An External I/O Interface for a Reconfigurable Computing System

by

Alex Kuperman

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

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

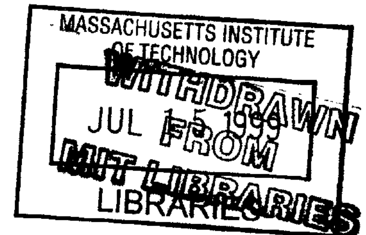
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Alex Kuperman, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.



Author ....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ....
Anant Agarwal
Professor of Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An External I/O Interface for a Reconfigurable Computing System

by

Alex Kuperman

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A box of FPGAs needs a simple, efficient and extensible I/O interface in order to function as a general-purpose reconfigurable computer. An end-to-end memory-mapped interface standard is developed and implemented for an IKOS VirtuaLogic emulator connected to a PC with an Annapolis Micro Systems WILD-ONE PCI board. Use of the interface to facilitate application development for the emulator, as well as device emulation on the PC, are demonstrated by description and performance analysis of several programs. Advantages of such a system over a single microprocessor-based machine are outlined for multimedia applications.

Thesis Supervisor: Anant Agarwal
Title: Professor of Computer Science

Acknowledgments

This work relies heavily on the system developed for the RAW Benchmarks [2] by Jonathan Babb and others. The members of the RAW group have provided me with many ideas and much inspiration throughout the project. I would especially like to thank Jonathan Babb, Russ Tessier, Matt Frank, Ben Greenwald, and Anne McCarthy for their patience and help with the endless technical problems that never failed to get in the way. The project would not have been successful without the support and direction of Anant Agarwal. The daughtercard for the WILD-ONE board was designed and produced by Eric Bovell.

Finally, I would like to thank my parents for their constant encouragement and ready advice.

Contents

1 Introduction **9**

2 Environment **11**

2.1 VirtuaLogic Emulator 11

2.2 Annapolis PCI Card 12

3 Implementation **13**

3.1 Requirements 13

3.2 System Overview 13

3.3 Clocking 15

3.4 WILD-ONE FPGA Logic 16

3.4.1 Bandwidth Matching and Allocation 16

3.4.2 Architecture of WILD-ONE Board 17

3.4.3 Datapath 18

3.4.4 Control 20

3.5 PC User Library 22

3.6 Memory-Mapped Verilog Module 24

4 Results **29**

4.1 Applications 29

4.1.1 Hello World! 29

4.1.2 Conway's Game of Life 32

4.2 Performance 34

5 Discussion **37**

5.1 Conclusions 37

5.2 Future Work 37

A Source Code **39**

A.1 Memory and character I/O driver 39

A.2 PC library 40

A.3 WILD-ONE FPGA Logic 42

A.4 Memory-Mapped Module 45

A.5 "Hello World!" Emulator Program 47

Bibliography **51**

List of Figures

3-1 System Overview 15

3-2 WILD-ONE Board Architecture 17

3-3 Interface Datapath Architecture 18

3-4 Interface Control FSM 21

3-5 Memory-Mapped Module Architecture 25

3-6 Memory-Mapped Module Read Timing 26

3-7 Memory-Mapped Module Control 26

Chapter 1

Introduction

In the search for higher performance computing, there has been a trend towards specialized hardware that is custom designed to do a specific task, relieving the load on the main processor of a computer. When a user demands high performance video, 3D, DSP, floating point, or other compute-intensive support for his applications, he is usually forced to go out and obtain additional specialized hardware to go beyond the capabilities of a single microprocessor. Such a scenario quickly becomes expensive without adding much in the way of flexibility. The user and/or programmer must tailor his applications to the specific hardware.

It would be much more desirable to design a program in an algorithmically efficient manner without having to worry about limited or unavailable hardware resources, then to create the custom hardware exactly tailored to the needs of the application, without having to wait several months for an ASIC to be produced.

A solution to this problem may already be available thanks to FPGA technology. It would be convenient to replace the traditional microprocessor, motherboard, and expansion cards with a sea of FPGA fabric that could be programmed to run any arbitrary application. Since it takes only seconds to program an FPGA, the system would work as easily as loading a program in software on a traditional PC, but provide performance comparable to custom hardware. The performance gains would result not only from using custom circuitry instead of processor instructions, but also from the ability to take advantage of large amounts of parallelism.

Currently, boxes of FPGAs, also known as logic emulators, have found applications in

ASIC prototyping and in-circuit emulation. Such a box can be converted to the general-purpose computer described above by adding the necessary runtime interfaces to memory, user I/O devices and other peripherals, in addition to the FPGA programming mechanism.

Some work has been done to enable a commercial logic emulator to run hardware subroutines from a host [4] and to use the emulator to benchmark parallelizable code [2]. Both approaches have used the emulator as a "slave" unit attached to a host computer. I propose to go one step further, to make the emulator itself the central computer which runs the entire application and controls peripheral devices.

In this thesis I will focus on designing an I/O interface that would enable this central computer to communicate with and control any devices that may be connected to it.

Chapter 2

Environment

While a commercial reconfigurable computer would likely be a single box with everything but the peripheral connectors hidden from the user, a somewhat less elegant prototype is used for the research. The basic setup is similar to the one used to run the RAW Benchmark Suite [2]. A VirtuaLogic Emulator from IKOS Systems [5] is used as the reconfigurable computer, and is programmed via a SCSI interface. To emulate the memory and various peripherals that may be connected to the computer, I use one or more off-the-shelf PCs connected to the emulator's data ports. In order to interface the PCs with the emulator, I use a programmable PCI card made by Annapolis Micro Systems [1].

2.1 VirtuaLogic Emulator

The emulator is a box containing five boards of 64 FPGAs each. In my case the FPGAs are Xilinx 4013, each containing 13,000 programmable gates, for a total of over four million gates in the entire emulator. Much larger FPGAs are currently available, allowing for even larger designs. The FPGAs are connected in a nearest-neighbor mesh, with inter-FPGA signals time-multiplexed over the pins by means of Virtual Wires technology [3]. The use of Virtual Wires allows full usage of the FPGA gates by removing pin limitations, but results in a clock speed penalty seen by the user's design. The final clock speed of this system is typically 1 MHz.

The VirtuaLogic software performs most of the steps necessary to place a design onto the emulator. The user need only write his design in Verilog, from which an LSI netlist is synthesized, partitioned with Virtual Wires, and finally compiled into FPGA bitstreams to

be loaded onto the emulator.

The feature most vital to a general-purpose reconfigurable computer is the I/O. The emulator fulfills this requirement quite well. Each board has six 100-pin connectors, each of which has 86 signal pins. All of the signal pins are completely programmable and can be tied to any signal or bus in the design. The design of the emulator makes it very easy and convenient for the user to view the system as a single giant FPGA with four million gates and 2580 signal pins.

2.2 Annapolis PCI Card

To implement the other end of the interface, I have chosen to use the Annapolis Micro Systems WILD-ONE programmable PCI card installed in an off-the shelf Pentium II PC. This configuration can easily be programmed to emulate an arbitrary external device. The WILD-ONE card contains two Xilinx 4036 FPGAs which I use to implement the interface logic of the device. Three 32-bit busses connect the FPGA pins directly to a connector on the face of the card. A custom daughtercard was designed to convert from this connector to a 100-pin connector on the backplate of the WILD-ONE card which is compatible with the emulator's data cable. Another advantage of the WILD-ONE card is that it provides a FIFO interface to the PCI bus, which reduces the complexity of my interface logic. Annapolis Micro Systems also provides the API necessary to talk to the card from within a C program, eliminating the need for me to write a custom driver.

Chapter 3

Implementation

3.1 Requirements

One of the most important features of an interface to a general purpose computer is flexibility. The interface must be flexible and general enough to be used for a variety of purposes and must be easy to adapt for use with devices that have widely varying interface requirements.

Furthermore, it is desirable to have a high-bandwidth, scalable interface that can take advantage to the generous number of I/O pins on the emulator. There is some disparity between the emulator's 86x30-pin, 1 Mhz interface and the PCI bus' 32-bit, 33 MHz interface. There must be some mechanism to translate between the two efficiently.

Finally, the complete end-to-end interface must be easy to use for the designer of the applications to be run on the emulator as well as the designer of external devices to be connected to the emulator, or in my case, the writer of the device emulation programs on the PC. This means that it should also be straightforward to adapt existing applications to the new computation model.

3.2 System Overview

To satisfy the flexibility and ease-of use requirements, I have decided to use a memory-mapped model for my high-level interface. Under such a model, the application in the emulator would access any external device as if it were a memory. This is of course an ideal mapping if the external device is indeed a memory, but the model also works quite well in

other cases. Writes to particular addresses could be used as specific commands to a device, and reads could be used to monitor status or to receive data streams. Such an interface is fairly easy to implement from any perspective. The device and the application need only to agree on a mapping of the address space, and any desired functionality may be built on top of the provided memory-mapped structure.

Such an approach provides a uniform, standard view to the application and device developers while hiding the wiring and timing details of the specific hardware implementation. This makes the system highly portable and invites improvement over the current emulator, PCI card, and workstation setup.

