A Field-Programmable System with Reconfigurable Memory

by

David Karchmer

A Thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science in the
Department of Electrical and Computer Engineering,
University of Toronto

A Field-Programmable System with Reconfigurable Memory

Master of Applied Science, 1994

David Karchmer

Department of Electrical and Computer Engineering

University of Toronto

Abstract

This thesis describes the construction of a Field-Programmable System (FPS) and a software tool for the implementation of memory on the FPS. This FPS, called the *Transmogrifier-1*, consists of Field-Programmable Gate Arrays (FPGAs), memory chips and programmable interconnect components and can be used both as a logic emulation system and as a compute engine.

This work defines a new optimization problem that arises from the use of Field-Programmable Systems. Circuits implemented on the FPS, will require a set of memories which may not match the number and aspect ratio of the physical memories available on the FPS. This may require that the physical memories be time-multiplexed to implement the required memories, in a circuit we call a *memory organizer.*

A precise definition of the packing optimization problem is given and an algorithm for its solution is presented. The algorithm has been implemented in a Computer Aided Design (CAD) tool that automatically produces a memory organizer circuit ready for synthesis by a commercial FPGA tool set.

Acknowledgements

This work would have been impossible without the help of Professor Jonathan Rose. He provided me with guidance and advice, but more important, with friendship throughout my degree. This thesis is all joint work with him.

I am grateful to all my friends who shared with me much more than just scientific problems. In particular, I thank Paul Chow, Sam Crapanzano, Gennady Feygin, Jess Lee, Tony Ngai, Ali Sheikholeslami, Michael Tresidder, Qiang Wang and Steve Wilton.

I am also greatful to all the people who were involved with the Transmogrifier-1 project: David Galloway, Professor David Lewis, Professor Paul Chow and the people at BNR. Special thanks to David Galloway who wrote part of the software necessary to program the TM-1 and helped me test all the hardware.

Finally, I want to thank my parents Samuel and Susana, and my brother and sisters for their support over the years. I especially want to thank my brother, Mauricio. He not only provided a continual source of encouragement, but also contributed to many aspects of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In today's competitive global economy, the electronic industry is fighting to reach the market with new products in the shortest possible time. By being able to build a prototype as early in the development cycle as possible, companies can start testing their products even before fabrication and hence, reduce time-to-market and financial risks.

Field-Programmable Gate Arrays (FPGAs) have been used as a solution to the above time-to-market and risk problems. FPGAs provide instant manufacturing and very low cost prototyping as well as re-programmability [1]. As illustrated in Figure 1.1, an FPGA is a regular array of programmable logic blocks that can be interconnected by a programmable-interconnect network. A digital circuit can be emulated by programming the function into the logic blocks and selecting the way these blocks are interconnected.

Current FPGAs, however, do not have sufficient logic capacity for large designs, and a system with multiple FPGAs is often needed. Such a *Field-Programmable System* (FPS), as shown in Figure 1.2, consists of several FPGAs, auxiliary special-purpose integrated circuits (*e.g.* memory chips), and an interconnection network or mechanism.

To date, different types of programmable systems exist for both logic emulation [2, 3, 4]

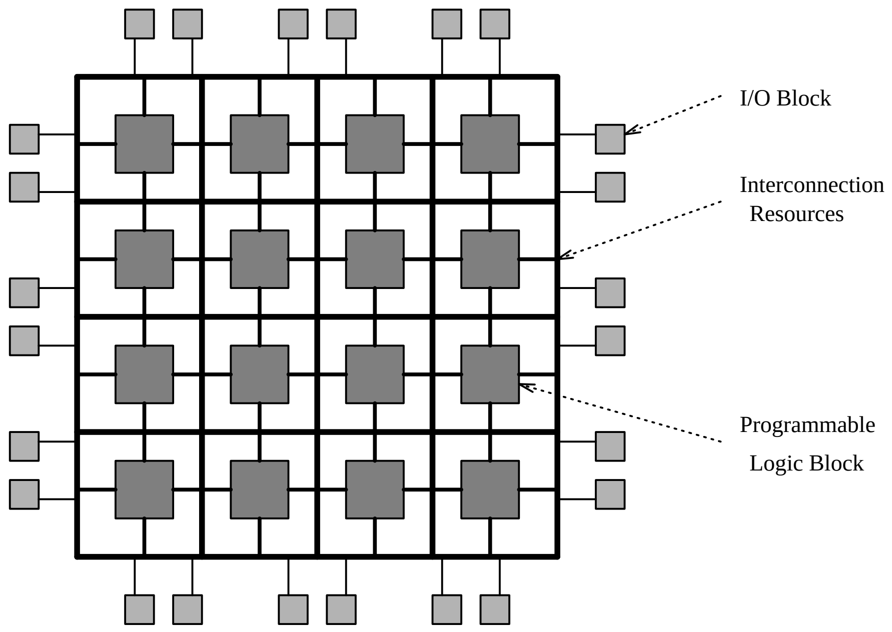

Figure 1.1: A Conceptual Field-Programmable Gate Array

and compute engines [5, 6, 7, 8, 9]. A logic emulator is a system of hardware and software that can take a large gate-level logic design (at least tens of thousands of gates), and emulate it in hardware form, by configuring a set of FPGAs [2]. A compute engine uses reconfigurable hardware to create a custom computer architecture for a particular application. It is designed to accelerate computations which exhibit at least a modest amount of temporal parallelism (pipelining) or data parallelism [10]. Many interesting issues arise in building these Field-Programmable Systems:

- How many and what kind of FPGAs should be used ?
- What kind of auxiliary special purpose integrated circuits should be used (if any) ?
- How these chips should be interconnected in order to maximize performance and minimize cost ?

Each of the existing Field-Programmable Systems uses different FPGAs and different interconnected structures. The relative advantages of each system will be described in Chapter 2.

In this thesis, an alternative Field-Programmable System architecture and working system are implemented and presented. This FPS, called the *Transmogrifier-1* (TM-1), consists

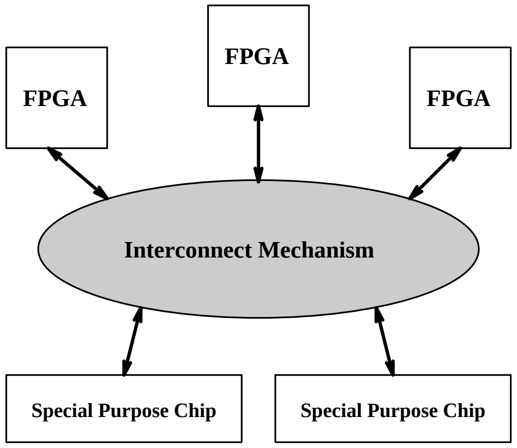

Figure 1.2: A Conceptual Field-Programmable System

of four 10K-gate FPGAs and four $32k \times 9$ memories connected together through two Field-Programmable Interconnect Components (FPIC). An FPIC is a new kind of programmable device which, like an FPGA, contains programmable routing circuitry that can connect each I/O pin to any number of other I/O pins. By programming the FPGAs and the way they are interconnected together (*i.e.* by programming the FPGA and FPIC devices), the TM-1 can emulate any digital circuit that does not exceed its logic and memory capacity.

A second contribution of this thesis is a method for dealing with memory in the context of Field-Programmable Systems. Memory is an essential part of almost any digital system, and hence forms a key element of an FPS. Each application, however, has vastly different memory needs. A telecommunication circuit, for example, may require many small queuing buffers, while a graphics engine will require a few larger frame buffers. Each may, in addition, require other shapes and sizes of independent memories.

In this thesis, we present a method that allows users to efficiently map the required set of memories for an application circuit into the available physical memories on an FPS. The context is a *low-cost* FPS in which there is a small number of pre-fabricated physical memories (such as in the TM-1), but with a large number of desired memories. Because the number and size of the desired memories may not match the number and size of the physical memories, two or more desired memories may need to share the same physical memory. In this case the physical memories must be time-multiplexed — each desired memory will have a different time slot in which it can access a separate portion of the physical memory.

The mapping becomes an optimization problem when the number of required memories exceeds the number of physical memories. There may be many different ways that the required (logical) memories can be packed into the physical memories. If the logical memories have different access-time requirements, the optimization problem is to find a packing that meets the timing requirements. Furthermore, different packings will require different amounts of multiplexing and control, and so it is desirable to minimize the area devoted to this part of the circuit.

For example, consider an application in which there are five logical memories as illustrated in Figure 1.3: There are three $3k \times 8$ memories, each with a required access time of $80ns$, one $4k \times 7$ memory with a required access time of $20ns$ and one $32k \times 8$ memory with a required access time of $100ns$. Assume that the FPS has three $32k \times 8$ physical memories, each with an access time of $20ns$. Since the logical memories are time-multiplexed, their final access time is approximately equal to the physical memory's nominal access time (in this case $20ns$) multiplied by the number of memories sharing the physical memory.

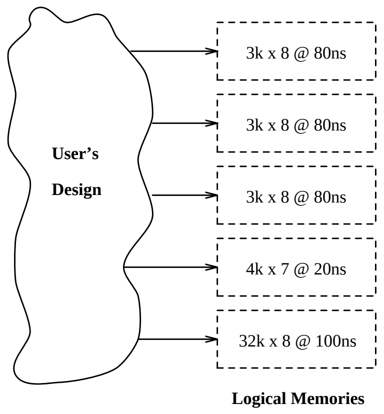

Figure 1.3: Application with 5 logical Memories and Timing Constraints

Figure 1.4a illustrates a packing in which all the $3k \times 8$ and $4k \times 7$ memories have a final access time of $40ns$ or less. This implementation does not meet the access time requirement of the $4k \times 7$ memory. Figure 1.4b shows the only possible packing that does meet all the access time constraints. The three $3k \times 8$ memories have a final access time of $60ns$ and both the $4k \times 7$ and the $32k \times 8$ memories have access time of $20ns$.

Note that in some cases, a logical memory may be larger than an individual physical memory, and so will have to be partitioned into smaller pieces before it can be packed.

The multiplexer and controller that implements the memory packing will be referred to as a *Memory Organizer.* In a Field-Programmable System, this circuit would be implemented

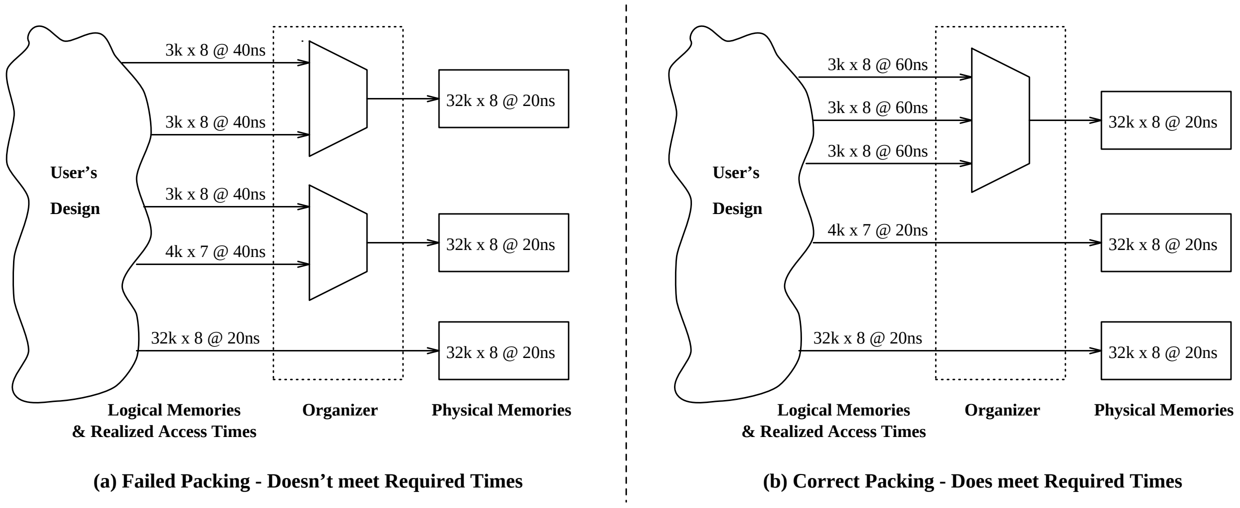

Figure 1.4: Possible Packing Solutions for Memories in Figure 1.3

using the FPGAs. Chapter 4 presents an algorithm to solve the optimization problem and describes a Computer-Aided-Design (CAD) tool which uses this algorithm to generate the memory-organizer netlist.

1.2 Thesis Overview

This thesis is divided into five chapters. In Chapter 2, some academic and commercially available Field-Programmable Systems, and the way they use their memory, are introduced. Chapter 3 describes the Transmogrifier-1 Field-Programmable System built at the University of Toronto as part of this thesis, along with the software used to program it. In Chapter 4, the Field-Reconfigurable Memory problem is defined and a solution is presented. Finally, Chapter 5 concludes this thesis and offers some suggestions for future work.

Chapter 2

Background

This chapter reviews related work on Field-Programmable Systems (FPS). It provides a detailed description of academic and commercially available Field-Programmable Systems. In addition, it describes how memory is used in existing Field-Programmable Gate Arrays (FPGAs) and Field-Programmable Systems.

The concept of reconfigurable hardware has been present for a number of years. Several Universities as well as private companies have been building systems with several FPGAs on it. Although there are many differences between these Field-Programmable Systems, probably the most important one is the way in which the FPGAs are connected together. In this thesis, these systems are categorized by their inter-chip connection architecture.

Section 2.1 presents the one-dimensional array interconnect architecture. Section 2.2 describes the two-dimensional nearest-neighbor mesh interconnect architecture. Section 2.3 presents the concentrated programmable-interconnect architecture and describes three variations of it.

In the final two sections of this chapter, the use of memory in programmable systems is described: Existing FPGAs with memory are presented in Section 2.4 and existing Field-Programmable Systems with memory are described in Section 2.5.

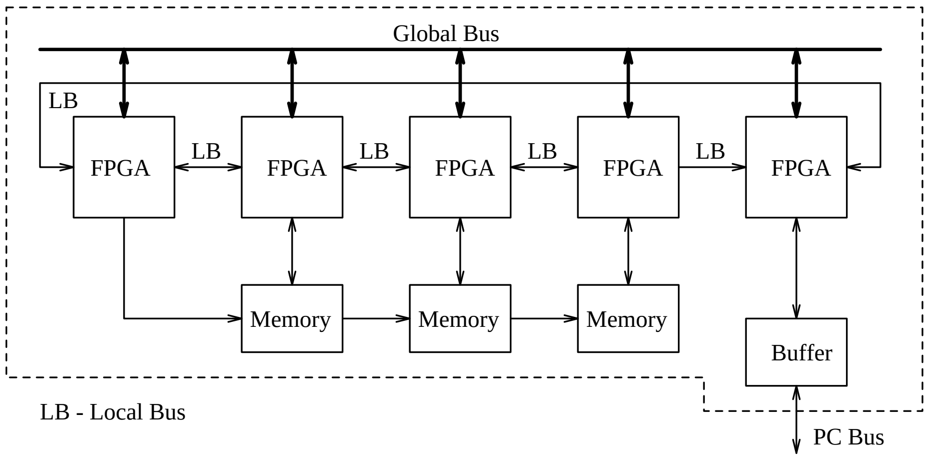

Figure 2.1: The AnyBoard Block Diagram

2.1 One-Dimensional Array

The one-dimensional array architecture is usually used in compute engines in which systolic processing is needed. A compute engine (as described in Section 1.1), uses reconfigurable hardware to accelerate the execution speed of an application.

In this architecture, the FPGAs are connected in a linear array. Each FPGA is connected to its nearest neighbor through a local bus. An additional global bus is usually used to access common resources such as the host computer's data bus. There are two notable examples of Field-Programmable Systems which use this style: The *AnyBoard* and the *Splash-2*.

2.1.1 The AnyBoard System

The *AnyBoard* is an FPGA-based Reconfigurable System built at North Carolina State University [5]. It contains five Xilinx XC3090 [11] FPGAs connected as illustrated in Figure 2.1. In addition, three of the FPGAs are connected to the data bus of three separate $128k \times 8$ SRAMs. The leftmost FPGA serves as an address generator. By using one FPGA to provide the address word to all memory chips, different FPGAs cannot access different memory locations. This results in less flexibility than connecting both data and address buses between each FPGA-memory pair. However, it requires fewer FPGA I/O pins.

The global bus is used for high fan-out signals, such as clocks, to connect to every FPGA. The global bus is also used to interface the AnyBoard with external systems. For programming and control, the AnyBoard is connected to a PC-compatible computer through its parallel bus and the rightmost FPGA.

2.1.2 Splash-2

Splash-2 is a compute engine built at the Supercomputing Research Center [12, 10]. This system, replacing the Splash-1 (a similar system but with older technology), contains 17 Xilinx XC4010 [11] FPGAs connected as shown in Figure 2.2. The systolic data path brings data from either the previous Splash-2 board, or from the host computer, into the FPGA labeled $X1$, through the linear array, and out from FPGA labeled $X16$ to either the next Splash-2 board or back to the host computer. Each FPGA is connected to a $256k \times 16$ RAM (labeled $M1$ to $M16$). These RAMs are also connected to a 16-bit global data bus. This global data bus allows the host computer to directly write or read the memories. The address lines are generated by the FPGAs.

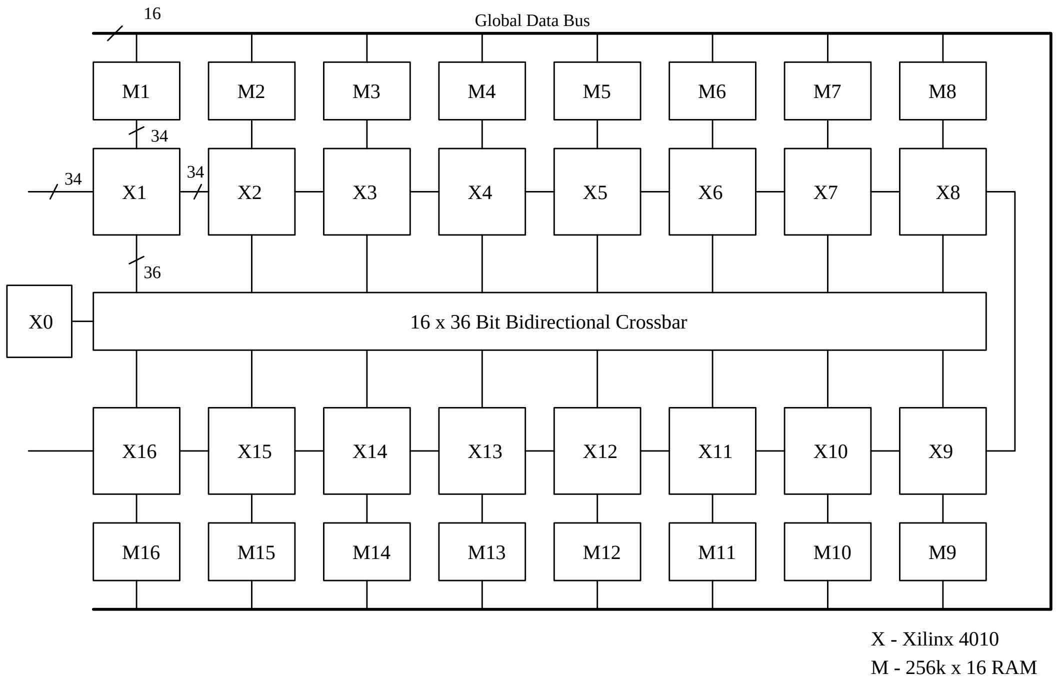

Figure 2.2: The Splash-2 Block Diagram

As an improvement upon the limited inter-FPGA communication found in other one-dimensional array systems, each FPGA in the Splash-2 board is connected to a 16×36 bi-directional crossbar. This crossbar is formed with nine crossbar chips, each with 16 4-

bit bi-directional ports. Using the FPGA labeled $X0$ (Figure 2.2), these crossbars can be dynamically programmed (*i.e.* switched cycle by cycle) with eight different configurations. A 4×4 mesh, for example, can be realized if in one of the two dimensions, only half the needed communication paths are available at one time [12].

