# A Hardware Testbed for

# Distributed Heterogeneous Adaptive Computing

*Christopher Bachmann*

**Technical Report No. CPDC-TR-9905-007**



May 1999

*Center for Parallel and Distributed Computing*

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

# A Hardware Testbed for Distributed Heterogeneous Adaptive Computing

by

Christopher Bachmann

THESIS

Submitted in partial fulfillment of the requirements

for the degree of Master of Science in Electrical and Computer Engineering

in the Graduate School of

Northwestern University, 1999

Evanston, Illinois

## Abstract

Adaptive computing systems are composed of reconfigurable hardware elements that are capable of adapting to various computation requirements. Using these reconfigurable hardware elements, adaptive computing systems provide significant performance improvements over conventional non-reconfigurable systems. However, completely reconfigurable systems, such as those based entirely around FPGAs, become very unsuitable for implementing complex applications. This is due to both the time to develop algorithms and then implement them, as well as the FPGAs being slower than processor's internal computational units when implementing the same functions. The solution, being developed by the MATCH project, is to merge various commercial-off-the-shelf (COTS) components into one cohesive unit. Such components would include not only FPGAs, but DSPs and embedded processing elements as well. This allows the advantages of each element to be combined, such as the ability of the FPGAs to deal with more custom processing. The primary goal of the MATCH project is to explore methods for bridging the gap between the high level of applications developers, and the low level control required by the adaptive computing environment. This would include new developments in compiler technologies to take advantage of such a heterogeneous environment. In order to test the new theories and ideas of the MATCH system developers a testbed was developed including the necessary custom components targeted by the MATCH system developers. The actual testbed itself consists of a Motorola embededded board, an Annapolis Microsystems WILDCHILD, a Transtech DSP board, and a Force 5V Sparc based board. This thesis will present prior work that has been done in the area of adaptive computing, the testbed that we have developed, and the results of various benchmarks on the testbed, including matrix multiply on the various components.

# Acknowledgements

I wish to express my gratitude and thanks to my advisor, Dr. Prithviraj Banerjee. I would also like to thank my committee members, Dr. Alok Choudhary, Dr. Scott Hauck and Dr. Nagaraj Shenoy, for their valuable comments and suggestions on my work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem Description

Adaptive computing systems are composed of reconfigurable hardware elements that are capable of adapting to various computation requirements. Using these reconfigurable hardware elements, adaptive computing systems provide significant performance improvements over conventional non-reconfigurable systems. However, completely reconfigurable systems, such as those based around FPGAs, become very unsuitable for implementing complex applications. This is due to both the time to develop algorithms and then implement them, as well as the FPGAs being slower than processor's internal computational units when implementing the same functions.

The solution, being developed by the MATCH project, is to merge various commercial-off-the-shelf (COTS) components into one cohesive unit. Such components would include not only FPGAs, but DSPs and embedded processing elements as well. This allows the advantages of each element to be combined, such as the ability of the FPGAs to deal with more custom processing. The primary goal of the MATCH project is to explore methods for bridging the gap between the high level of applications developers, and the low level control required by the adaptive computing environment. This would include new developments in compiler technologies to take advantage of such a heterogeneous environment.

In order to test the new theories and ideas of the MATCH system developers a testbed was developed including the necessary custom components targeted by the MATCH system developers. For the three basic parts of the compilation environment, the embedded processors, the FPGAs, and the DSPs, we needed to choose the most popular manufacturer used today. For this reason, we chose the Motorola PowerPC as our embedded processor, Xilinx as the manufacturer of our FPGAs, and Texas Instruments as the manufacturer of our DSPs. These components are merged into one heterogeneous environment through the use of a common chassis, thus resulting in the MATCH testbed. An overview of the testbed and MATCH project is shown in Figure 1.1. The MATCH project consists of various tasks dealing with the development of a MATLAB compiler. This thesis relates to Task 1 of the MATCH project as referenced in Figure 1.2.

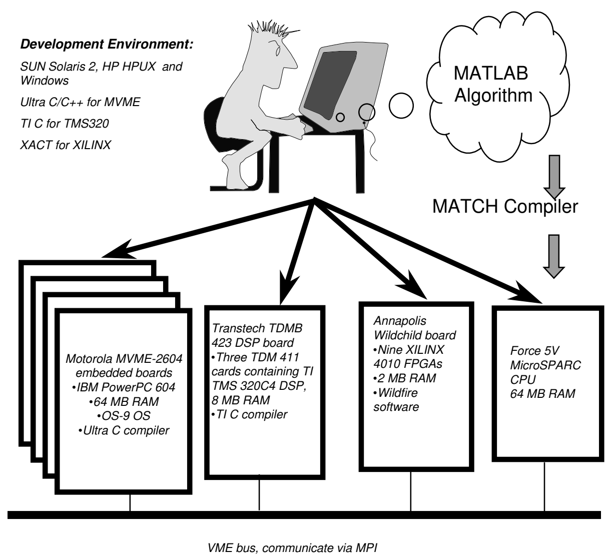


Figure 1.1: Testbed Overview

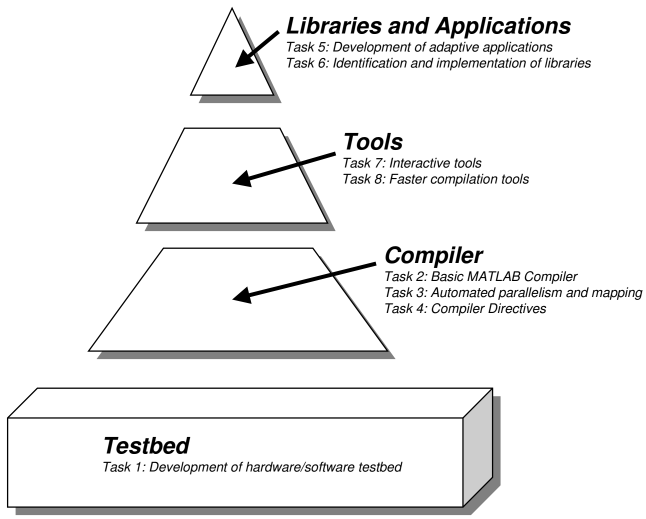


Figure 1.2: MATCH Project Overview

## 1.2 Related work

While our main work was in developing a heterogeneous hardware environment for adaptive computing, much work has been done within the adaptive computing area. These consist of software approaches to mapping applications to adaptive computing platforms, new hardware combining the technologies of processors and FPGAs, and new developments in the FPGA areas.

### 1.2.1 Hardware Developments

Within the hardware development area of adaptive computing are two basic classifications. The first class consists of devices which combine general purpose computing devices with reconfigurable coprocessors. By using this coprocessor technology they take advantage of the power of reconfigurable units as well as the general purpose processing devices. The second classification is those designs which use only reconfigurable technology.

These designs are typically used as the engines themselves for computations, where a majority of the processing for the application is done on the reconfigurable devices themselves and not on general purpose processing devices.

The RAW [W+97] architecture developed at MIT approaches the configurable logic with processor concept at a very fine grained level. Each processing unit consists of a small amount of instruction and data memory, an ALU for general purpose processing, a small amount of configurable logic, and a switch interconnect. The switch interconnect joins all the processing units together into a large mesh. The compiler can then allocate statically data streams and schedules for the data. The ALU for each tile is designed such that it supports both bit, byte, and word-level operations. The programmable logic within the tile also allows for application specific operations to be added into the tile's logic. Such a fine-grained approach simplifies the hardware cost of design and development, and allows for multiple processors to be combined easily into a single chip. This approach shows significant speedups compared to a standard SparcStation.

The NAPA1000 Adaptive Processor is currently being produced by National Semiconductor [G+99]. The NAPA1000 features a merging of FPGA and RISC processor technology. To help aid in merging these two functional units into a multiprocessing environment a new parallel processing bus was also developed. Combining the FPGA and RISC technologies together, they allow an application to be designed with a typical software approach, and then the core parts to be used on the FPGA array, while the processor controls the outer loops of the computations. The ToggleBus parallel processing bus allows for arrays of up to 1024 elements to be combined efficiently for large processing units without the need for external crossbars. The basic goals of it's final design hope to achieve 2 orders of magnitude in cost/performance improvement for bit and byte computation over traditional processors.

The Chimaera [H+99] project is based upon creating a new hardware system consisting of a microprocessor with an internal reconfigurable functional unit. This solution takes advantage of the microprocessor's ability to do the bulk of the work, while the reconfigurable unit performs the compute intensive kernels. The major difference with this reconfigurable unit is that it is not based upon a standard FPGA design, but a new design which is specifically geared towards interacting with the microprocessor. This interaction allows the reconfigurable unit to work tightly with the microprocessor, sharing it's registers and busses to compute results within the program stream, instead of as an outside coprocessor. Hand mappings of applications with small optimizations showed speedups of between one and two, however aggressive mappings showed significant speedups over normal applications.

At the University of California, Berkeley, the GARP architecture [HW97] has been developed. This architecture utilizes a MIPS processor core, tied with a reconfigurable unit. This reconfigurable unit is classified as a slave computational unit, similar to the Chimaera [H$^+$99] architecture above. However, in this implementation the reconfigurable unit is more of a coprocessor, in that it computes alongside the processor, instead of in line as the Chimaera unit does. Several new instructions are added to the instruction set to facilitate the loading and operation of the coprocessor, while the Chimaera unit loads it's configuration in line. Simulation results of the GARP architecture have shown significant speedups over a standard processor.

Virtual Computer Corporation's (VCC) [Sch98] HotWorks Development System consists of two basic parts. The first is a hardware board, called the H.O.T. Works PCI Board. This board fits in a PCI slot on a modern PC. It contains a fast reconfigurable processing unit (RPU) from Xilinx and an FPGA for interfacing with the PCI bus on the PC. It also contains memory and a mezzanine slot for expandibility. The software development environment allows for basic design entry, placement and routing, and a runtime program format. The final step allows a user to write custom C++ applications which embed the reconfigurable unit's bitstream into the application. Due to this technique, the application can quickly reconfigure the HOTworks board as it runs, allowing added flexibility. This rapid reprogrammibility offers a more flexible architecture in which to design applications.

The ACEcard [DH98] is a high-performance, run-time reconfigurable coprocessor. Its primary use is to help enable the use of reconfigurable computing in high-performance, multi-threaded, multi-user environments. It consists of an embedded processor (a microSPARC IIep), memory and two Xilinx 6264 reconfigurable processor units, as well as the expansion capabilities of PCI. This is coupled with a run-time environment that makes utilizing the capabilities of the ACEcard as simple as setting up the data, and performing the function call. The embedded processor on the ACEcard then performs all the tasks necessary to generate the desired computation. This allows the host processor to continue with other tasks, while the ACEcard is computing. Such an environment creates a powerful tool for applications that need fast reconfigurability. Recently TSI labs has announced the ACEcard II, consisting of two Xilinx 4035 FPGAs and a microSPARC II CPU with 64MB of DRAM .