Once the high-level interface and functionality have been defined, the challenge is to build the underlying hardware and firmware support it. In this case, the hardware consists of the following:

1. 30 100-pin connectors on the emulator directly accessible by the programmable fabric
2. 100-pin cables to connect the emulator to the WILD-ONE card
3. Programmable WILD-ONE card with FIFO interface to the PCI bus
4. Pentium II PC
5. WILD-ONE API for communicating with the PCI card

I need to come up with a design and a protocol specific to the listed hardware to provide the basic I/O functionality in an efficient manner. Since this protocol will likely not look like the desired memory-mapped interface, I also need to implement two additional modules:

1. A Verilog module to be linked with the application to be mapped into the emulator to translate between the specified memory-mapped functionality and the low-level protocol. The user's application will talk to the front end of this module as if it were a memory, while the back end will implement the protocol necessary to talk through the connector pins to the circuit in the WILD-ONE FPGA.
2. A C language library on the PC to wrap around the WILD-ONE API. The user will call the library functions as if they were memory operations, and the library functions will in turn call the appropriate API commands necessary to talk to the circuit in the WILD-ONE FPGA.

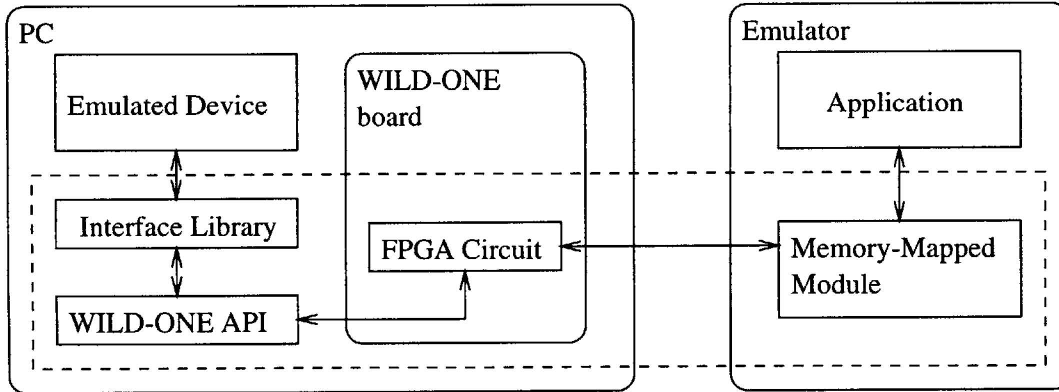


Figure 3-1: The components of the interface (dashed box) work together across hardware and software boundaries to provide a direct, high-level interface between the programs in the PC and in the emulator.

The final result is illustrated in Figure 3-1. The Verilog module, the WILD-ONE FPGA circuit, and the PC library all work together to create a layer that seamlessly connects the application in the emulator to the emulated device or other application in the PC.

3.3 Clocking

For reasons of both performance and simplicity, it is desirable for the interface to be synchronous. The only way to achieve this is to have a common clock input. The emulator is already set up to run on an external clock. While the FPGAs and Virtual Wires circuitry run on a fast internal clock (typically 20 MHz, though 27 MHz is possible), the user application and I/O pins sample an external clock input, which is limited to approximately 0.5 MHz to 2 Mhz, depending on the size and complexity of the design.

In order to operate synchronously with the emulator, the WILD-ONE FPGA circuit must sample the same clock and follow a set timing protocol compatible with that used by the emulator. It is convenient that the WILD-ONE card has the capability to generate an external I/O clock in addition to its master clock. This eliminates the need for a separate clock generator, simplifying the setup somewhat.

The fact that the interface clock is sampled in the presence of a faster master clock on both the emulator and the WILD-ONE card brings up some interesting design issues. As the designer of the WILD-ONE circuit I need to be aware of the fact that the emulator does not see the sampled clock edge until a few of its fast clock cycles later, and I need to know when the emulator *actually* asserts and/or reads the data lines to avoid bus contention. I

also need to be aware of such details as the emulator's ability to perform single-cycle reads — it may read the address at the beginning of an interface clock cycle, then look up and assert the data value by the next clock edge.

3.4 WILD-ONE FPGA Logic

The WILD-ONE logic is very much the central piece of the interface system. It arbitrates between the signals and timing of the emulator pins and that of the PCI bus in the PC. This is the level at which the interface's efficiency and underlying protocol are defined.

3.4.1 Bandwidth Matching and Allocation

The first thing that needs to be addressed is the disparity between the PCI bus and the emulator's connectors. The PCI bus has a maximum bandwidth of 33 MHz x 32 bits = 132 MBytes/sec, while a single emulator connector has 86 signal pins operating at 1 Mhz, giving a bandwidth of 10.75 MBytes/sec. It appears that we need to feed 12 emulator connectors into one PCI bus to achieve maximum efficiency.

However, since the PCI bus is shared, full bandwidth is rarely achieved. In my own tests, I have observed transfer rates from the WILD-ONE card to the PC's memory of up to 96 MB/s. A safe assumption might be that 50% of the PCI bandwidth is generally available. Furthermore, it is possible to run the emulator at up to 2 MHz for some applications. Taking these numbers into account, the ratio drops to 66/21.5 = 3 connectors/PCI bus. Since the WILD-ONE card has only three 32-bit busses routed to the external I/O connector, it only makes sense to have one emulator connector per card. Fortunately, the WILD-ONE drivers allow up to four cards per PC, so our ratio is comfortably achieved.

To harness the full bandwidth of the emulator, we would need 30 WILD-ONE cards in eight PCs, as well as some scheme of dividing the data among the PCs. Since the emulator is fully programmable, it easily scales to the use of one, all, or any number of the interface ports. Therefore, we need only spend design effort on a single-card setup and be confident that it will work for multiple cards and PCs with minimal modification.

Now if we consider a single card, it is convenient to divide the three external busses into one address bus and two data busses. Since the emulator connector limits us to 86 signal pins, it would be best to preserve a 64-bit data word and make do with only 22 bits of

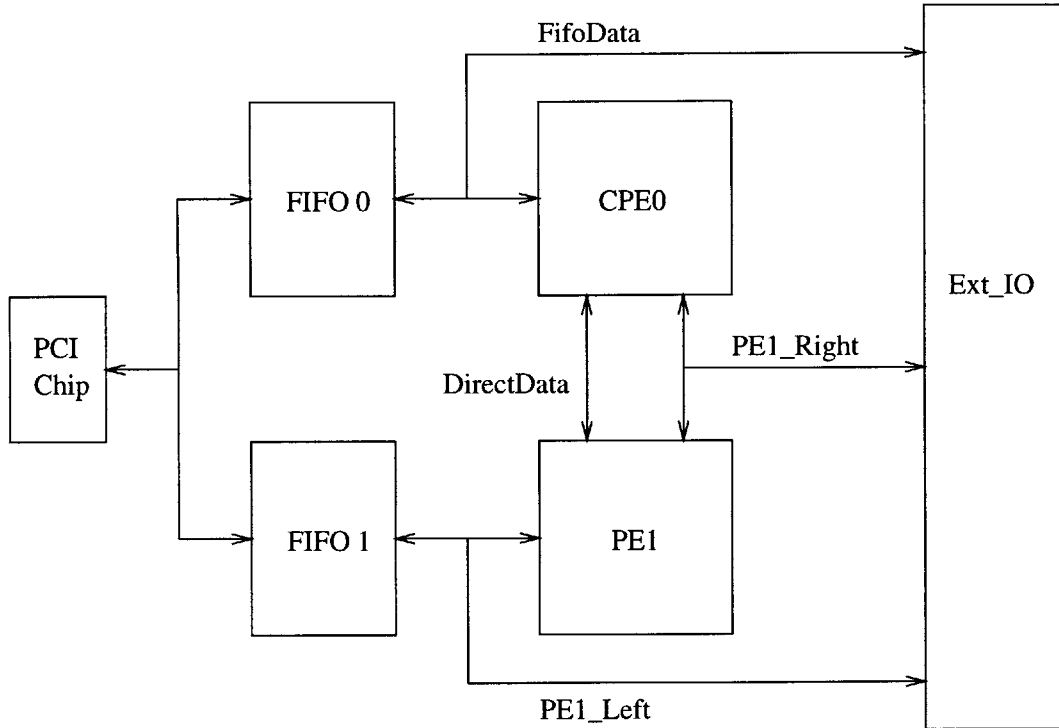


Figure 3-2: Selected components of the WILD-ONE PCI board architecture. CPE0 and PE1 are Xilinx 4036XL FPGAs, and Ext_IO is a connector on the face of the board. All connections shown are 36-bit busses.

address. This is not as big a problem as it may appear to be since this address will only be used for communication between the WILD-ONE logic and the memory-mapped module in the emulator. Actual addresses and data streams would flow over the 64-bit data bus.

Finally, the minimal control necessary to operate a synchronous protocol is a write signal and a read signal, so we must allocate two of the address bits to this function, leaving us with 64 data bits, 20 address bits, and 2 protocol bits. Now the goal of the FPGA logic is clear: To time-multiplex the three slow external buses over the fast 32-bit PCI bus, while adhering to a synchronous protocol with the emulator.