Several example applications have been implemented on Splash-2. A text searching program is capable of processing text at an estimated rate of 50 million characters per second with a system clock running at $25MHz$ [13]. Also, an implementation of a dynamic programming algorithm for DNA database searching has been implemented [14]. This system can search a database at a rate of 12 million characters per second, several orders of magnitude faster than implementations of the same algorithm on conventional computers.

2.1.3 Comments on One-Dimensional Arrays

In cases in which systolic processing arrays are desirable, one-dimensional arrays of FPGAs are very powerful. However, this architecture is not very efficient when circuits are not partitionable into an array of functional blocks small enough to fit into each FPGA.

In addition, the one-dimensional arrays suffer from poor chip interconnect flexibility. Systems with this interconnect architecture have only a single data path and therefore, if the circuit is not narrow enough to fit in one array of FPGAs, the design cannot be implemented. Splash-2, however, has much more flexibility than the AnyBoard. This is because in addition to the linear data path, Splash-2 uses a crossbar to interconnect (to some degree) the FPGAs, making this system more similar to those described in the following section.

2.2 Nearest-Neighbor Connection Mesh

One way to increase inter-FPGA connectivity is by using a two-dimensional nearest-neighbor connection mesh. In this architecture, as illustrated in Figure 2.3, each FPGA is hard-wired to the four closest FPGAs (*i.e.* to the north, south, east and west).

Below, three Field-Programmable Systems that use the nearest-neighbor connection mesh are described.

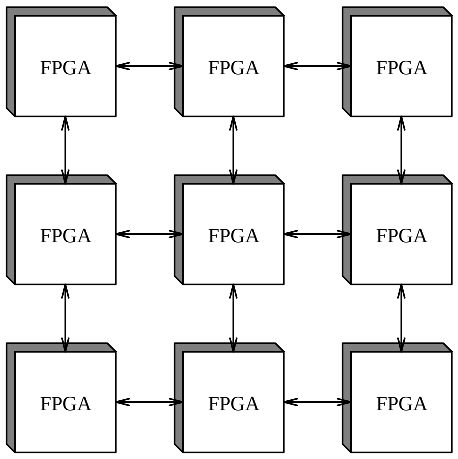

Figure 2.3: Example of a 3×3 array architecture

2.2.1 The DEC PeRLe-0 and PeRLe-1 systems

The *PeRLe-0* and *PeRLe-1* systems [6], built at the DEC Paris Research Laboratory, are two Field-Programmable Systems that, when connected to the system bus of a host computer, work as universal hardware co-processors. The *DEC PeRLe-0* is a 5×5 matrix of Xilinx XC3020 [11] FPGAs with two 32-bit wide RAM banks, and the *DEC PeRLe-1* is four times bigger than it predecessor. It uses 24 Xilinx XC3090 [11] FPGAs and larger RAMs. It is important to notice that both systems also have a global bus, used to access the memory banks in the east and south of the board. The characteristics of the two systems are shown in Table 2.1

System	**FPGAs**	F_{max} (MHz)	**RAM** (MB)	**Host Bus** (MB/s)
DEC PeRLe-0	25 XC3020	25	0.5	8
DEC PeRLe-1	24 XC3090	40	4	100

Table 2.1: Characteristics of the DEC PeRLe machines.

The systems are employed as compute engines to speed up many critical software applications running on the host, by executing part of the instructions in hardware. With this approach, DEC Paris was able to achieve better performances than a "super-computer" on several applications. Table 2.2 summarizes the practical DEC PeRLe-1 performance reported in [6]. The first column of this table gives the application name, the second gives the system clock speed in MHz, and the third column gives the computing power in *Gbops* (Billion of binary operations per second).

Design	Clock Speed (MHz)	Performance (Gbops)
Multiplier	33	264
RSA Cryptography	32	256
Discrete Cosine Transform	25	250
Newton's Mechanics	25	250
Laplace Equation	20	200
Boltzmann Machine	25	200
3D Geometry	25	75

Table 2.2: Example Applications of the DEC PeRLe-1 system.

2.2.2 Quickturn's RPM Emulation System

The RPM emulation system from Quickturn Systems is the first commercial FPGA-based emulation system[3]. This system was built to work as an Application Specific Integrated Circuit (ASIC) emulator or a microprocessor emulator. A logic emulator is a system of hardware and software, which can take a large gate-level logic design and emulate it in hardware form [2].

Quickturn also selected high-density FPGAs from Xilinx as the implementation medium. Each FPGA is hard-wired to all its nearest neighbor FPGAs forming an array of FPGAs, called an *emulation module*. Several of these modules are connected together to form a large array of FPGAs. The RPM includes an interface for connection to the target system. The complete emulation system can cost several hundred thousand dollars. It is capable of emulating large ASICs or Microprocessors.

The RPM system runs at very low speeds (approximately between $0.5MHz$ and $8MHz$). As an example, a group at Intel Co. have reported that their Quickturn system emulates the *Pentium* microprocessor at $0.5MHz$ [15]. We note however, that this was sufficient speed to boot an operating system.

2.2.3 The MIT Virtual Wires Emulation System

The Virtual Wires System is a sixteen FPGA emulation board built at the Massachusetts Institute of Technology [4]. The system uses Xilinx XC4005 [11] FPGAs connected in a two-dimensional array. Each FPGA has eight hard-wired connections to its nearest neighbors and 22 dedicated lines to a $64k \times 4$ RAM.

The low number of connections between FPGAs is overcome by using *Virtual Wires* [16]. The Virtual Wiring approach increases the usable bandwidth between two FPGAs by pipelining and multiplexing the physical connections between the two FPGAs. Several logical connections can share the same physical pins and wire by clocking this physical resource at the maximum FPGA frequency.

The Virtual Wires system was used to emulate several designs, including Sparcle [17], an 18k-gate ASIC implementation of the Sparc microprocessor with enhanced multiprocessor support. Using 24 FPGAs with an average of 718 gates and 119 I/Os per FPGA, the system was able to run at $0.18MHz$. The Virtual Wires system is capable of simulating/emulating $30K$ gates.

2.2.4 Comments on Nearest-Neighbor Connection Meshes

Although the two-dimensional nearest-neighbor interconnect architecture is widely used, there are three drawbacks in using this architecture:

1. The number of signals that can be passed from FPGA to FPGA is limited to the number of hard-wired connections between these two FPGAs.
2. Long distance communication is slow and expensive, since long connections must travel through many FPGAs.
3. The FPGA's place-and-route tool does not have the freedom to choose which I/O pins to use.

In order to pass a signal from one FPGA to another, the FPGA's place-and-route tool has to route the signal to one of the pins which are hard-wired to the other FPGA. If for example a logic block is in one end of the FPGA and it drives an output pin which is in the other end of the FPGA, this signal has to be routed across the whole device. This may result in larger delays and even in the CAD tool's inability to route.

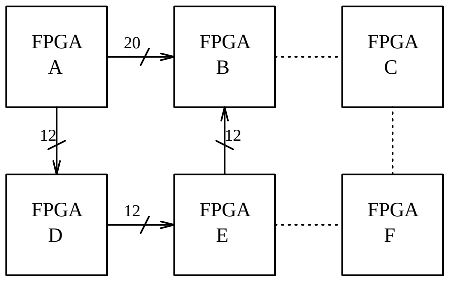

Figure 2.4: Example of a two-dimensional array with few hard-wired connections

Also, if the number of signals required to pass from one FPGA to another is greater than the number of hard-wired connections between these two FPGAs, then additional FPGAs may be needed for some of the signals. As an example, Figure 2.4 shows a two-dimensional array in which 32 signals need to pass from FPGA A to FPGA B. Since only 20 hard-wired connections exist between these two FPGAs, the extra 12 signals need to pass from FPGA A to FPGA B through FPGAs D and E. These 12 signals are going to have a larger delay than the 20 signals that pass directly form FPGA A to FPGA B. In addition, there are some wasted resources inside FPGAs D and E because signals have to pass through the FPGAs without using their internal logic.

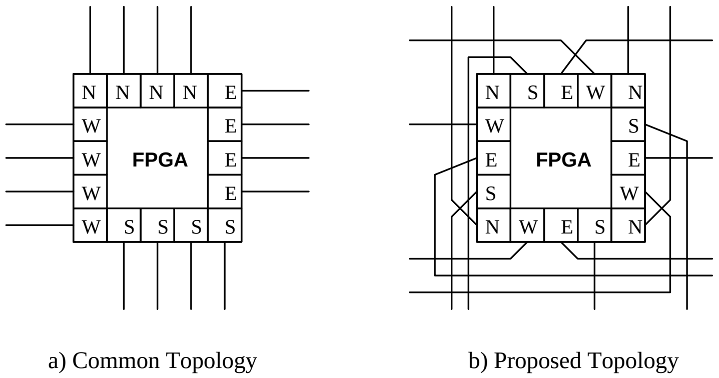

Figure 2.5: The commonly used mesh interconnect topology and a more efficient one.

As seen in the above example, a signal entering an FPGA from a given direction might want to leave in any other direction, wasting internal routing resources. Hauck, Borriello and Ebeling [18] suggest a method of interconnecting the FPGAs in a two-dimensional array in a way that reduces the FPGA-internal routing cost. By intermingling the connections, as shown in Figure 2.5b, they made sure that a signal entering on a given pin will have a nearby pin exiting in the direction it wishes to go.

2.3 Concentrated Programmable-Interconnect

One solution to the limited interconnect problems of the one-dimensional and two-dimensional architectures is found in the concentrated programmable-interconnect architecture. As illustrated in Figure 2.6a and 2.6b, in the one-dimensional and two-dimensional interconnect architectures, signals can only pass from one FPGA I/O pin to another pre-defined FPGA I/O pin. In the concentrated programmable-interconnect architecture, as shown in Figure 2.6c, all the FPGA I/O pins are hard-wired to a central - programmable chip-interconnect mechanism. The interconnect mechanism must contain a programmable switching circuitry capable of interconnecting any set of I/O pins. This interconnect mechanism can simply be an FPGA that links all the other FPGAs (*i.e.* the ones which are actually implementing the logic). However, FPGAs were designed to implement logic and not to interconnect chips. Therefore, the use of special purpose interconnect components may be preferred. These components, such as Aptix's FPIC [19] and I-Cube's FPID [20], have a larger amount of usable I/O ports and no logic blocks.

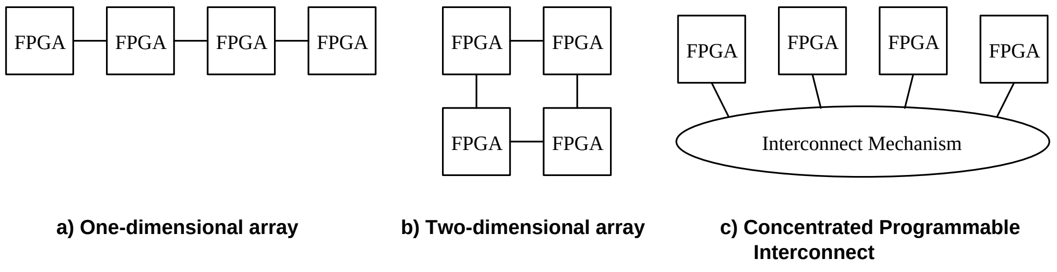

Figure 2.6: Inter-chip connection architectures.

In this section, we first describe the commercially available special-purpose interconnect components and then we present three different concentrated programmable-interconnect architectures: the full-crossbar interconnect architecture, the partial-crossbar interconnects architecture and the distributed-crossbar interconnect architecture.

2.3.1 Special-Purpose Interconnect Component

I-Cube's Field-Programmable Interconnect Devices

The I-Cube's Field-Programmable Interconnect Device (FPID) [20] has from 96 to 320 usable I/O ports with $10ns$ port-to-port delays. With a $0.8\mu m$ static RAM CMOS process, and active buffering, FPID devices can support signals switching at close to $100MHz$.

Internally, the FPID is a crossbar switch, allowing total flexibility in routing signals. A crossbar is an interconnect architecture which can connect any pin with any other pin or pins, without restrictions. In the FPID, only one transistor switch needs to be closed in order to make a connection between two I/O ports, allowing predictable delays. Each I/O in the device is identical and can be programmed as an input buffer, registered input, tristate output, registered output or bidirectional port. It is important to note that for bidirectional signals, the device detects the driving port and passes the signal to the receiving port.

The FPID device is programmed using the JTAG (IEEE 1149.1) serial bus or a parallel bus in which connections can be changed incrementally in under $40ns$.

Aptix's Field-Programmable Interconnect Component

When larger number of I/Os is needed, a larger crossbar is required. Aptix's Field - Programmable Interconnect Component (FPIC) [19] has 936 usable I/O pins. Each of the I/O pins can be connected to any number of other I/O pins through a programmable, routed array architecture, as illustrated in Figure 2.7. The pins are arranged in a 32×32 array with horizontal and vertical routing channels between rows and columns of pins.

Critical paths can be made with as few as two transistors. However, nets with high fanout can have more than four transistors. The delay between two pins ranges from 5 to $20ns$ depending on the number of pass transistors used to interconnect the two pins.

2.3.2 Full-Crossbar Interconnect

If a small number of FPGA pins are to be interconnected in an FPS, we can use a single programmable-interconnect device to connect all the FPGAs together. The full-crossbar

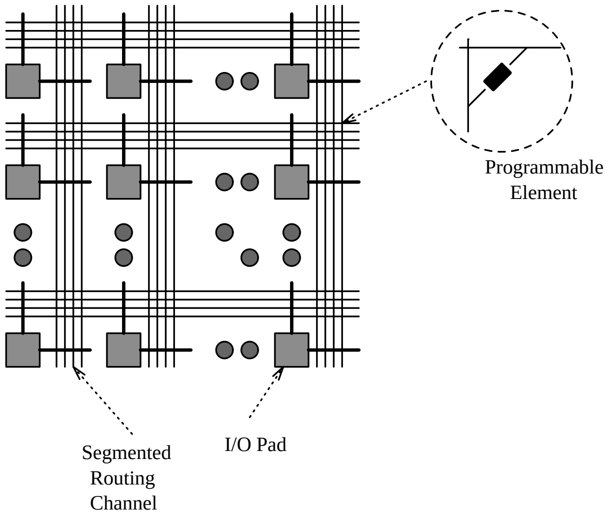

Figure 2.7: I/O array and routing channels of Aptix's FPIC.

interconnect architecture uses a single central-crossbar to interconnect every FPGA's I/O pins. As introduced in Section 2.3.1, a crossbar is a chip (or set of chips) which can connect any pin with any other pin or pins, without restrictions. Figure 2.8 shows how four FPGAs are connected together using a central-crossbar device. By programming this crossbar, connections can be made between any set of FPGA I/O pins. Because of this, the FPGA's place-and-route tool has the freedom of choosing the mapping of internal signals into specific I/O pins. This freedom results in a much better utilization of internal resources than in the one-dimensional or two-dimensional architectures.

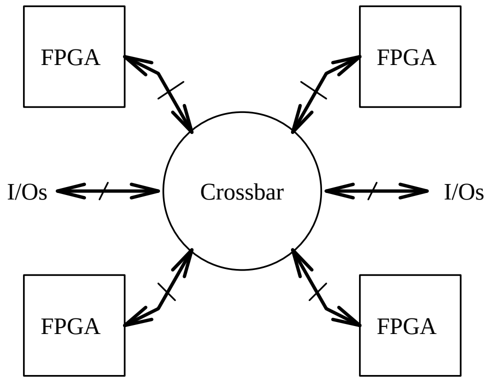

Figure 2.8: Full-Crossbar Architecture

The programmable-interconnect chips described in Section 2.3.1 can only be used as full-crossbars for a small FPS. None of them can handle more than four or five FPGAs with 160 pins each. Therefore, to date, only small Field-Programmable Systems exist with full-crossbar architectures.

2.3.3 Partial-Crossbar Interconnects

For systems with a large number of FPGA pins, many programmable-interconnect chips are needed. In this case, there may be different ways to connect these crossbars.

One way to interconnect several FPGAs is with the use of a partial-crossbar interconnect architecture [2]. As illustrated in Figure 2.9, it consists of a set of small full-crossbars, connected to a set of FPGAs, but not to each other. The FPGA's I/O pins are divided into subsets of pins. There are as many subset of pins (per FPGA) as there are crossbar chips. Each FPGA's subset of pins is connected to a different crossbar.

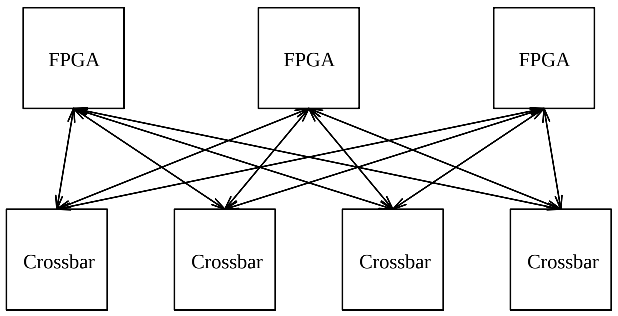

Figure 2.9: Partial-Crossbar Architecture

Since each crossbar is connected to a subset of pins on each FPGA, an interconnection between an FPGA's I/O pin in one subset and another FPGA's I/O pin in a different subset cannot be done. Therefore, the partial-crossbar interconnect will fail to interconnect a net when the available pin in the source FPGA is hard-wired to a crossbar which has no available pins connected to the destination FPGA.

The partial-crossbar architecture provides a more predictable routing delay between FPGAs, since every inter-FPGA signal moves through exactly one crossbar. This is an advantage of significant use by the system level partitioning tool — the one that subdivides a large design into the number of subblocks equal or less than the number of available FPGAs.

2.3.4 Distributed-Crossbar Interconnect

Another way to interconnect several crossbars is with a distributed approach. In a distributed-crossbar interconnect, the number of FPGAs is divided into the number of available crossbars. Then, each group of FPGAs is connected to one crossbar. The crossbars are interconnected using global buses. The crossbars can be interconnected in a linear array like the FPGAs in the one-dimensional array architecture or in a two-dimensional array like the FPGAs in a nearest-neighbor connection mesh. Figure 2.10 illustrates an example in which six FPGAs are interconnected using two crossbars and one global bus.

Like the one-dimensional and two-dimensional array of FPGAs, the distributed-crossbar interconnect architecture suffers from limited bandwidth between crossbars. This problem is reduced by increasing the number of global connections, at the cost of connecting fewer FPGAs per crossbar.

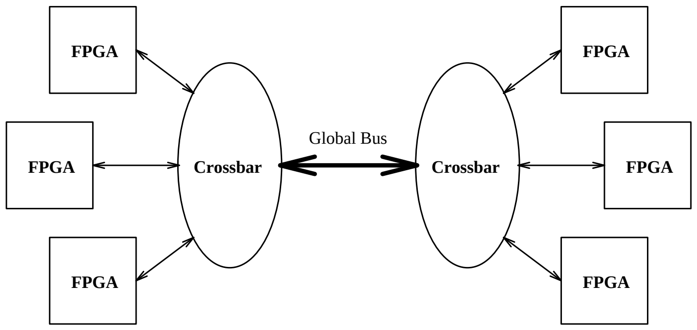

Figure 2.10: Using a global bus to interconnect more than one crossbar.

2.3.5 Comments on Concentrated Programmable-Interconnects

Ideally, a concentrated programmable-interconnect with one large full-crossbar chip is preferred over any other architecture. However, because of the limited size of commercially available crossbar devices, a partial-crossbar or distributed-interconnect architecture is required.

With partial-crossbar interconnect, the place-and-route tool is not as restricted (in terms of assigning I/O pins) as the one-dimensional or the two-dimensional array of FPGAs, but does not have the freedom of a full-crossbar interconnect. The capacity of a partial-crossbar interconnect depends on the number and size of the crossbars used. At one extreme, we have only one subset of pins per FPGA connected to one full crossbar. At the other extreme, we have as many subsets (per FPGA) as the number of pins. In this case, we need as many crossbars as pins in each FPGA. This will have the least flexibility because the FPGA's place-and-route tool does not have much freedom in assigning I/O pins to internal signals.

