in Chapter 5 and the Xilinx-based prototype hardware described in Section 6.2. The results described in this chapter are focused on the front-end of the Virtual Wires software system. This front-end includes all components up to and including the Virtual Router. Since the Route Embedder and FPGA-Specific APR tools are not included, we did not execute our benchmark designs on the hardware for this set of experiments. We used the toolset shown in Table 7.1 for our measurements. For reference, Table 7.2 show the tools which we have upgraded to since these measurements.

Except for the InCA partitioner, which can take hours to optimize a complex design, running times on a SPARC 2 workstation were usually 1 to 15 minutes for each front-end tool. Running times for the remaining tools were also from 1 to 15 minutes; however, the FPGA compiler must be executed for every FPGA partition in the design.

Tool Function	Software Used
Netlist Translator	InCA Translator
Technology Mapper	InCA Libraries
Partitioner	InCA Partitioner
Global Placer	New C program
Dependency Analyzer	Modified lisp parser program
Virtual Router	New C program

Table 7.1: Software Tools used For Experiments

Tool Function	Software Used
Netlist Translator	Synopsys [41]
Technology Mapper	Synopsys FPGA Compiler [41]
Partitioner	InCA Partitioner
Global Placer	New C program
Dependency Analyzer	Modified lisp parser program
Virtual Router	New C program
Route Embedder	New C program
FPGA-Specific APR	Xilinx XACT

Table 7.2: Current Software Tools

Since this analysis does not include the embed stage or the final FPGA-specific stage, we will estimate the FPGA *mapping efficiency*, E_m, and the Virtual Wires gate overhead, VW_i. The FPGA mapping efficiency is the percentage of vendor claimed FPGA gates, N_g^{fpga}, that can actually be used. We could factor this overhead out by normalizing all FPGA device capacity to *usable gates,* $N_g^{fpga} \times E_m$. However, to preserve claimed FPGA gate count numbers, we instead scale up our designs requirements to *mapped gates*:

$$N_g^{circuit} = \frac{N_g^{circuit_{unmapped}}}{E_m} \; mapped\ gates.$$

Of the usable gates, some will be consumed by the Virtual Wires logic. Our estimate for N_G^{vw}, the Virtual Wires overhead, is based on expected control logic, virtual inputs, intermediate hops, and tri-state drivers required to embed the route produced by our Virtual Router.

The first set of experiments analyzes the pin limitations imposed when we attempt to partition our benchmark circuits for current FPGA technologies. Then we estimate the Virtual Wires overhead based on a particular FPGA technology. With this information, we can estimate both the number of FPGAs required and the emulation clock speed when using Virtual Wires. We compare these estimates with the baseline hardwired case. Finally, we analyze the sensitivity of the emulation clock period when the design is bandwidth dominated, and we examine the benefits of a hybrid combination of Virtual Wires with hard wires.

Statistic	Sparcle	A-1000
LSI Gate Count	17,252	85,721
Element Count	4802	37,871
Element Complexity	3.6	2.3
% Sync. Elements	15%	25%
% Comb. Elements	85%	75%
External IO count	129	245

Table 7.3: Design Statitics

7.2 Benchmarks

The benchmarks in this chapter are both ASICs used in the Alewife Machine, a distributed shared memory machine being designed at MIT [2]. Alewife consists of a scalable number of homogeneous processing nodes connected in a 2-dimensional mesh network. Each Alewife node (Figure 7-1) consists of a RISC processor, a floating point coprocessor, a cache, a portion of globally-shared distributed memory, a controller memory management unit (CMMU), and a network switch.

The RISC microprocessor, Sparcle [3] [1], is an 18K gate SPARC processor with some modifications to the basic hardware. These modifications enhance the usefulness of Sparcle as the processing element in a multiprocessor. The CMMU [29] part is an 86K gate cache controller for this distributed shared memory machine being designed at MIT. It implements the shared memory abstraction and provides a message passing interface as well as performing the basic cache controller functionality.

For these experiments we use the production version of the Sparcle netlist, and an early working version of the A-1000 netlist. Both designs have SRAM memory (not included in the total gate count) which we temporarily removed for these measurements. Table 7.3 presents basic statistics of these two designs.

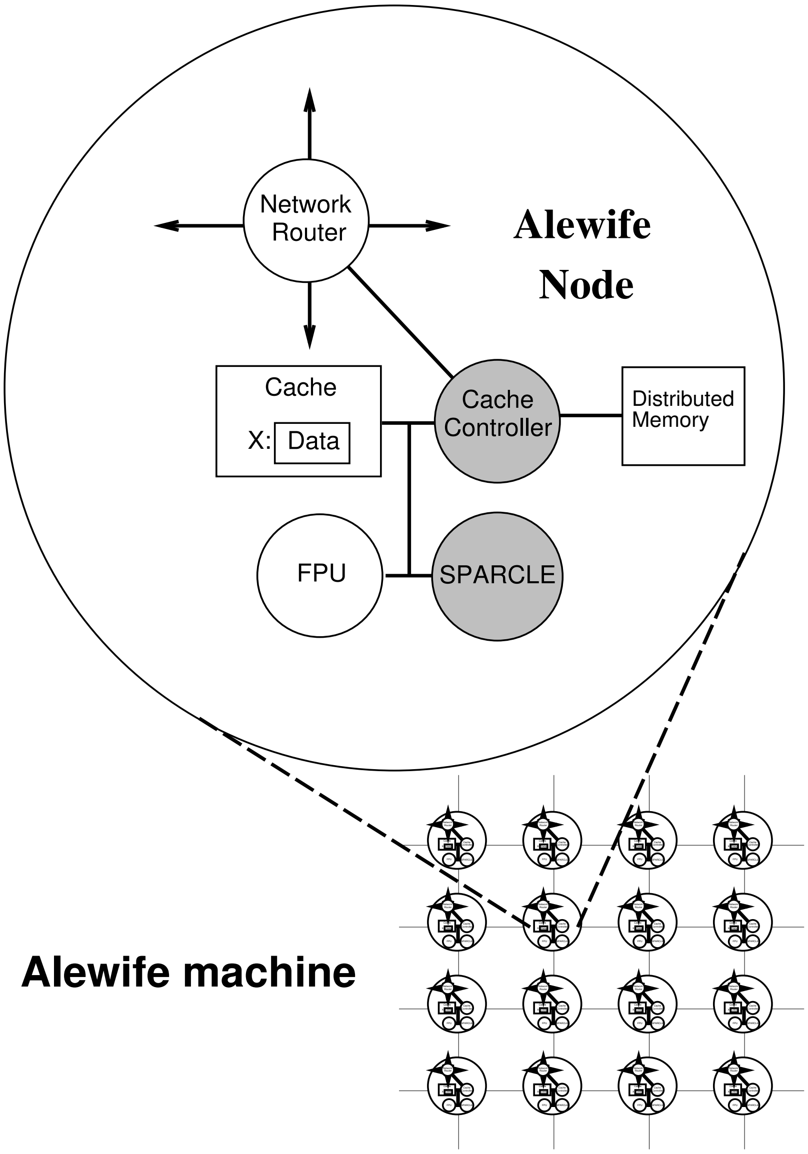

Figure 7-1: Topology of the Alewife Machine and Detail of One Node.