3.4.2 Architecture of WILD-ONE Board

A desirable platform for developing a PCI-to-emulator interface would be a programmable PCI card with an FPGA that has some pins connected to the PCI bus and a separate set of pins routed to one or two 100-pin emulator connectors. The desired interface logic would be inserted between the two sets of pins. The WILD-ONE card gives us somewhat more than we asked for. One convenient feature is the FIFO interface that sits between the PCI bus

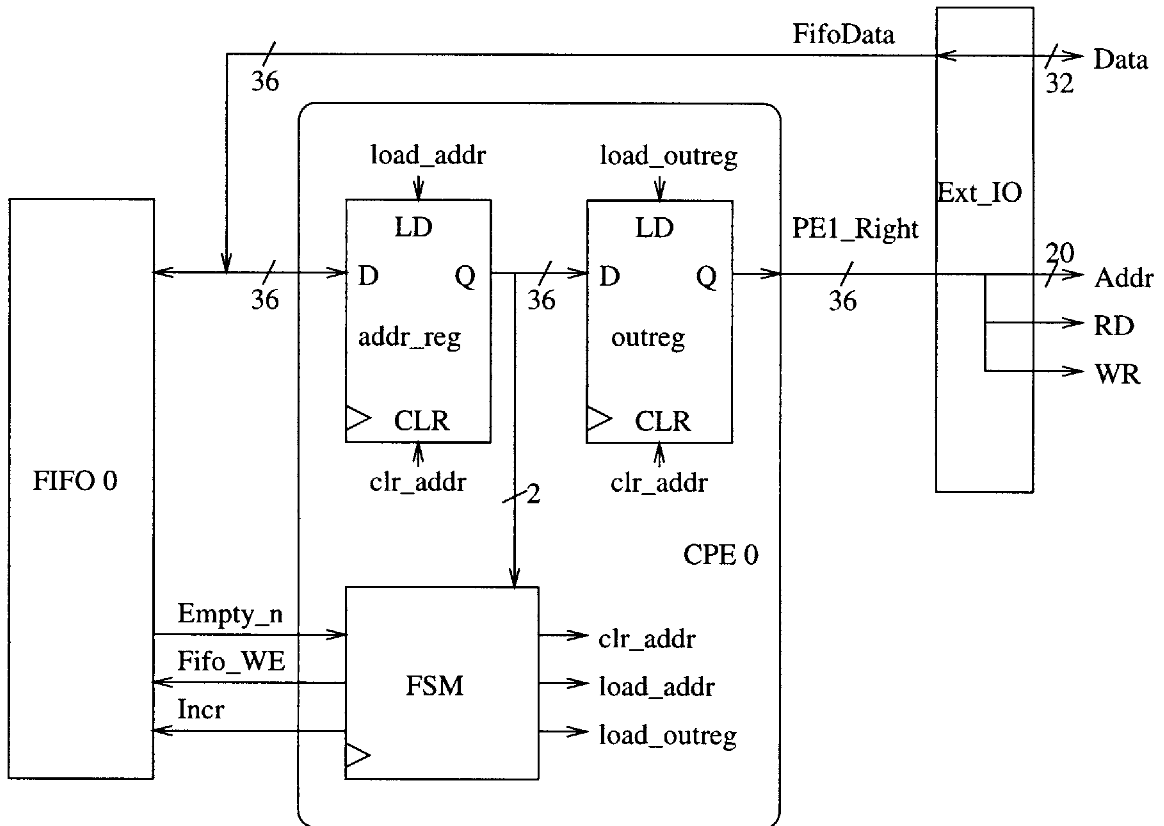


Figure 3-3: Interface Datapath Architecture

and the FPGA. This interface hides the complexity of dealing with the PCI bus directly and makes the design of the emulator interface logic much easier. However, the WILD-ONE has not one, but two FPGAs, each with its own FIFO, and an assortment of busses. As you can see from Figure 3-2, the buses are not so cleanly split between input and output — two of the three are shared with the FPGAs' inputs from the FIFO. This means the interface logic cannot read from the PCI bus and write to the emulator at the same time. Furthermore, the design must be split among two FPGAs to gain access to all available output busses.

3.4.3 Datapath

The design of the datapath must take into account not only the desired functionality of the interface but also the restrictions dictated by the specific architecture of the WILD-ONE card. To keep things simple, I shall describe an interface with one address and one (not two) data bus, as shown in Figure 3-3. This keeps the design to one FPGA while demonstrating all of the relevant points of the project.

One end of the datapath begins with the FIFO that sits between the FPGA and the PCI bus. The FIFO is bi-directional and 512 36-bit words deep. Both the software API and the FPGA logic can communicate directly with their respective ends of it. When the API writes a word to the FIFO, the *HostToPeFifoEmpty_n* flag is asserted on the FPGA end, indicating that there is at least one word waiting to be read from the FIFO. To read the word, the FPGA logic must de-assert the *Fifo_WE* signal to tell the FIFO it wants to read, not write, and then pulse the *FifoPtrIncr_EN* signal for one Pclk cycle, after which the data will be available on the *FifoData* bus. Similarly, to write to the FIFO, the logic must assert the *Fifo_WE* signal, set up the data on the *FifoData* bus, and pulse the *FifoPtrIncr_EN* signal for one Pclk cycle. Of course we must be careful not the drive the *FifoData* bus when we set the FIFO to read mode (*Fifo_WE de-asserted*).

Note that the WILD-ONE board is clocked by a fast clock, called Pclk, which may be set to run at up to 50 MHz. (My interface logic runs at 25 Mhz.) This clock is independent of the slower Eclk which is used to synchronize the interface between the WILD-ONE card and the emulator. All that is necessary is that there be enough Pclk cycles in one Eclk period to perform all the necessary protocol and data processing functions.

As mentioned in section 3.4.1, The 20-bit address and 64-bit data busses, as well as any control signals, must all be funneled through the 32-bit PCI bus, and therefore also the *FifoData* bus. In my design, the read, write, reset and any other control signals are bundled with the address word. Therefore, the address word needs to be read from the FIFO first and stored in a buffer (addr_reg in Figure 3-3). While the address is held in the buffer, the bits used to encode the control instructions are routed to the inputs of the control logic, which decides what to do next. If the operation is a write to the emulator, the next data word(s) need to be read from the FIFO. In the 32-bit case, we do not need to store the word since it is automatically visible on the Ext_IO connector due to the WILD-ONE bus architecture. If we had a 64-bit data word, we would need another buffer to store the first 32 bits of it, to be presented on the third bus visible to the Ext_IO connector. (i.e. The output of the data buffer would be routed via the *DirectData* bus through PE1 to the *PE1_Left* bus.) In the case of a read from the emulator, the data is driven by the emulator directly onto the *FifoData* bus, and is read into the FIFO on the appropriate signal. Of course, we would also need a 32-bit data input buffer for a 64-bit data word.

I have designed the control signals from the PC to the WILD-ONE to coincide with

the RD and WR signals to the emulator, so that the 22 bits of address and control can be passed directly to the Ext_IO connector without the need for any further processing. The only trick is to present the signals at the right time, i.e. synchronized with an Eclk edge. For this purpose I have a special output buffer (`outreg` in Figure 3-3) that is loaded with the address and control bits at the correct moment, and held at NULL when the interface is idle.

3.4.4 Control

The reading and writing of the FIFO as well as the loading and clearing of the buffers is handled by the protocol and control logic indicated by the "FSM" box in Figure 3-3. This logic is implemented as a Finite State Machine which takes the control bits of the address word, the *HostToPeFifoEmpty_n* flag, and Eclk as inputs, and generates the appropriate control signals for the FIFO and buffers. This FSM is coded in VHDL, as is the datapath and all other elements of the FPGA logic. The VHDL is synthesized into gates and compiled into the bitstream that is provided to the user, to be loaded into the FPGA at runtime.

A Mealy machine representation of the FSM is shown in Figure 3-4. The Eclk signal to the FSM is derived from the raw Eclk input as follows: The input signal is passed through two flip-flops in series to synchronize the edges to Pclk, then the output of the first flip-flop is ANDed with the inverse of the output of the second flip-flop to generate a 1-Pclk duration pulse whenever an Eclk rising edge occurs. In the figure and for the rest of this subsection, Eclk will refer to this rising-edge pulse.