With distributed-crossbar interconnects, the place-and-route tool has no restriction in terms of assigning I/O pins to internal signals. However, in some case, the number of hard-wired connections between two crossbars may not be enough to pass the required signals.

2.4 FPGAs with memory

Since part of this thesis concerns memory in Field-Programmable Systems. We first describe how memory is implemented directly on a Field-Programmable Gate Array.

Company	**FPGA**	**SRAM Blocks**	**Number of Blocks**	**Maximum User Bits**
AT&T	Orca 2C15	two 16×2 or one 16×4 per PLC	400 PLCs	25,600
Xilinx	XC4013	two 16×1 or one 32×1 per CLB	576 CLBs	18,432
Intel	iFX780	128×10 per CFB	8 CFBs	10,240
Crosspoint	CP22000	1 bit per RLM	3,684 RLMs	3,684

Table 2.3: FPGAs with on-chip SRAM.

To date, several commercial FPGAs have the capability to implement small memories. Both Xilinx 4000 series [11] and AT&T's Optimized Reconfigurable Cell Array (ORCA) [21] FPGAs have the option of configuring their logic blocks as memory blocks. Xilinx's Look-Up tables (LUTs) can be configured as two 16×1 memories or one 32×1 memory while AT&T's LUTs can be configured as a two 16×2 memories or one 16×4 memory.

The Crosspoint CP20K series FPGAs use RAM-Logic Tiles (RLM) designed to efficiently implement memory structures [22]. Intel's iFX780 [23], the first member of the FLEXlogic FPGA family, uses eight Configurable Function Blocks (CFBs) which can be configured as a 24V10-type Programmable Logic Device (PLD) or as a 128×10 bit memory.

Table 2.3 summarized the implementation of memory in these FPGAs. It only shows one member of each of the FPGA family described above. The only purpose of this table is to show the amount of memory that can be implemented in each of the FPGAs. It is important to say that the maximum number of user bits shown in column five does not necessarily mean that the user can implement a memory of that size. A $4k \times 1$ bit memory, for example, can be implemented in a Xilinx FPGA using 244 logic blocks (*i.e.* 42% of the CLBs of a XC4013).

2.5 Memory in existing Field-Programmable Systems

As described above, some FPGAs have the capacity to implement memory blocks. However to date, these FPGAs cannot implement large memories. Therefore, off-the-shelf memory chips are still needed in Field-Programmable Systems. Most of the systems described in Section 2.1, 2.2, and 2.3 have some auxiliary memory on board. Both the AnyBoard and Splash-2 have memory chips with fixed, hard-wired connections to the FPGAs. The AnyBoard provides three $128k \times 8$ RAMs. As illustrated in Figure 2.1, the data bus of these RAMs are connected to the three central FPGAs, and the leftmost FPGA serves as an address generator. This architecture limits the flexibility of RAM addressing, but saves a large number of I/O pins on all FPGA. The Splash-2, as shown in Figure 2.2, has one $256k \times 16$ memory chip connected to each of the 16 FPGAs, through a 34-bit bus (a 17-bit address bus and a 16-bit data bus). The memories can also be directly read or written from the Sun host over the Sbus and the 16-bit global data bus on the Splash-2 board.

Quickturn's emulation system does not provide any on board memory. However, they do sell an add-on memory board which can be connected to the emulation-module boards. The memory on the DEC Paris' PeRLe system is not directly connected to each FPGA but to a global bus accessible to any FPGA. This memory block is divided into two memory banks located on the east and south sides of the FPGA matrix. MIT's Virtual Wires emulation system, on the other hand, has a $64k \times 4$ memory hard-wired to each of the FPGAs on the board.

The Virtual Wires system is the only one capable of multiplexing (using the virtual wires approach) the address, data and control signals in order to increase the number of memory accesses to the same SRAM chip during each emulation clock cycle.

A serious shortcoming of all of the above Field-Programmable Systems (except, to a limited degree, the Virtual Wires system) lies in a fact that these systems can only emulate a number of required memories equal to the number of physical memories. If for some reason, a larger number is needed, the user is forced to manually map and time-multiplex the memory access.

2.6 Summary

In this chapter, several Field-Programmable Systems were described and categorized by their chip interconnect architecture. These architectures are:

The one-dimensional array. FPGAs are connected to their nearest neighbor through a local bus. The systems have only a single data path.

The nearest-neighbor connection mesh. FPGAs are connected together in a two dimensional array. Each FPGA is hard-wired to the four closest FPGAs.

The concentrated programmable-interconnect. FPGAs are connected to a central-programmable chip-interconnect mechanism. The mechanism is programmed to connect any set of pins.

Also in this chapter, the memory capabilities of some FPGAs as well as the memory capabilities of some Field-Programmable Systems was described. It was shown how, none of the existing Field-Programmable Systems have the capability to emulate a number of required memories larger than the number of physical memories.

Chapter 3

The Transmogrifier-1

As described in Chapter 1, a Field-Programmable System (FPS) may consist of several Field-Programmable Gate Arrays (FPGAs), auxiliary special-purpose integrated circuits (*e.g.* memory chips), and an interconnection mechanism. In Chapter 2, several Field-Programmable Systems were categorized with respect to their chip-interconnect architecture. We presented three interconnect styles: the one-dimensional array, the nearest-neighbor connection mesh and the concentrated programmable-interconnects.

In this chapter, a Field-Programmable System called the *Transmogrifier-1* (TM-1) that was constructed as part of this thesis is described. Both the hardware and software that were developed are presented. The TM-1 interconnect architecture belongs to the concentrated programmable-interconnects class. It is a distributed-crossbar interconnect architecture.

The hardware of this system, including a host computer interface, is presented in Section 3.1. Various software programs needed to use the TM-1 are described in Section 3.2. Finally, some example applications are described in Section 3.3.

3.1 Hardware Description

As described in Chapter 1 and 2, a Field-Programmable Interconnect Component (FPIC) is a new kind of programmable interconnect device. This device can be programmed to connect together any set of I/O pins.

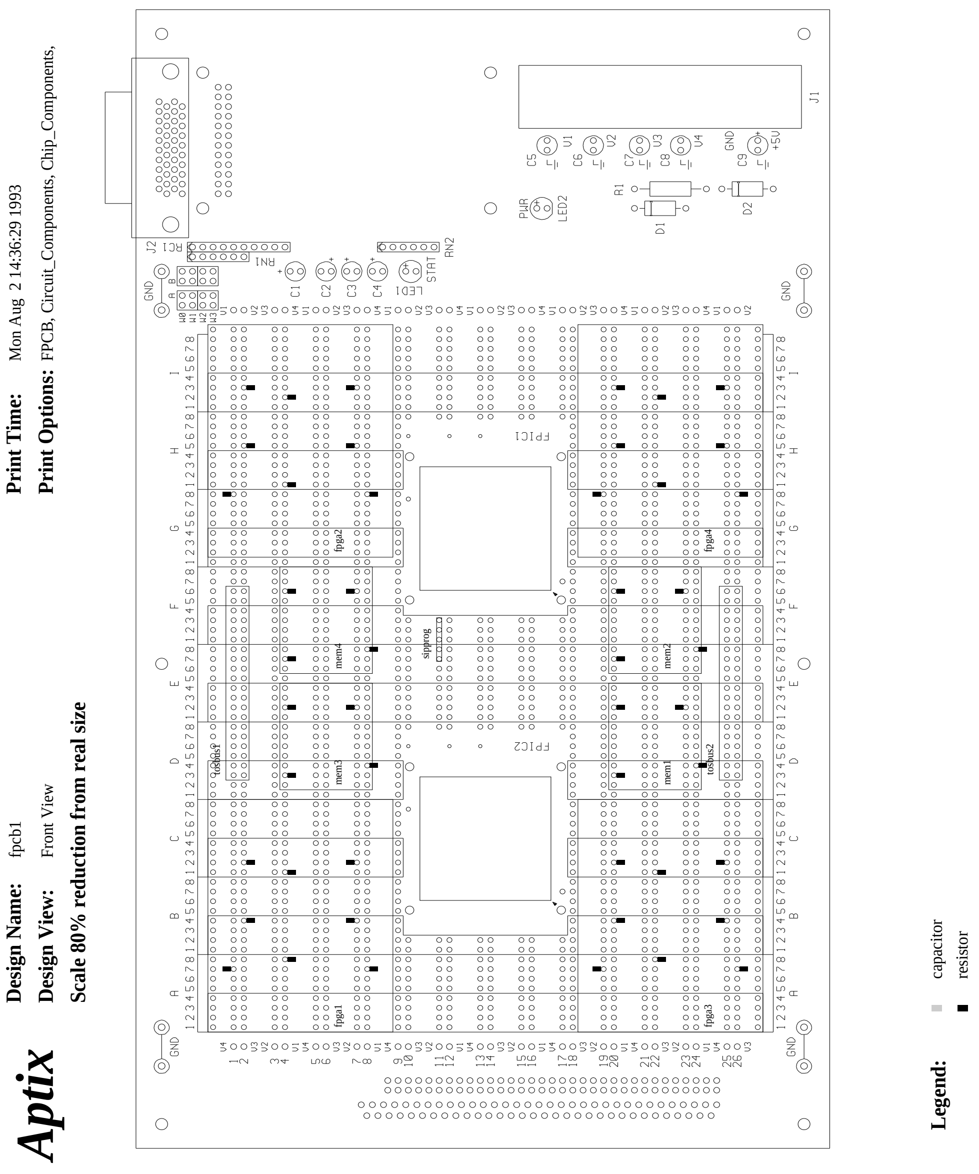

Figure 3.1: Aptix's Field-Programmable Circuit Board.

The Transmogrifier-1 (TM-1) [24] is based on the Aptix AXB-GP2 Field-Programmable Circuit Board [19]. As shown in Figure 3.1, this board contains two Aptix AX1024 FPIC devices [19] hard-wired to 1700 socket holes. These holes can be used to mount standard DIP packages. By programming the FPIC devices, any set of holes can be connected together.

In the TM-1, four Xilinx 4010 FPGAs [11] and four $32k \times 9$ SRAM chips [25] are mounted, using special PGA and PLCC adaptors, to the Aptix board. In addition, two 40-pin connectors carry 72 bidirectional signals (and eight grounds) to an interface board. A block diagram of the TM-1 is shown in Figure 3.2.

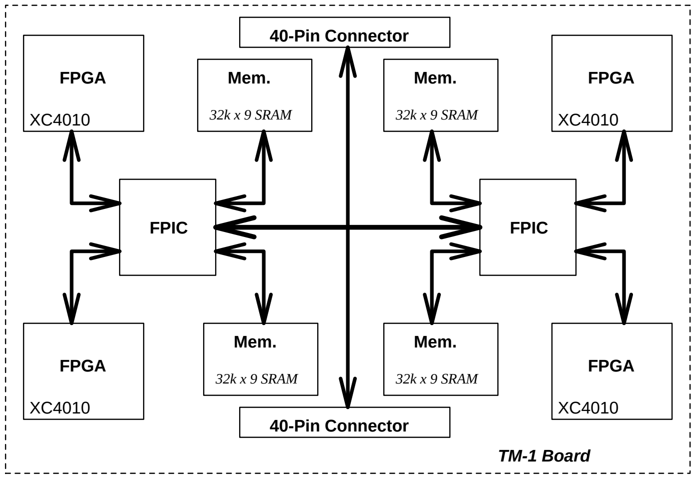

Figure 3.2: Block diagram of the Transmogrifier-1.

The FPGAs can be programmed through a serial interface from the Xilinx Development System [26]. The FPIC devices can be programmed through the Aptix Field-Programmable Circuit Board's Development System [27].

3.1.1 The Interface Board

For the TM-1 to work properly, either as a compute engine or as a logic emulator, it needs some kind of communication with a host computer. In this way, data can be transferred back and forth for testing and debugging purposes. The TM-1 system uses two auxiliary boards in order to interface with a Sparc workstation. Firstly, it uses an S16D interface from *Engineering Design Team, Inc.* (EDT) [28]. The S16D is a single-slot, 16-bit parallel input/output interface for SBus-based computer systems. Secondly, the system uses an additional interface board, built at the University of Toronto as part of this thesis, that contains a Xilinx XC4010 FPGA as shown in Figure 3.3. This FPGA acts as a communication controller, implementing simple protocols that allow a program on the host computer to communicate with the circuit running on the TM-1.

As described in Section 3.1, the TM-1 board contains two 40-pin connectors. These connectors are used to carry 72 bidirectional signals to the interface board. Two circuit designs were implemented and tested on the interface board: A 72-bit programmable bidirectional interface and a 16-bit direct memory access (DMA) [29] interface.

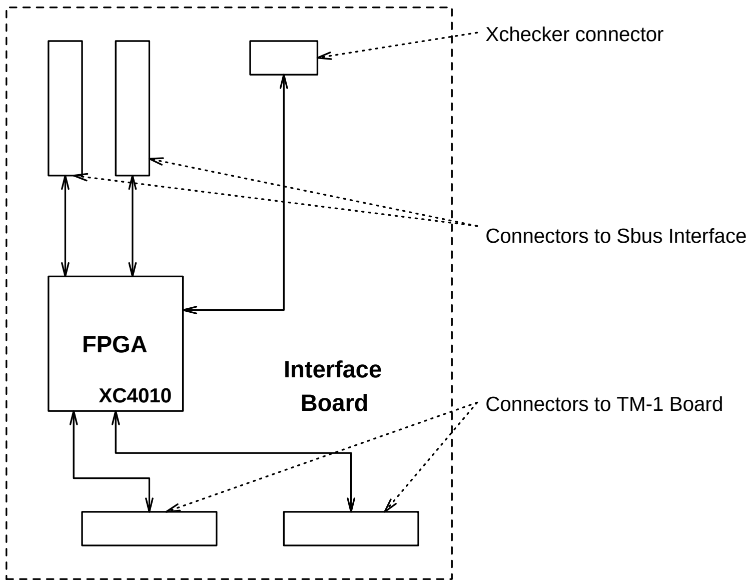

Figure 3.3: Interface's Block Diagram.

72-Bit Programmable Bidirectional Interface

This circuit allows the Sparc workstation to communicate to the TM-1 board through six 12-bit ports. Each of these 72 bits can be configured as an input or output by storing the appropriate data in the configuration registers. Figure 3.4 illustrates the block diagram of this circuit. When the host computer wants to write to the TM-1, a 16-bit word is placed in the S16D data-in bus. The four most significant bits (MSB) of this 16-bit word are used to determine the destination address (either one of the six data ports or one of the six configuration registers) and the remaining 12 least significant bits (LSB) are used to transfer the actual data. When the computer wants to read from the TM-1 board, it places the address of the port to read in the four most significant bits of the 16-bit data-in bus and then it reads the actual data from the 12 least significant bits of the 16-bit data-out bus.

In order to synchronize the transfers between the interface board and the TM-1 board (if needed), a software program is required to implement the communication protocol between these two boards. For example, if the computer wants to write 12 bits of data to the TM-1 board, it can use one port as control. In this way, the computer can send a logical one as a ready signal through one of the pins and then poll a second pin until it receives an acknowledge signal from the TM-1 board. At this point, the computer can send the 12-bits of data through a second port.

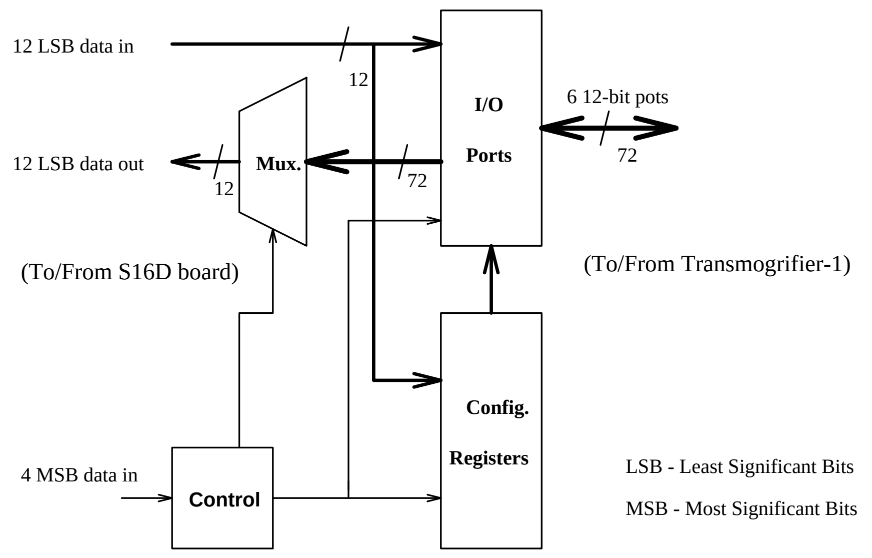

Figure 3.4: 72-bit programmable bidirectional interface.

The actual schematics of this circuit as well as instructions for its use are presented in Appendix A. This circuit was tested by several TM-1 users as described later in Section 3.3. The 72-bit programmable bidirectional interface can achieve transfer rates up to $320K$ *12-bit transfers per second.*

16-Bit DMA Interface

The host computer can also use **read()** and **write()** system calls to read or write large blocks of data (using DMA — direct memory access) from or to the S16D board. The 16-bit DMA interface circuit, illustrated in Figure 3.5, implements the interface handshake between the S16D board and the TM-1 board.

The interface board's circuit, as illustrated in Figure 3.5, has a control block, created with VHDL [30], with a small state machine that performs the necessary handshaking between the S16D and the interface boards The handshaking between these two boards is as follows [28]:

1. The S16D sends the DMA direction through the DMAINPUT line.
2. To input data to the S16D, the interface board pulses low the DCLK signal as soon as the data is available in the DIN bus.
3. The output data from the S16D is valid on the DOUT bus $40ns$ after OUTVALID goes low. The interface board pulses low the DCLK signal as soon as it accepts the data.

4. If the input FIFO on the S16D becomes full, or the output FIFO becomes empty, the DACK signal stays low until the FIFO or data is again available. The OUTVALID signal pulses high between data words as long as new data is available.

The actual VHDL code that supports the above handshaking as well as the actual schematic are shown in Appendix A. Instructions on how to use the DMA interface are also described in Appendix A.

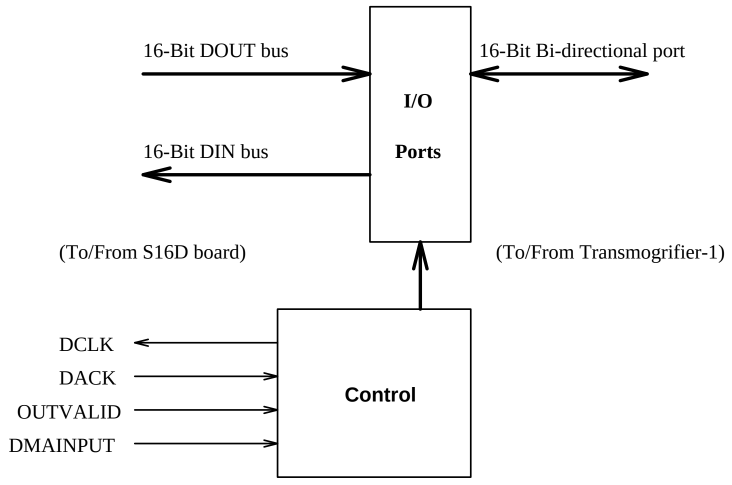

Figure 3.5: 16-bit DMA interface.

The 16-bit DMA interface can be used when unidirectional 16-bit transfers at a rate of equal or less than $7.14 MBytes/sec$ are needed.

3.2 Software Description

The CAD Synthesis tools needed for the TM-1 are a mixture of Xilinx's synthesis tools, Aptix's development tools and custom made Unix scripts.

Figure 3.6 illustrates the design flow used to implement a design on the *Transmogrifier-1*. The TM-1 user begins by creating a Xilinx Netlist Format file (XNF file) [26] of his/her design. This task can be done with schematic capture CAD tools, such as ViewLogic [31], with hardware description languages, such as VHDL [30], or with a mixture of the two.