Device	XC4005	XC4010	XC4003H	XC4005H	Cli6002	Cli6005
Gates	5K gates	10K gates	3K gates	5K gates	2K gates	5K gates
Cells	14X14	20X20	30X30	10X10	32X32	56X56
Flip-Flops	616 bits	1,120 bits	200 bits	392 bits	1,024bits	3136 bits
I/Os	112 pins	160 pins	160 pins	192 pins	96 pins	108 pins

Table 7.4: A Few FPGA Device Characteristics

7.3 Target FPGAs

For target FPGAs, we consider the Xilinx 3000 and 4000 series (including the new 4000H series) [46] [47] and the Concurrent Logic Cli6000 series [13]. These are all SRAM-based FPGAs. The Xilinx 3000 series is the previous generation of Xilinx FPGAs, and the Xilinx 4000 series is the latest generation. The largest size FPGA in this series will contain 20,000 claimed gates and 240 external I/O pins. The Xilinx 4000H series is based on the same internal architecture as the 4000 series, except for a higher I/O count.

The Concurrent Logic Cli6000 series contains finer grained programmable logic cells than the Xilinx FPGAs. This series also has a higher proportion of registers than the Xilinx series. Table 7.4 shows some feature of a few of these devices.

For these experiments, we must estimate the mapping efficiency, E_m, for any FPGA we consider. Since E_m is a function of both the FPGA technology and the circuit to be mapped, we can't measure E_m without actually mapping our benchmarks to each FPGA technology. Based on previous experience with these devices, we will use an assumed mapping efficiency of $E_m = 50\%$ for all calculations. Thus each design gate will consume 2 *mapped gates*.

7.4 Pin Limitation Severity

Before compiling the two test designs, we compared their communication requirements to the communication resources of available FPGA technologies. For this comparison, we used the InCA partitioner to partition each design for various mapped gate counts and measured the pin requirements. Figure 7-2 shows the resulting curves, plotted on a log-log scale (note that partition gate count is scaled up to represent a mapping inefficiency of 50%).

Both design curves and the technology curves fit Rent's Rule as discussed in Chapter 3. Table 7.5 shows the resulting Rent equations. For the technology curve, B near 0.5 roughly corresponds to the area versus perimeter for the FPGA die.[1] For the circuit implementations, the lower B, the more locality there is within the circuit. Thus, the A-1000 has more locality than Sparcle, although it has higher total communication requirement.

As Figure 7-2 shows, both Sparcle and the A-1000 will be pin-limited for any choice of FPGA size. In hardwired designs with pin-limited partition sizes, usable gate count is determined solely by available pin resources. For example, a 5000 gate FPGA with 100 pins can only utilize 1000 Sparcle gates or 250 A-1000 gates.

Parameter	FPGA Technology	High I/O FPGAs	Sparcle	A-1000
B	0.57	0.60	0.62	0.45
K_P	0.84	0.93	1.3	8.4

Table 7.5: Rent's Rule Parameters (slope, offset of log-log curve)

[1] If we include all pins: the configuration programming I/Os, JTAG, and power and grounds, then B is almost exactly 0.5 for the FPGA devices.

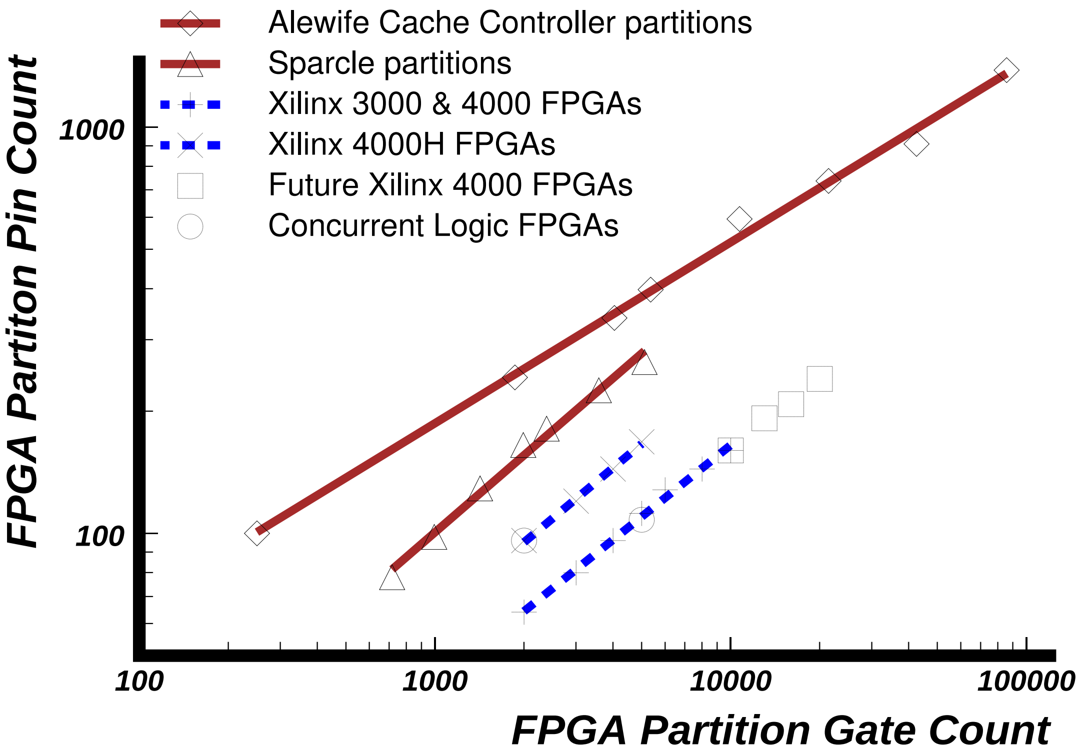

Figure 7-2: Pin Count as a Function of FPGA Partition Size

7.5 Virtual Wires Overhead

In the following analysis, we estimate the FPGA gate costs of Virtual Wires based on the architecture described in Section 6.1. In this architecture, gate overhead is consumed by phase control logic, shift loops, and intermediate hops. Each shift loops is composed of a number of shift register bits combined with one tri-state driver. Thus for FPGA partition i, the Virtual Wires gate overhead, N_g^{vw}, consist of:

1. **Phase Control Logic**: One state machine.
2. **Inputs:** I_i shift register bits for partition inputs.
3. **Hops:** H_i shift register bits for intermediate hops.
4. **Muxing:** M_{i_p} tri-state drivers to multiplex the shift loops on each pin p.

Note that we do not count partition output shift register bits because the storage of logical outputs can be overlapped with logic inputs.

Assuming the state machine takes C_p gates and the shift register bits and tri-state drivers each take C_s gates, the Virtual Wires Gate overhead for partition i is then:

$$(N_g^{vw})_i = C_p + C_s \times \left(I_i \ + \ H_i \ + \ \sum_p M_{i_p} \right) \ \textit{mapped gates.} \tag{7.1}$$

For our empirical estimates, we use the actual number of logical inputs, I_i, for each partition as determined by our partitioning tool. When routing in a mesh or torus, intermediate hops cost one shift register bit per hop. The number of tri-state drivers needed per physical wire is equal to the number of shift loops connected to that physical wire. We use our Virtual Router to determine L_i for each partition and M_{pi} for each partition pin.

For numerical comparisons in the remaining analysis, we use estimated FPGA gate costs based on the Concurrent Logic CLI6000 series FPGA. We estimate the phase control logic will be $C_p \approx 300$ mapped gates based on hand design of a few sample state machine. In the Cli6000, both a single-bit shift register and a tri-state driver take 1 of 3136 cells in the 5K gate part, which implies that $C_s \approx 3$ mapped gates. Table 7.6 shows these costs along with our assumed gate and pin count, N_g^{fpga} and N_p^{fpga}.