When the interface is reset, the FSM begins in the `Master` state, and waits there until the software sends a command through the FIFO. When the command arrives, as indicated by the *HostToPeFifoEmpty_n* flag, the FSM asserts *HostToPeFifoPtrIncr_EN* to drive the word onto the *FifoData* bus and proceeds to the next state, `GetAddr`, where the word is loaded from the bus into the address buffer. Note that *Fifo_WE* is not asserted at this time. On the next Pclk cycle, the FSM enters the `Dispatch` state, which is the main switchboard for interface operations. This state examines the control bits [WR,RD] of the data word and proceeds as follows:

11: This is the code used for reset of the application in the emulator. Note that this is different from the reset signal used for the FPGA logic. In this case the FSM loads the address word into the `outreg` on the next Eclk edge. The emulator will translate

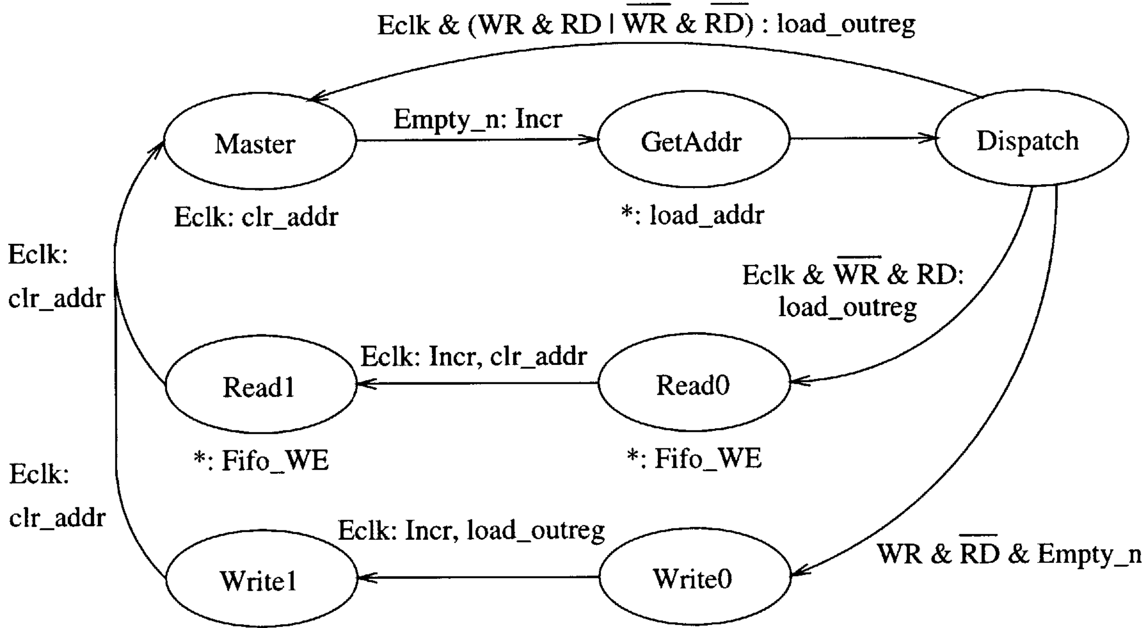


Figure 3-4: Interface Control FSM

the bit pattern into a reset signal and will ignore the rest of the address. Since all the work is done, the FSM now returns to the `Master` state. If no new commands from the software appear before the next Eclk edge, the buffers are cleared to prevent command from being sent to the emulator more than once.

00: This code is unused and acts as a `nop`. It is passed to the emulator just as the reset signal, but has no effect.

01: The asserted RD signal indicates that the software wants to read from the emulator. Since the address is supplied in the rest of the word, we can go ahead and load the `outreg` on the next Eclk edge and then proceed to the `Read0` state, which will wait for the data to come back from the emulator. The `Read0` state asserts *Fifo_WE* to prevent bus contention and to make the FIFO ready to accept the data that will be presented by the emulator.

Since we know that the application in the emulator will run synchronous to Eclk, we can make the memory holding the emulator I/O output registers combinational, so that we will not have to wait an extra Eclk cycle to see the data. I found that the output buffer structure I used in the interface on the emulator side had a turnaround time of 500ns after the Eclk edge, so there should be no problem when running the

Eclk at 1 MHz. The WILD-ONE logic can go ahead and load the data bus into the FIFO on the next Eclk edge. Of course, we must clear the addr_reg and outreg buffers to prevent a second read of the same address, then wait an extra Eclk cycle (state `Read1`) for the emulator to stop driving the bus before we can return to the `Master` state.

10: For the write transaction, we must wait until the data word arrives on the FIFO by again checking *HostToPeFifoEmpty_n*. When it arrives, we go to state `Write0` to wait for the next Eclk edge, upon which we simultaneously load outreg with the address and write signal, and increment the FIFO to drive the data word onto the *FifoData* bus. Then we must wait one Eclk cycle for the data to be written before clearing the buffers and returning to the `Master` state.

Such is the control process for the data flow through the WILD-ONE board. Adaptation to a 64-bit data bus involves only the addition an extra state in both the read and the write paths to load the extra data buffer mentioned in the datapath description (Section 3.4.3).

In order to maintain simplicity, no special error detection and recovery features are included. The correct operation of the FPGA logic depends on specific and consistent operation of the PC software and of the emulator. Fortunately, I am in control of both, so I can ensure that my design does not generate any incorrect situations.

3.5 PC User Library

The PC software may communicate with the WILD-ONE board by means of the provided API functions, but I have written a set of wrappers to preserve the memory-mapped abstraction and to simplify the interface for the user. These wrappers are contained in the files `interface.h` and `interface.c`, which the user would link to when developing his application.

The functionality of these wrappers is best described by stepping through the code. First we need to set up some bit masks to help us construct the 20-bit address and control word:

```
#define READ_OFFSET   (0x00100000)
#define WRITE_OFFSET  (0x00200000)
#define RESET_OFFSET  (0x00300000)
```

```
#define ADDR_MASK      (0x000FFFFF)
```

Before using the interface, the program should call `interfaceOpen()`, which initializes the WILD-ONE board, sets the clocks, and loads the FPGA images containing the interface logic. Similarly, the program should call `interfaceClose()` when done.

In order to write to the interface, the program should call `interfaceWrite(addr,data)` to write a word, `data`, to an address, `addr`, on the emulator. Keep in mind that this address must be no larger than 20 bits. Here is the code for the `interfaceWrite()` wrapper:

```
#define interfaceWrite(addr, data) ifaceWrite((int)addr, data)

void ifaceWrite(int addr, int data)
{
  DWORD fifoDataOut[1],
        wordsWritten;

  fifoDataOut[0] = (addr & ADDR_MASK)|WRITE_OFFSET;
  fifoDataOut[1] = data;
  WF1_FifoWrite( BOARD, WF1_Fifo_Pe0, fifoDataOut, 2, &wordsWritten, 0);
}
```

Notice how the address word is constructed by masking the passed `addr` to extract the lower 20 bits, then adding the control code for a write operation (ie. bits 21,20 set to 1,0). Then the address and data words are placed into an array and passed to the `WF1_FifoWrite()` API function which performs the write to the appropriate FIFO. Since `int`s are 32-bit, a 64-bit design would take two data words and pass a three-word array to the API function.

The `interfaceRead()` function is not much more complicated:

```
#define interfaceRead(addr, data)  data = ifaceRead((int)addr)

int ifaceRead(int addr)
{
  DWORD fifoDataOut,
        fifoDataIn,
        wordsWritten,
        wordsRead;

  fifoDataOut = (addr & ADDR_MASK)|READ_OFFSET;
  fifoDataIn = 0;
  WF1_FifoWrite( BOARD, WF1_Fifo_Pe0, &fifoDataOut, 1, &wordsWritten, 0);
```

```
  WF1_FifoRead( BOARD, WF1_Fifo_PeO, &fifoDataIn, 1, &wordsRead, 0);

  return fifoDataIn;
}
```

The address word is constructed as before and written to the FIFO first. The `WF1_FifoRead()` API call will place the next word coming from the FIFO into `fifoDataIn`. If everything works properly, this will be the value of the given memory location in the emulator.

Finally, a Reset command may be passed to the emulator by calling `interfaceReset()`:

```
void interfaceReset()
{
  DWORD fifoDataOut,
        wordsWritten;

  WF1_FifoReset( BOARD, WF1_Fifo_PeO);
  fifoDataOut = RESET_OFFSET;
  WF1_FifoWrite( BOARD, WF1_Fifo_PeO, &fifoDataOut, 1, &wordsWritten, 0);
}
```

Just for good measure, this function also resets the FIFO, so it is a good idea to call it when the application is first launched, after `interfaceOpen()`.

This set of functions provides a simple and standard interface to an application on the emulator, but can also serve as a base for building the more complex protocols that may be used by devices that are emulated on the PC.

3.6 Memory-Mapped Verilog Module

The final piece of the puzzle is the memory-mapped module that talks directly to the application in the emulator. This Verilog module hides the details of the interface implementation from the application designer. The designer instantiates one or more of these modules to be used like memories in his application. The front end (the signals on the left beginning with "Ext" in Figure 3-5) is a standard set of signals and busses with a specified functionality, while the back end connects to the emulator's pins and communicates directly with the WILD-ONE logic. Thus, the interface itself may be changed or replaced, or the module tweaked for performance or for other reasons without requiring any changes in the application.