After a single design is created (one XNF file), the design has to be partitioned into four XNF files, such that each of the subblocks is small enough to fit into a single XC4010 FPGA.

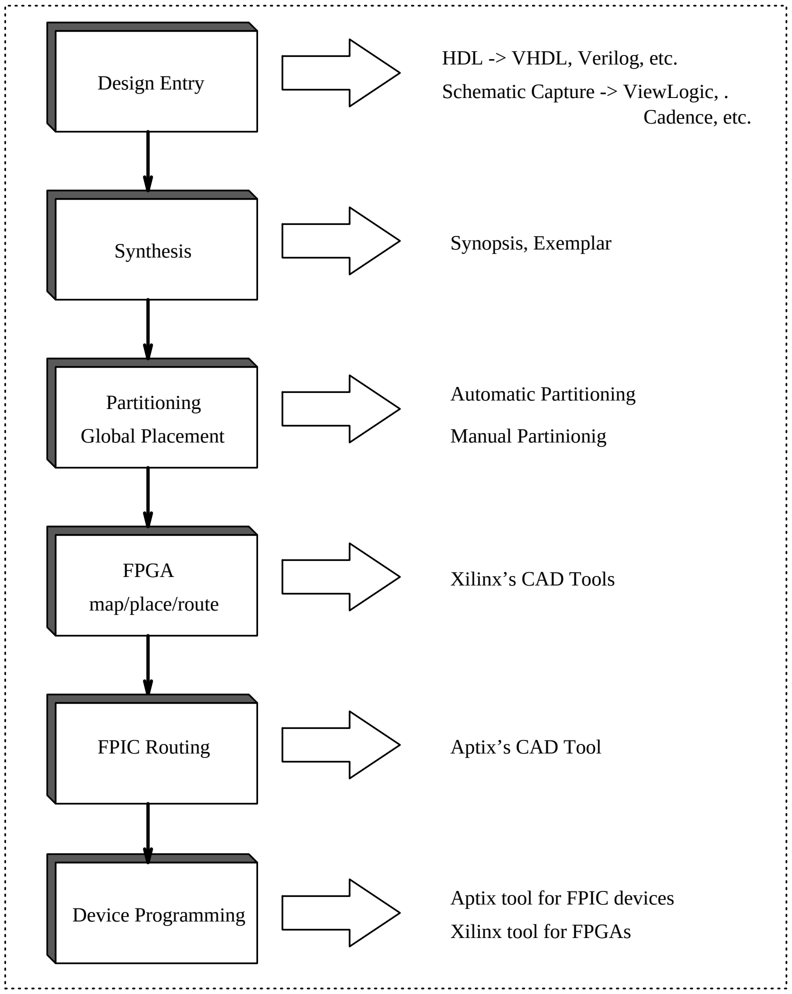

Figure 3.6: The Transmogrifier-1's design flow.

This partitioning can be done manually by creating four individual designs from the very beginning (*e.g.* four individual VHDL files or schematic files), or automatically by using a commercial or university developed partitioner CAD tool [1]. This partitioner must take an XNF file as an input and generate four XNF files small enough to fit into the FPGAs. If one net is broken into two (in order to partition the circuitry), the same name has to be assigned to each divided net so that the FPIC devices can later join them together.

Once a partitioner successfully generates four XNF files, the Xilinx technology map, place-and-route tool (**ppr**) [26] is executed for each of these XNF files. The technology mapper takes the logic circuit and divides it to fit into the available logic blocks in the

[1]A partitioner called **part** is currently under development by Dave Galloway at the University of Toronto [24].

FPGA. Afterwards, the placer decides the best location for the blocks, depending on their connectivity. Finally, the router interconnects the placed blocks. **ppr** also chooses the appropriate I/O pin for every input and output signal [32]. After **ppr**, another Xilinx tool: **makebits** [26], is executed to create the final bitstream files (BIT files) used to program the FPGAs.

Using two custom made unix scripts: **xtopins** and **toaptix**, the pin assignments are extracted from the **ppr** output report file (RPT file [26]). This information is then used to generate an Aptix netlist format file (SCI file [27]) which describes the inter-chip connectivity. This SCI file is then taken by the Aptix **axess** software [27], and used to route the FPIC devices.

Finally, after the FPGA and FPIC devices are ready to be programmed, the Aptix **axess** software is used to program the FPIC devices. The system's power supply is then turned on and the Xilinx **xchecker** software is used to serially program the FPGAs. The **xchecker** software uses a Motorola EXORMAX PROM format (EXO file) [26] as the file to download into the four FPGAs. This EXO file is created by the Xilinx **makeprom** tool [26], using the four bitstream files (BIT files), previously generated by **ppr**.

3.2.1 Custom Made TM-1 programs

As described above, the *Transmogrifier-1* requires a number of commercially available software tools, such as the software provided by both Xilinx [26] and Aptix [27], and a number of custom made software tools. The important programs, developed as part of this thesis are:

xnf2tm1: This Unix script, illustrated in Figure 3.7, is the main program which executes several other programs in order to generate the final files needed to program both the FPGA and FPIC devices.

xtopins: This Unix script uses the Unix **sed** and **awk** filters [33] to extract the pin assignment from the **ppr** report file (RPT file). The output arranges the information in a TM-1 netlist format (NET file) that contains the inter-chip connectivity. The NET file is created as follows: In the first column, the component name (*i.e.* fpga1, fpga2, fpga3

```
for each FPGA do
    {
        Technology map, Place and Route ( ppr )
        Make Bitstream files ( makebits )
        Extract pin locations from RPT file ( xtopins )
    }
Generate SCI file (toaptix: Using the output of the above xtopins )
Generate EXO file ( makeprom )
```

Figure 3.7: xnf2tm1 algorithm

or fpga4) is written. The signal name is written in the second column, and finally the pin number is written in the third column.

```
------------------------------------------------------------------------
                          fpga1   A    C1
                          fpga1   B    B5
                          fpga2   A    C7
                          fpga2   B    A4
------------------------------------------------------------------------
```

Figure 3.8: Example of a NET file

Figure 3.8 shows an example of a NET file created by executing **xtopins** on two files (one for each of two example FPGAs: fpga1 and fpga2). In this example, the shown NET file represents that signal A passes between pin C1 in FPGA No. 1 and pin C7 in FPGA No. 2 and that signal B passes between pin B5 in FPGA No. 1 and pin A4 in FPGA No. 2.

toaptix: This Unix script written by Dave Galloway, also uses some Unix filters to generate the appropriate SCI file. The script takes several NET files as input files (with the same format as the one shown in Figure 3.8) and generates the SCI file in the exact format as described in Appendix C-1 of the Aptix manual [27].

3.2.2 Takeme: TM-1's Design Tool

Even though the *Transmogrifier-1* is very easy to use, the design process, as described above, consist of several steps. In each step, one or multiple commercial or custom made programs have to be executed. Because of this, a front-end program was developed as part of this thesis to help an unfamiliar user design with the TM-1.

This design tool, called **takeme**, was developed to *take* the user through the complete design cycle. The X11 windows based program, written in TCL and TK [34], takes some initial information, such as the name of the design files and the project directory name, and executes the appropriate Xilinx and custom made programs in order to create the final EXO file (needed to program the FPGAs) and the SCI file (needed to route and program the FPIC devices).

The program executes different Unix scripts depending on the number of FPGAs and/or memory chips used. It also allows the user to manually partition, place and route the FPGAs (when manual optimization is required) or to use the partitioner. Appendix B describes a step by step instructions for the **takeme** software and the TM-1 design cycle.

3.3 Example Applications

3.3.1 A Viterbi Decoder

A multiprocessor Viterbi decoder is currently being built at the University of Toronto [35]. Viterbi decoding is an error correcting algorithm that provides a maximum-likelihood decoding scheme for convolutional codes. The system will comprise FPGAs organized to form a ring of 13 switches and 13 processor chips [36]. For each pair of processor chips, there is a single RAM controller controlling the RAM associated with each processor.

The processor and memory system have been implemented on the TM-1 by David Yeh [35]. The test ran at $10MHz$ and was limited by the clock distribution through the interconnect chips. It is important to notice that no real effort was made to increase this clock speed. By giving higher priorities to the critical path and clock nets, the system may have ran at higher speeds.

3.3.2 A Logarithmic Arithmetic Chip

To test the TM-1 as a logic emulator, Jess Lee has implemented a Logarithmic Arithmetic Chip [37]. This chip was previously implemented in a full-custom $1.2\mu m$ CMOS process [38].

In order to map full-custom logic into FPGA logic, parts of the circuit were redesigned. To reduce the amount of routing resources for example, barrel shifters were redesigned out of multiplexers and a *one-hot* encoded state machine became a *binary* encoded one. The algorithm was also redesigned to reduce its complexity at the cost of increased memory size. The original full custom design used 2392×32 bits of memory to store the values of a function $f(x)$. The memory was then used to calculate $f(x-1)$, $f(x)$, and $f(x+1)$. In the TM-1 implementation, three 3292×32 memories are used to store the actual $f(x-1)$, $f(x)$, and $f(x+1)$ function values. It is important to note, however, that the memory cost was much less in the TM-1 design, due to the available SRAM.

The TM-1 implementation uses all four FPGAs and all four memories. It has been simulated with a $5MHz$ clock rate. The low speed is due to the internal FPGA's delays rather than to the use of programmable interconnects. A 19×8 bit multiplier, which is used twice per clock cycle, is the critical path of this implementation.

If we assume that the Xilinx XC4010 FPGA has a die size of approximately $144mm^2$ in a $0.8\mu m$ technology [2], then we can calculate how many times the TM-1 implementation is larger than the full custom implementation. Knowing that the full-custom implementation has a die size of $10mm^2$ in a $1.2\mu m$ technology, we can calculate the size ratio with the following equation,

$$\frac{Size_{TM-1}}{Size_{Chip}} = \left(\frac{1.2\mu m}{0.8\mu m}\right)^2 \cdot \left(\frac{4 \times 140mm^2}{10mm^2}\right) = 129.6 \tag{3.1}$$

Speed-wise, the TM-1 implementation simulated at $5MHz$, which is 10 times slower than the $50MHz$ full-custom design. It is important to notice that the actual implementation has not been fully operational, but is expected to work shortly.

[2]Xilinx does not provide die sizes in their documentation.

3.4 Summary

In this chapter, the *Transmogrifier-1* Field-Programmable System built at the University of Toronto as part of this thesis, was described. The complete *Transmogrifier-1* system consists of:

- The TM-1 board which consists of four Xilinx XC4010 FPGAs, four $32k \times 9$ SRAMs, two Aptix FPIC devices and one Aptix Field-Programmable Circuit Board.
- The interface board which consists of one Xilinx XC4010 FPGA that acts as a communication controller. The FPGA can be configured as a 72-bit programmable bidirectional interface or as 16-bit DMA interface. Both circuits communicate to a commercial board (S16D) connected to a Sparc workstation.
- Several Unix scripts (**xtopins**, **toaptix** and **xnf2tm1**) that help experienced users implement a circuit on the TM-1.
- The front-end software program, called **takeme**, which helps an unexperienced user implement and test a circuit on the TM-1.

Two example applications have been implemented on the TM-1 board: part of a multiprocessor Viterbi decoder was successfully tested with a system clock rate of $10MHz$ and a logarithmic arithmetic full-custom chip was emulated using the TM-1 board. This last circuit did not work completely because of an error with the memory controls. A working implementation is expected shortly.

Chapter 4

The Memory Packing Problem for Field-Programmable Systems

As mentioned in Chapter 1, memory is an essential part of almost any digital system, and so forms a key element of a Field-Programmable System.

In this chapter, a method to efficiently map an application circuit's required set of memories into the available physical memories on an FPS is presented [39]. The context is a *low-cost* FPS in which there is a small number of pre-fabricated physical memories, but a large number of desired memories. The physical memories must be time-multiplexed to create the desired memories.

The mapping becomes an optimization problem when the number of required memories exceeds the number of physical memories. There may be many different ways that the required (logical) memories can be packed into the physical memories. If the logical memories have different access time requirements, the optimization problem is to find a packing that meets the timing requirements. Furthermore, different packings will require different amounts of multiplexing and control, and so it is desirable to minimize the amount of logic devoted to this part of the circuit.

Note that in some cases, a logical memory may be larger than an individual physical memory, and so will have to be partitioned into smaller pieces before it can be packed.

The multiplexer and controller that implements the packing will be referred to as a *Memory Organizer*. In a Field-Programmable System this circuit would be implemented using the FPGAs. This chapter presents an algorithm to solve the optimization problem and describes a CAD tool which uses this algorithm to generate the memory organizer circuit. This tool has been used on the Field-Programmable System described in Chapter 3.

This chapter is organized as follows: Section 4.1 provides a precise definition of the optimization problem to be solved. Section 4.2 describes the memory organizer architecture, and the models needed to estimate the organizer speed and area. Section 4.3 gives an algorithm for the solution of the optimization problem. Section 4.4 presents several examples of the use of the CAD tool based on this algorithm, and measurements corroborating the models.

4.1 Problem Definition

In this section we first describe our notation and then give a precise definition of the Field-Reconfigurable Memory packing problem.

There are three key parameters that characterize any memory: The *width* is the number of bits per memory word, the *depth* is the number of words in the memory, and the *access time* is the minimum amount of time that must pass between two consecutive memory access requests.

We will refer to each memory required by the user as a *Logical Memory*, abbreviated as LM. Each fixed available memory on the FPS is referred to as a *Physical Memory* (PM). There are n logical memories in the set $L = \{LM_0, \ldots, LM_{n-1}\}$ and m physical memories in the set $P = \{PM_0, \ldots, PM_{m-1}\}$. The width, depth and access time of the j^{th} PM are denoted by Pw_j, Pd_j and Pt_j, and the width, depth and required access time of the i^{th} LM are denoted by Lw_i, Ld_i and Lt_i. If a logical memory is larger in depth or width than the physical memory, then it must be partitioned into pieces. Section 4.1.1 shows how this partitioning is done.

A *packing* is a partition of the logical memories together with a mapping from each of the partitioned logical memories to one of the physical memories.

The *occupancy* of a physical memory in a packing is the number of logical memories that are mapped into the physical memory. For example, the occupancy of the topmost physical memory in Figure 1.4b is three. We denote the occupancy of PM_j by OC_j.

A *legal packing* is a packing which meets the access times required by each of the logical memories. The access time of a realized logical memory is a function of the occupancy of the physical memory in which it resides, and of the delay introduced by the organizer multiplexer. If we denote the *organizer delay* of logical memory LM_i by OD_i, then a packing is legal if the following holds for all logical memories LM_i packed into physical memory PM_j:

$$Lt_i \leq OC_j \cdot Pt_j + OD_i \tag{4.1}$$

If a logical memory is partitioned into smaller pieces in order to fit into a physical memory, then this access time requirement is also placed on the individual pieces.

The *area* used by the organizer is the amount of logic used in the FPGAs to implement the multiplexing hardware and control. It depends on the number and size of the multiplexers in the organizer, which in turn depends on the occupancy of each physical memory, and the width and depth of the logical memories.

For example, suppose that six logical memories of the same width and depth are to be packed into two physical memories. Assume there are only two legal packings, one that maps four logical memories into one physical memory and two logical memories into the other physical memory (i.e. $OC_0 = 4$ and $OC_1 = 2$). The other legal packing maps three logical memories into one physical memory and three logical memories into the other physical memory ($OC_0 = 3$ and $OC_1 = 3$). Assume also that both a *2 to 1* and a *3 to 1* multiplexer can be implemented in a single FPGA logic block, but a *4 to 1* multiplexer requires two logic blocks, as is the case for a Xilinx 2000 series logic block. Under these assumptions, the $OC_0 = 4$, $OC_1 = 2$ solution requires a larger area (3 logic blocks per multiplexed bit) than the $OC_0 = 3$, $OC_1 = 3$ solution which requires 2 logic blocks per multiplexed bit.

With this notation, we can state a general version of the **Field-Reconfigurable Memory packing problem:**

Given: The sets $L = \{LM_0, \ldots, LM_{n-1}\}$ and $P = \{PM_0, \ldots, PM_{m-1}\}$ and associated width, depth and access time parameters,

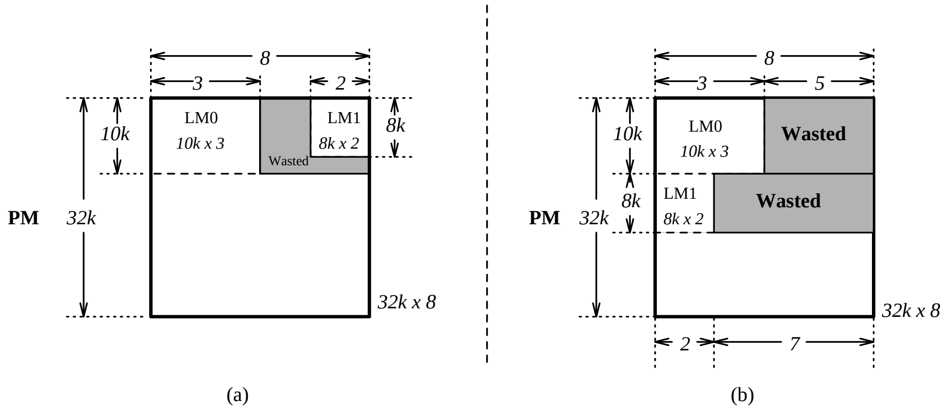

Figure 4.1: Contrast in Efficiency With and Without Word Sharing.

Find: A legal packing of L into P, which minimizes the area of the organizer.

4.1.1 Practical Issues and Assumptions

To both create a practical organizer and to simplify the description in this chapter, we make the following assumptions about the logical and physical memories:

1. **Homogeneous Physical Memories**

   In the above definition each physical memory is allowed to be different. For simplicity we will assume all physical memories are identical. The notation for the width, depth and access time of all physical memories thus simplifies to Pw, Pd and Pt.

2. **Prevention of Word Sharing**

   Word sharing occurs when two or more logical memories share the same word in a physical memory. We will not allow this because it would cause a memory *write* to become significantly slower and more complex — since part of a word has to be preserved, each write must be preceded by a read to record the un-altered portion of the physical memory's word. Note that this assumption implies that more of the physical memory will be wasted, as illustrated in Figure 4.1. Figure 4.1a shows an example packing with word sharing and Figure 4.1b shows an example packing without word sharing. The latter uses a greater portion of the physical memory.

   To implement this restriction, we re-define the width of the logical memory to be equal to the next nearest multiple of Pw. We call this adjusted width, Lw_i', and it is given by,

$$Lw'_i = \left\lceil \frac{Lw_i}{Pw} \right\rceil \cdot Pw \tag{4.2}$$

3. **Logical Memory Depth Restricted to be a Power of Two**

   If the depth of each logical memory is anything other than a power of two, the addressing circuitry connecting the logical address to the physical memory address bus will require physical adders to create the correct offsets into the physical memory space. Since adders are expensive in space and time, we restrict all logical memory's depths to be a power of two.

   If we restrict the offset to be a power of two value, we can replace the adders by simply including extra bits (*i.e.* logical ones or zeros) in the logical memories' address bus. We implement this restriction by re-defining a depth for each logical memory, Ld'_i, given by,

$$Ld'_i = 2^{\lceil \log_2 Ld_i \rceil} \tag{4.3}$$

   Note that this assumption also has the potential to waste part of the physical memory.

4. **Logical Memory Partition Limit**

   When a logical memory is bigger than a physical memory, it is necessary to partition the logical memory into smaller pieces. For simplicity, we will partition logical memory LM_i only if:

$$(Lw'_i > Pw) \text{ or } (Ld'_i > Pd) \tag{4.4}$$

   In this case LM_i is partitioned into exactly nb_i pieces where nb_i is given by:

$$nb_i = \frac{Lw'_i}{Pw} \cdot \left\lceil \frac{Ld'_i}{Pd} \right\rceil \tag{4.5}$$

   From this point on we will assume that all logical memories have been partitioned in this way, and we will refer to the pieces simply as logical memories. Note that the CAD tool that generates the organizer does in fact generate the control and multiplexing to handle the larger memories.