The WILDFORCE XL [FHP98] is classified as a reconfigurable engine, containing five Xilinx 4062XL FPGAs as processing elements. This board is designed such that the five FPGAs are connected both in a systolic array and by a crossbar switch, so that the parallelism of a given application may be exploited. Also attached to each processing element is a mezzanine card connection for the addition of memory local to the processing element. The board is connected through three FIFOs to the PCI interface, which can then be connected to the host processor. The board also contains several external I/O connections for direct integration of digital I/O into the board. The software accompanying the WILDFORCE XL allows the user to synthesis VHDL, map it, and then move it to the board, where an application can make direct use of it. This easily adds high-performance reconfigurable abilities to a standard PC. Annapolis has recently announced their WILDSTAR series which uses Xilinx Virtex FPGAs, instead of the older Xilinx 4000 series FPGAs.

### 1.2.2 Software Developments

Several different works have been done on the development of software which can help reduce the amount of time to take a high level application and map it to a hardware system. The CORDS [DJ98] system takes an application and maps it embedded systems containing reconfigurable FPGAs. RTExpress [BBB+98] takes MATLAB as input and then maps it to high performance computers and embedded systems. Work [GDM92] has also been done in taking applications and mapping them to processors with additional hardware coprocessors.

The CORDS [DJ98] system takes advantage of new FPGAs ability to be reconfigured while the system is operational. It synthesizes an embedded system containing dynamically reconfigurable FPGAs. Normally, FPGAs require a time-consuming reconfiguration delay. However, the CORDS system deals with this by scheduling tasks by using a preemptive, dynamic priority, multi-rate scheduler. This scheduler takes as input a task graph, a set of communication processors, general purpose processors, and FPGAs. It assigns the tasks to the processors, and then develops the schedule for the tasks and communication events. It's algorithm attempts to minimize the total reconfiguration time while still maintaining the priorities of the tasks. The results show that using dynamically reconfigurable FPGAs in this way, has the ability to allow synthesis of hardware which might not be normally possible, and with a possible reduction in price.

In [GDM92], they take the problem of converting an algorithm described by complex behavioral descriptions, into hardware and software working together. The hardware targeted by this method is a general purpose processor combined with application specific hardware support. The process first consists of partitioning the problem into hardware and software sections, and then mapping those partitions onto the available system hardware. The result is a general purpose processor with part of the task, and an application specific coprocessor that handles the rest. The combined work often requires less hardware than a full hardware solution, and offers the expandibility of a software solution, with similar performance.

RTExpress [BBB+98] is a software tool, which provides the user with an environment to assist them in the developing of software for real-time embedded systems. This tool takes as input MATLAB, and then compiles it so that it may be directly executed on a target high performance computer. This set of tools reduces the normal design cycle by eliminating the task of taking MATLAB code and turning it into compilable high performance code by hand. This is accomplished by using RTExpress to generate parallel C code for the required high performance machine. Once this is accomplished, the C code can be compiled and linked with vender specific libraries as well as th RTExpress parallel function library. The result is executable code that will perform the functions described by the original MATLAB script. This reduces the design time, and helps to eliminate flaws that would normally be introduced in the normal MATLAB to C conversion process.

## 1.3 Contributions

The main contributions of this thesis include identifying the components for building the adaptive computing testbed, integrating the various hardware and software elements to make the testbed function as a whole, and preliminary micro-benchmarking of the testbed. For the first part, we had to choose the various components of the testbed, the basic backplane type to be used, and how the components would be installed. Then we needed to assemble the components, and load their software in such a manner that they could communicate. Finally, we put together a small suite of benchmark programs to give us an idea of the performance of the testbed.

## 1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we describe the architecture of the testbed, and its major components. And finally, in Chapter 3, we discuss some basic measurements made to help characterize the testbed's performance.

CHAPTER 2

# Adaptive Computing Testbed

The testbed that we have designed to work with the MATCH project consists of five major parts. The first part consists of the communication backbone for the various other parts. This consists of the VME Chassis and the networking subsystem. The second part is the Force 5V board, which constitutes the central point of control. The third part is the Annapolis WILDCHILD, which holds the FPGA subsystem. The fourth part is the Transtech DSP board. The final part of the system is the Motorola MVME2604 boards which constitute the primary general purpose embedded computing environment.

## 2.1 The Communication Subsystems

The major part of the communication subsystem is a VME chassis from Annapolis Micro Systems. This not only physically houses the various other components of the MATCH testbed, but it also allows them to communicate and interact with each other over the VMEbus. The second part of the communication subsystem is the 10/100BaseT ethernet connection. Combined, these offer several methods for the various parts of the MATCH testbed to communicate.

The VME chassis was chosen over other backplanes for several reasons. The first was that the embedded board that we had decided to use was only available in a VME configuration. The availability of a DSP board and the FPGA board with a VME interface made our decision to pick the VME chassis easier.

Before deciding on the VME chassis we also looked at PCI. However, several components of our project were not available in PCI versions. The most notable component was the PowerPC board. Because of this, we discovered that we would have to use a VME to PCI bridge, which would cause an additional point of failure and contention. We also discovered that the FPGA boards that came in PCI form were greatly limited in the number of actual FPGAs that could be placed upon them. Such design constraints caused us to make our decision in favor of a VMEbus based design.

VMEbus is a standard that has been around for over 17 years. Its long run has resulted in the implementation of many off-the-shelf parts designed to work with it. VMEbus standardizes the form factor, control interface, address/data bus, and multiprocessing utilities. It allows for up to 21 slots, with multiple masters and slaves to coexist.

The initial design of VMEbus allowed for up to 32 bit address and 32 bit data transfers to allow for a theoretical bandwidth of approximately 40MB/s. Current implementations, which we do not use since they aren't supported by all of our equipment, double this by allowing 64 bit data transfers, giving a theoretical maximum of 80MB/s.

Our particular VME backplane consists of twelve slots, so that we have a large amount of expandability. Figure 2.1 shows a picture of how we have inserted the boards in the VME chassis. We use every other slot to help minimize heat related problems, and even that allows for quite a bit of expandability. The first slot is the Force 5V, which acts as the bus arbitrator. The third slot contains the WILDCHILD FPGA board. Next, in the fifth slot is the Motorola PowerPC board. Following that we have left an empty slot for another Motorola board, and finally in the ninth slot we have inserted the Transtech DSP board.

In addition to the VMEbus, we have also installed a 10/100BaseT network connecting the main processors in the system. Each Motorola PowerPC board contains a 100BaseT interface embedded on it. The Force 5V also contains a 10BaseT interface, which we have connected through a 3com hub to each other. One port on the hub also connects to a Sun Ultra 5, which is the connection to the outside world. This network allows us to run commonly available TCP/IP protocols to interface the embedded, DSP, and FPGA boards to the outside

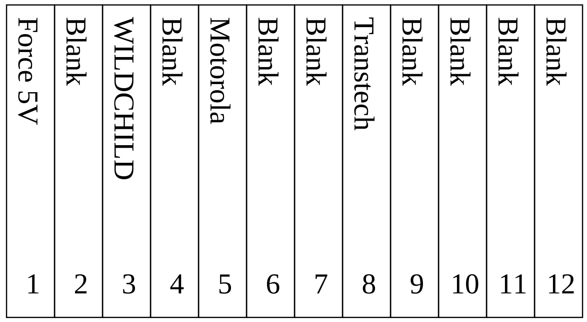


Figure 2.1: VME Chassis Layout

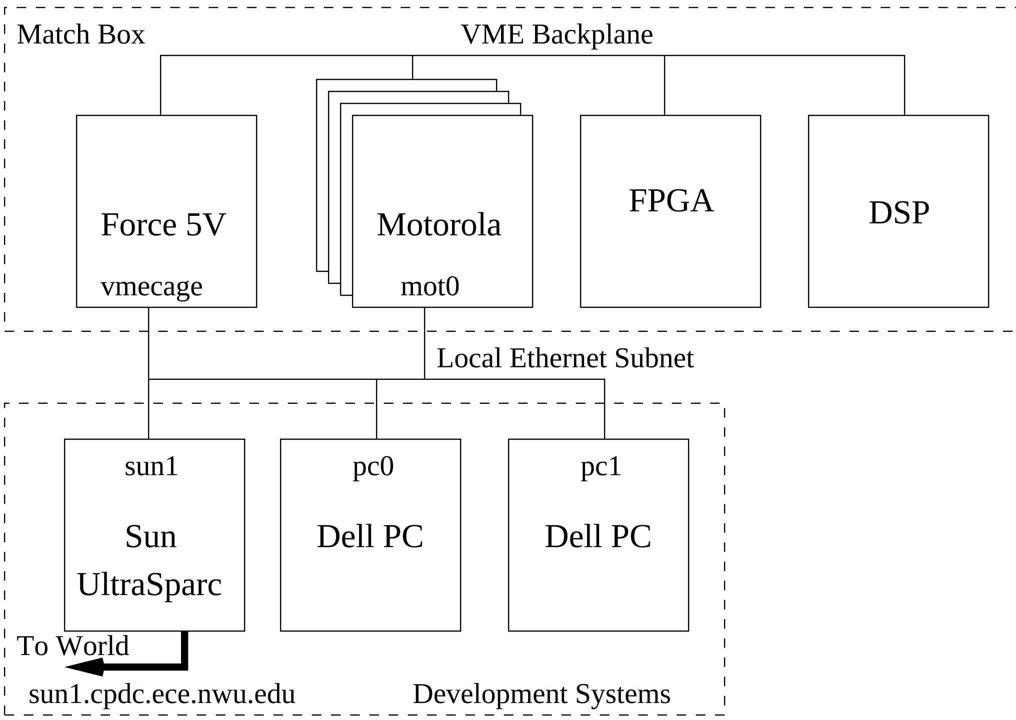


Figure 2.2: Testbed System

world. Figure 2.2 shows a diagram of how the network connects the various parts of the testbed.

## 2.2 Control Processor

The Force 5V acts as the central point of control for our testbed. As such, it is the most important point in the design.

### 2.2.1 Force 5V Hardware

The Force 5V is basically a SparcStation 5 on a VME card. Figure 2.3 shows a diagram of the internal layout of the Force 5V board. The Force 5V is powered by the microSPARC-II processor running at 85MHz. The board that we have has 64MB of RAM, and an attached 2.1 Gigabyte SCSI disk drive. The board also has a 10Mbps ethernet interface, as well as dual serial ports for console communication. Also available are two SBus slots for standard communication with SBus devices. This allows for future expansion of the Force5V.