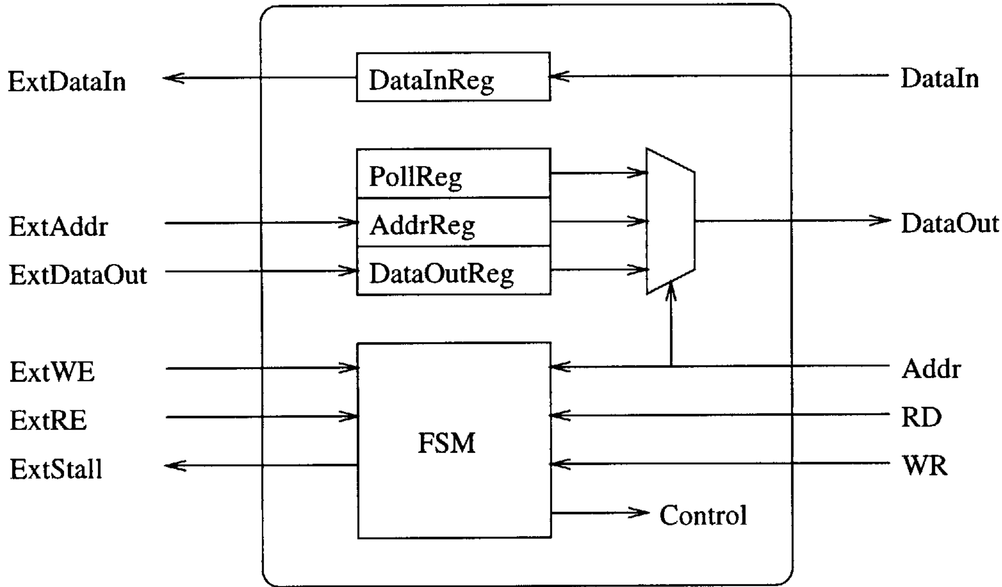


Figure 3-5: The Memory-Mapped module is implemented in Verilog and separates the emulator application (on the left) from the pin-level interface (on the right). It consists simply of a few registers and control logic to arbitrate between the two different protocols. Any of the outgoing registers may drive the *DataOut* bus, as selected by the *Addr* bits from the WILD-ONE board.

Regardless of the underlying low-level interface, the memory-mapped module will always act like a memory. This means that the application in the emulator must have control over reads and writes to the memory. To achieve this, we must reverse the role of the PC as a host by making it a slave device. This functionality is achieved by having the PC poll an address (PollReg in Figure 3-5) in the memory-mapped module which contains the emulator's commands. The module takes care of setting this address to read, write, or null commands when appropriate. Based on these commands, the PC application would perform the desired operations — transferring data between its memory and the interface, for example.

To better understand the process, refer to the timing diagram in Figure 3-6. The module cannot act exactly like an SRAM because the data will probably not show up on the next clock cycle after the address is given. In fact, the delay is unknown and variable because the memory is implemented in software on the PC. To overcome this problem, I have implemented a request/delay mechanism as follows: The emulator would request to read or write data by pulsing the *ExtRE* or *ExtWE* signal for one Eclk cycle simultaneously with presenting valid data on the address lines. At this point, the memory-mapped module's

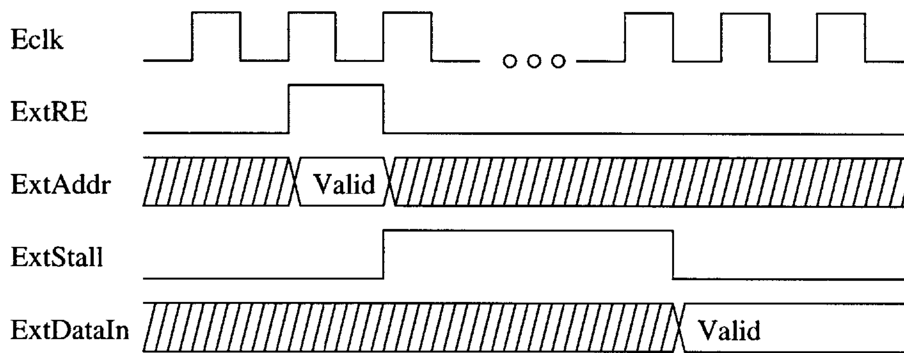


Figure 3-6: Timing of an emulator read from the memory-mapped module. The *ExtStall* signal is necessary since the amount of time the interface takes to retrieve the data from the PC is unspecified.

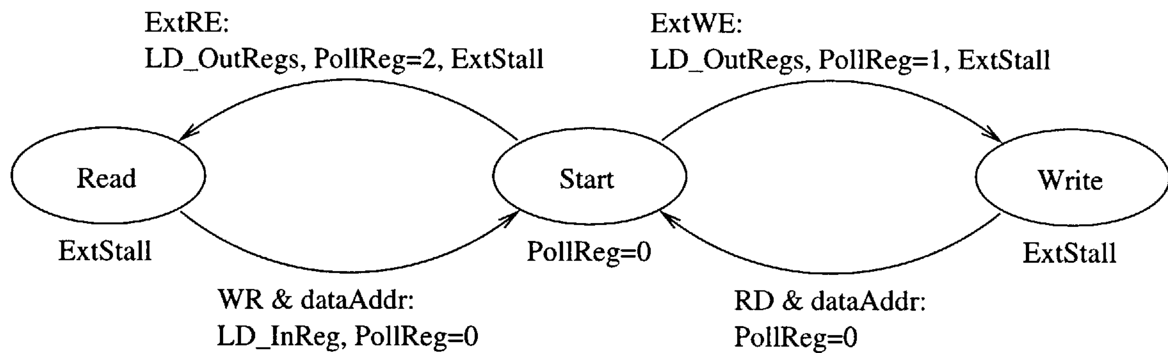


Figure 3-7: Memory-Mapped Module Control

control (Figure 3-7), which normally sits in the `Start` state, would see the *ExtRE* signal and will do the following on the next Eclk edge:

1. Load the outgoing registers (AddrReg and DataOutReg) from the input lines,
2. Set PollReg to the "Read" code, and
3. Raise the *ExtStall* flag.

At this point the PC must poll the PollReg, recognize the "read" command, grab the address from AddrReg, look up the corresponding location in its memory, and finally write the data to the interface. Meanwhile, the memory-mapped module must sit in the `Read` state with the *ExtStall* signal asserted, so that the emulator application will know that the data is not yet ready. The application may of course perform other operations while it is waiting for data, but it may not initiate another memory operation to the same module.

When the module's control logic detects a *WR* signal from the WILD-ONE board, together with an address corresponding to *dataAddr*, it knows that the data has finally

arrived on the *DataIn* bus, so it can load the DataInReg and lower the *ExtStall* flag. It must also return to the `Start` state and reset the outbound registers, especially the PollReg, so that the PC will not attempt an uncalled-for operation. When the emulator sees that the *ExtStall* flag has been lowered, it can use the data on the *ExtDataIn* bus, which will remain valid until the next memory operation.

Notice that the *ExtAddr* bus can be a full 32 or 64 bits, since its value is actually passed over the interface's data bus. The 20-bit interface address bus is used only for internal control in this scenario, to read from the various registers in the memory-mapped module. What may have seemed to be a problematic pin limitation turns out to be more than adequate. The emulator can use the full address space of 64 bits, or more with the addition of more WILD-ONE cards.

Chapter 4

Results

4.1 Applications

4.1.1 Hello World!

Typically, the first program that is written to test a new computer, programming language, or other software system is one that displays "Hello World!" on an output device. In keeping with tradition, this was the first program that I implemented on the emulator, proving that it is indeed a general-purpose computer.

The most vital piece of equipment necessary to run "Hello World!" is the display device. Since the express purpose of the PC is to emulate external devices to be connected to the emulator, I first had to write the driver that would turn it into an output terminal for the emulator. All that was necessary was to decide on a memory address that would correspond to character screen output. The emulator would write to that address via the memory-mapped Verilog module. When the PC driver reads the data from the emulator, it would recognize the address and interpret the accompanying data as a character to be printed to the screen.

Since most of the work is done for us by the memory-mapped module and interface library, the actual driver program is very simple:

```
#include "interface.h"

#define MEMSIZE   (1048576)
#define POLLLOC   (0)
#define ADDRLOC   (1)
#define DATALOC   (2)
```

```
#define WRITEOP   (1)
#define READOP    (2)

#define PUTCADDR  (1048576)

int main()
{
  int memory[MEMSIZE];
  int opcode, addr, data;

  interfaceOpen();     /* set up the WILD-ONE board and library */
  interfaceReset();

  while(1) {
    interfaceRead(POLLLOC,opcode);   /* read instruction from PollReg */
    switch (opcode) {
    case WRITEOP:                    /* if emulator wants to write */
      interfaceRead(ADDRLOC,addr);   /*   read address and data */
      interfaceRead(DATALOC,data);
      if (addr == PUTCADDR) {        /*   check for print command */
        putc(data, stdout);          /*     write char to screen */
      } else {
        memory[addr] = data;         /*     else store in memory */
      }
      break;
    case READOP:                     /* emulator wants to read */
      interfaceRead(ADDRLOC,addr);   /*   get requested address */
      interfaceWrite(DATALOC,memory[addr]);   /* and return the data */
      break;
    default:
      break;
    }
  }
  interfaceClose();   /* don't forget to clean up */
  return(0);
}
```

First, notice how the `#define`'d address locations and opcodes match those in the memory-mapped Verilog module:

```
parameter pollAddr = 2'd0, addrAddr = 2'd1, dataAddr = 2'd2;

assign DataOut = (Addr[1:0] == pollAddr) ? PollReg : 'bz;
assign DataOut = (Addr[1:0] == addrAddr) ? AddrReg : 'bz;
assign DataOut = (Addr[1:0] == dataAddr) ? DataOutReg : 'bz;
...
  if (ExtWE) begin
```

```
    PollReg = 1;
...
  if (ExtRE) begin
    PollReg = 2;
...
  else
   PollReg = 0;
```