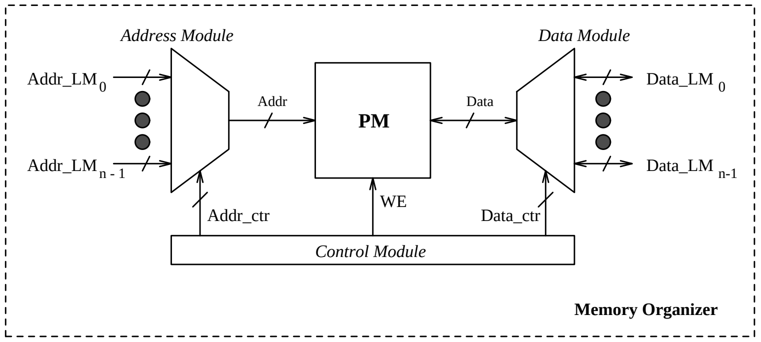

Figure 4.2: Block diagram of the Memory Organizer.

4.2 Organizer Architecture and Area/Delay Models

In the introduction of this chapter we discussed how different packings of the same set of logical memories could require different amounts of area to implement the Organizer. In this section, we describe the general structure of the organizer and derive models for the area and delay of the organizer hardware as a function of the occupancy of the physical memories and the width and depth of the logical memories. These models are used in the packing algorithm described in Section 4.3.

In the following discussion we describe area and delay models for an organizer for a single physical memory and some number of logical memories.

As illustrated in Figure 4.2, the organizer is divided into three modules: The Address module, which organizes the address busses, the Data module, which organizes the data busses and the Control module, which generates the timing and multiplexer control signals.

Some aspects of the model are independent of the FPGAs and memories in the actual Field-Programmable System. These *technology-independent* aspects will be described first. After describing the specific FPS that we have built, we will use it to give a technology-specific delay model.

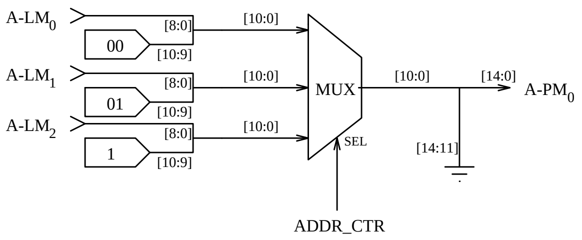

Figure 4.3: Address Module for two 512×8 LMs and one $1k \times 8$ LMs

4.2.1 Technology-Independent Area Model

Address Module

When more than one logical memory is mapped into a single physical memory, the logical memories' address buses are multiplexed to form a single physical memory address. The address module organizes these busses so that accesses to the logical memories are mapped to non-overlapping portions of the physical memory.

Figure 4.3 illustrates the address module multiplexer for an example with two 512×8, and one $1k \times 8$ logical memories packed into a single physical memory. The $ADDR\_CTR$ bus in the figure is generated in the control module.

The area required for this multiplexer depends on the occupancy of the physical memory and the width of the maximum offset of the logical memories' address buses. If we denote by Lo_i the offset (*i.e.* the based memory address) of the i^{th} LM mapped into the PM, the area of the address multiplexer for physical memory PM_j is given by:

$$A_{Addr} = \left[1 + \max(\log_2(Lo_0), \log_2(Lo_1), \ldots, \log_2(Lo_{OC_j-1}))\right] \cdot MUX(OC_j) \tag{4.6}$$

Here the function $MUX(x)$ returns the number of logic blocks needed to implement a one bit *x to one* multiplexer. The function depends on the type of FPGA being used. It returns a larger number as the occupancy of the physical memory increases and therefore depends on the packing.

Equation (4.6) illustrates that different packings will result in a different sized organizer: If there are wide address buses and thin address buses in the set of logical memories, it

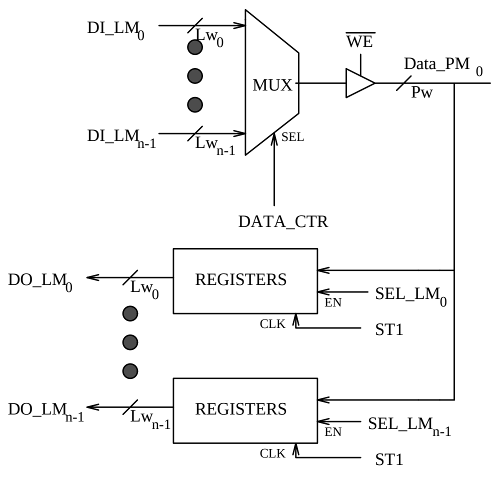

Figure 4.4: Block diagram of the Data Module

is better to pack the wide addresses together and the thin addresses together because the address module area is determined by the maximum width bus in the physical memory.

Data Module

The data bus multiplexing is more complex than the address bus as illustrated in Figure 4.4. For *write* accesses, the logical memories' data buses are multiplexed and passed through a tri-state driver into the physical memory's data bus. For *read* accesses, it is necessary to capture each logical memory's read data in a separate register, as illustrated in the figure. The $DATA\_CTR$, SEL_LM_i and *ST1* signals are generated by the Control module.

Each bit in the data bus of all logical memories is multiplexed independently. For example, consider the following three logical memories: one $1k \times 3$, one $1k \times 6$ and one $1k \times 8$. In order to multiplex the data buses, we need a *3 to 1* multiplexer for data bits 0, 1 and 2 of all the logical memories and a *2 to 1* multiplexer for data bits 3, 4 and 5 of the $1k \times 6$ and $1k \times 8$ logical memories. Data bits 6 and 7 of the $1k \times 8$ logical memory are connected directly to the physical memory. Figure 4.5 gives a block diagram for this example.

To calculate the area of the data multiplexer in the data module, we first sort the width of the logical memories so that $Lw_0 \leq Lw_1 \leq \ldots \leq Lw_{n-1}$. To minimize the area, we first multiplex the first Lw_0 data bits of all the logical memories. Then we multiplex the $Lw_1 - Lw_0$ data bits of $LM_1, \ldots, LM_{n-1}$ and so on. The area of the organizer depends on the occupancy of the physical memory and the width of the logical memories. This is given

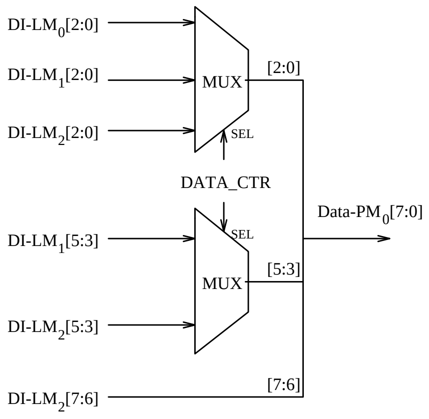

Figure 4.5: Data Multiplexer for a $1k \times 3$, a $1k \times 6$ and a $1k \times 8$ LMs

by:

$$A_{Data\_Mux} = \sum_{k=1}^{OC_j - 1} (Lw_k - Lw_{k-1}) \cdot MUX(OC_j - k + 1) \tag{4.7}$$

The technology-dependent function MUX is equivalent to the one used in Equation (4.6). Equation (4.7) also illustrates that different packings will result in a different sized organizer: If the logical memories consists of both wide and thin data busses, it is more area-efficient to pack thin busses with wide busses, because most of the wide bus part will not require multiplexing. On the other hand, if the wide busses are packed together, then all bits will have to be multiplexed.

The area of the data registers in the data module only depends on the width of the logical memories. Hence, it is independent of the packing. This area is given by:

$$A_{Data\_Reg} = \sum_{k=0}^{OC_j - 1} REG(Lw_k) \tag{4.8}$$

Here $REG(x)$ is a technology-dependent function that returns the number of logic blocks needed to implement a register of x bits.

Control Module

The Control module consists of a finite state machine and decoders that generate the timing, clocking and multiplexer control signals.

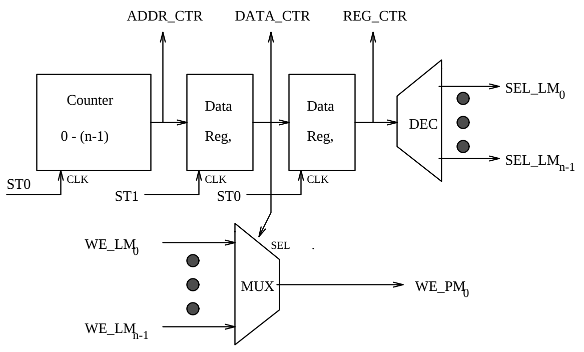

Figure 4.6: Block diagram of the Control Module.

Figure 4.6 shows the block diagram for the Control module. The state signals ST0 and ST1 are generated centrally; the same signals are used for all the physical memories. The control is pipelined as described below:

ST0. In this first state, the counter in Figure 4.6 is clocked, generating the ADDR_CTR bus used to multiplex the address buses. In this same state, the physical memory's clock [1] latches both the data bus and control signals (*i.e.* write enable, input and output enable) of the previous memory access (*i.e.* the ones generated in ST1).

ST1. In this second state, the first data register in Figure 4.6 is clocked, generating the DATA_CTR bus used to multiplex the data buses. In this same state, the physical memory latches the address bus generated at ST0. In case of a memory read access, the data registers in the data module latches the data coming from the physical memory (of the previous memory access).

The area of the controller is given by:

$$A_{Control} = COUNTER(OC_j) + 2 \cdot REG(\log_2 OC_j) + DEC(OC_j) \tag{4.9}$$

Here $COUNTER(x)$ is a function that returns the number of logic blocks needed to implement a 0 *to* $x - 1$ counter and $DEC(x)$ is a function that returns the number of logic blocks needed to implement a $\log_2 x$ *to* x decoder. All of these functions return larger numbers as the occupancy of the physical memory increases.

[1]The TM-1 uses four Motorola MCM62110 Synchronous RAM chips: the address bus is latched at the falling edge of the clock signal and the data bus and control signals are latched at the rising edge of the clock signal.

4.2.2 Area Model Accuracy

Figure 4.7 presents a comparison between the area model presented in Section 4.2.1 and the actual area used in the FPGA. The actual area used is taken from the output of the Xilinx's technology mapping tool) after generating the Organizer. In the graph shown in this figure, the x-axis represents the number of logical memories packed into a single PM. The y-axis represents the number of logic blocks that the organizer needs to multiplex the logical memories.

As shown in the graph, the model is very accurate. The discrepancies are due to the fact that the model uses manual technology mapping, which can give different results than the automatic technology mapping of **ppr**. For example, when doing manual mapping, we decide how many logic blocks are needed to implement a multiplexer and the result is always the same. With automatic technology mapping, the multiplexer is first translated into a binary equation and then combined to all the other equations to form a large equation. An algorithm then decides how to map this equation into the logic blocks. The same multiplexer that can manually be mapped into one logic block, may be mapped into two separate logic blocks with an automatic technology mapper.

It is important to note that **ppr** itself produces little differences each time is executed. The graph shows the average logic blocks used in five **ppr** runs.

4.2.3 Technology-Dependent Delay Model

In Chapter 3, the *Transmogrifier-1* Field-Programmable System built at the University of Toronto was presented. As described, the TM-1 consists of four Xilinx's 4010 FPGAs [11], two Aptix's Field-Programmable Interconnect Components (FPIC) [27] and four Motorola's MC62110 $32k \times 9$ Synchronous SRAM chips [25]. Figure 3.2 gives a block diagram of the TM-1. In this section, the TM-1 is used to create a technology specific delay model.

Finding an exact delay model for the Organizer is difficult because the circuits implemented in most FPGAs have delays that are hard to predict. We now present a worst-case delay model for our implementation in the TM-1.

The control module uses two states for every physical memory access and so the best

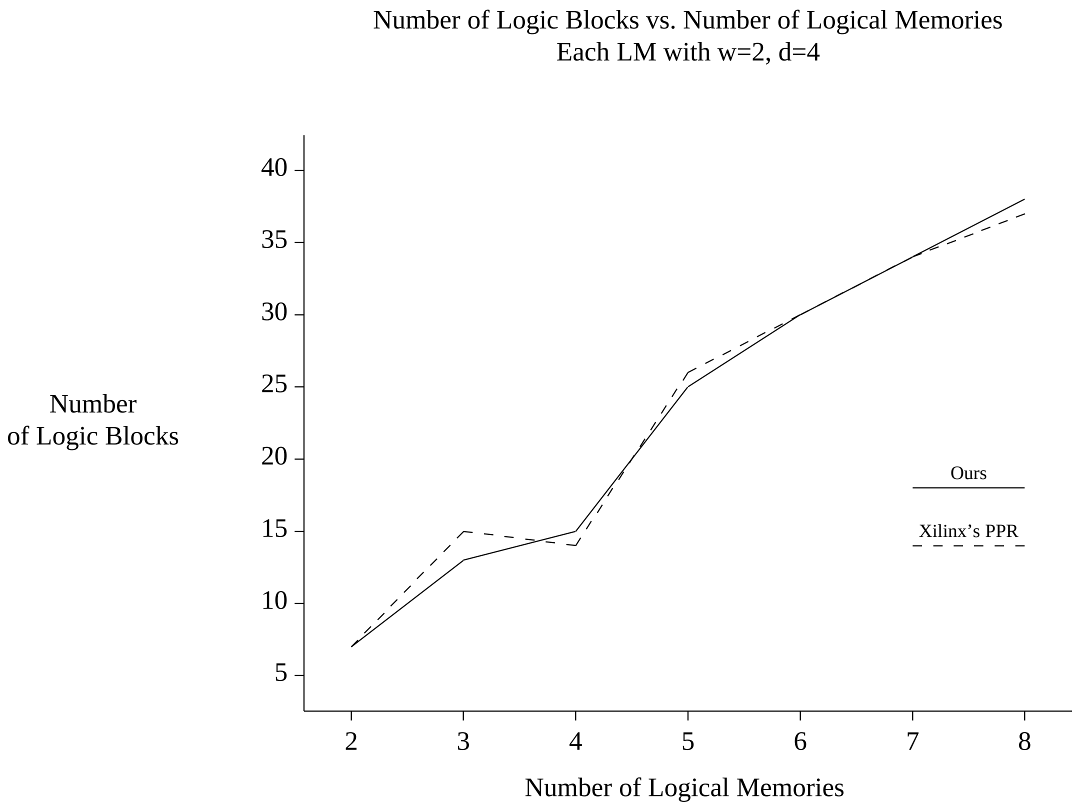

Figure 4.7: Our area model vs. Xilinx ppr's area model.

possible access time for LM_i packed in PM_j is:

$$Lt_i = 2 \cdot (OC_j \cdot Pt + OD_i) \tag{4.10}$$

For the TM-1 FPS, the access time of the physical memories, Pt, is equal to $20ns$. The delay due to the programmable interconnect component incurs an additional $15ns$. Thus, the physical memory access time is $35ns$.

The Organizer Delay for LM_i, OD_i, is a function of the packing. It depends on the number of logic blocks in the critical path and the routing needed to connect these blocks. For simplicity, we assume that the routing delay is constant and that all the logical memories mapped into one physical memory have the same delay. We denote by OD_j, the organizer delay of all the logical memories mapped into PM_j. From Figures 4.3 and 4.6, if we assume that the critical path includes the counter in the local control module and the multiplexer in the address module, then the Organizer delay for PM_j is given by,

$$OD_j = OBUF + COUNTdelay(OC_j) + MUXdelay(OC_j) + SMdelay \tag{4.11}$$

Here $OBUF$ is the delay of the output buffer, $COUNTdelay(x)$ and $MUXdelay(x)$ are the delays for a *0 to* $x - 1$ counter and an x *to 1* multiplexer, and $SMdelay$ is the delay of the state machine. If we assume $8ns$ delay per Xilinx XC4000 family logic block, (*i.e.* the $6ns$ nominal delay plus $2ns$ for routing), then $COUNTdelay(x)$ and $MUXdelay(x)$ have a worst-case delay of $8ns$ each for $x = 2, 3, 4$ and of $16ns$ for $x = 5, 6, 7, 8$. $OBUF$ is approximately equal to $10ns$ and the worst-case delay of the state machine is approximately $8ns$.

With the above data and Equations (4.10) and (4.11), the final delay model is given by,

$$Lt_i = \begin{cases} 2 \cdot OC_j \cdot (35 + 34), & \text{if } OC_j = 2, 3, 4 \\ 2 \cdot OC_j \cdot (35 + 50), & \text{if } OC_j = 5, 6, 7, 8 \end{cases} \tag{4.12}$$

This model assumes that the entire Organizer is placed in a single FPGA, and not split across multiple FPGAs. This is a reasonable assumption, as the organizer is small.

4.2.4 Delay Model Accuracy

Figure 4.8 presents a comparison between the delay model presented in Section 4.2.3 and the actual delay used in the FPGA. The actual delay used is taken from the output of the Xilinx's technology map, place and route tool (taken from PPR [2]) after generating the Organizer. In the graph shown in this figure, the x-axis represents the number of logical memories packed into a single PM. The y-axis represents the delay in nanoseconds that the organizer needs to multiplex the logical memories. The difference between the two graphs is due to the fact that the delay model does not take into account the routing delays. It just adds a $2ns$ delay to the nominal $6ns$ logic block delay. However, the error in the model is less than 10%.

It is important to notice that the graph only gives the delay inside the FPGA, as opposed to the total delay: FPGA delay plus FPIC delay plus physical memory delay. Therefore, to

[2]PPR was run five times to determine the average delay.

calculate the total organizer delay, the following equation needs to be used:

$$Lt_i = 2 \cdot OC_j \cdot (35 + FPGA_{delay}) \tag{4.13}$$

where OC_j is the number of logic memories and $FPGA_{delay}$ is the delay shown in Figure 4.8 for the OC_j number of logic memories.

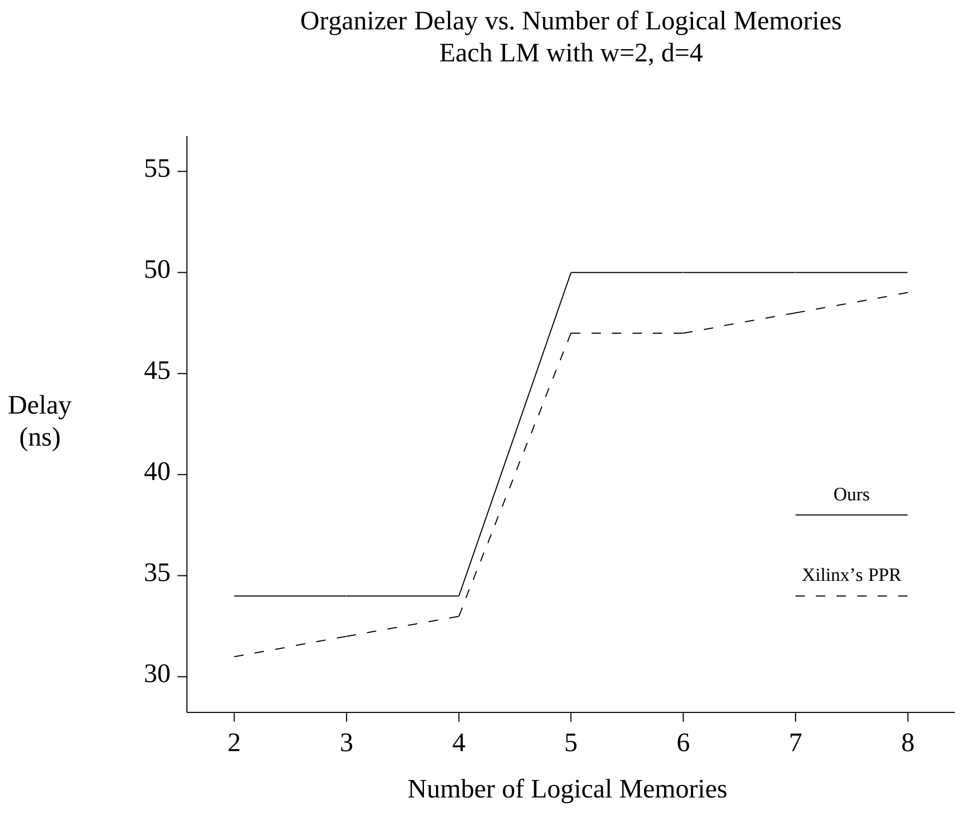

Figure 4.8: Our delay model vs. Xilinx ppr's delay model.