Phase control logic	$C_p \approx 300$ *mapped gates*
Shift register bit or tri-state driver	$C_s \approx 3$ *mapped gates*
FPGA Gate Count	$N_g^{fpga} \approx 5000$ *mapped gates*
FPGA Pin Count	$N_p^{fpga} \approx 100$ *pins*

Table 7.6: Parameters for Empirical Analysis

7.6 Number of Component Comparison

To determine empirically the number of components needed for our benchmark designs, we compiled the netlists with the front-end of the Virtual Wires compiler and estimate the mapping efficiency and Virtual Wires overhead. We compiled both designs for a two dimensional torus and for a full crossbar interconnect of 5000 mapped gate, 100 pin FPGAs, assuming a 50 percent mapping efficiency. Virtual Wires overhead estimation is discussed in Section 7.5. Table 7.7 shows the results for both hard wires and Virtual Wires. Compiling the A-1000 to a torus, hardwires only, was not practical with our partitioning software. The gate utilizations obtained for the hardwired cases agree with reports in the literature [25] [44] on designs of similar complexity.

Design	*Hardwires Only*		*Virtual Wires Only*	
	2-D Torus	Full Crossbar	2-D Torus	Full Crossbar
Sparcle (18K gates)	>100 (<7%)	31 (23%)	9 (80%)	9 (80%)
A-1000 (86K gates)	Not Practical	>400 (<10%)	49 (71%)	42 (83%)

Number of FPGAs (Mapped Gate Utilization)

Table 7.7: Required number of 5K Gate, 100 Pin, 50% mapping efficient FPGAs

For the crossbar, the use of Virtual Wires can decrease the number of FPGA devices needed by a factor of 3 for Sparcle and a factor of 10 for the A-1000. This decrease will results in a direct saving in emulation costs.

The torus topology is impractical for the hardwired case, costing over 300% due to increased pin limitations for Sparcle and making the A-1000 circuit practically impossible to emulate. However, when using Virtual Wires, the extra FPGAs needed to implement intermediate hops did not increase the chip count for Sparcle, and only increased the chip count by 17% for the A-1000.

The gate utilization percentages in Table 7.7 reflect the average number of mapped gates available to each design's FPGA partition. In the hardwired case, the utilization is low because of pin limitations, while in the Virtual Wires case this utilization is less than 100% because of the Virtual Wires gate overhead in each partition. Chapter 8 provides further analysis of these overheads.

7.7 Emulation Speed Comparison

Emulation clock cycle time T_E is determined by:

- Communication delay per hop, t_c: the time required to transmit a single bit on a wire between a pair of FPGAs.

T_E	Emulation clock period
t_c	Inter-FPGA wire delay
L	Longest dependency path
T_L	Total critical path delay
N	Total pipeline cycles
D	Network diameter

Table 7.8: Parameter's Used for Speed Estimation

- Length of longest path in dependency graph, L, in terms of number of FPGA partitions.
- Total FPGA gate delay along the longest path T_L, which is the sum of the FPGA partition delays in the longest path (not counting communication time).
- Sum of pipeline cycles across all phases, N.
- Network diameter, D ($D = 1$ for a crossbar).

These parameter are summarized in Table 7.8.

The emulation clock period is directly related to the number of phases in an emulation clock, and the number of pipeline clocks within each of the phases. The total number of phases in an emulation clock is at least equal to the largest number of partitions through which a combinatorial path passes, L. The number of pipeline cycles in each phase is directly related to network diameter and/or physical wire contention.

If the emulation is *latency dominated*, then the minimal number of phases will be exactly L, and the pipeline cycles per phase should be no greater than D, giving:

$$N = L \times D.$$

The upper bound of the network diameter, D, is imposed by the worst case number of intermediate hops.

On the other hand, if the emulation is *bandwidth dominated*, then the total pipeline cycles (summed over all phases) will be at least:

$$N = MAX_i \left(\frac{V_i}{P_i} \right),$$

where V_i and P_i are the number of virtual and physical wires respectively for FPGA partition i. We call the ratio V_i/P_i the *pin multiplication factor*, PMF_i. If there are hot spots in the network (not possible with a crossbar), the bandwidth dominated delay will be higher due to the extra contention.

Emulation speeds for Sparcle and the A-1000 were both latency dominated. Using the worst case number of pipeline cycles, N, we compare estimated execution speeds for the Virtual Wires and hardwired cases. Without performing timing analysis on the underlying circuit, we can estimate emulation speed by considering a *computation only* delay component, T_{EP}, and a *communication only* delay component T_{EC}. This dichotomy is defined as follows.

Computation only delay:

$$T_{EP}^{hw} = T_L$$

$$T_{EP}^{vw} = T_L + t_c \times N$$

The computation-only bound assumes that communication time between chips is negligible. For the Virtual Wires case we add in a component equal to $t_c \times N$ to reflect the extra cost of multiplexing at every chip boundary.

Communication only delay:

$$T_{EC}^{hw} = T_{EC}^{vw} = t_c \times N.$$

This delay is the inter-chip delay multiplied by the total number of pipeline cycles. For the hardwired case, N is not the number of pipeline cycles, as we earlier defined it, but the total number of FPGAs along the critical path (including intermediate hops). Since we are clocking the FPGA at the maximum clock rate for the Virtual Wires case, then t_c will also be the period of the pipeline clock.[2]

We combine these two components to estimated the emulation clock period:

$$T_E^{hw} = T_L + t_c \times N \tag{7.2}$$

$$T_E^{vw} = T_L + 2 \times t_c \times N. \tag{7.3}$$

Based on CLi6000 specifications we assumed that $t_c = 20ns$ (a $50MHZ$ pipeline clock). Based on a $40MhZ$ design clock rate and an FPGA slowdown factor of 10 (the ratio of our ASIC technology to the CLi6000 technology), we assume that $T_L = 250ns$. From our compilation of both designs, we determined N. Table 7.9 shows the resulting emulation speeds for Virtual Wires and hardwires for the crossbar topology. Note that we have made the conservative assumption that T_L for Virtual Wires remains the same as that for hardwires, even though Virtual Wires yields fewer partitions. Since the use of Virtual Wires allows a design to be partitioned across fewer FPGAs, N is decreased, decreasing T_{EC}. However, the pipeline stages will increase T_{EP} proportional to t_c per pipeline cycle. Table 7.10 shows the resulting worst case speed for Virtual Wires on the torus topology. Virtual Wire torus speeds are slower by a factor of D, the network diameter.

[2]The extra delay introduced by the multiplexor (assumed to be t_c here, but is typically much smaller) will actually increase the clock period to 2 t_c; however, in a bandwidth limited design we may choose the alternative of adding an extra pipeline stage, increasing the number of pipeline cycles per hop to 2 pipeline clocks, but allowing maximum inter-FPGA bandwidth utilization. Ignoring any added register delays, the factor of 2 in Equation 7.3 accounts for either case.