Looking back at the driver code, you can see that is just sits in a loop and polls the memory-mapped module, waiting for commands. When it reads a code it recognizes, it springs into action. It reads or writes to the PC's memory as requested, with one exception: when the address that the emulator wants to write to corresponds to the predetermined code for character output, the driver program instead translates the accompanying data to a character value and outputs it through the `putc` command. Here the character output address (1048576) has been chosen to be outside of the range assigned to the memory (this example shows a one-megaword integer array) to prevent conflicts. This address space mapping is arbitrary and can be tailored to the specific application without any changes to the interface. Of course, this mapping must match the one used by the application in the emulator:

```
`define PutcAddr 1048576

module hello (Clk, Reset, ExtAddr, ExtDataIn, ExtDataOut,
              ExtWE, ExtRE, ExtStall);
...
always @(posedge Clk)
begin
  if (Reset) begin
    state = 0;
    ExtRE = 0;
    ExtWE = 0;
  end else case (state)
  0:
    begin
      ExtRE = 0;
      ExtWE = 0;
      state = 1;
    end
  1:
    begin
      ExtRE = 0;
      if (ExtStall) begin
```

```
        ExtWE = 0;
        state = 1;
      end else begin
        ExtAddr     = 'PutcAddr;
        ExtDataOut = 72;
        ExtWE = 1;
        state = 2;
      end
    end
  2: ...
```

This hello module would be linked with the memory-mapped module in the high-level netlist and compiled onto the emulator. Writing characters to the interface is no different than writing any other data. In state 1, the application must first wait until the memory-mapped module is free and ready to accept input by checking the *ExtStall* flag. Then it must simultaneously set up the address and data lines with the character output address and the ASCII character value, respectively, and assert the *ExtWE* signal before proceeding to the next state. State 2 is identical to state 1 except for a different character value. It waits for the previous write to complete before writing its value.

That completes the description of the "Hello World!" program. This application exhibits most of the details relevant to developing any application on the emulator and the associated device emulation drivers on the PC. The simplicity of the necessary code illustrates the elegance of my interface design and indicates a significant reduction in the design effort required to program a hardware reconfigurable computer.

4.1.2 Conway's Game of Life

While the "Hello World!" example demonstrated the basic functionality of the interface, it would be desirable to see what can be achieved with a real application. For this experiment, I decided to draw on the RAW benchmark suite and adapt one of the benchmarks, Conway's Game of Life, to my computation model. This is an ideal application for a large reconfigurable computer because it has a large amount of parallelism, as well as a simple computation that easily translates into a circuit.

Conway's Game of Life is a computer science problem that models an array of "cells" over time. Each cell may be alive or dead at any given time step. The state of the cell at time $t + 1$ depends on the state of it and its eight neighbors at time t as follows:

a) If the cell is dead but exactly three of its neighbors are alive, it will come to life, otherwise it will remain dead.

b) If the cell is alive and less than two or more than three of its neighbors are alive it will die from either starvation or overcrowding, otherwise it will remain alive.

This calculation is repeated over and over again for the entire array, making the sea of cells evolve into different patterns over time. Implementing the calculation in software on a single processor machine would require the machine to iterate through each cell of the array in sequence, adding up eight values for each cell. For a two-dimensional array, the compute time grows exponentially with the size of the array.

However, the large sea of gates available to us in the emulator allows us to lay out the entire array of cell calculations at once, limited only by the number of gates in the emulator. Such an approach would transform the algorithm into one that takes constant time, regardless of the size of the array.

On a microprocessor-based computer, we could write the application so that the array resides in memory, is updated by the processor, and is displayed on the screen. Under the reconfigurable computer model, we would like to have the array reside in the emulator and build the circuit so that the array updates itself. We would still want to display it on a screen, so we use the PC to emulate a screen device that can be plugged into the emulator's I/O ports.

Conveniently, the first part has already been done for us — the Life benchmark was developed in Verilog with the emulator as a target. In fact, it turns out that we hardly need to change the code at all. The benchmark already includes a "control" module that takes care of starting and stopping the calculation, as well as reading the array in and writing it out by means of a scan chain mechanism. This module is similar enough in its communication with the emulator pins that it can be used in place of my memory-mapped module. There is a counter at address 0 (conveniently the same address that we defined for PollReg) which the PC writes to in order to start the calculation. The array recalculates and the counter decrements while it is nonzero. By continually polling the counter (address 0) the PC will know when the calculation is done, at which point it may read back the data by means of the familiar `interfaceRead()` library function, reading from the "scan" address as defined in the control module. Whenever the control module detects the resulting read

signals generated by the WILD-ONE circuit, it updates the value in the "scan" address location by rotating the array.

If we wanted to preserve the interface as a monolithic abstraction, we would change the control module to talk to the memory-mapped module instead of directly to the pins. Acting as a true host module, the Life application could use the `getchar` / `putchar` mechanism demonstrated in "Hello World!" to ask the user for a number, and take the initiative to write the state of the array to external memory (on the PC) whenever the counter reaches that number. The drivers on the PC would be continually standing by for input, as well as displaying the relevant section of memory on the screen. In effect, the PC would function as a user terminal and as a raster display device. The section of memory that the emulator writes the array to would function like the video memory of an LCD panel, for example. In fact, we could replace the back end of the memory-mapped module with an LCD interface, plug an LCD screen directly into emulator, and the Life application would never know the difference.

4.2 Performance

With the PC driver program listed in Section 4.1.1 running on a 300 MHz Pentium II PC, reads and writes by the emulator to the PC's memory were measured using a logic analyzer, and found to take on the order of 30 microseconds. This is equivalent to approximately 30 cycles of a 1 MHz Eclk. This ratio is close to the 18 clock cycles the same 300 Mhz PC would take to read from a 60 ns DRAM, and thus works well when porting an application from a PC to the emulator.

However, what one loses in clock speed, one must regain from a direct hardware implementation and from parallelism. The Life application is a case in point. When a 64x64 cell array was run on the 300 Mhz Pentium II, the array was recalculated approximately 2400 times per second. While the emulator runs at only 1 MHz, it is able to calculate the entire array every clock cycle, resulting in performance of one million "generations" of Life per second — a speedup of over 400x compared to the PC!

Since we want to continually display the array, we must take a small hit in performance to take time out to transfer the array data. We only need one bit to encode each cell, so we can transfer the 4096 cells in 128 32-bit words. Using this encoding, I wrote a driver

program that attempts to read the array every (1,000,000/30) generations and obtained a calculation rate of approximately 940,000 generations per second, at an actual frame rate of 28 frames per second. This means that we spend 60 milliseconds of every second reading data from the emulator. At 28 frames per second, we perform 28x128 read operations, so each read operation takes on average 16.7 microseconds. This is somewhat faster than the measured number for a single read presented earlier, because in this case the software read loop is tighter. In any case, we still get a tremendous performance boost by running the Life application on the reconfigurable hardware.

Chapter 5

Discussion

5.1 Conclusions

I have successfully designed and implemented an I/O interface that allows the IKOS VirtuaLogic emulator to be used as a general-purpose reconfigurable host computer. It can have full control of any devices attached to it, since almost any device interface can be adapted for use with the emulator by writing a Verilog driver that conforms internally to my defined memory-mapped model. Furthermore, I have provided the mechanism by which any device can be modeled on a PC and have a simple, standard, and extensible interface to the emulator.

Because of the large amount of hardware available, such an FPGA-based computer can be a viable alternative to the traditional microprocessor model, especially for applications that can take advantage of extensive parallelism. The multimedia applications that are becoming more commonplace today would benefit from this architecture because they often have simple computations can be translated to hardware and replicated many times, and can take advantage of the wide, modular direct I/O that is now available on the emulator.

5.2 Future Work

There are many areas of possible future research. Perhaps the next step now that I have one WILD-ONE card up and running is to develop a framework for adding more cards, using more of the emulator's I/O ports. We would need some simple and effective method for distributing data across the multiple cards on the PCs, and also a technique for consolidating

the data inside the emulator, perhaps by means of the memory-mapped module. Since the emulator and WILD-ONE cards all sample an external clock, the synchronization problem is greatly simplified by the ability to daisy-chain or otherwise distribute a central clock to all the devices.

For many applications, the 30 microsecond latency for interface operations may be a serious bottleneck. This latency can be improved by optimizing the PC driver code, using faster PCs, or using real hardware devices instead. Another approach would be to hide the latency by developing a direct memory access (DMA) model. After some amount of handshaking, the emulator would be able to blast data in or out every Eclk cycle, and we would use the WILD-ONE API's DMA functions to stream the data to the PC's memory.

Another area possibly worth looking at is the emulator itself. The Virtual Wires overhead reduces our clock speed by a factor of 20. With larger and faster FPGAs now available we may be able to significantly reduce or eliminate this penalty, perhaps using more intelligent strategies to partition the computation.