4.3 Memory Packing Algorithm

In this section, an exhaustive algorithm which solves the constrained problem defined in Section 4.1 is presented. A fast heuristic algorithm is also described at the end of this section.

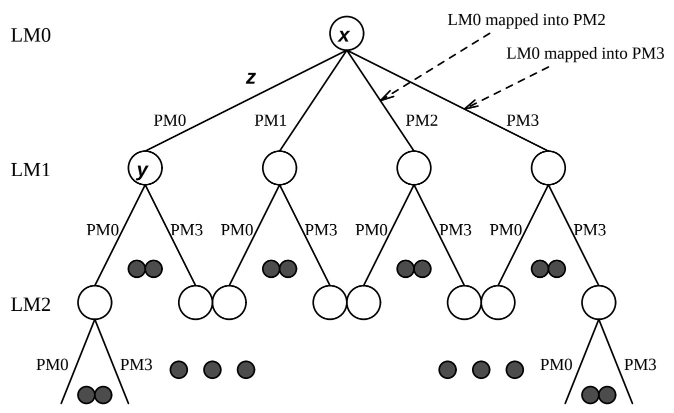

Figure 4.9: Decision Tree of Packing Solutions for problem with 3 LMs and 4 PMs.

4.3.1 Exhaustive Algorithm

The memory mapping problem can be solved using a Branch and Bound approach [40]. Figure 4.9 illustrates the branch and bound decision tree that represents all the possible solutions to the memory mapping problem. The nodes represent the logical memories. The edges represent the physical memories into which the logical memories are packed. For example, Figure 4.9 has two nodes, x and y joined by edge z. The partial solution represented by node y means that LM_0 was mapped into PM_0.

The decision tree is traversed depth-first from the root. Pruning occurs in the following manner: assume that node x is the current node and has child node y connected by edge z. The tree is pruned at node y if LM_x cannot be mapped into PM_z, which occurs when any one of the following is *not* true:

1. LM_x physically fits in the size remaining in PM_z.
2. After placing LM_x into PM_z, the required access time of LM_x (Lt_x) is achieved.
3. By placing LM_x into PM_z, the logical memories already placed into PM_z also meet their required access times.

If the bottom of the tree is reached then a legal packing is found. The algorithm continues to traverse the tree to find the solution with the minimal area as determine by the area model described in Section 4.2.

The tree is also pruned using a bounding function on the area of the partial solutions: this lower bound is calculated as the area needed to implement an Organizer for all the logical memories already mapped in the sub-tree above the current node. If there are n logical memories and m physical memories, the worst-case complexity of the algorithm is $O(m^n)$. However, in most of the cases, a large portion of the tree is pruned and only a small fraction of the tree is visited. Table 4.1 gives the required memories of a few example applications: Column one gives the required logical memories and columns three and four give the total number of nodes in the tree and the number of visited nodes. Column six gives the CPU time — the time a SUN Sparc I workstation took to execute the program. Note that although the number of visited nodes is smaller than the total number of nodes (*i.e.* m^n), it still grows exponentially with the number of logical memories. Therefore, the execution time also grows exponentially.

Required Logical Memories	Number of Subdivided Logical Memories	Total No. of Nodes	Visited No. of Nodes	CPU Time
three 28x16 one 28x3 LIFO	7	16,384	398	0.11s
15x24, 16x4 (RAM) 256x32 (ROM)	8	65,536	818	0.21s
2048x56 (ROM) 4096x12 (ROM)	9	262,144	7,708	3.69s
16x80, 160x8 16x16, 32x8	14	2.68×10^8	4.60×10^6	3601.52s

Table 4.1: Total and visited number of nodes in the Branch & Bound tree.

4.3.2 Heuristic Algorithm

In this section, a heuristic algorithm is presented. This algorithm employs a *mapping priority function* (described later in this section) to order all logical memories, and then it takes one LM at a time and maps it into a PM.

As described in Section 4.1, the goal of the memory packing problem is to fit the logical memories within the physical memories, meet the required access times, and minimize the area of the multiplexing hardware. The goals of fitting the logical memories and meeting their access time may be in direct conflict because they are completely orthogonal. To create the prioritized list of logical memories for packing, a mapping priority function must be derived from the two competing goals. Although it is difficult to determine, in advance, which goal (fitting or meeting) will be the hardest to achieve, we calculate a "normalized" measure for each goal, and sum the two normalized values to create the mapping priority function.

The first measurement, called *normalized depth*, is calculated by dividing the depth of the LM by the depth of the PM and then multiplying it by $\lceil n/m \rceil$, where n is the total number of logical memories, and m is the total number of physical memories. Therefore, $\lceil n/m \rceil$ is the number of logical memories that need to be packed into each PM to achieve a balanced packing — a legal packing in which each PM has the same number of logical memories. More formally,

$$Normalized\ depth = \frac{Ld_i}{Pd} \cdot \left\lceil \frac{n}{m} \right\rceil \tag{4.14}$$

The second measurement, called *normalized access time* is calculated by dividing the access time of the PM by the required access time of the LM and then multiplying it by $\lceil n/m \rceil$. More formally,

$$Normalized\ access\ time = \frac{Pt}{Lt_i} \cdot \left\lceil \frac{n}{m} \right\rceil \tag{4.15}$$

Using the normalized depth and normalized access time, the *mapping priority function* is given by,

$$Priority_i = \exp\left(\frac{Ld_i \cdot \lceil \frac{n}{m} \rceil}{Pd}\right) + \exp\left(\frac{Pt \cdot \lceil \frac{n}{m} \rceil}{Lt_i}\right) \tag{4.16}$$

Note that an exponential form is given to the mapping priority function to emphasize the terms that are equal to or greater than one. The exponential form was empirically found to be better than the linear form. With this equation, for example, an LM with normalized depth equal to 1.8, and normalized access time equal to 0.3, is mapped before an LM with both normalized depth, and normalized access time equal to 1.1.

Also note that, if all logical memories have both normalized depth and normalized access time less than or equal to one, then a balanced packing can easily be found by simply packing $\lceil n/m \rceil$ logical memories into each PM. If this is not the case, then an algorithm is needed to some how try to map all the logical memories into the physical memories: The *First-Fit* (FF)[41] algorithm takes one LM at a time (in order from the prioritized list), and maps it into the first PM in which it fits. A legal packing found with this algorithm tends to result in an unbalanced packing — a legal packing in which each PM has a different number of logical memories. Furthermore, the *FF* algorithm will try to map all logical memories into a single PM if is possible.

Improving the heuristic for area minimization

The algorithm described above does not try to find the best possible organizer's area. This is because, as described in Section 4.1, an unbalanced packing sometimes results in a better area but most of the time does not.

```
Heuristic Algorithm
{
    // $n$ is the total number of logical memories
    // $m$ is the total number of physical memories
    Best Area = 0;
    $i = 0$;
    Repeat $n$ times:
        $Priority[i] = \exp\left(\frac{Ld[i]\cdot\lceil\frac{n}{m}\rceil}{Pd}\right) + \exp\left(\frac{Pt\cdot\lceil\frac{n}{m}\rceil}{Lt[i]}\right)$;
        $i = i + 1$;
    Sort_Down all the $Priority$ values;
    $i = 1$;
    Repeat $n$ times:
        Pack-FF $(LM[0]\ldots LM[i-1])$;
        Pack-BF $(LM[i]\ldots LM[n-1])$;
        if Best Area $\geq$ Area of this packing
            $\Longrightarrow$ Best Area = Area of this packing;
        $i = i + 1$;
}
```

Figure 4.10: Pseudo Code for the heuristic algorithm

The heuristic algorithm shown in Figure 4.10, tries to explore both the balanced and unbalanced legal packing by using a combination of the *First-Fit* (FF) algorithm, and an algorithm we call *Balanced-Fit* (BF) [3]. The *BF* algorithm takes one LM at a time and maps it into the PM that fits and has the least number of logical memories.

As shown in Figure 4.10, the algorithm iterates n times (where n is the number of logical

[3]Not to be confused with *Best-Fit*

memories). In the first iteration, the algorithm only uses BF to pack all the logical memories. In the second iteration, the algorithm packs one LM using FF and the rest using BF. In the i^{th} iteration, the algorithm uses FF to pack the first i logical memories and BF to pack the rest (*i.e.* $n - i$ memories). At the end, the algorithm uses only FF to pack all logical memories. From all the legal packings, the one with minimum area is chosen as the solution.

Since it was not the primary goal of this thesis to find a heuristic for the memory packing problem, there has not been enough analytical and practical investigation to determine if this algorithm can always find a legal packing, if one exists. It is important to notice however, that this heuristic algorithm has worst-case complexity of $O(2 \cdot n \cdot m)$. This means that the heuristic can run orders of magnitude faster than the exhaustive algorithm which has worst-case complexity of $O(m^n)$. Table 4.2 gives the same example applications presented in Table 4.1. Columns three and four presents the area cost given by the exhaustive and the heuristic algorithms. Column five gives the ratio between the time a Sparc workstation takes to run the exhaustive algorithm and the time it takes to run the heuristic algorithm. As the number of logical memories increases, the difference in speed between the two algorithms increases, making the heuristic essential for circuits with large numbers of required logical memories. The CPU time of the exhaustive algorithm is shown in Table 4.1. The final example in Table 4.2 shows a case in which the heuristic algorithm was 45 thousand times faster than the exhaustive algorithm, in finding a packing with only a 9.1% increase in area.

Required Logical Memories	Number of Memory Pieces	Area (Exhaustive)	Area (Heuristic)	Heuristic Speedup
three 28x16 one 28x3 LIFO	7	53	53	1.83
15x24, 16x4 (RAM) 256x32 (ROM)	8	67	67	4.20
2048x56 (ROM) 4096x12 (ROM)	9	96	96	92.25
16x80, 160x8 16x16, 32x8	14	121	132	45,019

Table 4.2: Comparison between the exhaustive and heuristic algorithms.

4.4 Results

The above algorithms have been implemented in a CAD tool called **MemPacker**. It is currently targeted towards the TM-1 FPS described in Chapter 3. The inputs to **MemPacker** are the logical memory parameters (width, depth and access time) and the output is the FPGA design of the organizer. **MemPacker** produces a Xilinx 4000 series netlist format (XNF) file that can be directly synthesized by the native Xilinx tools to produce a programming bit stream. We note that this tool saves the designer a significant amount of time by automatically generating the memory multiplexing logic and control. Manual generation of such circuits may take many hours. Appendix C describes how to use the **MemPacker** program.

4.4.1 Using MemPacker for Architecture Exploration

MemPacker can be used to explore the architectural space of a design from the perspective of the memory access times. It is often true that memory access times are the limiting factor in the overall speed of an application [6]. **MemPacker** can be used to determine the minimum access time for a set of logical memories implemented on a set of physical memories by iterating the algorithm with successively smaller required access times. The iteration prior to the one in which the algorithm fails to find a legal packing gives the minimum access time for all of the memories (using the exhaustive approach). The algorithm will also determine the smallest sized organizer that will achieve maximum performance.

Table 4.3 gives a set of example applications for which the maximum operating frequency of the memories has been determined. In this example, we assume that all of the memories will have the same access time. **MemPacker** is iteratively invoked to determine the smallest possible access time. The first column of Table 4.3 gives the source and/or name of the circuit from which the set of logical memories was derived. The second column describes the set of logical memories. These are packed into the TM-1, which consists of four $32k \times 8$ physical memories. Column three indicates the number of pieces the logical memories are partitioned into, as described in Section 4.1.1. Column four gives the minimum area of the organizer achieved, in terms of the number of Xilinx 4000-series logic blocks. Column five gives the maximum operating frequency achievable if the memory access time is the limiting factor

System	Logical Memories	Number of Subdivided Logical Memories	Area (XC4000 CLBs)	Max Op. Freq
Industrial Example 1	three 736x16	6	52	$3.6MHz$
Viterbi decoder	three 28x16 one 28x3 LIFO	7	53	$3.6MHz$
DMA Chip for LAN	15x24, 16x4 (RAM) 256x32 (ROM)	8	67	$3.6MHz$
Industrial Example 2	six 88x8 one 64x24	9	87	$2.4MHz$
Fast Divider	2048x56 (ROM) 4096x12 (ROM)	9	96	$2.4MHz$
Industrial Example 3	four 368x16 one 736x16	10	109	$2.4MHz$
Neural Network Chip	16x80, 160x8 16x16, 32x8	14	121	$1.8MHz$

Table 4.3: Maximum Operating Frequency and Area for Example Circuits. The Industrial Examples are Telecommunication-related circuits.

in the system performance. This illustrates how **MemPacker** can be used to explore the performance and area costs of different memory architectures.

4.4.2 Illustration of Area Dependency on the Packing

In Section 4.2 we discussed the effect of the packing on both the address and data module area. Here we illustrate these effects by an example. Consider the following set of logical memories: four $4k \times 8$, two 32×3, two 64×2 and two $8k \times 2$. These will be packed into the same physical memories as above.

Using the heuristic algorithm described in Section 4.3.2, these memories would be packed as illustrated in Figure 4.11a. The area cost of this packing is 84 Xilinx 4000-series logic blocks. Using the exhaustive algorithm described in Section 4.3, the packing illustrated in Figure 4.11b results. The area cost of the latter packing is only 78 logic blocks. The reason for the major difference is a better matching of address and data busses within each physical memory to minimize the amount of multiplexing. Note that both packings achieve the same minimum access time over all the memories. If this constraint is relaxed and the minimum area solution is generated, the packing of Figure 4.11c results, which has an area cost of only 66 logic blocks. This solution is 21% smaller than the heuristic solution.

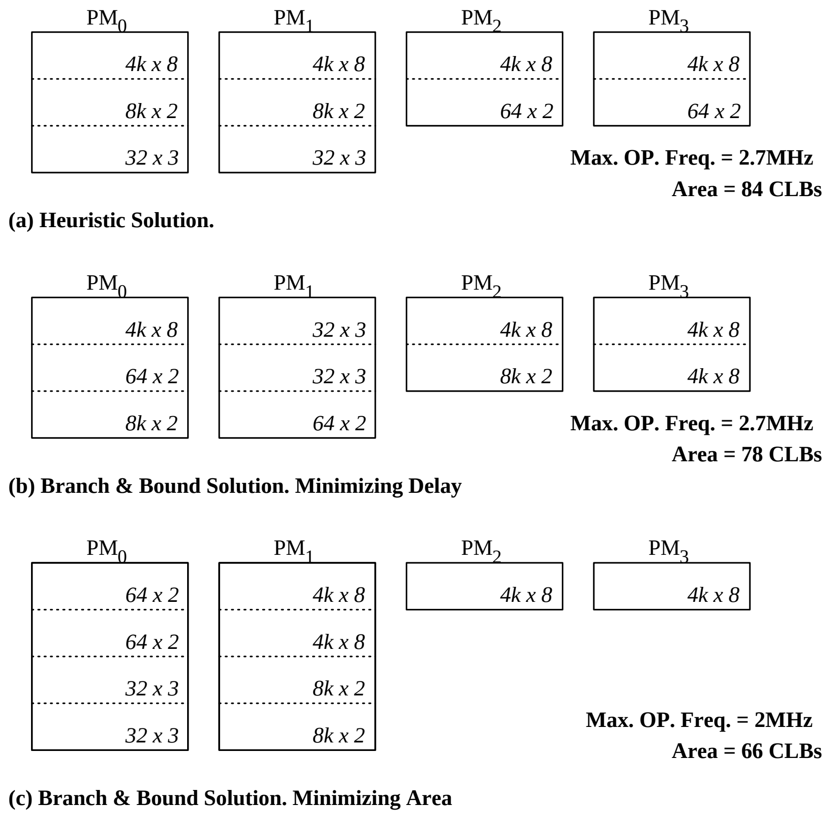

Figure 4.11: Effect of different packings on the Area and Delay Cost.

4.5 Summary

This chapter motivates and defines the memory packing problem for Field-Programmable Systems. Memory packing is necessary when the number of application logical memories exceeds the number of physical memories. Because different packings result in both different access times and area requirements, it is an optimization problem to select the fastest and most area-efficient packing. This chapter has presented a precise definition of this problem and an algorithm for its solution. The resulting CAD tool, **MemPacker** was used to synthesize area-efficient and delay-minimal packings for a set of application circuit examples.

Chapter 5

Conclusions and Future Work

5.1 Summary and Contributions

This thesis makes two contributions. First, a working Field-Programmable System that can be used for instant prototyping of digital circuits was implemented. Both the hardware and software needed to support the system were presented in Chapter 3. This FPS, called the *Transmogrifier-1* (TM-1) has been used to emulate and test some example applications.

Second, a method for mapping a set of required logical memories into a fixed set of physical memories in a Field-Programmable System was presented. It was described how the access time of the logical memories and the area of the multiplexing circuitry depend on how the logical memories are packed into the physical memories.

A CAD tool called **MemPacker** was developed with the algorithms and models presented in Chapter 4. The effectiveness of the tool was demonstrated by showing how it can be used to determine maximum memory operating frequency with minimum area cost.

5.2 Realities

The *Transmogrifier-1*, like any other Field-Programmable System, runs at very low speeds. From the two large designs that were implemented using the TM-1, only one ran at $10MHz$. It is important to notice however, that the low speed is due mainly to the slowness of today's FPGAs. The memory organizer also ran at very low speeds due to the low speeds of the FPGAs in which the organizer was implemented.

Another important issue is the density of the TM-1. The TM-1 is supposed to have 40,000 equivalent gates. This is not a very large number if we consider that by the time the system was functional, many FPGA companies had announced 25,000 equivalent gate FPGAs. This means that probably in a year, FPGA companies will announce 50,000 equivalent-gate devices, making the TM-1 obsolete.

Finally, synthesis software for FPGAs is not a mature field and therefore, designing with many FPGAs is not as easy as it should be. A CAD tool that understands the architecture of the programmable system, and not only the individual FPGAs, is going to be needed in order to get higher logic utilization. However, we believe that it is only a matter of time before better synthesis tools are developed.

5.3 Future Work

Bell Northern Research has been constructing a multi-chip module (MCM) almost identical to the *Transmogrifier-1*. BNR's MCM has a $2.4'' \times 2.4''$ footprint, and is suitable for replacing small-gate count ASICs with reasonable amounts of memory. The main differences between the board level TM-1 and the MCM TM-1 are that the MCM contains one FPIC device, and some control lines as well as global clock signals that are hard-wired between the FPGAs without passing through the FPIC device.

A next generation Field-Programmable System (*i.e.* the *Transmogrifier-2*) can be built using several of BNR's MCM or future versions of it. However, if a more efficient FPS is wanted, several issues need to be addressed before building the next *Transmogrifier* system:

1. It is important to decide how many and what kind of FPGAs are going to be used in

the next generation FPS: Should the TM-2 contain the largest available Xilinx FPGAs, or should it use Altera's, AT&T's, or other FPGAs ?

2. The type of interconnect architecture to be used in the *Transmogrifier-2* will be very important: Should the TM-2 also use a concentrated, distributed programmable interconnect architecture, or should it use a partial programmable architecture or something else ? Should the TM-2 contain some hard-wired connections ? If so, how many ?

3. The type and size of the memories also imposes many interesting questions: How many and what kind of memories will be needed ? Should the TM-2 use hard-wired connections between the FPGAs and the memories to reduce memory access times and hence increase the practicality of a memory organizer ?

Another important topic for future work has to do with benchmarking. To the best of our knowledge, there are no set of benchmark circuits that can be used to compare Field-Programmable Systems. Therefore, nobody really knows the advantages or disadvantages of all the available systems. Benchmark circuits can also be used to develop better synthesis tools, such as better partitioners.