		Hardwire Only	Virtual Wire Only
Sparcle	Longest path, L	9 partitions	6 partitions
	Network diameter, D	1 hop	1 hop
	Pipeline cycles, N	9 cycles	6 cycles
	Computation delay, T_{EP}	250 ns (58%)	370 ns (76%)
	Communication delay, T_{EC}	180 ns (42%)	120 ns (24%)
	Total delay, T_E	430 ns (100%)	490 ns (100%)
	Estimated emulation clock, $1/T_E$	**2.3 MHz**	**2.0 MHz**
A-1000	Longest path, L	27 partitions	17 partitions
	Network diameter, D	1 hop	1 hop
	Pipeline cycles, N	27 cycles	17 cycles
	Computation only delay, T_{EP}	250 ns (32%)	590 ns (63%)
	Communication only delay, T_{EC}	540 ns (68%)	340 ns (37%)
	Total delay, T_E	790 ns (100%)	930 ns (100%)
	Estimated emulation clock, $1/T_E$	**1.3 MHz**	**1.1 MHz**

Table 7.9: Emulation Clock Speed Estimate (Crossbar Topology)

		Virtual Wire Only
Sparcle	Longest path, L	6 partitions
	Network diameter, D	2 hops
	Pipeline cycles, N	12 cycles
	Computation delay, T_{EP}	370 ns (61%)
	Communication delay, T_{EC}	240 ns (39%)
	Total delay, T_E	610 ns (100%)
	Estimated emulation clock, $1/T_E$	**1.6 MHz**
A-1000	Longest path, L	17 hops
	Network diameter, D	6 hops
	Pipeline cycles, N	102 cycles
	Computation only delay, T_{EP}	590 ns (22%)
	Communication only delay, T_{EC}	2040 ns (78%)
	Total delay, T_E	2630 ns (100%)
	Estimated emulation clock, $1/T_E$	**0.38 MHz**

Table 7.10: Emulation Clock Speed Estimate (Torus Topology)

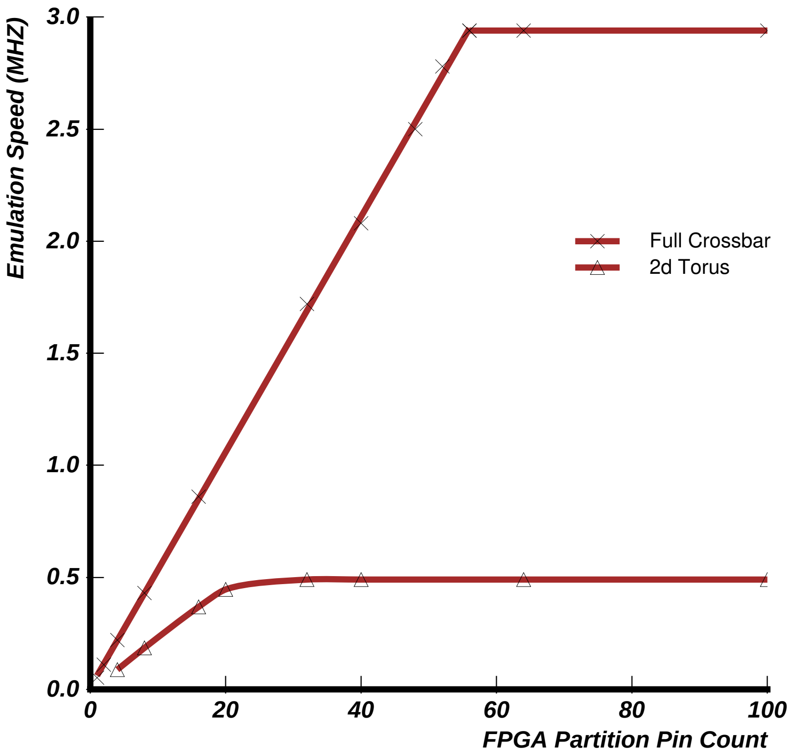

Figure 7-3: A-1000 Emulation Speed (Communication only Component)

7.8 Bandwidth Requirements

In Table 7.9, the Virtual Wire emulation clock was determined solely by the length of the longest path; the emulation clock period was latency dominated. In order to determine what happens when bandwidth limitations dominate, we reduced the pin count and recorded the resulting emulation clock (based on T_{EC} only) for both a crossbar and torus topology. Since the Virtual Router can use an arbitrary FPGA topology file, we can generate these data points by varying the FPGA pin counts in the topology file and re-running the Virtual Router. Using the resulting total pipeline cycles, N, we can then estimate the emulation clock. Figure 7-3 shows the results for the A-1000.

Emulation speeds switch from bandwidth dominated to latency dominated at the knee of the curves in Figure 7-3. The torus is slower because it has a larger diameter, D. However, it moves out of the latency dominated region sooner because it exploits locality; several short wires can be routed during the time of a single long wire. This analysis has assumed that the crossbar can be clocked as fast as the torus; the increase in emulation speed obtained with the crossbar is lower if t_c is adjusted accordingly.

Circuit	Hard Wires Only	Virtual Wires Only	Hybrid
Sparcle	9 hops	6 hops	3 hops
A-1000	27 hops	17 hops	15 hops

Table 7.11: Reduction Of Critical Path with Hybrid Wiring

7.9 Combination with Hard Wires

With Virtual Wires, neither design was bandwidth limited, but rather limited by its respective critical paths. As shown in Figure 7-3, the A-1000 only needs about 20 pins per FPGA to run at the maximum emulation frequency. While this allows the use of lower pin count (and thus cheaper) FPGAs, another option is to trade this surplus bandwidth for speed. This tradeoff is accomplished by *hardwiring* logical wires at both ends of the critical paths. Critical wires can be hardwired until there is no more surplus bandwidth, thus fully utilizing both gate and pin resources. We added this feature to the Virtual Router to obtain the results shown in Table 7.11. Besides the added feature in the hybrid case, all other steps are the same as those used for the previous measurements. This hybrid combination reduces the longest virtual path in Sparcle by 50% and in Alewife by 12%. Since part of the path is now hardwired, there will be a corresponding speedup of the emulation clock cycle and reduction in Virtual Wires overhead.

Chapter 8

Analysis

In this chapter we derive theoretical gate utilization for logic emulation with and without Virtual Wires. Much of our analysis relies on Rent's Rule, as introduced in Chapter 3. For the hardwired case, we derive relative Rent's rule factors, which we call *Rent ratios*. We also derive a *topological factor* which explains why a mesh topology is not effective without Virtual Wires. For the Virtual Wires case, we mathematically determine the optimal FPGA partition size. In conclusion, we show how the derived Virtual Wire utilization scales with increasing FPGA device size, while hardwired utilization may not.

8.1 Hard Wires Gate Utilization

For circuits which obey Rent's rule, we can determine the overhead for hardwires, under pin-limited conditions (Figure 8-1). Given pin limitations, the number of FPGA pins dictates the number of FPGA partition pins available for the circuit:[1]

$$N_P^{circuit} = \frac{1}{\overline{d}} \times N_P^{fpga} \; pins. \tag{8.1}$$

[1] For this analysis, we work with average partition pin and gate requirements. Pin limitation effects are worse when the circuit is non-uniform.

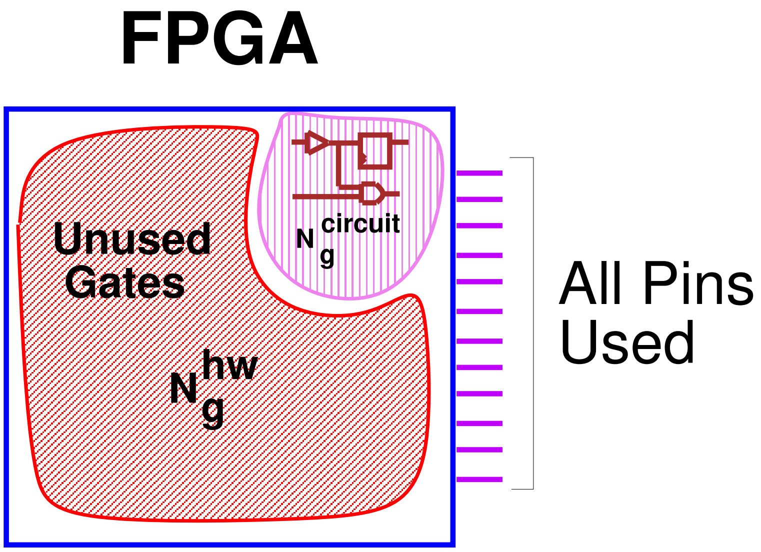

Figure 8-1: Gate Utilization without Virtual Wires

In Equation 8.1, $\overline{d}$ is the average number of FPGA hops for each wire. This factor accounts for pins consumed by intermediate hop routing. We next substitute Rent's equation for both sides of Equation 8.1:

$$K_P^{circuit}(N_g^{circuit})^{B^{circuit}} = \frac{1}{\overline{d}} K_P^{fpga}(N_G^{fpga})^{B^{fpga}} \quad pins. \tag{8.2}$$