The final step in making the emulator a true general purpose computer would be to eliminate the need to code the application in Verilog and to set up the interfaces manually. We need to be able to take the same high-level language (eg. C) code used for microprocessor machines and automatically generate the hardware and set up the entire system, perhaps even from a remote location.

Appendix A

Source Code

A.1 Memory and character I/O driver

```
/* driver.c
 *
 * extmem module test interface
 *
 * Author: Alex Kuperman
 */

#include <stdio.h>

#include "interface.h"                                              10

#define MEMSIZE      (1048576)
#define POLLLOC      (0)
#define ADDRLOC      (1)
#define DATALOC      (2)
#define SIZELOC      (3)
#define WRITEOP      (1)
#define READOP       (2)

#define PUTCADDR     (1048576)                                      20

int
main()
{
  int *memory;
  int i, opcode, addr, data;

  /* Initialize Memory */
  memory = malloc(MEMSIZE*sizeof(int));
  for(i=0; i<MEMSIZE; i++) {                                        30
    memory[i] = 0;
  }

  interfaceOpen();
  interfaceReset();
```

```
 while(1) {
  interfacefaceRead(POLLLOC,opcode);
  switch (opcode) {
  case WRITEOP:                                                         40
    interfaceRead(ADDRLOC,addr);
    interfaceRead(DATALOC,data);
    if (addr == PUTCADDR) {
         if (data == 13) {        /* For MS-DOS Compatibility */
           putc('\n', stdout);
         } else {
           putc(data, stdout);
         }
    } else {
         memory[addr] = data;                                          50
    }
    break;
  case READOP:
    interfaceRead(ADDRLOC,addr);
    interfaceWrite(DATALOC,memory[addr]);
    break;
  default:
    break;
  }
 }                                                                     60

 interfaceClose();
 return 0;
}
```

A.2 PC library

```
/* interface.h */

void interfaceOpen();
void interfaceReset();
void interfaceClose();

/* Read & Write to Wildfire card interface */

int ifaceRead(int addr);
void ifaceWrite(int addr, int data);                                   10

#define interfaceRead(addr, data)  data = ifaceRead((int)addr)
#define interfaceWrite(addr, data) ifaceWrite((int)addr, data)
```

```
/* interface.c */

#include <stdio.h>
#include <stdlib.h>
#include "interface.h"

/* WILD-ONE include files */
#include <WF1errs.h>
#include <WF1api.h>
                                                                       10
/* Constants */
```

```
#define PCLK_FREQ     (25.0f)
#define ECLK_FREQ     (1.0f)
#define BOARD         (0)
#define READ_OFFSET  (0x00100000)
#define WRITE_OFFSET  (0x00200000)
#define RESET_OFFSET  (0x00300000)
#define ADDR_MASK    (0x000FFFFF)

/* Prototypes */

WF1_RetCode
BoardInitialize();

/* Interface Functions */

WF1_RetCode
interfaceOpen()
{
  WF1_RetCode rc = WF1_SUCCESS;

  rc= BoardInitialize();

  return rc;
}


void
interfaceClose()
{
  WF1_Close( BOARD );
}


void
ifaceWrite(int addr, int data)
{
  DWORD
    fifoDataOut[1],
    wordsWritten;

  fifoDataOut[0] = (addr & ADDR_MASK)|WRITE_OFFSET;
  fifoDataOut[1] = data;
  WF1_FifoWrite( BOARD, WF1_Fifo_Pe0, fifoDataOut, 2, &wordsWritten, 0);
}


int
ifaceRead(int addr)
{
  DWORD
    fifoDataOut,
    fifoDataIn,
    wordsWritten,
    wordsRead;

  fifoDataOut = (addr & ADDR_MASK)|READ_OFFSET;
  fifoDataIn = 0;
  WF1_FifoWrite( BOARD, WF1_Fifo_Pe0, &fifoDataOut, 1, &wordsWritten, 0);
```

```
 WF1_FifoRead( BOARD, WF1_Fifo_Pe0, &fifoDataIn, 1, &wordsRead, 0);

 return (fifoDataIn);
}


void
interfaceReset()
{
 DWORD                                                                    80
   fifoDataOut,
   wordsWritten;

 WF1_FifoReset( BOARD, WF1_Fifo_Pe0);
 fifoDataOut = RESET_OFFSET;
 WF1_FifoWrite( BOARD, WF1_Fifo_Pe0, &fifoDataOut, 1, &wordsWritten, 0);
}
```

A.3 WILD-ONE FPGA Logic

```
-------------------------------------------------------------------------------
-- Entity        : CPE0_Logic_Core
-- Architecture  : Iface
-- Filename      : cpe0lca.vhd
-- Description   : VirtuaLogic-PC Interface guts
-- Date          : 5/21/99
-------------------------------------------------------------------------------

------------------------------- Library Declarations -------------------------------
                                                                          10
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

---------------------------- Architecture Declaration ----------------------------

architecture iface of CPE0_Logic_Core is

  type states is (Master, Dispatch, GetAddr, Read0, Read1, Write0, Write1);
  signal Present_State, Next_State: states;                               20

  signal Address_Reg: std_logic_vector (35 downto 0);
  signal load_addr, clr_addr, load_outreg, load_counter: std_logic;
  signal eclk_1, eclk_2, eclk_rising_edge: std_logic;

 begin

  eclk_detect: process (CPE_Pclk, CPE_Eclk, eclk_1)
  begin
    if ( rising_edge(CPE_Pclk) ) then                                     30
      eclk_1 <= CPE_Eclk;
      eclk_2 <= eclk_1;
    end if;
  end process eclk_detect;

  eclk_rising_edge <= eclk_1 AND NOT eclk_2;

  addr_reg: process (CPE_Pclk, CPE_Reset, load_addr, clr_addr, load_outreg)
```

```
begin
  if ( CPE_Reset = '1' ) then
    Address_Reg(35 downto 0) <= (others => '0');
    CPE_PE1_Right_Out(35 downto 0) <= (others => '0');
  else
    if ( rising_edge(CPE_Pclk) ) then
      if ( clr_addr = '1' ) then
        Address_Reg(35 downto 0) <= (others => '0');
        CPE_PE1_Right_Out(35 downto 0) <= (others => '0');
      else
        if ( load_addr = '1' ) then
          Address_Reg(35 downto 0) <= CPE_HostToPeFifoData_In(35 downto 0);
        else
          Address_Reg(35 downto 0) <= Address_Reg(35 downto 0);
        end if;
        if ( load_outreg = '1' ) then
          CPE_PE1_Right_Out(35 downto 0) <= Address_Reg(35 downto 0);
        end if;
      end if;
    end if;
  end if;
end process addr_reg;

CPE_PE1_Right_OE <= ( others => '1');
CPE_FifoSelect <= "10";                    -- Select on-board fifo
CPE_PeToHostFifoData_OE <= ( others => '0' );   -- Disable fifo data output

state_clocked: process(CPE_Reset, CPE_Pclk)
begin
  if (CPE_Reset = '1') then
    Present_State <= Master;
  elsif ( rising_edge(CPE_Pclk) ) then
    Present_State <= Next_State;
  end if;
end process state_clocked;

control_fsm: process(Present_State, eclk_rising_edge, Address_Reg,
                    CPE_HostToPeFifoEmpty_n)
begin
  case Present_State is
  when Master =>
    load_outreg <= '0';
    load_addr <= '0';
    CPE_Fifo_WE_n <= '1';
    if ( eclk_rising_edge = '1' ) then
      clr_addr <= '1';
    else
      clr_addr <= '0';
    end if;
    if ( CPE_HostToPeFifoEmpty_n = '1' ) then
      CPE_FifoPtrIncr_EN <= '1';
      Next_State <= GetAddr;
    else
      CPE_FifoPtrIncr_EN <= '0';
      Next_State <= Master;
    end if;
  when GetAddr =>
    clr_addr <= '0';
    load_addr <= '1';
```

```
load_outreg <= '0';
CPE_Fifo_WE_n <= '1';
CPE_FifoPtrIncr_EN <= '0';                                          100
Next_State <= Dispatch;
when Dispatch =>
 clr_addr <= '0';
 load_addr <= '0';
 CPE_Fifo_WE_n <= '1';
 case Address_Reg(21 downto 20) is
 when "00" =>                          -- nop
  CPE_FifoPtrIncr_EN <= '0';
  if ( eclk_rising_edge = '1' ) then
   load_outreg <= '1';                                              110
   Next_State <= Master;
  else
   load_outreg <= '0';
   Next_State <= Dispatch;
  end if;
 when "01" =>                          -- read
  CPE_FifoPtrIncr_EN <= '0';
  if ( eclk_rising_edge = '1' ) then
   load_outreg <= '1';
   Next_State <= Read0;                                             120
  else
   load_outreg <= '0';
   Next_State <= Dispatch;
  end if;
 when "10" =>                          -- write
  load_outreg <= '0';
  CPE_FifoPtrIncr_EN <= '0';
  if ( CPE_HostToPeFifoEmpty_n = '1' ) then
   Next_State <= Write0;
  else                                                              130
   Next_State <= Dispatch;
  end if;
 when "11" =>                          -- reset
  CPE_FifoPtrIncr_EN <= '0';
  if ( eclk_rising_edge = '1' ) then
   load_outreg <= '1';
   Next_State <= Master;
  else
   load_outreg <= '0';
   Next_State <= Dispatch;                                          140
  end if;
 end case;
when Read0 =>
 load_addr <= '0';
 load_outreg <= '0';
 CPE_Fifo_WE_n <= '0';
 if ( eclk_rising_edge = '1' ) then
  CPE_FifoPtrIncr_EN <= '1';
  clr_addr <= '1';
  Next_State <= Read1;                                              150
 else
  CPE_FifoPtrIncr_EN <= '0';
  clr_addr <= '0';
  Next_State <= Read0;
 end if;
when Read1 =>
```

```
      load_addr <= '0';
      load_outreg <= '0';
      CPE_Fifo_WE_n <= '0';
      CPE_FifoPtrIncr_EN <= '0';                          160
      if ( eclk_rising_edge = '1' ) then
        clr_addr <= '1';
        Next_State <= Master;
      else
        clr_addr <= '0';
        Next_State <= Read1;
      end if;
    when Write0 =>
      load_addr <= '0';
      clr_addr <= '0';                                    170
      CPE_Fifo_WE_n <= '1';
      if ( eclk_rising_edge = '1' ) then
        CPE_FifoPtrIncr_EN <= '1';
        load_outreg <= '1';
        Next_State <= Write1;
      else
        CPE_FifoPtrIncr_EN <= '0';
        load_outreg <= '0';
        Next_State <= Write0;
      end if;                                             180
    when Write1 =>
      load_addr <= '0';
      load_outreg <= '0';
      CPE_Fifo_WE_n <= '1';
      CPE_FifoPtrIncr_EN <= '0';
      if ( eclk_rising_edge = '1' ) then
        clr_addr <= '1';
        Next_State <= Master;
      else
        clr_addr <= '0';                                  190
        Next_State <= Write1;
      end if;
    end case;
  end process control_fsm;

end iface;
```