The Force 5V communicates with the VME backplane through a custom chip designed by Force Corporation. This chip allows the Force 5V to function as a VMEbus Master or Slave. It also handles the interrupt, DMA, and arbiter functions of the VMEbus. This chip takes up two virtual SBus slots and as such glues the microSPARC-II based system onto the VMEbus. This interfaces moves data at 40Bbytes/sec between local memory and the VMEbus. It provides the maximum VME bandwidth, helping to transfer data to and from other boards on the VME backplane.

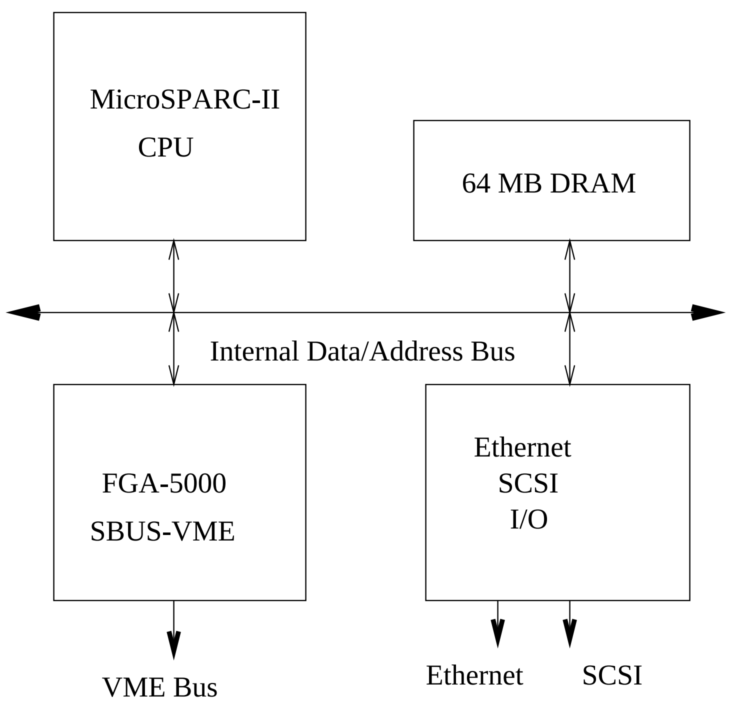


Figure 2.3: Force 5V Board

### 2.2.2 Force Software

The Force 5V runs Solaris 2.5.1 from Sun Microsystems. It is set up as a basic Solaris standalone machine with a private IP address for the local network. It has only one account for remote logins set up for the MATCH project. We have installed specific drivers for talking to the VME backplane. On top of these run the two drivers for talking with the Transtech DSP board and the WILDCHILD board.

Also installed on the Solaris system is the software for compiling for the WILDCHILD and the Transtech DSP board. This software consists of gcc, as well as the custom headers and programs needed for the DSP and FPGA boards.

## 2.3 Digital Signal Processors

The third component of our system is a DSP board containing th Texas Instruments TMS320C40 DSP processor. While we realize that the C40 is not the latest state of the art DSP processor (the latest are the C60 series), we chose this since the C compiler and software technology for this processor is relatively stable and we wanted to develop a testbed as a proof of concept of our MATCH compiler as opposed to the highest performance testbed.

The Transtech TDMB423 DSP board provides for the mounting and control of the DSP portion of our system.

### 2.3.1 Transtech TDMB423 Hardware

The Transtech DSP board consists of three mounting points for TIM modules, a JTAG Master, and the glue logic necessary to interface the TIM modules to the VME Bus. Figure 2.4 shows the internal layout of the TDMB423. This board was chosen because it has the necessary JTAG Master for the debugging of the modules on the board. It also has a maximum of three installable TIM modules, of which we use two currently. The JTAG master capabilities of this board support debugging of the DSP modules through the Transtech C40 debugging software.

The TMS320C40 provides six communication ports which are bidirectional and can theoretically perform data transfers up to 20MBytes/Sec. A DMA coprocessor on chip allows for communications I/O and computation simultaneously, allowing the CPU to perform computations and improving performance. Our C40 processors are running at 60MHz, which delivers a performance rating of approximately 60MFLOPS/300MOPS.

The TDMB423 connects comport (communication port) 3 of the TIM module at site one to the VME bus, through FIFO logic, so that repeated polling is unnecessary. The board then connects comport 0 of TIM 1 to comport 3 of TIM 2. Comport 0 of TIM 2 is then connected to comport 3 on TIM 3. Comport 0 on TIM 3 is left unconnected, however comports 1, 2, 4, and 5 on each TIM module are connected to straps on the outer edge of the TDMB423. This is diagrammed in Figure 2.4.

Each TIM module is a Transtech TDM411-4-4-60. This means that each TIM contains a TMS320C40 DSP running at 60MHz combined with 8 MBytes of ultra fast EDRAM memory. These 8 MBytes are split into two groups of 4 MBytes each. One block of memory is designated as local to the processor, and the other block is designated as global. The EDRAM memory combines a 2Kbit SRAM cache with a 4Mbit DRAM. This allows for single cycle reads, and two cycle page misses, which both fit within the zero wait state read on the C40's 100MByte/sec data bus.

Internally, the C40 utilizes a 40/32 bit floating/fixed point multiplier/ALU. It has hardware support for divide and inverse square root operations. Memory can be addressed in linear, circular, and bit reversed modes. Fast bit manipulation is aided with a single cycle barrel shifter. Also the DSP includes a on-chip 512-byte instruction cache, 8Kbytes of single cycle dual-access instruction or data RAM, a ROM based bootloader and an onboard JTAG-controlled analysis module for aiding in debugging.

### 2.3.2 Transtech Software

The Transtech DSP system includes the PaCE development system. This includes the tools necessary to develop C code, execute the code, and then debug the code for the TI DSP chips. The PaCE development system contains the TI C compiler for the C4x processors, TI development tools such as the assembler, linker and archiver, and then the TI debugger. Transtech adds to these basic tools their own TOPS host server, which allows the user to access the host I/O facilities including the console and file systems.

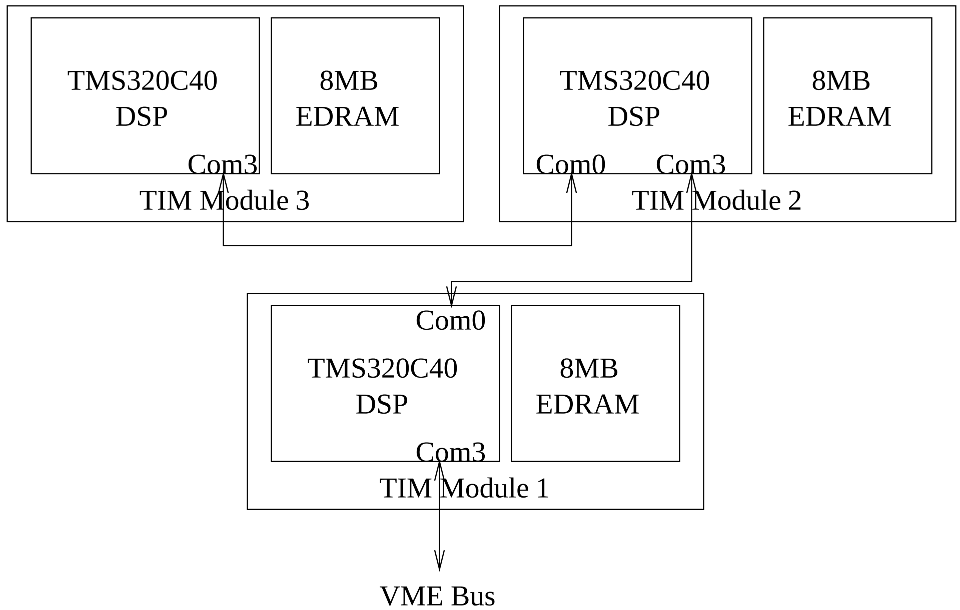


Figure 2.4: Transtech DSP Board

## 2.4 Reconfigurable Processors

The Annapolis Micro Systems WILDCHILD provides the interface and the mounting of the FPGA part of our system. We chose the WILDCHILD board since it uses the Xilinx 4000 series FPGAs which are the most popular FPGAs used in industry today, and the fact that it has eight FPGAs so that we could test designs with a moderate number of FPGAs in parallel.

### 2.4.1 WILDCHILD Hardware

The Annapolis WILDCHILD board provides several highly reconfigurable resources. This includes several means of getting data onto the board, the fixed and reconfigurable data paths, and many output paths. Figure 2.5 shows a diagram of how these parts are connected on the WILDCHILD.

Internally, the WILDCHILD is made up of eight identical FPGAs and one larger processing element. For our implementation the eight identical PFGAs (numbered PE1 through PE8) are Xilinx 4010 FPGAs. The single processing element (numbered PE0) is a Xilinx 4028 FPGA. This processing element is special in that it offers control of the FIFOs as well as access to the PE0 FIFO and crossbar.

The host system communicates to the WILDCHILD through the use of the FIFOs, memory, and handshaking signals built into the WILDCHILD. A set of two handshaking signals are connected to each processing element. The direction of each of these is completely under the configuration designer's control. The memory components on the WILDCHILD are dual ported, so that both the host system and the processing elements can access the same memory. The WILDCHILDs dual ported memory controller handles the arbitration of the memory, and offers blocked as well as arbitrated modes. This allows the processing element to be blocked when the host is writing, or for the processing element to take priority and block the host when it requires access to the memory elements. Finally, the FIFOs offer a method to pipe large amounts of data to and from the WILDCHILD board. One of the three FIFOs is connected to processing element 0, and is 32K by 36 bits deep in each direction. The other two FIFOs are attached to processing element 1 and processing element 8. These both hold 16K 36-bit words in each direction.

The crossbar provides the user with an efficient means of connecting any of the processing elements to another. It contains 10 ports on the WILDCHILD, with two from PE0, and one from each of the other processing elements. The crossbar may be used to broadcast data from one processing element to a single one, a group of elements, or even all other processing elements. The crossbar offers the ability to control the direction, as well as to partition and only use parts of the crossbar buss to connect various processing elements to each other.

Also included in the WILDCHILD is the systolic bus that connects each processing element with it's neighbor. Thus, with the two FIFOs on the end, processing elements PE1 through PE8 form a ring, from one end to the other with the host as the connector. The external systolic connectors allow the WILDCHILD to be connected together with another WILDCHILD in the same VME chassis, offering much larger rings.

The GOR signals are the last major set of internal signals. These signals are called GORR (for global or result) and GORV (for global or valid). Each set connects PE0 to each individual processing element. These bi-directional signals can be used in whatever manner the user desires and are often used for starting a process, signaling completion, or transmitting serial data.