Solving for $N_G^{circuit}$ yield the predicted gates available to each FPGA partition:

$$N_g^{circuit} = \left(\frac{K_P^{fpga}}{\overline{d} K_P^{circuit}} (N_g^{fpga})^{B^{fpga}} \right)^{1/B^{circuit}} \quad mapped\ gates. \tag{8.3}$$

8.1.1 Rent Ratios

We can simplify Equation 8.3 by defining *Rent ratios* for mapping a circuit on to an FPGA, $K^{f \leftarrow c}$ and $B^{f \leftarrow c}$. We also need a *topological factor*, $\overline{d}^{f \leftarrow c}$, for multiple hops in the network.[2]

[2] Note that $\overline{d}^{f \leftarrow c} = 1$ for a crossbar topology, and can be expressed purely as a function of $N_g^{circuit}$, $N_c^{circuit}$ (number of chips), and $B^{circuit}$ for a mesh topology, as we shall we in Equation 8.11.

$K_P^{f\leftarrow c} = \left(\frac{K_P^{fpga}}{K_P^{circuit}}\right)^{1/B^{circuit}}$
$B^{f\leftarrow c} = \frac{B^{fpga}}{B^{circuit}}$
$\overline{d}^{f\leftarrow c} = (\overline{d})^{1/B^{circuit}}$

Table 8.1: Rent Ratios and Topological Factor

Substitution of these newly defined Rent ratios and topological factor, summarized in Table 8.1, into Equation 8.3 yields

$$N_g^{circuit} = \frac{1}{\overline{d}^{f\leftarrow c}} K_P^{f\leftarrow c} (N_g^{fpga})^{B^{f\leftarrow c}} \ mapped\ gates. \tag{8.4}$$

The average pin-limited hardwire overhead per FPGA partition is then

$$\overline{N_g^{hw}} = N_g^{fpga} - \frac{1}{\overline{d}^{f\leftarrow c}} K_P^{f\leftarrow c} (N_g^{fpga})^{B^{f\leftarrow c}} \ mapped\ gates, \tag{8.5}$$

and the FPGA utilization with hardwires is

$$U_{hw} = \frac{1}{\overline{d}^{f\leftarrow c}} K_P^{f\leftarrow c} (N_g^{fpga})^{B^{f\leftarrow c}-1} \times 100\ percent. \tag{8.6}$$

Here we can see the significance of $B^{f\leftarrow c}$: if $B^{f\leftarrow c} < 1$ FPGA utilization will decline with increasing FPGA device size. Utilization is directly proportional to the ratio $K^{f\leftarrow c}$ for a fixed device size, and inversely proportional to the topological factor $\overline{d}^{f\leftarrow c}$.

8.1.2 Topological Factor

To see why a non-crossbar topology is so difficult for the hardwired case, we simplify Equation 8.6 by assuming that the circuit is balanced for the FPGA device. That is, $B^{f\leftarrow c} = 1$ and $K_P^{f\leftarrow c} = 1$:

$$U_{hw}^{balanced} = \frac{1}{\overline{d}^{f\leftarrow c}} \times 100\ percent. \tag{8.7}$$

Equation 8.7 captures the effect of multiple hop overhead. For example, if $B^{circuit} = 0.5$, we can express this component directly in terms of the average wire length:

$$U_{hw}^{balanced} = (\frac{1}{\overline{d}})^2 \times 100 \ percent. \tag{8.8}$$

For a crossbar, $\overline{d} = 1$; there will be no pin-limitations and thus a hardwired emulation can achieve 100% utilization. However, for any other topology there will be serious degradation. For example, in a 4X4 mesh topology with $N_c^{circuit} = 16$ and $B^{circuit} = 0.6$, Rent's rule predicts $\overline{d} \approx 2$ *hops*. This average distance drops utilization to $U_{hw}^{balanced} \approx 30$ percent! As another example, if Quickturn were to mesh connect all 528 FPGAs in a 330K gate Enterprise Emulation System [33], with the same $B^{circuit} = 0.6$ design, a predicted $\overline{d} \approx 4$ *hops* would drop utilization to $U_{hw}^{balanced} \approx 10$ percent of their current crossbar topology's efficiency.[3]

8.2 Virtual Wires Gate Utilization

If we consider the entire set of partitions, we can express the *average Virtual Wires gate overhead*, $\overline{N_g^{vw}}$ (Figure 8-2), in terms of the virtual logic costs, C_p and C_s (Section 7.5), and the average number of virtual inputs, bits per shift register, and hops per Virtual Wire: $\overline{I}$, $\overline{S}$, and $\overline{d}$:

$$\overline{N_g^{vw}} = C_p + C_s \times \overline{I} + (1 + (\overline{d} - 1) + \overline{d}/\overline{S}) \ mapped \ gates. \tag{8.9}$$

Each Virtual Wire consumes an average of C_s gates for the initial input register, $C_s \times (\overline{d} - 1)$ for intermediate hops, and $\overline{d}/\overline{S}$ gates for each multiplexor it passes through. For $S_{avg} \gg 1$ (many bits per shift register):

$$\overline{N_g^{vw-avg}} = C_p + C_s \times \overline{I} \times \overline{d} \ mapped \ gates. \tag{8.10}$$

[3] Assuming their partial crossbar can achieve the same efficiency as a full crossbar.

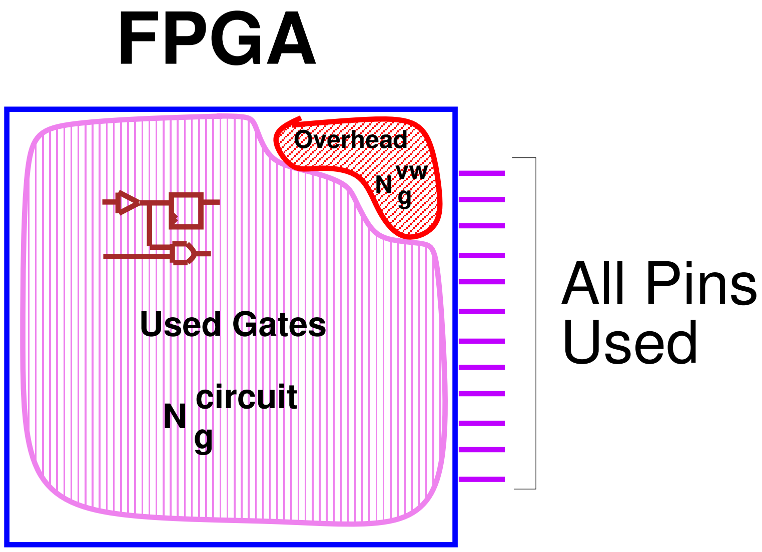

Figure 8-2: Gate Utilization with Virtual Wires

In this case the overhead is directly proportional to the average number of inputs per partition and to the average wire length.