A.4 Memory-Mapped Module

```
`include "main_define.v"

module extmem (Clk, Reset, RD, WR, Addr, DataIn, DataOut,
               ExtDataIn, ExtDataOut, ExtAddrIn, ExtAddrOut,
               ExtWE, ExtRE, ExtStall );

  input                          Clk, Reset, RD, WR;
  input [`GlobalAddrWidth-1:0]   Addr;
  input [`GlobalDataWidth-1:0]   DataIn;
  output [`GlobalDataWidth-1:0]  DataOut;                 10

  input [`GlobalDataWidth-1:0]   ExtAddrOut, ExtDataOut, ExtAddrIn;
  output [`GlobalDataWidth-1:0]  ExtDataIn;
  input                          ExtWE, ExtRE;
  output                         ExtStall;
```

```
reg [`GlobalDataWidth-1:0]      AddrReg, DataInReg, DataOutReg;
reg [2:0]                       PollReg;
reg                             ExtStall;
reg [2:0]                       state;                                          20

wire [1:0]                      lowAddr;

// state declarations
parameter                       start = 3'b000, write = 3'b001, read = 3'b010,
                                DMAwrite = 3'b101, DMAread = 3'b110;
parameter                       pollAddr = 2'd0, addrAddr = 2'd1, dataAddr = 2'd2;

assign lowAddr = Addr[1:0];
                                                                                30
// Set output to PC based on addr bits (combinational)
assign DataOut = (lowAddr == pollAddr) ? PollReg : `GlobalDataHighZ;
assign DataOut = (lowAddr == addrAddr) ? AddrReg : `GlobalDataHighZ;
assign DataOut = (lowAddr == dataAddr) ? DataOutReg : `GlobalDataHighZ;

assign ExtDataIn = DataInReg;

// main state machine
always @(posedge Clk or posedge Reset)
  begin                                                                         40
      if (Reset) begin
        state = start;
        PollReg = 0;
        AddrReg = 0;
        DataInReg = 0;
        DataOutReg = 0;
      end else
      case (state)
        start:
          begin                                                                 50
              if (ExtWE) begin
                AddrReg = ExtAddrOut;
                DataOutReg = ExtDataOut;
                PollReg = 1; // Mem Write opcode
                state = write;
              end else if (ExtRE) begin
                AddrReg = ExtAddrIn;
                DataOutReg = ExtDataOut;
                PollReg = 2; // Mem Read opcode
                state = read;                                                   60
              end else begin
                PollReg = 0;
                AddrReg = 0;
                DataOutReg = 0;
                state = start;
              end
          end
        write:
          begin
              if (RD && (lowAddr == dataAddr)) begin                            70
                PollReg = 0;
                state = start;
              end else state = write;
          end
```

```
      read:
        begin
            if (WR && (lowAddr == dataAddr)) begin
              DataInReg = DataIn;
              PollReg = 0;
              state = start;                                          80
            end else state = read;
        end
      default: state = 'bx;
    endcase
  end // always @ (posedge Clk)

always @(state or ExtWE or ExtRE or RD or WR or lowAddr)
  begin
    case(state)
      start:  ExtStall = ExtWE || ExtRE;                               90
      write:  ExtStall = 1;
      read:    ExtStall = 1;
      default:  ExtStall = 0;
    endcase // case(state)
  end // always @ (state)

endmodule
```

A.5 "Hello World!" Emulator Program

```
`define PutcAddr 1048576

module hello (Clk, Reset, ExtAddr, ExtDataIn, ExtDataOut,
         ExtWE, ExtRE, ExtStall);

input Clk,Reset;
input  [`GlobalAddrWidth-1:0] ExtAddr;
input  [`GlobalDataWidth-1:0] ExtDataIn;
output [`GlobalDataWidth-1:0] ExtDataOut;
output ExtStall;                                                       10

reg [`GlobalDataWidth-1:0] ExtAddrOut, ExtDataOut, ExtAddrIn;
wire ExtStall;

reg [3:0] state;

always @(posedge Clk)
begin
        if (Reset) begin
          state = 0;                                                   20
          ExtRE = 0;
          ExtWE = 0;
        end else case (state)
        0:
                begin
                        ExtRE = 0;
                        ExtWE = 0;
                        state = 1;
                end
        1:                                                             30
                begin
```

```
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 72;
                        ExtWE = 1;
                        state = 2;
                end                                             40
        end
2:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 101;
                        ExtWE = 1;                              50
                        state = 3;
                end
        end
3:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;                 60
                        ExtDataOut = 108;
                        ExtWE = 1;
                        state = 4;
                end
        end
4:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;                              70
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 108;
                        ExtWE = 1;
                        state = 5;
                end
        end
5:
        begin
                ExtRE = 0;                                      80
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 111;
                        ExtWE = 1;
                        state = 6;
                end
        end
6:                                                              90
```

```
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 32;
                        ExtWE = 1;
                        state = 7;
                end                                                     100
        end
7:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 87;
                        ExtWE = 1;                                      110
                        state = 8;
                end
        end
8:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;                         120
                        ExtDataOut = 111;
                        ExtWE = 1;
                        state = 9;
                end
        end
9:
        begin
                ExtRE = 0;
                if (ExtStall) begin
                        ExtWE = 0;                                      130
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 114;
                        ExtWE = 1;
                        state = 10;
                end
        end
10:
        begin
                ExtRE = 0;                                              140
                if (ExtStall) begin
                        ExtWE = 0;
                end else begin
                        ExtAddrOut = `PutcAddr;
                        ExtDataOut = 108;
                        ExtWE = 1;
                        state = 11;
                end
        end
```

```
    11:                                                              150
            begin
                ExtRE = 0;
                if (ExtStall) begin
                    ExtWE = 0;
                end else begin
                    ExtAddrOut = `PutcAddr;
                    ExtDataOut = 100;
                    ExtWE = 1;
                    state = 12;
                end                                                  160
            end
    12:
            begin
                ExtRE = 0;
                if (ExtStall) begin
                    ExtWE = 0;
                end else begin
                    ExtAddrOut = `PutcAddr;
                    ExtDataOut = 33;
                    ExtWE = 1;                                       170
                    state = 13;
                end
            end
    13:
            begin
                ExtRE = 0;
                if (ExtStall) begin
                    ExtWE = 0;
                end else begin
                    ExtAddrOut = `PutcAddr;                          180
                    ExtDataOut = 13;
                    ExtWE = 1;
                    state = 14;
                end
            end
    14:
      begin
        ExtRE = 0;
        ExtWE = 0;
        state = 14;                                                  190
      end
   default: state = 'bx;
     endcase
end

endmodule // hello
```

Bibliography

[1] Annapolis Micro Systems, Inc., Annapolis, MD. *WILD-ONE(tm) Reference Manual*, 1999. Revision 3.3.

[2] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW benchmark suite: Computation structures for general purpose computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[3] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming pin limitations in FPGA-based logic emulators. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142–151, Napa, CA, April 1993.

[4] T. Bauer. The design of an efficient hardware subroutine protocol for FPGAs. Master's thesis, MIT, Department of Electrical Engineering and Computer Science, May 1994.

[5] IKOS Systems, Inc. *VirtuaLogic Emulation System Documentation*, 1996. Version 1.2.