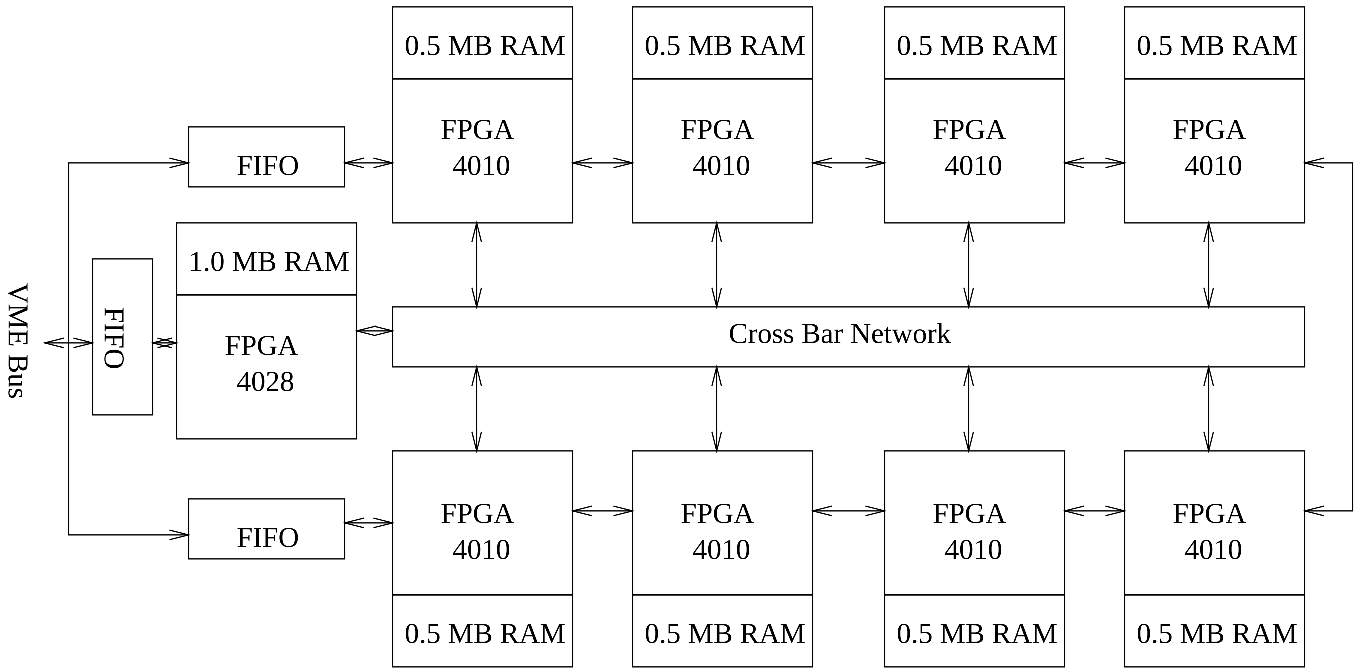


Figure 2.5: WILDCHILD FPGA Board

### 2.4.2 WILDCHILD Software

The Annapolis WILDCHILD's software consists of two parts. The first is the hardware description of the FPGA which is written in VHDL, and the second is the controlling software which runs on the host and actually initializes the WILDCHILD, starts the process, and then retrieves the final results.

The hardware design process goes through several phases. The first is the partitioning of the problem so that it may fit on the individual FPGAs on the board. After the problem is partitioned, it requires the design of each individual FPGAs description in VHDL. Tools are available so that the entire board may be simulated at the VHDL level as a whole, or as parts, and even in conjunction with other boards.

Once the initial VHDL is debugged and designed, it requires processing through the Synplicity tools. This translates the VHDL code for each FPGA on the board into a gate-level description. This gate level description is then processed by the Xilinx place and route tools to create the bitstream used to configure the actual hardware device.

The second major part of the software is the host program. This program links with an API designed by Annapolis allowing the host program to control the various aspects of the WILDCHILD board, including reading and writing to memory, using the FIFOs, and initializing the various aspects of the board.

The main steps of the host program are to open (initialize) the board, configure the board, transfer data to the memories if required, start the computation, and then finally retrieve the results. This pattern is highly job specific though, and depends much on the task being accomplished. A host program using the FIFOs will be required to keep the input FIFO filled, while emptying the output FIFO, for optimum usage of the WILDCHILD.

Calling the API from the host program requires compiling and linking with the headers from Annapolis and the correct libraries. The host uses the open call to open a connection to the board. It them loads the FPGAs being used in the design, the crossbar configuration, and any other memories or control functions as necessary. Depending on how the boards hardware design was configured the host activates the design, and using API calls controls it until it is completed. The host can then use the same API calls to read back the results, or read them while the board is processing.

The Annapolis WILDCHILD libraries which provide an interface through the board interface directly to the Force VME drivers. These drivers in turn interface to the hardware on the VME bus, and thus the WILDCHILD. The result is a set of layers, upon which the API builds. The result is that a general purpose interface to the WILDCHILD has been developed which allows the host to not have to worry about the VMEbus control at all.

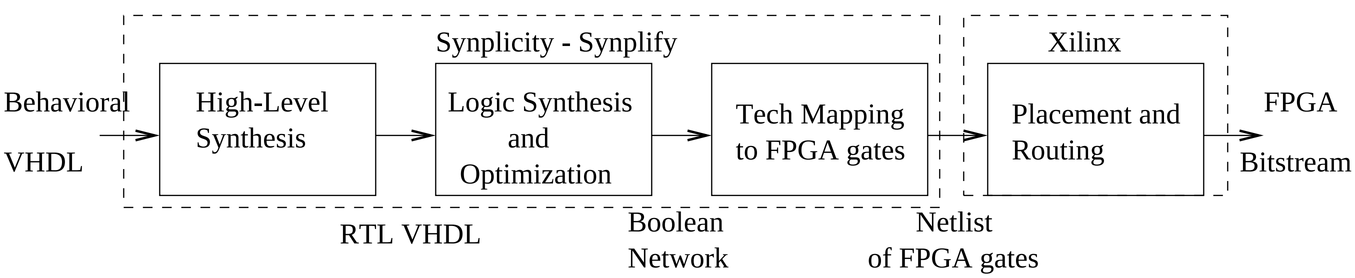


Figure 2.6: FPGA Software Design

## 2.5 Embedded Processors

The Motorola MVME2604 provides the platform for the general purpose computing. This board was chosen because the PowerPC is viewed as a leading embedded processor with very high performance.

### 2.5.1 Motorola Hardware

The Motorola MVME2604 system contains a nearly complete computer system all on one VME board, and an extension module, the MVME761. Figure 2.7 diagrams how the internal components of the MVME2604 are connected to each other.

The primary element of the Motorola MVME2604 is its processor and memory system. The PowerPC 604 processor runs at 200MHz, and contains a 32KB/32KB L1 cache. The MVME2604 extends this cache further with a 256KB L2 cache. Main memory is organized as two 72bit (64 data bits + 8 ECC bits) blocks. On the version of the MVME2604 used in the testbed, the memory mezzanine contains 64MB of ECC DRAM (60ns FPM). This memory can be accessed via a 66MHz (64-bit) memory bus, giving a corresponding processor/memory bandwidth in excess of 400MB/s. Also included in the memory system, is the provision for two banks of FLASH memory. In our particular installation, this is one bank of 1MB, and a second bank of 4MB. One of the banks contains the Open Firmware boot system, and the other can contain user code.

The MVME2604 is designed around a 64-bit PCI bus. The implementation used by the MVME2604 is version 2.0 of the specification which allows the bus to be clocked at 33MHz. Combining this with the 64-bit bus results in bandwidth of nearly 264MB/s. The MVME2604 contains one PMC (PCI Mezzanine Card) slot onboard, with provisions for additional expansion via the P2 connector on the VMEbus.

The PCI to VME bridge is accomplished through the use of the Tundra Universe VME controller. This acts as a PCI bus master, allowing burst mode data transfers from the VME bus onto the PCI bus, and ultimately to the PowerPC processor. The Tundra Universe will also allow for a PCI slave mode, which allows it to free up the PCI bus for other uses, while still maintaining connection to the VME bus. The Tundra Universe also controls the relaying of the VME interrupts onto the PCI bus, which passes them to the PowerPC processor for immediate attention.

The PCI bus also contains three more components built into the MVME2604. The first is the PCI to ISA bridge. This allows for the common Super I/O chip and the standard Zilog serial chips to add a parallel port, synchronous and asynchronous serial ports, PS/2 mouse and keyboard ports, and floppy disk support to the MVME2604. The second attachment to the PCI bus is a SCSI-2 controller, allowing SCSI hard drives to be connected to the MVME2604. Finally, a DEC21140 ethernet controller is added to the system to allow for 10/100Base-T ethernet connections.

These additional controllers are connected to the outside world via the MVME761 transition module. This provides for standard connections to the Ethernet via a RJ-45 connector. This also provides connections to the synchronous and asynchronous serial ports, and the parallel port. Access to the SCSI system is gained internally through the transition module's internal connector.

For our implementation, the MVME2604 and MVME761 transition module are connected to only a few things. The serial port allows us to connect a console via a standard serial cable. The ethernet is connected to the 10/100Base-T ethernet hub. And finally, the MVME2604 is connected to the VME bus. This allows for maximum flexibility in how it may be controlled and data may be passed through it.

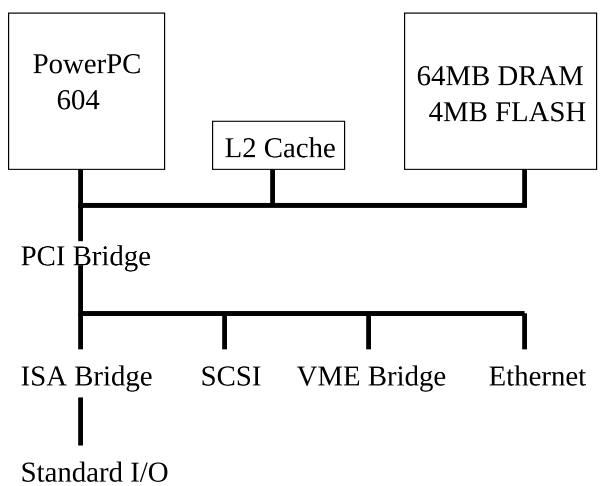


Figure 2.7: Motorola PowerPC Board

### 2.5.2 Motorola Software

The Motorola MVME2604 runs Microware's OS-9 operating system for the PowerPC chip. This operating system has several key ideas that were used in building this system. The first is that it is a real/time operating system and thus it can be used to help meet real/time constraints. One other advantage is that it can be run entirely out of the RAM on the MVME2604, thus eliminating the need for disks.