Benchmark circuits can also be used to further study better memory organizers. By having a better idea of the set of memories that most of the times are required, better models can be developed. At the same time, heuristic algorithms can be investigated.

Appendix A

Interface Board's Schematics

As described in Section 3.1.1, for the TM-1 to work properly, either as a compute engine or as an emulator, it needs some kind of communication with a host computer.

The TM-1 system uses two auxiliary boards in order to interface to a host computer. First, it uses an S16D interface from *Engineering Design Team, Inc.* (EDT) [28]. The S16D is a single-slot, 16-bit parallel input/output interface for SBus-based computer systems. Second, the system uses an additional interface board, built at the University of Toronto as part of this thesis, that contains a Xilinx XC4010 FPGA.

Two circuit designs were implemented and tested on the interface board: Figure A.1 shows the schematics of the 72-bit programmable bi-directional interface, and Figure A.2 illustrates the schematics of the 16-bit direct memory access (DMA) [29] interface.

Figure A.3 shows the contents of the port symbol used in both of the above designs. Finally, the control symbol (in Figure A.2) contains the VHDL file shown in Figure A.4:

A.1 Instructions

A.1.1 72-bit programmable bi-directional interface

To use the 72-bit programmable bi-directional interface:

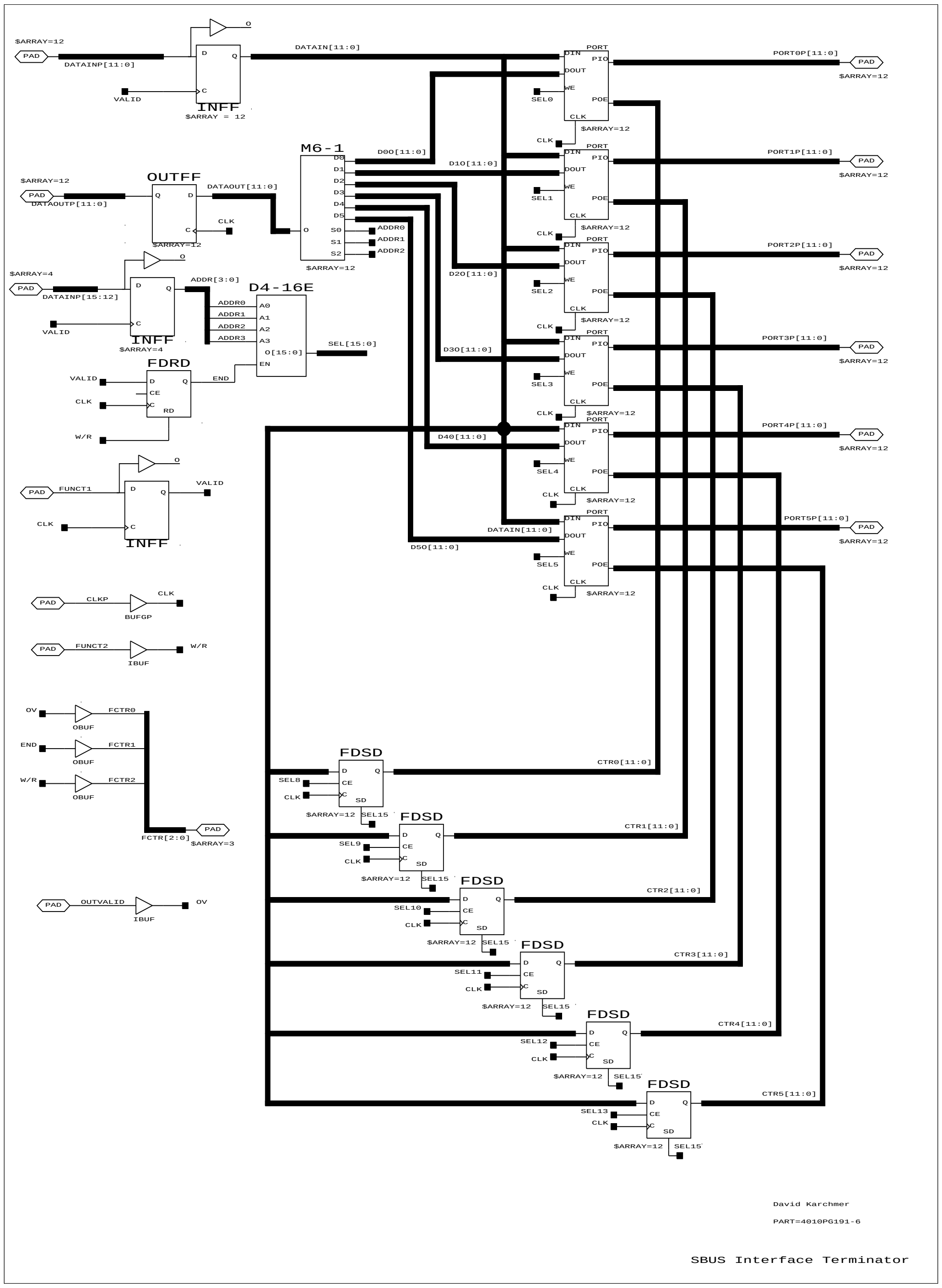

Figure A.1: 72-Bit bi-directional Interface.

1. Download the circuit into the FPGA in the interface board. To do so, turn switch 1 to the left (see Figure B.4) and type:
   "xchecker -port /dev/ttya /jayar/d0/davka/FPS/s16d/io72".
   If you have any problem downloading the fpga in the interface board, you can disconnect the two sbus cables, execute the **xchecker** command again, and then reconnect

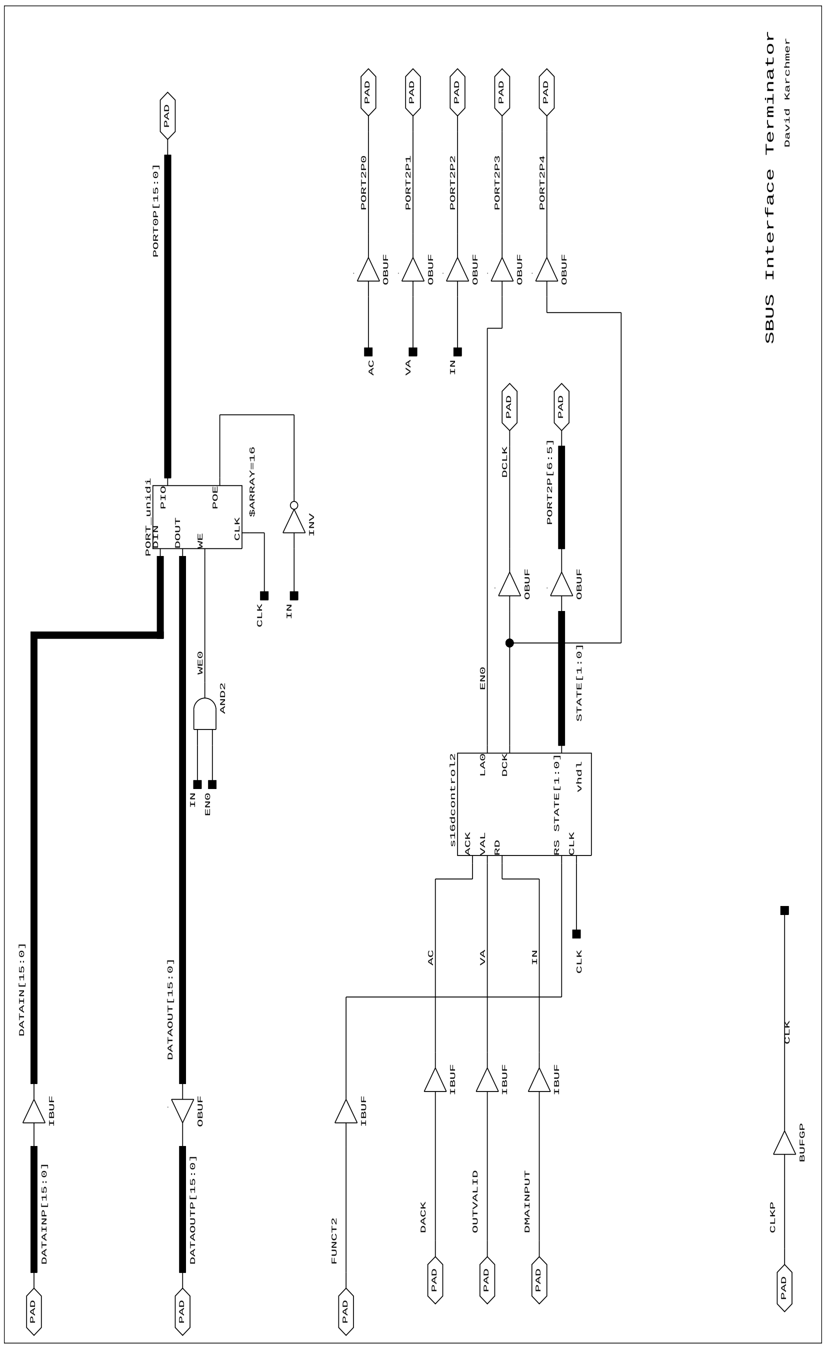

Figure A.2: 16-Bit DMA Interface.

the two cables without turning the power off.

2. To run the interface program type:
   "/jayar/d0/davka/FPS/bin/interface".
   Follow the below instructions to use the interface:

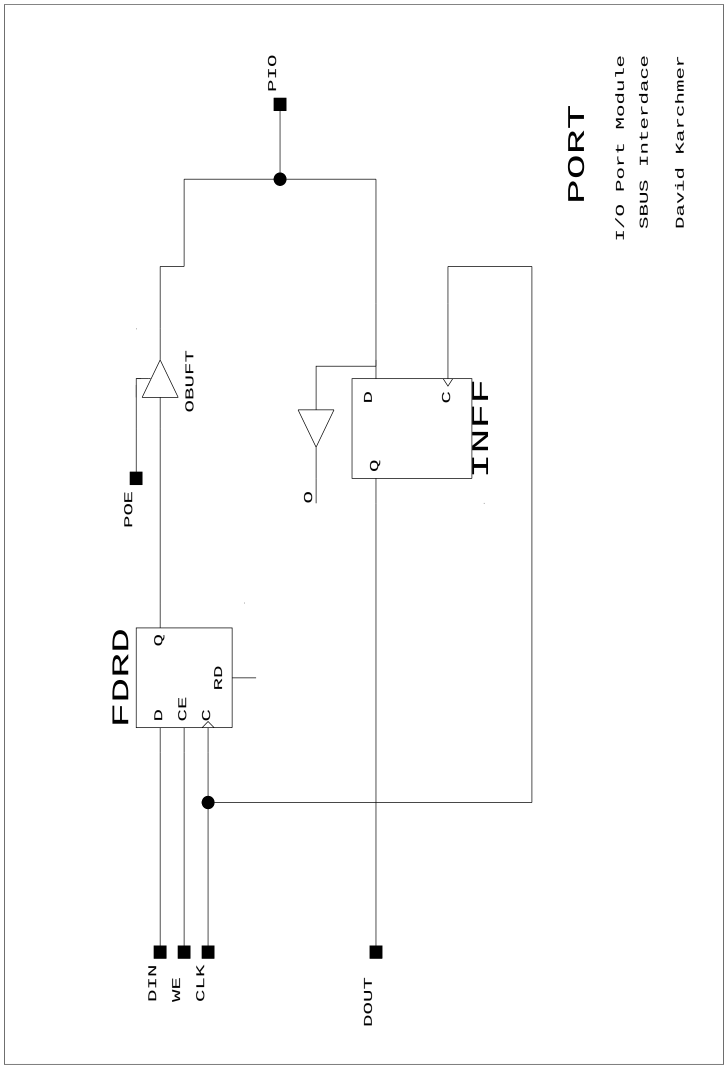

Figure A.3: Port's Schematics.

- Use **c P ccc** to configure each port where *P* is the port number (0 to 5) and *ccc* is the configuration data. *ccc* is a hex number representing the 12 port pins. Each pin can be configured with a zero for an output (*i.e.* from the computer to the TM-1 board) or with a one for an input (*i.e.* from the TM-1 board to the computer).
- Use **w P www** to write data where *P* is the port number (0 to 5) and *www* is the data to be written. This number is also a hex number representing the 12 port bits.
- Use **r P** to read data where *P* is the port number (0 to 5) from where you want to read.

- Use **q** to quit the interface program.

A.1.2 16-bit DMA interface

To use the 16-bit DMA interface:

1. Download the circuit into the FPGA in the interface board. To do so, turn switch 1 to the left (see Figure B.4) and type:
   "xchecker -port /dev/ttya /jayar/d0/davka/FPS/s16d/dma16_u1"
   if you are using a $10 MHz$ oscillator, or
   "xchecker -port /dev/ttya /jayar/d0/davka/FPS/s16d/dma16_u2"
   if you are using a $33 MHz$ oscillator. If you have any problem downloading the fpga in the interface board, you can disconnect the two sbus cables, execute the **xchecker** command again, and then reconnect the two cables without turning the power off.

2. The interface can be tested by typing:
   "/jayar/d0/davka/FPS/s16d/testdma -o 1 size repeat"
   where *size* is the size of the counter and *repeat* is the number of times you want to repeat this counter. This command only writes a counter from 0 to *size*, *repeat* times.

3. To read data type:
   "/jayar/d0/davka/FPS/s16d/testdma -o 0 size repeat".

4. Use the **testdma** program as a template when more elaborated programs are needed.

```
-- -----------------------------------------------------------
--   This is a state machine that controls the protocol needed
--   to talk to the S16D sbus interface card. It needs
--   a clock of approx. 10MHz.
-- -----------------------------------------------------------
library exemplar;
use work.exemplar.all;

entity control is
        port(   clk, ack, val, rd, rs: IN bit;
                state: OUT bit_vector(0 to 1);
                dck, la0: OUT bit
        );
end control;

architecture exemplar of control is
        signal st, nx: bit_vector(0 to 1);

begin

-- update the state on the clock edge (Initialize at st(0))
        dffp(nx(0), rs, clk, st(0));
        dffc(nx(1), rs, clk, st(1));

-- set the outputs
        dck <= '0' when st(1)='1' else '1';
        la0 <= '1' when st(1)='1' and rd='1' else '0';
        state(0) <= st(0);
        state(1) <= st(1);

-- set next state
        nx(0) <= '1' when (st(1)='1' and st(0)='0')
                        or (st(0)='1' and ack='0' )
                        or (st(0)='1' and val='1' and rd='1')
                        or (st(0)='0' and st(1)='0')
                else '0';
        nx(1) <= '1' when (st(0)='1' and ack='1' and rd='0')
                        or (st(0)='1' and val='0' and ack='1' and rd='1')
                else '0';

end exemplar;
```

Figure A.4: VHDL Control Circuit for the DMA Interface.

Appendix B

TM-1 Design Instructions

This Appendix contains the instructions to setup and use the *Takeme* CAD tool. This tool was written to help inexperience users to implement or emulate digital circuits using the Transmogrifier-1 Field-Programmable System.

B.1 General Information

To setup your environment, be sure to follow the next steps:

- It is very important that you include the following lines in your .cshrc:

  1. *source ~pga/xact/SOURCE*
  2. *set path=($path /jayar/e0/aptix/aptix/bin)*
  3. *set path=($path /jayar/d0/davka/FPS/bin)*

- If you have never used *axess* before, execute:
  cat /jayar/e0/aptix/aptix/etc/Xdefaults >> ~/.X11defaults

- You can use the tutorial directory (*/jayar/d0/davka/FPS/designs/tutorial*)
  Read the README file for instructions.

B.2 Design Limitations

Although the user can configure the chip interconnections as he/she wishes, some connections are fixed to facilitate the FPGA downloading. Figure B.1 illustrates the fixed connections needed to serially download the FPGAs. In order to assure that the Xilinx's partition, place and route tool (**ppr**) [26] does not route any internal signal to the unavailable pins, a constraint file (*i.e.* using a *.cst* extension) has to be created before executing **ppr**. An example of this file is shown in Figure B.2.

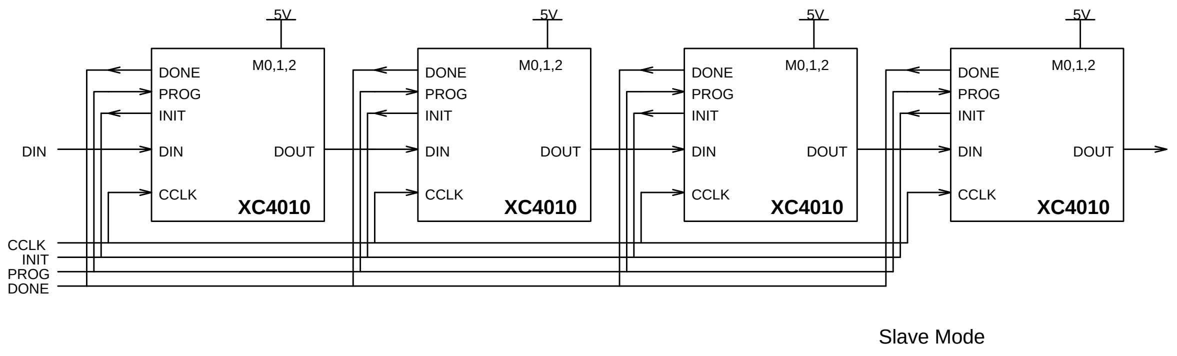

Figure B.1: Fixed connections in the Transmogrifier-1.

```
-------------------------------------------------------------------------
                      notplace instance *: j16;
                      notplace instance *: t4;
                      notplace instance *: u3;
-------------------------------------------------------------------------
```

Figure B.2: Example of a *.cst* constraint file

B.3 Step by step instructions

Takeme is a program that generates all the files needed to download the routing configuration into the Aptix FPIC devices (*i.e.* using **axess**), and the logic design into the FPGAs (*i.e.* using the **xchecker**). For the development process, follow the next steps:

1. Type **takeme**.

2. Click the *SETUP* button, and in the *Setup* window,
   - (a) Enter the name of the directory that you want to create or use.
   - (b) Select *New* or *Old*. New to create a new directory or Old to use an old one.
   - (c) Select what type of format you are using as input.
     - Selecting bit/rpt means that you have run **ppr** already. If so, check the report file (rpt) and be sure that the $j16$, $t4$ and $u3$ pins were not used. To be sure that **ppr** does not use these pins, you need to include a *.cst* file before **ppr** is executed. Figure B.2 shows an example of the *.cst* file.
   - (d) Click *OK* to go to the next step.

3. In the *Resources* window,
   - (a) If you are using an old project and you already modified the *.net* files, then Click *skip init* to go to the next window, else:
   - (b) Select the number of FPGAs required or *use partitioner* if you want to use Dave Galloway's Partitioner.
   - (c) Select the number of memories required.
   - (d) Select the number of 40-pin connectors required. These connectors can be used to interface your design with the computer or to debug your design with the use of a Logic Analyzer.
   - (e) Click *init* to go to the next step.

4. In the *Load* window,
   - (a) Enter the name of your source files. Do not use extensions (*i.e.* .bit, .xnf, etc).
   - (b) Enter the output files name. This is the name that the program will use when generating the SCI netlist file and the xchecker EXO file.
   - (c) Click *OK* to go to the next step.

5. In the *Edit* window,

   (a) Select the editor to use.

   (b) Click the button of the file you want to edit.

   - *connectors.net* This file contains the signal name and location in the two 40-pin connectors.
   - *mem.net* This file contains the memories address and data port names and locations. For example:
     If you have a design in FPGA No. 1 that uses one of the memories (say Memory No. 1) and you call the data bus *data* and the address bus *addr*, you need to edit the file *mem.net* as shown in Figure B.3 [1].

```
--------------------------------------------------------------------
                  mem1 DATA0          35
                  mem1 DATA1          19
                  .
                  .
                  mem1 ADDR0          47
                  .
                  .
                  mem1 ADDR14         21
--------------------------------------------------------------------
```

Figure B.3: Example of a *mem.net* file

   (c) Click *OK* to go back to the main menu.