For mapping a circuit which obey Rent's rule onto a mesh topology, Equation 8.10 can be expressed as:

$$\overline{N_g^{vw}} = C_p + C_s \times \overbrace{\frac{1}{2}\left(K_P N_g^B\right)}^{\bar{I}} \times \overbrace{\frac{2}{9}\left(7\frac{N_c^{B-0.5}-1}{4^{B-0.5}-1} - \frac{1-N_c^{B-1.5}}{1-4^{B-1.5}}\right)\frac{1-4^{B-1}}{1-N_c^{B-1}}}^{\bar{d}} \ \textit{mapped gates}. \tag{8.11}$$

Where $N_C = N_g^{total}/N_g^{circuit}$, the total number of FPGAs needed for a design of size N_g^{total}, and K_P and B are for the circuit. The term for $\bar{I}$ is Rent's rule for the circuit with a factor of 1/2 to allow for overlapping inputs with outputs. The term for $\bar{d}$ is an upper limit to the average wire length, in Manhattan hops, which can also be derived from Rent's rule [7]. Thus given Rent's parameters, $K_p^{circuit}$ and $B^{circuit}$, for a given design, the Virtual Wires overhead, N_g^{vw}, can be expressed strictly in terms of design and partition sizes.

For a crossbar toplogy, $\overline{d} = 1$, and Equation 8.10 reduces to:

$$\overline{N_g^{vw}} = C_p + C_s \times \overline{I} \ mapped\ gates \tag{8.12}$$

Or, in terms of the Rent equation:

$$\overline{N_g^{vw}} = C_p + C_s \times \frac{1}{2} K_P^{circuit} (N_g^{circuit})^{B^{circuit}} \ mapped\ gates. \tag{8.13}$$

Given that Rent's Rule holds, we can use the general Virtual Wires equation, Equation 8.9, to estimate the number of mapped gates available to each FPGA partition:

$$N_g^{circuit} = N_g^{fpga} - \left(C_p + C_s \times \frac{1}{2}\overline{d}\left(1 + \frac{1}{\overline{S}}\right) K_P^{circuit} (N_g^{circuit})^{B^{circuit}} \right) \ mapped\ gates. \tag{8.14}$$

Solving Equation 8.14 for $N_g^{circuit}$ yields the optimal Virtual Wires FPGA partition size, $N_g^{circuit*}$, for a particular circuit. Why is the equation not a function of N_p^{fpga}? As long as there is at least one available pin, we can multiplex that one pin as often as we need.[4] To get utilization, we divide this optimal partition size by the FPGA device size:

$$U_{vw} = \frac{N_g^{circuit*}}{N_g^{fpga}} \times 100 \ percent, \tag{8.15}$$

8.2.1 Optimal Partition Size

The optimal partition size, $N_g^{circuit*}$, is the largest design partition size which will fit in the FPGA device. By comparing the optimal partition size for both the hardwired and the Virtual Wires case, we illustrate the tradeoffs allowed by Virtual Wires. For the hardwired case, we use N_g^{fpga} and N_p^{fpga}, the maximum gate and pin count of the device, to define a feasible region for each partition. For the Virtual Wires case, we

[4] In fact, in our initial demonstration of Virtual Wires we successfully executed circuits on a 9-node wire-wrapped board with only one wire connecting nearest neighbors.

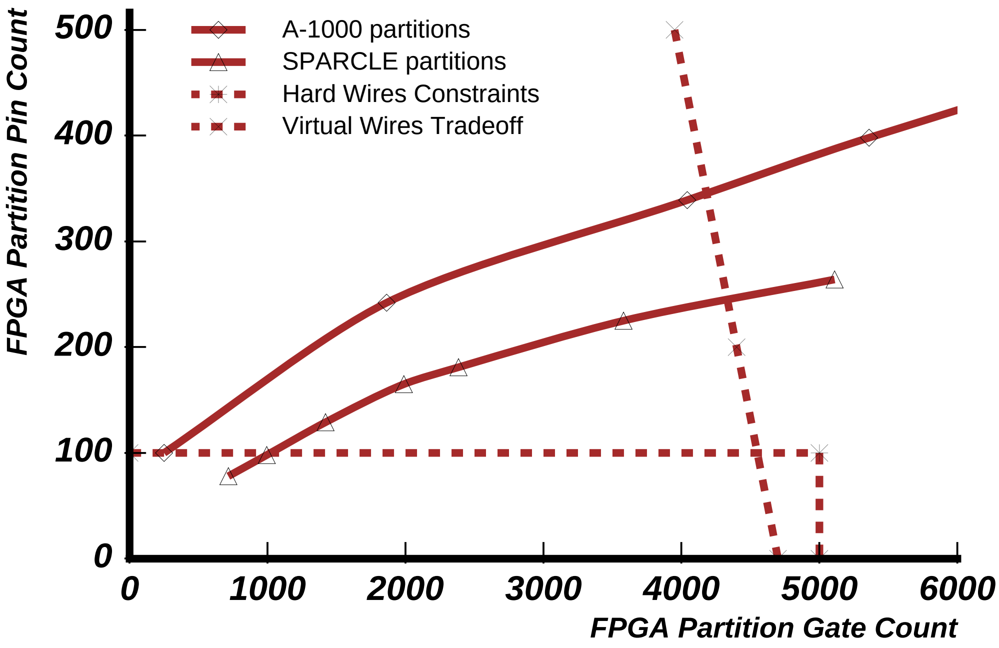

Figure 8-3: Determination of Optimal Partition Size

must consider the tradeoff between gates and pins as the FPGA partitions pin count increases beyond the number of available pins on the FPGA device. Assuming $\overline{d} = 1$ and $\overline{S} \gg 1$, we evaluate Equation 8.12 with our FPGA numbers from Table 7.6 to determine the Virtual Wires pin/gate constraint curve. This curve is for a 5K gate, 100 pin FPGA, with $C_p = 300$ mapped gates and $C_s = 3$ mapped gates.

Figure 8-3 shows the both the hard wires pin/gate constraint and the Virtual Wires pin/gate constraint curve plotted against the partition curves for our benchmark designs (Section 7.2). The regions enclosed by the axes and the constraint curves represents feasible regions in the design space. The hard wires constraint region corresponds to the area where FPGA partition pin and gate requirements are less than the FPGA constraints. On the other hand, the Virtual Wires curve corresponds to the area where the combined FPGA partition gate count and Virtual Wire overhead gate count is less than the FPGA gate count. The intersection of the partition curves and the wire curves gives the optimal partition and sizes. Notice how Virtual Wires add the flexibility of trading gate resources for pin resources.

8.3 Scalability

To summarize this analysis, Figure 8-4 compares FPGA device utilization for Virtual Wires and hardwires on both a 4X4 mesh and a 16 chip crossbar topology for increasing FPGA device size. This graph is on a semi-log scale, with the y-axis measuring percent of usable gates and the x-axis logarithmically measuring FPGA device size. Table 8.2 shows the assumed parameter for this graph. Note that since the FPGA device size is increasing, with a constant number of devices, we are increasing the circuit size as well. Also, the FPGA pin count is increasing with the gate count in accordance with Rent's rule.

The slope of both hardwired curves is $B^{f \leftarrow c}$. Thus if $B^{f \leftarrow c} > 1$ this curve would slope upwards instead. The log-intercept is similarly $K_P^{f \leftarrow c}$ for the hardwires crossbar. For the hardwired mesh, the offset is lower than the crossbar by $1/\overline{d}^{f \leftarrow c}$ in accordance with Equation 8.6. For this example, the FPGA pin count decreases with decreasing gate count such that the FPGA can never be fully utilized in the hardwired case.

For Virtual Wires, the utilization is low for small gate count due to the constant factor of $C_p = 300$ control logic overhead. However, with increasing circuit size, this overhead is rapidly diluted : the Virtual Wires gate utilization approach 100% with increasing FPGA size. For a mesh toplogy, the utilization increases more slowly, however it too asymptotically approaches 100%.[5]

[5] By working in terms of mapped gates, this analysis hides the associated decrease in mapping efficiency, E_m, when we increase FPGA device size and $B^{f \leftarrow c} < 1$ for the internal FPGA wiring. This problem is due to hardwires *within* the FPGA device itself! If we overcome this problem by using Virtual Wires for routing intra-FPGA signals, then Figure 8-4 will hold.