The main idea with OS-9 is the central kernel. This kernel holds all the necessary routines and programs for the running of the OS-9 system. Once the necessary modules for the running OS-9 system have been placed in the kernel file, this can easily be loaded over a network and used to initialize the MVME2604 board.

The second half of the OS-9 software is the FasTrak system which runs on a PC networked with the MVME2604. This offers the development environment on which to compile, debug, and run programs ultimately destined for the OS-9 system. The FasTrak system does this by cross-compiling the code destined for the PowerPC on the PC, and then transferring it to the MVME2604 over the network. Once it has been transfered it can be run and debugged.

## 2.6 Putting it all together

In theoretical terms, the above parts can all be attached to result in a cohesive object that does what it is intended. In the real world, there is a need to get information in and out of the system, and this results in other additions to the testbed. In our case, we have needed to add a Sun Ultra 5 system to the basic design of the testbed. This is necessitated to act as a gateway between the university network, and the private network that runs the MATCH testbed.

This provides us with several advantages, such as security, privacy, administrative control, and speed. Such things also bring disadvantages, such as making accessing the testbed slightly more complex. However, the many advantages far outweigh the disadvantages. The end result is that a user must log into the Sun Ultra system or one of the PCs on the MATCH subnet, before they can access the MATCH testbed.

## 2.7 Summary

In this chapter we have discussed the hardware testbed for the MATCH project on adaptive computing. The testbed consists of a VME chassis, a Motorola embedded board, a Transtech DSP board, an Annapolis WILDCHILD FPGA board, and a Force controller board. Even though it appears that we put together a number of commercial off-the-shelf boards in a commercial VME chassis, we had significant problems in integrating the various hardware and software components such that they would work together. From the time of designing the testbed (March 1998), identifying the boards to be used, acquiring the components, and getting all the components to work together, the entire process took one year (March 1999). In the next chapter, we will report some preliminary measurements on the testbed.

Chapter 3

# Measurements on Experimental Testbed

This chapter presents a set of measurements taken on the hardware testbed and the methodology used in obtaining the results. They consist of a set of measurements for each board. These measurements will ultimately be used in the MATCH compiler for scheduling computations.

## 3.1 WILDCHILD FPGA Board

Three sets of measurements were taken on the WILDCHILD FPGA board. The first was a basic timing of the API calls, and the WILDCHILD libraries. This helps to test the theoretical limits of the WILDCHILD board. The second test was a fixed point addition of individual elements of two arrays. The final test was the fixed point multiplication of the arrays.

The basic measurements performed tested the times to make simple API calls. The main calls dealt with the memory and the FPGA programming system. The tests included reading and writing to/from memory from/to a buffer, filling the memories with a constant, clearing the memories to zero, and finally loading the memories from a file on disk. The cost of programming the FPGAs with their bitstream from disk was also timed.

The addition measurements performed tested the time to add two vectors of data from RAM, and place them back into RAM. Each individual element from the vector is added to its corresponding element in the second vector, resulting in a third vector.

The multiplication test performs virtually the same test as the addition, only a multiplication operator was substituted in for the addition.

### 3.1.1 Reading FPGA RAM

The first measurement done was reading from the FPGA memories. This was done by simply setting up a fixed buffer in memory, and then timing the API call to read a block of the memory from the onboard RAM, into the Force 5V's memory.

```
WF_ERROR WFireRead (WF_BOARD hBoard,
                    WF_PE_TYPE Pe,
                    DWORD dDWordOffset,
                    DWORD dDWordCount,
                    DWORD *pbuff)
```

For the WILDCHILD API Read Memory call, the following parameters are required: Board, Processing Element, Offset into Memory, Memory Count, and Buffer. It reads `dDWordCount` DWORDs starting at `dDWordOffset` in the RAM connected to processing element `Pe` and transfers it to the buffer in `pbuff`.

The time that it took to perform reads of varying sizes is listed in Table 3.1 For this test we took blocks of data from PE1's RAM and transfer them into a buffer in the host's memory. By evaluating the data, we can see that it takes approximately 1 millisecond for each 1K of data to be transfered.

### 3.1.2 Writing FPGA Ram

The next measurement accomplished was writing into the FPGA memories. Using the fixed buffer from the first test, it was simply transfered into the WILDCHILD RAMs from the host using the API call for writing to an FPGA memory.

```
WF_ERROR WFireWrite (WF_BOARD hBoard,
                     WF_PE_TYPE Pe,
                     DWORD dDWordOffset,
                     DWORD dDWordCount,
                     DWORD *pbuff)
```

For the WILDCHILD API Write Memory call, the following parameters are required: Board, Processing Element, Offset into Memory, Memory Count, and Buffer. It writes `dDWordCount` DWORDS starting at `dDWordOffset` in the RAM connected to processing element `Pe`, transferring it from the buffer in `pbuff`

The time that it took to perform writes of varying sizes is also listed in Table 3.1 For this test we took blocks of data from a buffer in the host's RAM and transfered it into PE1's RAM. By evaluating the data, we can see that it takes approximately 0.4 milliseconds for each 1K of data to be transfered. One interesting thing to note is that it takes longer to read data than to write it. This is expected due to that writes are bursted across the databus, whereas reads require a two way communication.

Table 3.1: FPGA Read and Write Timing in Seconds

| Array Size (Kwords) | 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 128 |
|---|---|---|---|---|---|---|---|---|---|
| Read (seconds) | 0.006657 | 0.016457 | 0.032249 | 0.048144 | 0.063874 | 0.079653 | 0.095486 | 0.111077 | 0.127332 |
| Write (seconds) | 0.000030 | 0.006753 | 0.012882 | 0.018949 | 0.024969 | 0.031125 | 0.037105 | 0.043256 | 0.049271 |

### 3.1.3 Pattern Fill FPGA RAM

The test program then times how long it takes to fill the FPGAs RAM with a given pattern. In our case, this was done for processing elements zero and one, thus accounting for the different sizes of the processing element's RAM.

```
WF_ERROR WFireFillMem (WF_BOARD hBoard,
```

```
                  WF_PE_TYPE Pe,
                  DWORD dPattern)
```

For the WILDCHILD API Fill Memory call, the following parameters are required: Board, Processing Element, Pattern to write. It writes `dPattern` in the RAM connected to processing element `Pe`.

### 3.1.4 Clear FPGA RAM

Clearing the processing element's RAM is the next task. Once again we limited our test to elements zero and one.

```
WF_ERROR WFireClearMem (WF_BOARD hBoard,
                        WF_PE_TYPE Pe)
```

For the WILDCHILD API Clear Memory call, the following parameters are required: Board, Processing Element. It writes zeros into the RAM connected to processing element `Pe`.

### 3.1.5 Load FPGA RAM

For the final test of the memory API calls, we loaded the FPGAs RAM from a file. This reads in the data from a file of binary data. For this we used two files, one the size of the PE0 RAM, and one the size of the PEx RAM.

```
WF_ERROR WFireLoadMem (WF_BOARD hBoard,
                       WF_PE_TYPE Pe,
                       char*pFilename);
```

For the WILDCHILD API Load Memory call, the following parameters are required: Board, Processing Element, File to read from. It writes the data contained in the binary file `pFilename` in the RAM connected to processing element `Pe`.

Table 3.2: FPGA Memory Timing in Seconds

| Test | PE0 (seconds) | PEx (seconds) |
|---|---|---|
| Fill Memory | 0.129665 | 0.067453 |
| Clear Memory | 0.129799 | 0.067992 |
| File Load | 0.281845 | 0.094312 |

### 3.1.6 Load FPGA Bitstream

Programming the onboard FPGAs is accomplished through a single API call. The normal operation of the board is to initially program the FPGAs with a blank program, then to selectively program the needed FPGAs with their correct programming. This is done through two calls to the same API routine listed below. The first call will load blank bitstreams into all of the FPGAs. Then the board will load the FPGAs being used with their respective bitstreams.

```
WF_ERROR WFireProgramPE (WF_BOARD hBoard,
                         DWORD dPeBitmap,
                         char *pFilename);
```

For the WILDCHILD API Program Processing Element call, the following parameters are required: Board, Processing Element bitmask, File to read from. This call loads the bitstream from `pFilename` into the FPGAs designated by the bits set to one in `dPeBitmap`.

Program timing was performed for the two different types of FPGA on the WILDCHILD board. The first set was for PE0, the second for a single PEx, and third for ALL of the PEx with the same configuration (typical when programming the array with blank). In our tests, the timings directly correlated with the size of the bitstream. As PE0 is larger on our board, the time to program is correspondingly larger. Also the time to program all 8 PEx FPGAs corresponds well with the time to program a single PEx FPGA. It is expected that configurations consisting of a combinations of FPGAs will simply take n*single PEx time to configure.

Table 3.3: FPGA Program Timing in Seconds

| PE0 (seconds) | PEx (seconds) | ALL PEx (seconds) |
|---|---|---|
| 1.248781 | 0.329655 | 1.022495 |

### 3.1.7 Addition

The addition test operation consists of taking two arrays of 16 bit integers, and adding each element together, placing the results in a third array. The operation extends this functionality a little, allowing for a start, end, and skip values to be input. The corresponding MATLAB code computes `c=a+b`, or with the extend syntax `c(start:skip:end)=a(start:skip:end)+b(start:skip:end)`. This effectively computes (in C):

```
for (i=start;i<end;i+=skip) {
    c[i]=a[i]+b[i];
}
```

Performing this test requires software for the host to control the WILDCHILD and VHDL code to describe the hardware that the WILDCHILD must operate as. In addition to the normal host operations, such as configuring the WILDCHILD, the host must also fill the memory used by the FPGA with the appropriate data to be added, as well as the parameters used to define the operation. The memory map for the operation is shown in Table 3.4

One thing that should be noted is the differentiation between hardware, and software memory addresses. In the host software, each word is 32 bits long. However, in the hardware, each 32 bit word has been split into two 16 bit data values. As such, the length of the array is actually the number of 16 bit data values, not the length of the array in the host's memory.

The host's code for performing the core initialization, execution, and readback stages are shown in Figure 3.1. Initialization starts with the creation of the buffer to hold the data to be placed into memory, and then read back in. This step is accomplished in Line 1 using the `WildfireAlloc` call to allocate the memory.

After creating the buffer, line 2 clears all the elements to zero. Lines 3 and 4 load the arrays that are to be added. Finally, lines 5 through 10 initialize the memory map shown in Table 3.4. The first value loaded is

Table 3.4: Memory Map

| Software Address | Hardware Address | Use |
|---|---|---|
| 0 | 0 | start index |
| 1 | 2 | end index |
| 2 | 4 | skip |
| 3 | 6 | starting hardware address for a |
| 4 | 8 | starting hardware address for b |
| 5 | 10 | starting hardware address for c |

the starting index into the arrays, next the final ending index, and then the skip value. It should be noted that these are hardware values, and thus must be adjusted by a factor of two to account for the 16 bit nature of the data. Finally, the last three values are the hardware locations in the RAMs of the arrays.