6. Click the *EXECUTE* button, and in the *Execute* window, Click *OK* to produce the .sci and .exo files. This process may take a long time if your input format is *xnf* or *toyc*.

7. Check the .sci file. You can do this by Clicking *MISC* in the main menu and selecting the *Edit SCI File* button. Make extra changes to the net files if needed (using the *Edit* window and then Click the *Regenerate SCI* button).

[1]Do not change the pin numbers (*i.e.* the number at the end of each line)

8. Click the *DOWNLOAD* button to see the instructions of how to download both the FPIC and FPGA devices. Refer to the Aptix FPCB development system user's manual.

   (a) The *AXESS* window shows the instructions for the FPIC downloading.

      i. Type *Run Axess* [2]: From axess,

         A. *File — Open FPCB Design*

            - Select the *fpcb1* design in your working directory.

         B. *Setup — Import Netlist*

            - Select your SCI file

         C. *Route — Auto Route All*

      ii. Connect the cable that comes from the HIM to the TM-1 (See Figure B.4).

      iii. Turn ON the HIM.

      iv. Wait for one minute.

      v. From axess: *Program — Program FPIC Devices*

      vi. Click *OK* to go to the next window.

   (b) In the *Download Interface* window.

      i. Make all the appropriate cable connections as shown in Figure B.4 (if the download of the FPIC devices was successful):

         - Power and Ground cable from the TM-1 board to the power supply.
         - The two 40-pin connectors from the TM-1 board to the interface board.
         - The 10-pin connector from the TM-1 board to the interface board.

      ii. **TURN ON** the power supply.

      iii. Turn switch 1 to the left. (See Figure B.4)

      iv. Click the *Download Interface* button and hit **Return** from the new window. [3].

      v. Click *OK* to go to the next window.

   (c) In the *Download Design* window:

[2]The Axess software is not X11 compatible and therefore, *twm* may die at any time. If this happens, go to the *console* window and enter **twm**.

[3]If you have any problem downloading the fpga in the interface board, you can disconnect the two sbus cables and then click *Download Interface* again. At this point, you can connect the two cables without turning the power off.

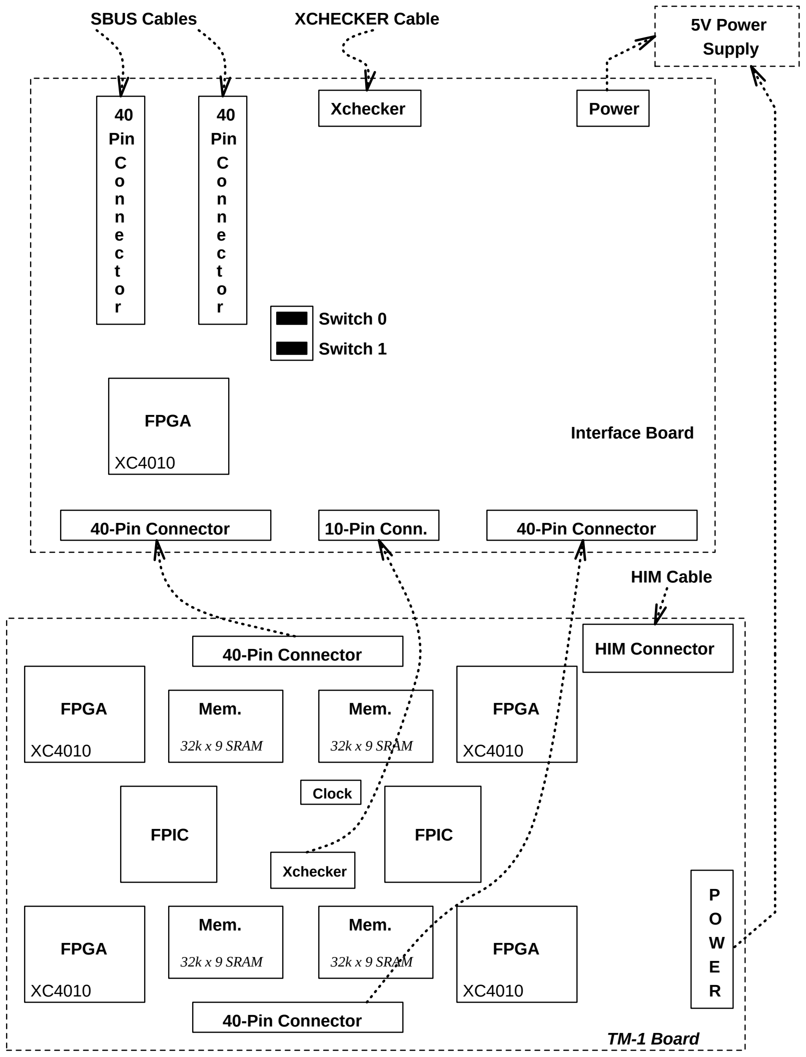

Figure B.4: Block Diagram of the FPS.

i. Turn switch 1 to the right.

ii. Click the *download Design* button and hit **Return** from the new window.

iii. Click *OK* to go to the next window.

(d) In the *Interface* window:

i. Click the *interface* button.

ii. From the new xterm window:

- Use **c P ccc** to configure each port where *P* is the port number (0 to 5) and *ccc* is the configuration data. *ccc* is a hex number representing the 12 port pins. Each pin can be configured with a zero for an output (*i.e.* from the computer to the TM-1 board) or with a one for an input (*i.e.*

from the TM-1 board to the computer).

- Use **w P www** to write data where *P* is the port number (0 to 5) and *www* is the data to be written. This number is also a hex number representing the 12 port bits.
- Use **r P** to read data where *P* is the port number (0 to 5) from where you want to read.
- Use **q** to quit the interface program.

iii. Click *OK* to go to the main menu.

9. Click the *POWERDOWN* button for instructions of how to turn off the system.

   (a) **TURN OFF** the power supply. **WAIT** for a few seconds until the red light in the power supply turns off.

   (b) From axess:

      i. *Program — Power down FPIC devices*

      ii. *File — exit.* It is not necessary to save the changes. In fact, it is faster if you discard them

      iii. TURN OFF the HIM

10. Click the *Quit* Button to exit Takeme.

Appendix C

Using MemPacker

To run **MemPacker**:

1. Create the input file with the Logic Memory Requirements. Each line represent a logic memory and contains three fields: depth, width and access time (in ns). Example:

```
12000 5 35
512 16 35
```

2. Execute the program as follows:
   MemPacker [*option option ...*] < *input_file*
   where the available options are listed in Table C.1

Figure C.1 illustrates the complete **MemPacker** design flow. As shown in this figure, the **MemPacker** XNF output is merged with the original user's XNF file. The resulting merged XNF file is then pass through the **xblox** filter [42]. Xblox synthesizes a delay- and area-efficient logic level design from an input specification consisting of a network of generic modules. After **xblox**, the design can be partitioned, placed and routed using **ppr** and **makebits**.

Option	**Meaning**
-m *num*	Select algorithm: 0: Exhaustive, 1: Heuristic
-s	Show debugging data while executing the algorithm
-t	Include test pads in XNF file. Use for testing
-p	Force powers of two
-x	Generate XNF file
-v	Generate Symbol for Viewlogic
-n	Generate *net* file for the TM-1
-o *name*	Output file name, for the xnf, symbol and net files Default: MemPacker_out

Table C.1: Available options for MemPacker.

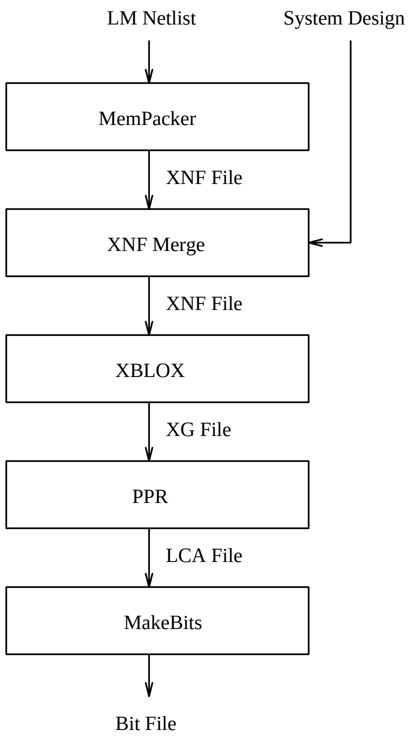

Figure C.1: Design Flow for MemPacker.

C.1 Tutorial

In this tutorial, we will use ViewLogic and **MemPacker** to design a memory organizer.

We will design a $64k \times 5$ logical memory and a 16×12 logical memory. Although the packing of these logical memories is easy, we will use **MemPacker** to quickly generate the XNF file, the symbol file used by ViewLogic and the NET file used by the TM-1.

This tutorial is intended for people with previous experience in designing using ViewLogic, Xilinx FPGAs and the TM-1 system.

1. Create an input file, named *organizer.lm*, as shown in Figure C.2. The first line tells **MemPacker** to use four physical memories, and the second and third line tells **MemPacker** that the required memories are: a $64k \times 5$ with required access time of $80ns$, and a 16×12 with the same access time.

```
------------------------------------------------------------------------
                          4
                          66536 5 80
                          16 12 80
------------------------------------------------------------------------
```

Figure C.2: Example input file for **MemPacker**.

2. Run **MemPacker**:
   Type: *MemPacker -s -x -v -n -m1 -o organizer < organizer.lm*
   This command will run **MemPacker** showing the debugging data (*-s*), and generating an "organizer.xnf" (*-x*), an "organizer.1" (*-v*) and an "organizer.net" (*-n*) files, containing the organizer Xilinx netlist file, the ViewLogic symbol file and the TM-1 mem.net file. It uses the heuristic algorithm (*-m0*)

3. Move the "organizer.1" file to your ViewLogic sym directory.

4. Use Viewdraw to capture the design shown in Figure C.3. Name this file: "top". (The organizer symbol can be retrieved by typing "< *space* > com organizer")

5. Generate the top.xnf file:
   Type: *wir2xnf top*

6. Merge the top.xnf file with the "organizer.xnf" file:
   Type: *xnfmerge top.xnf top_merged.xnf*

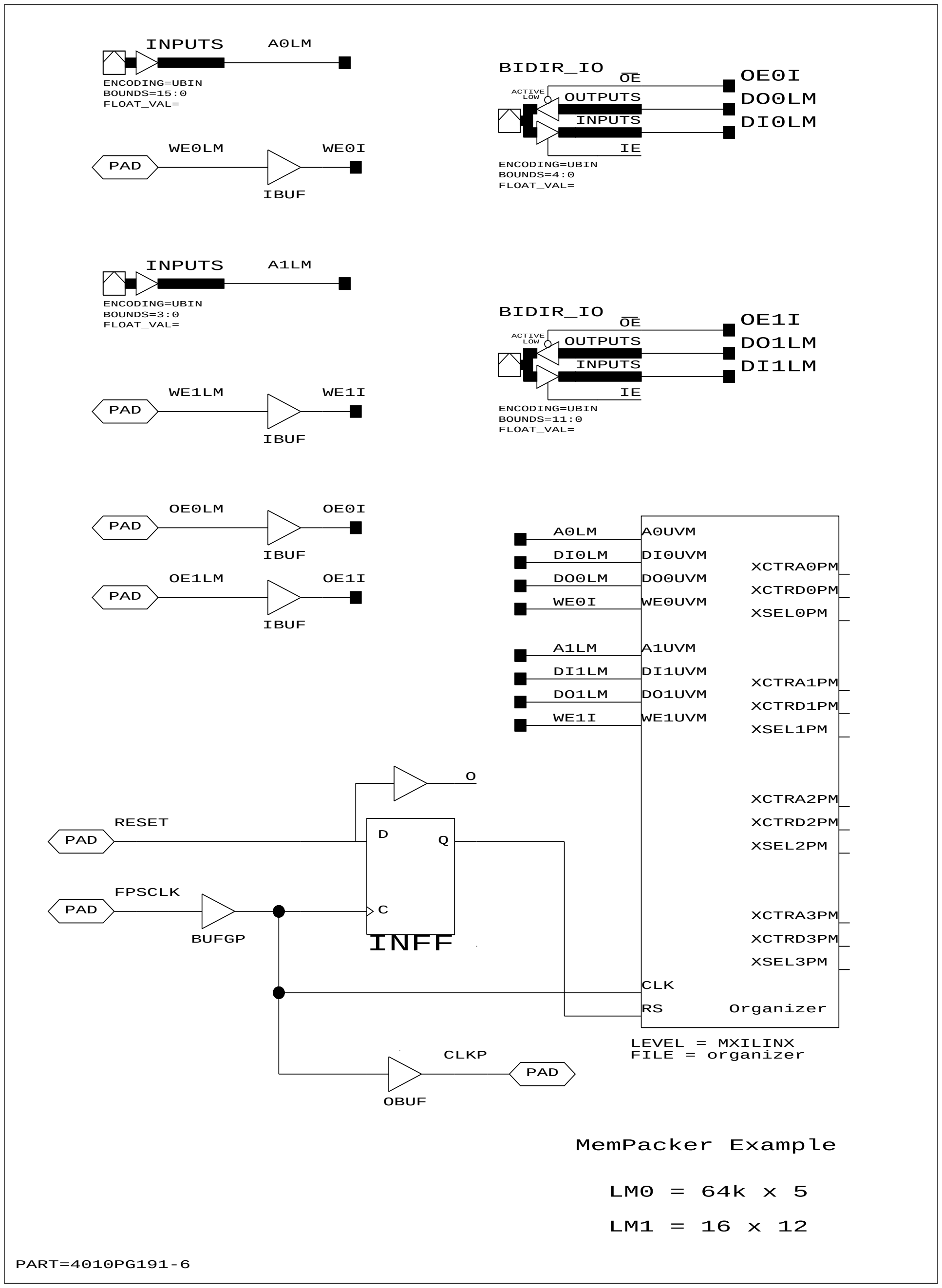

Figure C.3: Viewdraw schematics of the organizer example.

7. Filter the merged xnf file with xblox:

   Type: *xblox top_merged*

8. Generate the configuration bits for the FPGA:

   Type: *ppr top_merged*

Type: *makebits -w top_merged*

9. If you want to actually test the circuit on the TM-1, run **takeme** and:

   (a) Click *Setup*, and

      i. Enter "emulation.fps" as the working directory.

      ii. Select one FPGA and two connectors.

      iii. Enter "top_merged" as your source file.

      iv. Modify the "connectors.net" file to include the A0LM[15:0], DI0LM[4:0], WE0LM, OE0LM, A1LM[3:0], DI1LM[11:0], WE1LM, OE1LM, and RESET nets.

   (b) From a unix shell, copy the "organizer.net" file to "emulation.fps/mem.net"

   (c) From *takeme*, execute steps 6 to 9 of Section B.3.

Bibliography

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

[2] M. Butts, J. Batcheller, and J. Varghese, "An Efficient Logic Emulation System," in *IEEE International Conference on Computer Design*, October 1992.

[3] S. Walters, "Computer-aided prototyping for ASIC-Based systems," *IEEE Design and Test of Computers*, pp. 4–10, June 1991.

[4] R. Tessier, J. Babb, M. Dahl, S. Hanono, and A. Agarwal, "The virtual wires emulation system: A gate-efficient asic prototyping environment," in *FPGA 94*, February 1994.

[5] D. Van den Bout, J. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman, "Any-Board: An FPGA-Based reconfigurable system," *IEEE Design and Test of Computers*, pp. 21–30, September 1992.

[6] P. Bertin, D. Roncin, and J. Vuillemin, "Programmable Active Memories: A Performance Assessment," in *Research on Integrated Systems: Proceedings of the 1993 Symposium*, MIT Press, 1993.

[7] S. Casselman, "Virtual Computing and The Virtual Computer," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993.

[8] J. Darnauer, P. Garay, T. Isshiki, J. Ramirez, and W. Wei-Ming, "A Field Programmable Multichip Module," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, April 1994.

[9] D. Van den Bout, "The RIPP Architecture," in *4th. Anual PLD Design Conference*, April 1994.

[10] J. Arnold, "The Splash 2 software environment," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993.

[11] Xilinx Inc., San Jose, CA, *The Programmable Logic Data Book*, 1993.

[12] J. Arnold, D. Buell, and E. Davis, "Splash 2," in *4th. Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316–322, 1992.

[13] D. Pryor, M. Thistle, and N. Shirazi, "Text Searching on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993.

[14] T. Hoang, "Searching Genetic Databases on Splash 2," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993.

[15] "Emulation: Niche or Future Standard for Design Verification?." DAC'93's panel, 1993.

[16] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: Overcoming pin limitations in fpga-based logic emulators," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*, April 1993.

[17] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal, "Emulation of a Sparc Microprocessor with the MIT Virtual Wires Emulation System," in *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, April 1994.

[18] S. Hauck, G. Borriello, and C. Ebeling, "Mesh Routing Topologies For FPGA Arrays," in *FPGA 94*, February 1994.

[19] Aptix Corporation, San Jose, CA, *Aptix System Data Book*, November 1993.

[20] I-Cube, Inc., Santa Clara, CA, *The FPID Family Data Sheet*, February 1994.

[21] AT&T Microelectronics, *Product Brief: AT&T Optimized Reconfigurable Cell Array (ORCA) Series Field-Programmable Gate Arrays (FPGAs)*, April 1993.

[22] Crosspoint, *CP20K Field Programmable Gate Arrays Data Book*, November 1992.

[23] D. Smith, "Intel's FLEXlogic FPGA architecture," in *Compcon Spring '93*, February 1993.

[24] D. Galloway, D. Karchmer, P. Chow, D. Lewis, and J. Rose, "The Transmogrifier: The University of Toronto Field-Programmable System," in *Canadian Workshop on Field-Programmable Devices*, June 1994.

[25] Motorola Semiconductor, *MCM62110 Technical Data*, March 1992.

[26] Xilinx Inc., San Jose, CA, *XACT Development System*, October 1992.

[27] Aptix Corporation, San Jose, CA, *FPCB Development System User's Manual, V1.2*, September 1993.

[28] Engineering Design Team, Inc., *S16D High-speed 16-bit I/O Interface. User's Guide*, March 1993.

[29] Z. Vranesic and S. Zaky, *Microcomputer Structures.* Saunders College Publishing, 1989.

[30] A. Dewey and A. De Geus, "VHDL: Towards a Unified View of Design," *IEEE Design and Test of Computers*, June 1992.

[31] Viewlogic Systems, Inc., Marlboro, MA, *Workview Series II Manual*, May 1991.

[32] Aptix Corporation, San Jose, CA, *FPCB AXB-AP4: Field-Programmable Circuit Board for ASIC Prototypes*, August 1993.

[33] B. Kernighan and R. Pike, *The UNIX Programming Environment.* Prentice-Hall, Inc., 1984.

[34] J. Ousterhout, "TCL and TK: A New Approach to X11 and GUI Programming," in *USENIX Winter 1993 Technical Conference*, January 1993.

[35] D. Yeh, P. Chow, and G. Feygin, "A Multiprocessor Viterbi Decoder Using Xilinx FPGAs," in *Canadian Workshop on Field-Programmable Devices*, June 1994.

[36] G. Feygin, P. Gulak, and P. Chow, "A Multiprocessor Architecture for Viterbi Decoders with Linear Speed-Up," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 2907–2917, 1993.

[37] J. Lee, "A Field-Programmable System Emulation of an LNS Processor," tech. rep., University of Toronto, Toronto, Canada, 1994.

[38] D. Lewis, "An Accurate LNS Arithmetic Unit Using Interleaved Memory Funtion Interpolator," in *Proc. of Arith-11*, pp. 2–9, June 1993.

[39] D. Karchmer and J. Rose, "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems," in *IEEE International Conference on Computer-Aided Design*, November 1994.

[40] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley and Sons, 1990.

[41] M. Garey and R. Graham, "Worst-case analysis of memory allocation algorithms," in *4th. Annual ACM Symposium on Theory of Computing*, 1972.

[42] S. Kelem and J. Seidel, "Shortening the design cycle for programmable logic devices," *IEEE Design and Test of Computers*, pp. 40–50, December 1992.