Parameter	Value
$B^{circuit}$	0.60
B^{fpga}	0.55
$K_P^{circuit}$	2.0
K_P^{fpga}	1.0
$\overline{d}^{\,crossbar}$	1 hop
$\overline{d}^{\,mesh}$	2 hops
C_p	300 mapped gates
C_s	3 mapped gates

Table 8.2: Parameters for Scalability Comparison

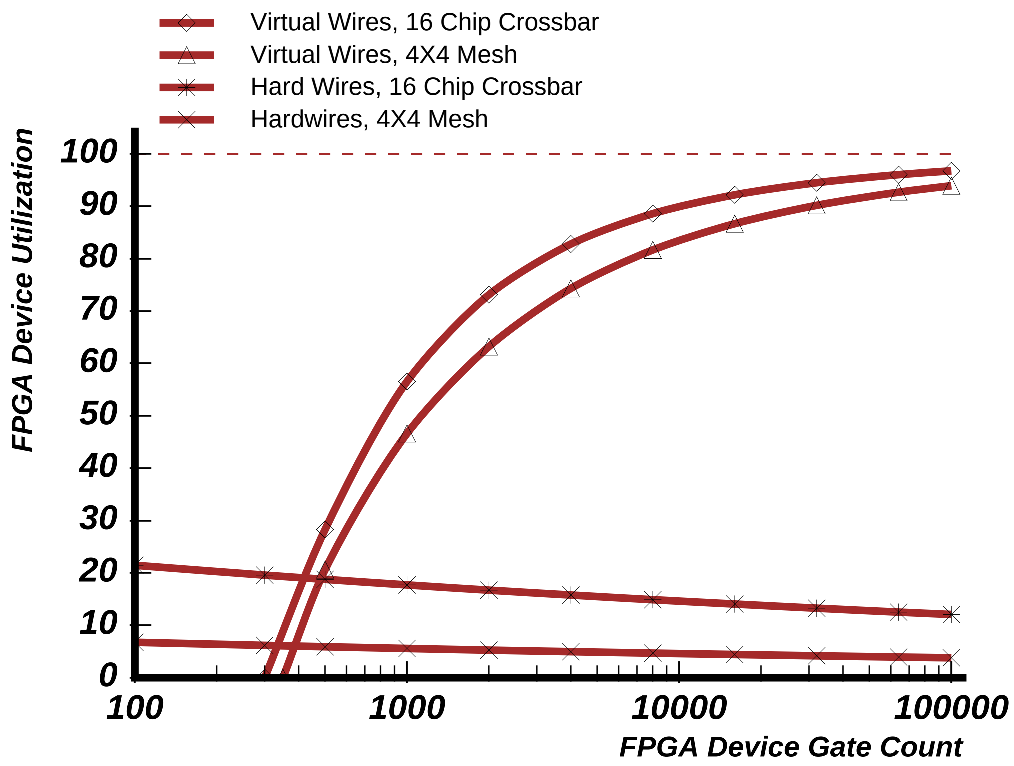

Figure 8-4: Scalability with FPGA Device Size

Chapter 9

Conclusions and Future Research

9.1 Conclusions

This thesis has presented detailed evaluation of the Virtual Wires compilation technique for overcoming pin limitations in FPGA-based logic emulations. We have described the software portion of a project at MIT to produce a scalable, low cost FPGA-based emulation system which maximizes FPGA resource utilization.

Careful analysis of existing logic partitioning schemes for hardwired logic emulation revealed that the pin limitations of current FPGA technology cause emulation resource efficiency to be as low as 10 to 20 percent. As FPGA technologies scale, this efficiency will decrease without the use of Virtual Wires (or some similar technique). This efficiency is low because of pin limitations; however, we found that in the hardwired scheme most of the usable off-chip bandwidth is wasted.

The Virtual Wires technique is independent of topology. Investigation of emulation on both a torus and a full crossbar reveal some interesting results: Virtual Wires allows the use of the less complex topologies, the torus, in cases where only the crossbar topology was practical before.

The Virtual Wires concept has been successfully implemented. Construction of a robust Virtual Wires compiler and the hardware to support such a system is nearing completion. Preliminary results from this system show that the use of Virtual Wires allows maximum utilization of FPGA gate resources, at estimated emulation speeds competitive with existing hardwired techniques. While this thesis has focused on using Virtual Wires for improving performance in FPGA-based logic emulation systems, the Virtual Wires concept is also applicable to other types of FPGA-based systems.

9.2 Future Research

This project has uncovered several possible areas for future research, in both the context of FPGA-based logic emulation with Virtual Wires, high-level digital logic synthesis, and FPGA computing.

Virtual Wires Research Group Directions

In the Virtual Wires research group at MIT, we are implementing a complete emulation system based on Virtual Wires, including the Virtual Wires compiler, simulator interfaces, and hardware prototype. By utilizing the pin multiplication effect of virtual wires in combination with a low dimensional interconnect, we are producing a logic emulation system capable of simulating/emulating 30K gates for less than $3000. By using this compiler and the scalability of our current system, we plan to emulate circuits in the 100K+ gate range.

We are continuing with the design of an automatic memory compiler – in our initial system, capable of emulating up to 1 Megabit of SRAM, the user must still hand-partition ASIC memory. With a high speed Sbus host interface, a tightly coupled interface between a software simulator and the emulation system will be used for *cosimulation.* We are also exploring using the Virtual Wires Prototype board as a general purpose computing engine. This will involve compiling a parallel language, such as parallel C, directly into the Virtual Wires hardware.

Virtual Wires Improvements

Using timing and/or locality sensitive partitioning with Virtual Wires has potential for reducing the required number of routing sub-cycles. Communication bandwidth can be further increased with *pipeline compaction*, a technique for overlapping the start and end of long virtual paths with shorter paths traveling in the same direction. A more robust *dataflow* implementation of Virtual Wires replaces the global barrier imposed by routing phases with a finer granularity of communication scheduling, possibility overlapping computation and communication as well.

Logic Synthesis Applications of Virtual Wires

Further logic resource improvements can be obtained if we apply Virtual Wires to high-level digital circuit synthesis. Using the information gained from dependency analysis, we can now predict which portions of the design are active during which parts of the synchronous clock cycle. If new FPGA devices support fast partial reconfiguration, this information can be used to implement *virtual logic* via invocation of hardware subroutines [23]. Even without fast reconfiguration, we can use Virtual Wires to automate combinational logic *resource sharing*. Because of the long routing delays and high register count in FPGAs, Virtual Wires could also be used to implement a form of *wave pipelining* in these shared resources. For multiple chip ASIC design, Virtual Wires can be used to automate the inter-chip interface design.