Next, PE1's RAM is placed into blocked mode at line 11. This will prevent the PE from accessing the RAM while we are initializing it, which occurs in line 12. We then reset the memory mode to allow the PE to access the RAM we have initialized, and by disabling the reset in line 14, start the entire process. When the process is complete, it raises an interrupt, which we watch for and catch in line 16. Finally, we disable the PE's access to the RAM in line 18, and read back the final data values in line 19. The read back data, will reside in the appropriate location as defined in line 10.

The second part of the design for the test is the VHDL code which describes the hardware description for PE1, the only PE used in this design. The first basic step involves reading in the memory map, and initializing the internal registers. Once this is accomplished, the design loops through the values, performing the addition, and writing back the results to the result array. Finally, the design sets an interrupt when it completes, signaling the host that it is done.

To help simplify the design, we used a single state machine for the memory reads and writes. The core of this state machine is shown in Figure 3.2. Complete description of the codes are provided in the Appendixes. This state machine waits for a request, and then performs the necessary steps to read or write to memory. This allows each memory access request in the main design to consist of only one state. The memory state machine handles the request to access memory (1), waiting until memory is available (2), and then the reading or writing

```
(1)    pbuffer = ( DWORD * ) WildfireAlloc ( 128*1024 * sizeof ( DWORD ));

(2)    for (i=0; i < 128*1024;i++) pbuffer[i] = 0L;

(3)    for (i=256;i<512;i++) { pbuffer[i] = i-256; }
(4)    for (i=512;i<768;i++) { pbuffer[i] = i-512; }

(5)    pbuffer[0] = 0;
(6)    pbuffer[1] = 512;
(7)    pbuffer[2] = 1;
(8)    pbuffer[3] = 512;
(9)    pbuffer[4] = 1024;
(10)   pbuffer[5] = 1536;

(11)   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_BLOCKED);
(12)   ret_val = WFireWrite(hBoard, 1, 0, 128*1024, pbuffer);

(13)   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_SW_ARBITRATE);

(14)   ret_val = WFireResetPE( hBoard, WF_ALL_PEs, WF_DISABLE );
(15)   RETURN_FAILURE( ret_val );

(16)   ret_val = WFireWaitForPEInt (hBoard, WF_PE1_BIT, WF_ALL, 6000);
(17)   RETURN_FAILURE( ret_val );

(18)   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_BLOCKED);

(19)   ret_val = WFireRead (hBoard, 1, 0, 128*1024, pbuffer);
(20)   RETURN_FAILURE( ret_val );
```

Figure 3.1: FPGA Host Code

of the data (3). This requires at least four cycles for a read, and two for a write. It should be noted that in an optimal situation the reads and writes can be pipelined, so as to shorten the overal time to perform the operations.

The main state machine consists of three parts, the states to initialize the internal registers, the main process loop, and the final interrupt signal. The initialization states perform a basic read from the local FPGA RAM into the state machine's internal registers. Each register is 32 bits, and thus takes two read cycles to read each of the 16 bits from the FPGA RAM. The index, skip, step, and memory locations for A, B, and C are read into their respective registers.

```
(1)  when mem_start =>
      mem_request <= '0';
      mem_state <= mem_req_wait;
(2)  when mem_req_wait =>
      mem_request <= '1';
      if mem_grant = '1' then
        mem_state <= mem_req_wait;
      else
       if mem_rdwr = '0' then
        mem_state <= mem_end;
       else
        mem_state <= mem_wait;
       end if;
      end if;
     when mem_wait =>
      mem_state <= mem_end;
(3)  when mem_end =>
      if mem_rdwr = '1' then
        mem_data <= read_data;
      end if;
      mem_done <= '1';
      mem_state <= mem_init;
```

Figure 3.2: FPGA VHDL Memory access

Next, the basic computation is performed. This is accomplished by reading into registers the contents of the A and B elements from the FPGA's RAM. An addition is performed, and then the results written back to the final C element location by once more using the memory access state machine.

The final test results from the FPGA were run using a FPGA clock rate of 1MHz. While the maximum rate was 33MHz, this would allow for easy scaling of the results to any clock rate, and was performed to show that the hardware and software flow works together. [Peri299] discusses a more optimal solution to this problem.

Table 3.5 shows the results for taking arrays of 16-bit words and performing the above addition operation on them. It can be seen that the time to perform the total vector add is fairly linear as the size of the array increases $t_r = n * 2.7E - 5$ (where n is the number of 16-bit words).

```
-- Load the A register from FPGA RAM
  when rd_A =>
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseA (17 downto 0);
    mem_rdwr <= '1';
    if (mem_done = '1') then
      A <= mem_data;
      mem_doit <= '0';
      state <= rd_B;
    else
      mem_doit <= '1';
      state <= rd_A;
    end if;
-- Load the B register from FPGA RAM
  when rd_B =>
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseB (17 downto 0);
    mem_rdwr <= '1';
    if (mem_done = '1') then
      B <= mem_data;
      mem_doit <= '0';
      state <= compute_C;
    else
      mem_doit <= '1';
      state <= rd_B;
    end if;
-- compute C
  when compute_C =>
    C <= A+B;
    state <= wr_C;
-- write C back to the FPGA RAM
  when wr_C =>
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseC (17 downto 0);
    mem_wr_data <= C;
    mem_rdwr <= '0';
    if (mem_done = '1') then
     mem_doit <= '0';
     state <= inc_index;
    else
     mem_doit <= '1';
     state <= wr_C;
    end if;
```

Figure 3.3: FPGA VHDL Computation

### 3.1.8 Multiplication

Multiplication on the FPGA array is performed exactly like the addition, only we substituted a multiplication operator for the addition in the PEx VHDL code. The results shown in Table 3.5 are virtually identical to those from the addition operation. Given the very small change in the design, this was not surprising.

Table 3.5: FPGA Addition and Multiplication Timing

| Array Size (Kwords) | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| Addition (ms) | 30.825 | 58.406 | 113.669 | 224.222 | 445.424 | 887.543 |
| Multiplication (ms) | 30.704 | 58.335 | 113.597 | 224.153 | 445.309 | 887.526 |

## 3.2 Transtech DSP Board

Two sets of tests were run on the Transtech DSP board, involving basic timing runs of additions and multiplications similar to the operations performed on the FPGA board. Each test allocates three vectors in memory, and then proceeds to add (or multiply) each corresponding element from the first two and places the result in the third vector.

### 3.2.1 Addition

Programming of the Transtech DSP board in single processor mode is as simple as writing C code to do the processing, and then to compile and execute it on the board. Figure 3.4 shows a simple vector vector add, the same used in the FPGA Addition test. This code is compiled using the TI DSP C compiler into an executable code. The code is then run on the DSP using the Transtech OPerating System (TOPS) loader.

Results from running the code in Figure 3.4 are shown in Table 3.6. As shown, for this very simple code, the execution time scales linearly with the size of the array.

```
timer_init(1000);

a = malloc(ARRAY_SIZE * sizeof(int));
b = malloc(ARRAY_SIZE * sizeof(int));
c = malloc(ARRAY_SIZE * sizeof(int));

etstart = get_time();

for (i=0; i<ARRAY_SIZE; i++)
  c[i] = a[i] + b[i];

etstop = get_time();
```

Figure 3.4: DSP Example Code

### 3.2.2 Multiplication

The multiplication tests on the Transtech DSP board for a single processor were done the same way as the addition. The only change made to the code in Figure 3.4 was that a multiplication was used instead of an addition in the computation loop. Results for this test also turned out to scale linearly, and to be identical to those for an addition, since the addition and multiplication take the same amount of time in the DSP. Results for the multiplication are shown in Table 3.6.

Table 3.6: DSP Addition and Multiplication Timing

| Array Size (Kwords) | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| Addition (ms) | 3 | 6 | 12 | 24 | 49 | 97 |
| Multiplication (ms) | 3 | 6 | 12 | 24 | 49 | 97 |

## 3.3 Benchmarking Matrix Multiplication Performance

For benchmarking purposes, a Matrix Matrix multiplication of two matrices (with 16 bit fixed point precision) was performed on each part of the testbed. For benchmarking purposes each part had four different sizes of the arrays. Results of these tests are shown in Table 3.7 Results from the FPGA matrix multiply were generated by Periyacheri [Peri99], based on a design implemented by him. Results for the other three test environments were generated by Walkden [Walk99], based on his designs for a matrix multiply on the Force, single TI DSP, and a single PowerPC.

Several results should be noted from Table 3.7. First, the computation time for a particular board (Force, Transtech, Motorola, or WILDCHILD) increases as $O(n^3)$ with the increased problem size. For example the computation time of 0.37 seconds for a 64x64 matrix to 179.3 seconds for a 496x496 matrix on the force board. Secondly, for a particular matrix size (ex. 496x496) the computation cost is highest on the Force 5V board (179.3 s), followed by the DSP board (39.6 s), followed by the PowerPC board (22.6 s), and the fastest time obtained is from the FPGA board (0.795 s). Given these particular implementations only, if we include the time to transfer the data to and from the host Force board (0.038 s), it is still worth it to execute matrix multiply on the FPGA board.

Table 3.7: Performance of Matrix Multiplication on various boards of the testbed

| size (words) | FPGA (data) (s) | Force (s) | DSP (data) (s) | PowerPC (s) |
|---|---|---|---|---|
| 496x496 | 0.795 (0.96) | 179.314 | 36.371 (6.94) | 22.66 |
| 248x248 | 0.103 (0.25) | 22.986 | 4.601 (1.74) | 3 |
| 128x128 | 0.015 (0.082) | 2.957 | 0.635 (0.46) | 0.33 |
| 64x64 | 0.00195 (0.038) | 0.371 | 0.080 (0.12) | 0.04 |

Chapter 4

# Concluding Remarks

In this thesis, the design of an adaptive computing testbed was presented. This system was motivated by a need to make an environment available on which to develop and compare codes on several different platforms. For this purpose, the testbed combines an FPGA environment, the WILDCHILD, a DSP environment, the Transtech DSP board, and a general purpose computing environment, the Motorola PowerPC platform. This is all glued together into one cohesive unit with the addition of the VME chassis and ethernet for communication, and the Force 5V platform for the control.

Basic tests were done to evaluate the computation speed and potential of the various components of the testbed. The communication speeds between the Force controller and the WILDCHILD FPGA boards were measured, and then small tests were run on the boards themselves to evaluate their potential. The same tests were run on the DSPs in an attempt to gauge their potential, and to compare them to the other environments in the system.

Finally, a single benchmark was run on all environments of the testbed in an effort to compare the power of each one. This resulted in information on the power of each individual platform for the benchmark chosen. This also demonstrates the various tradeoffs for the particular implementation of this benchmark.

Future bottlenecks are likely with this system though. We have already encountered problems fitting designs on the Xilinx 4010 processors, and the worry exists about the amount of memory on the FPGA board. Also, the DSP connection design is fairly limiting, a problem we hope to rectify in the future. Finally, the VMEbus and ethernet are likely to be saturated when a large parallel design is implemented. This could be solved in the future by adding additional communications methods.

# Bibliography


[BBB+98] Milissa Benincasa, Richard Besler, Diane Brassaw, Ralph L. Kohler, Jr. "Rapid Development of Real-Time Systems Using RTExpress". *Proceedings of the 12th International Parallel Processing Symposium*, pages 594-599, Mar 30 - Apr 3, 1998.

[DH98] Don Davis, Jonathan Harris. "ACEcard: A High-Performance Architecture for Run-Time Reconfiguration". *Proceedings of the 12th International Parallel Processing Symposium*, pages 616-619, Mar 30 - Apr 3, 1998.

[DJ98] Robert P. Dick, Niraj K. Jha. "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems". *IEEE/ACM International Conference on Computer Aided Design*, pages 62-68, San Jose, California, November, 1998.

[FHP98] Bradley K. Fross, Dennis M. Hawver, James B. Peterson. "WILDFIRE Heterogeneous Adaptive Parallel Processing Systems". *Proceedings of the 12th International Parallel Processing Symposium*, pages 611-615, Mar 30 - Apr 3, 1998.

[G+99] Tim Gaverick et al. Napa 1000. *http://www.national.com/appinfo/milaero/napa1000/*,1999

[GDM92] Rajesh K. Gupta, Giovanni De Micheli. "System-level Synthesis using Re-programmable Components". *Proceedings of The European Conference on Design Automation*, pages 2-7, Belgium, Mar 1992.

[H+99] Scott Hauck, Thomas W. Fry, Mathew M. Hosler, Jeffrey P. Kao. "The Cimaera Reconfigurable Functional Unit". *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 16-18, 1997.

[HW97] John R. Hauser, John Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor". *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 16-18, 1997.

[Peri99] Suresh Periyacheri, Private Communication, Mar 1999.

[Peri299] Suresh Periyacheri. "MATLAB Library Functions on a Reconfigurable Processor". *MS Thesis*, ECE, NWU, June 1999.

[Sch98] John Schewel. "A Hardware/Software Co-Design System using Configurable Computing Technology". *Proceedings of the 12th International Parallel Processing Symposium*, Pages 620-625, Mar 30 - Apr 3, 1998.

[W+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. "Baring it all to Software: Raw Machines" *IEEE Computer*, September 1997, Pages 86-93.

[Walk99] Mike Walkden, Private Communication, Mar 1999.

Appendix A

# Vector Addition: FPGA VHDL Code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity acc is
    port (
        read_data       : in STD_LOGIC_VECTOR (15 downto 0);
        address         : out STD_LOGIC_VECTOR (17 downto 0);
        clock           : in STD_LOGIC;
        reset           : in STD_LOGIC;
        write_data      : out STD_LOGIC_VECTOR (15 downto 0);
        mem_request     : out STD_LOGIC;
mem_grant : in STD_LOGIC;
        mem_readwrite   : out STD_LOGIC;
        done            : out STD_LOGIC
    );
end acc;

architecture mixed of acc is
-- Signals
signal index : STD_LOGIC_VECTOR (31 downto 0);
signal iskip : STD_LOGIC_VECTOR (31 downto 0);
signal iend : STD_LOGIC_VECTOR (31 downto 0);

signal baseA : STD_LOGIC_VECTOR (31 downto 0);
signal baseB : STD_LOGIC_VECTOR (31 downto 0);
signal baseC : STD_LOGIC_VECTOR (31 downto 0);

signal A : STD_LOGIC_VECTOR (15 downto 0);
signal B : STD_LOGIC_VECTOR (15 downto 0);
signal C : STD_LOGIC_VECTOR (15 downto 0);
```

```
-- States
type state_value is (init_state,
rd_indexL,
rd_indexH,
rd_iendL,
rd_iendH,
rd_iskipL,
rd_iskipH,
rd_baseAL,
rd_baseAH,
rd_baseBL,
rd_baseBH,
rd_baseCL,
rd_baseCH,
forloop, loop_body,
rd_A,
rd_B,
compute_c,
wr_C,
inc_index,
end_state,end_state1,final);
signal state  : state_value;

-- memory signals
type mem_state_value is (mem_init, mem_start, mem_req_wait, mem_wait, mem_end, mem_end2);
signal mem_state : mem_state_value;

signal mem_base : STD_LOGIC_VECTOR (17 downto 0);
signal mem_width : STD_LOGIC_VECTOR (17 downto 0);
signal mem_index_length : STD_LOGIC_VECTOR (17 downto 0);
signal mem_index_width : STD_LOGIC_VECTOR (17 downto 0);
signal mem_data : STD_LOGIC_VECTOR (15 downto 0);
signal mem_wr_data : STD_LOGIC_VECTOR (15 downto 0);
signal mem_doit : STD_LOGIC;
signal mem_rdwr : STD_LOGIC;
signal mem_done : STD_LOGIC;

begin

memory: process (clock, reset, mem_grant, read_data, mem_doit) is
  variable mult : std_logic_vector (35 downto 0);
  variable mult2 : std_logic_vector (17 downto 0);
begin

if reset = '1' then
 address <= b"00_0000_0000_0000_0000";
 mem_request <= '1';
 mem_readwrite <= '1';
 mem_state <= mem_init;
 mem_done <= '0';
```

```
elsif rising_edge(clock) then
 mem_request <= '1';
 mem_readwrite <= '1';
 mem_done <= '0';
 write_data <= mem_wr_data;
 if ((mem_index_width = b"00_0000_0000_0000_0000") or
     (mem_width = b"00_0000_0000_0000_0000")) then
   address <= mem_base + mem_index_length;
 else
   mult := mem_index_width*mem_width;
   mult2 := mult (17 downto 0);
   address <= mem_base+ mem_index_length + std_logic_vector(mult2);
 end if;
 mem_readwrite <= mem_rdwr;

 if mem_doit = '1' then
  case mem_state is
  when mem_init =>
    mem_state <= mem_start;
  when mem_start =>
    mem_request <= '0';
    mem_state <= mem_req_wait;
  when mem_req_wait =>
    mem_request <= '1';
    if mem_grant = '1' then
      mem_state <= mem_req_wait;
    else
     if mem_rdwr = '0' then
      mem_state <= mem_end;
     else
      mem_state <= mem_wait;
     end if;
    end if;
  when mem_wait =>
    mem_state <= mem_end;
  when mem_end =>
    if mem_rdwr = '1' then
      mem_data <= read_data;
    end if;
    mem_done <= '1';
    mem_state <= mem_init;
  when others =>
    mem_state <= mem_init;
  end case;
 else -- mem_doit = '0'
  mem_state <= mem_init;
 end if;
end if;

end process;
```

```
control: process (clock, reset, mem_grant, read_data) is
begin

if reset='1' then
  done <= '0';
  state <= init_state;
  mem_doit <= '0';
  mem_rdwr <= '1';

elsif rising_edge(clock) then

  case state is
-- Initialization
  when init_state =>
    state <= rd_indexL;

-- read the index value
  when rd_indexL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0001";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      index (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_indexH;
    else
      mem_doit <= '1';
      state <= rd_indexL;
    end if;

  when rd_indexH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0000";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      index (31 downto 16) <= mem_data;
      mem_doit <= '0';
      state <= rd_iendL;
    else
      mem_doit <= '1';
      state <= rd_indexH;
    end if;

-- read the end value
  when rd_iendL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
```

```
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0011";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      iend (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_iendH;
    else
      mem_doit <= '1';
      state <= rd_iendL;
    end if;

  when rd_iendH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0010";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      iend (31 downto 16) <= mem_data;
      mem_doit <= '0';
      state <= rd_iskipL;
    else
      mem_doit <= '1';
      state <= rd_iendH;
    end if;

-- read the skip value
  when rd_iskipL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0101";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      iskip (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_iskipH;
    else
      mem_doit <= '1';
      state <= rd_iskipL;
    end if;

  when rd_iskipH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0100";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      iskip (31 downto 16) <= mem_data;
```

```
      mem_doit <= '0';
      state <= rd_BaseAL;
    else
      mem_doit <= '1';
      state <= rd_iskipH;
    end if;

-- read the Base A value
  when rd_BaseAL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0111";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      baseA (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_BaseAH;
    else
      mem_doit <= '1';
      state <= rd_BaseAL;
    end if;

  when rd_BaseAH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_0100";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      BaseA (31 downto 16) <= mem_data;
      mem_doit <= '0';
      state <= rd_BaseBL;
    else
      mem_doit <= '1';
      state <= rd_BaseAH;
    end if;

-- read the Base B value
  when rd_BaseBL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_1001";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      BaseB (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_BaseBH;
    else
      mem_doit <= '1';
```

```
      state <= rd_BaseBL;
    end if;

  when rd_BaseBH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_1000";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      BaseB (31 downto 16) <= mem_data;
      mem_doit <= '0';
      state <= rd_BaseCL;
    else
      mem_doit <= '1';
      state <= rd_BaseBH;
    end if;

-- read the Base C value
  when rd_BaseCL =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_1011";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      BaseC (15 downto 0) <= mem_data;
      mem_doit <= '0';
      state <= rd_BaseCH;
    else
      mem_doit <= '1';
      state <= rd_BaseCL;
    end if;

  when rd_BaseCH =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= b"00_0000_0000_0000_0000";
    mem_base <= b"00_0000_0000_0000_1010";
    mem_rdwr <= '1';
    if (mem_done = '1') then
      BaseC (31 downto 16) <= mem_data;
      mem_doit <= '0';
      state <= forloop;
    else
      mem_doit <= '1';
      state <= rd_BaseCH;
    end if;

-- Comparison Test of For Loop
  when forloop =>
```

```
    if (index > iend) then
     state <= end_state;
    else
     state <= loop_body;
    end if;

-- Start of body of For Loop
  when loop_body =>
    state <= rd_A;

-- Load the A register
  when rd_A =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseA (17 downto 0);
    mem_rdwr <= '1';
    if (mem_done = '1') then
      A <= mem_data;
      mem_doit <= '0';
      state <= rd_B;
    else
      mem_doit <= '1';
      state <= rd_A;
    end if;