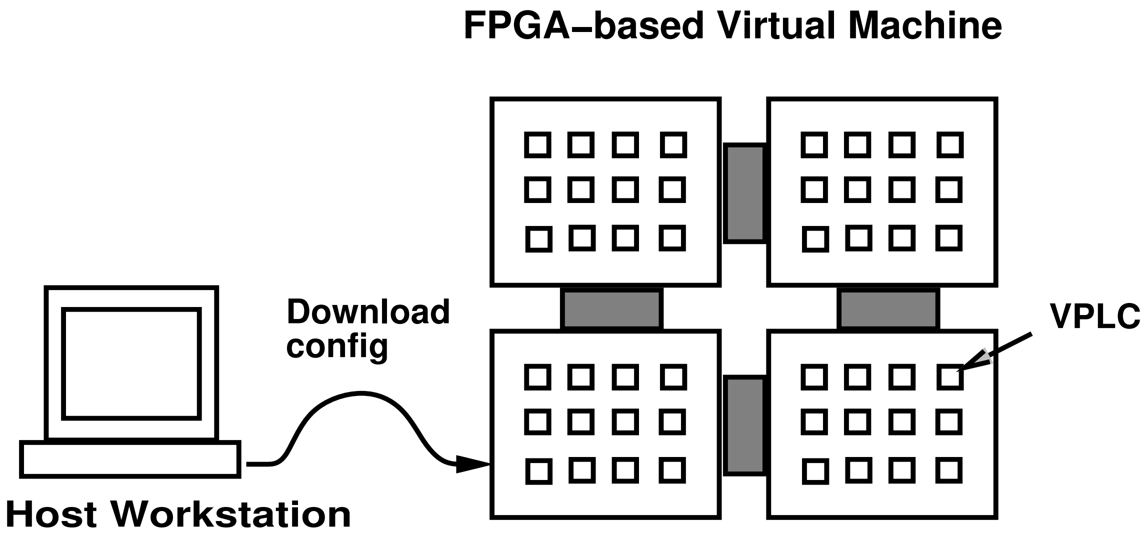

Figure 9-1: Virtual Machine Computing Engine

Virtual Machines

Virtual Wires allow us to treat an array of FPGAs as one large FPGA, while only using nearest neighbor interconnect. This allows the possibilty of a *Virtual Programmable Logic Chip* (VPLC) incorporates multiple FPGAs on an MCM substrate module. Arbitrary logic could be automatically synthesized onto the module, and could be used directly in systems as one giant FPGA.

A *Virtual Machine* is a customized computing engine based on FPGA substrates (Figure 9-1), potentially in the form of VPLC's. By including appropriate interfaces to the external world, such a machine could replace entire customized hardware systems. This concept would allow designers to extend the philosophies behind using FPGA's to the entire system - in much the same way Manning (and Von Neumann before) originally imagined with their cellular arrays.

Bibliography

[1] A. Agarwal, J. Babb, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, G. Maa, K. MacKenzie, D. Nussbaum, M. Parkin, and D. Yeung. Sparcle: Today's Micro for Tomorrow's Multiprocessor. In *HOTCHIPS*, August 1992.

[2] A. Agarwal et al. The MIT Alewife machine: A large-scale distributed memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Press, 1991.

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] Aptix, Inc. *Aptix AXB-AP4 Data Sheet*, oct 1993.

[5] J. Babb and R. Tessier. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings 1993 MIT Student Workshop on Supercomputing Technologies*, pages 4.0 – 4.1, Plimoth Plantation, August 1993.

[6] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *Proceedings IEEE Workshop on FPGA-based Custom Computing Machines*, pages 142–151, Napa, CA, April 1993. IEEE. Also as MIT/LCS TM-491, January 1993.

[7] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[8] Z. Barzilai, J. Carter, B. Rosen, and J. Rutledge. HSS - A High Speed Simulator. Technical Report RC 11738, IBM T. J. Watson Research Center, March 1986.

[9] D. Bertsekas and R. Gallagher, editors. *Data Networks.* Prentice Hall, Englewood Cliffs, N.J., 1992.

[10] T. Blank. A survey of hardware accelerators used in computer aided design. *IEEE Design and Test of Computers*, Aug. 1984.

[11] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Norwell, Mass., 1992.

[12] J. Cocke and R. E. Miller. Configurable Computer System. Technical Report 9, IBM Technical Disclosure Bulletin, Feb. 1973.

[13] Concurrent Logic, Inc. *CLi6000 Series Field-Programmable Gate Arrays*, May 1992. Revision 1c.

[14] D. Conner. IC prototyping: When simulation isn't enough. *Electronic Design News*, pages 74–74, July 1993.

[15] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.

[16] W. J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), Mar. 1992.

[17] D. V. den Bout, J. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman. Anyboard: An FPGA-based, reconfigurable system. *IEEE Design and Test of Computers*, Sept. 1992.

[18] S. Devadas and K. Keutzer. Synthesis of robust delay-fault-testable circuits: Practice. *IEEE Transactions on Computer Aided Design*, 11(3):277–300, March 1992.

[19] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[20] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and using a highly parallel programmable logic array. *Computer*, 24(1), Jan. 1991.

[21] A. Gupta. *Formal Hardware Verification Methods: A Survey*. PhD thesis, Carnegie Mellon University, School of Computer Science, October 1991.

[22] S. Hanono. Lsi and verilog serial interface to fpga board. Alewife Systems Memo 39, MIT Computer Architecture Group, Oct. 1993.

[23] N. Hastie and R. Cliff. The implementation of hardware subroutines on field programmable gate arrays. In *IEEE Custom Integrated Circuits Conference*, May 1990.

[24] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, California, 1990.

[25] InCA Inc. *Concept Silicon Reference Manual*, Nov. 1992. Version 1.1.

[26] B. Kerninghan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. Technical Report 2, Bell Syst. Tech. J., Feb. 1990.

[27] T. Kirkland and M. Mercer. Algorithms for automatic-test pattern generation. *IEEE Design & Test of Computers*, 5(3):43–55, June 1988.

[28] S. Kirkpatrick, C. D. Gellatt, and M. P. Vecchi. Simulated annealing. *Science*, 220, 1983.

[29] J. Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[30] H. T. Kung. Systolic communication. In *Proceedings of the International Conference on Systolic Arrays*, San Diego, California, May 1988.

[31] B. Landman and R. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on Computers*, C-20(12), Dec. 1971.

[32] C. Y. Lee. Representation of switching circuits by binary-decision progams. *Bell. Syst. tech. J.*, 38:958–999, July 1959.

[33] L. Maliniak. Multiplexing enhances hardware emulation. *Electronic Design*, Nov. 1992.

[34] F. P. Manning. *Automatic Test, Configuration, and Repair of Cellular Arrays*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1975.

[35] M. Minsky. Scheduled routing for the numesh. Master's thesis, EECS Department, MIT, Sept. 1993.

[36] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory or NP-Completeness*. Freeman, San Francisco, 1979.

[37] J. V. Neumann. *Theory of Self-Reproducing Automata.* edited and completed by A.W. Burks, U. of Ill. Press, Urbana and London, 1966.

[38] G. F. Pfister. The yorktown simulation engine: Introduction. In *Proc. 19th Design Automation Conference*, pages 51–54. IEEE Computer Society Press, 1982.

[39] R. Shoup. *Programmable Cellular Logic Arrays.* PhD thesis, Carnegie Mellon University, School of Computer Science, March 1970.

[40] L. P. Soulé. *Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms.* PhD thesis, Standford University, Department of Electrical Engineering and Computer Science, June 1992.

[41] Synopsys, Inc. *Command Reference Manual, Version 3.0*, dec 1992.

[42] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal. The Virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environment. In *Submitted to 1994 ACM International Workshop on Field-Programmable Gate Arrays*, Berkeley, CA, February 1994. ACM.

[43] S. Walters. Prototyping ASICs in reprogrammable hardware meets system requirements. *Computer Design*, pages 76–77, May 1990.

[44] S. Walters. Computer-aided prototyping for ASIC-based systems. *IEEE Design and Test of Computers*, June 1992.

[45] Y.-C. Wei, C.-K. Cheng, and Z. Wurman. Multiple-level partitioning: An application for the very large-scale hardware simulator. *IEEE Journal of Solid-State Circuits*, 26(5), May 1991.

[46] XILINX, Inc., 2100 Logic Drive, San Jose, California, 95214. *The Programmable Gate Array Data Book*, Aug. 1992.

[47] XILINX, Inc., 2100 Logic Drive, San Jose, California, 95214. *The XC4000 Data Book*, Aug. 1992.