-- Load the B register
  when rd_B =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseB (17 downto 0);
    mem_rdwr <= '1';
    if (mem_done = '1') then
      B <= mem_data;
      mem_doit <= '0';
      state <= compute_C;
    else
      mem_doit <= '1';
      state <= rd_B;
    end if;

-- compute C
  when compute_C =>
    C <= A+B;
    state <= wr_C;

-- write C back to the ram
  when wr_C =>
    mem_width <= b"00_0000_0000_0000_0000";
    mem_index_width <= b"00_0000_0000_0000_0000";
```

```
    mem_index_length <= index (17 downto 0);
    mem_base <= BaseC (17 downto 0);
    mem_wr_data <= C;
    mem_rdwr <= '0';
    if (mem_done = '1') then
     mem_doit <= '0';
     state <= inc_index;
    else
     mem_doit <= '1';
     state <= wr_C;
    end if;

-- increment the Index register
  when inc_index =>
  index <=  index + iskip;
  state <= forloop;

-- All done
  when end_state =>
    mem_doit <= '0';
    done <= '1';
    state <= end_state1;

  when end_state1 =>
    mem_doit <= '0';
    done <= '1';
    state  <= final;

  when final =>
    mem_doit <= '0';
    done <= '0';
    state <= final;

  when others =>
    state <= init_state;

  end case;
  end if;
end process;

end mixed;
```

Appendix B

# Vector Addition: FPGA Host Code

```
/***************************** Includes ************************************/
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

#ifdef MULTI_THREAD
#include <thread.h>
#endif

/* Wildfire specific */
#include <wildfire.h>

#undef _RESET_BOARD_
#undef  FIFO_DBUG

/***************************** Constants ************************************/

#define TRUE 1
#define FALSE 0

/*  Define TAG constants */
#define NO_TAG    0x0
#define START_TAG 0x1
#define STOP_TAG  0x2
#define VALID_TAG 0x8

/*  Define AF and AE flag offset constants */
#define AF_OFFSET 12
#define AE_OFFSET 12

/*  Define file buffer constants */
```

```
#define INFILE_BUF_SIZE  16384
#define OUTFILE_BUF_SIZE 8192
#define MAX_FIFO_DELTA   8192

#define ERR_FIFOXMPL 999
#define RETURN_FAILURE( x ) \
   if ( x != WF_SUCCESS ) { fprintf ( stdout, "command failed.\n" ); return x; }

/****************************** Globals ************************************/
WF_BOARD hBoard;
/* Input file and data variables */
FILE *infile;
DWORD *infile_buf;

/* Output file and data variables */
FILE *outfile;
DWORD *outfile_buf;

#define PREFIX_INVALID  -1
#define PREFIX_4010_208 0
#define PREFIX_4013E208 1
#define PREFIX_4020E208 2
#define PREFIX_4028X208 3
#define PREFIX_4036X208 4
#define PREFIX_4025_304 5
#define PREFIX_4036X432 6
#define PREFIX_4044X208 7
#define PREFIX_4062X432 8
#define PREFIX_4028X304 9
#define PREFIX_MAX     10

int iPe0Prefix = PREFIX_INVALID;
int iPexPrefix = PREFIX_INVALID;

CONST char * pathPrefix[PREFIX_MAX] = { "4010_208", "4013e208", "4020e208",
                                        "4028x208", "4036x208", "4025_304",
                                        "4036x432", "4044x208", "4062x432",
                                        "4028x304" };

/* Systolic PE Xilinx type */
DWORD dBoardType;
DWORD dBoardVersion;

/* Systolic PE Xilinx type */
DWORD pex_type;
DWORD pe0_type;

DWORD trace_count = 5;
BOOL bDataInFifo = TRUE;
BOOL bVerify = FALSE;
BOOL bDebug = FALSE;
```

```
DWORD dwords_recvd = 0;
DWORD data_reads = 0;
char * os_environment = "UNIX";
char * file_extension = ".m68";
char * file_slash = "/";
char filename[ 80 ];
char *p_filename;

DWORD dwords_sent = 0;

/* Data file names */
char pexfilename [9][ 128 ];
char memfilename [9][ 128 ];
char configfilename [ 128 ];

/**************************** Prototypes **********************************/
#ifdef PROTO_DEF

WF_ERROR WF_PrintError ( WF_ERROR error_value );
WF_ERROR WF_ConfigureWildfire ( WF_BOARD hBoard, float *clk_freq );
WF_ERROR WF_ShutdownWildfire ( WORD unit, WF_BOARD hBoard );

#else

WF_ERROR WF_PrintError ( );
WF_ERROR WF_ConfigureWildfire ( );
WF_ERROR WF_ShutdownWildfire ( );

#endif


/****************************** Local Function   ***************************
 * Function:    loadconfig
 * Returns:     void
 * Description: Load the configuration from the text file in configfilename
 ***************************************************************************/
void loadconfig () {
  FILE *input;
  int ret_val;
  int i;

  for ( i = 0; i < 9;  i++ ) {
   pexfilename [ i ][ 0 ] = '\0';
   memfilename [ i ][ 0 ] = '\0';
  }

  if (configfilename[0] == '\0') return;

  input = fopen (configfilename, "r");
  if (input == NULL) {
```

```
    fprintf (stdout,"ERROR: Configuration file %s unreadable\n",configfilename);
    return;
  }

  for ( i = 0; i < 9; i++) {
   ret_val = fscanf (input, "pex %s\n",pexfilename[i]);
   fprintf (stdout,"%s\n",pexfilename[i]);
  }
  for ( i = 0; i < 9; i++) {
   ret_val = fscanf (input, "mem %s\n",memfilename[i]);
   fprintf (stdout,"%s\n",memfilename[i]);
  }

  fclose (input);
}

/****************************** Local Function   ***************************
 * Function:    main
 * Parameters:  argc, argv - Command line parameters
 * Returns:     void
 ***************************************************************************/
void main ( argc, argv )
     int argc;
     char *argv [ ];
{
  /* WF_ERROR return value */
WF_ERROR ret_val;
int bad_args;

/* Argument counter */
int argi;

/* Unit and slot variables */
WORD unit=0;
WORD slot=0;

/* Pclk frequency variable */
float clk_freq;

CONST char * help_string =
     "Usage: loadfpga -s <slot> -c <filename> " ;

/* Initialize variables */
ret_val = WF_SUCCESS;
bad_args = 0;
clk_freq = FLOAT(10.0);
configfilename [ 0 ] = '\0';

   for ( argi = 1; argi < argc; argi++ )
     {
      if ( argv [ argi ][ 0 ] == '-' )
```

```
{
         switch ( toupper( argv [ argi ] [ 1 ]) )
   {
   case 'S':
               argi++;
               slot = atoi ( argv [ argi ] );
               break;
   case '0':
               argi++;
               strcpy ( pexfilename[0], argv [ argi ] );
               break;
   case '1':
               argi++;
               strcpy ( pexfilename[1], argv [ argi ] );
               break;
           case 'C':
               argi++;
               strcpy ( configfilename, argv [ argi ] );
               break;
   case 'F':
       argi++;
       clk_freq = atoi (argv [ argi ]);
       break;
   default:
               fprintf ( stderr, " Unknown option: \"%s\"\n", argv [ argi ] );
               fprintf ( stderr, "%s\n\n", help_string);
               bad_args = TRUE;
   } /* switch */
}
      else
{
         fprintf ( stderr, " Unknown option: \"%s\"\n", argv [ argi ] );
         fprintf ( stderr, "%s\n\n", help_string);
         bad_args = TRUE;
}
     } /* for loop */
  if ( bad_args == FALSE )
    {
      loadconfig();
      ret_val = WFireOpen ( unit, slot, WF_OPEN_SHARED_BIT, &hBoard, 0L );
      if (  ret_val == WF_SUCCESS )
{
  /*  Configure the WILDFIRE system specified by unit and slot */
         ret_val = WF_ConfigureWildfire ( hBoard, &clk_freq );

         if ( ret_val == WF_SUCCESS )
  {
            ret_val = WF_ShutdownWildfire ( unit, hBoard );
          }
}
      else
```

```
{
        fprintf ( stdout, "Wildfire Configuration failed.\n" );
}
    }

   if ( ret_val )
     {
      WF_PrintError ( ret_val );
     }
   fprintf ( stdout, "Example Done.\n\n" );

   exit ( ret_val );
}

/******************************* Local Function   **************************
 * Function:    WF_PrintError
 * Parameters:  error_value
 * Returns:     WF_ERROR
 * Description: This program is used to test for non-zero WF_ERROR
 *              conditions.  If an error condition is detected, then
 *              the error message is printed to stderr.
 ***********************************************************************/
#ifdef PROTO_DEF
WF_ERROR WF_PrintError ( WF_ERROR error_value )
#else
WF_ERROR WF_PrintError ( error_value )
     DWORD error_value;
#endif
{
static WF_ERROR last_error = WF_SUCCESS;

   if ( ( error_value != WF_SUCCESS ) &&
        ( error_value != last_error ) )
     {
      if ( error_value == ERR_FIFOXMPL )
{
         fprintf(stdout,"* Error:  FIFO EXAMPLE PROGRAM\n");
}
      else
{
         fprintf(stdout,"* Error:  %s (%d) (0x%X)\n",
              WFireErrorString( error_value ),
              (int)error_value , (WORD)error_value);
}
      fflush( stdout );
     }
   last_error = error_value;
   return error_value;
}

/******************************* Local Function   **************************
```

```
 * Function    : WF_ConfigureWildfire
 * Parameters  : board
 *               clk_freq : Processor clock (Pclk) frequency
 * Returns     : WF_ERROR
 * Description : Configures a WILDFIRE system specified by unit and slot.
 *               The WILDFIRE system must have already been opened by a call
 *               to WFireOpen.
 ***********************************************************************/
#ifdef PROTO_DEF
WF_ERROR WF_ConfigureWildfire ( WF_BOARD hBoard, float *clk_freq )
#else
WF_ERROR WF_ConfigureWildfire ( hBoard, clk_freq )
     WF_BOARD hBoard;
     float *clk_freq;
#endif
{
  /* Function return value */
WF_ERROR ret_val;
DWORD bpc_mode = WF_BPC0;

/* Timer variables */
struct timeval first,second,elapsed;
struct timezone tzp;
int i,j;

/* buffer variables */
DWORD *pbuffer;

/* Reset the Wildfire board (this may take about 10-15 seconds) */
#if defined (_RESET_BOARD_)
   fprintf ( stdout, "Resetting Wildfire, please wait ... " );
   fflush ( stdout );
   ret_val = WFResetBoard ( hBoard );
   RETURN_FAILURE( ret_val );
#endif
   /* Get the board capability info */
   ret_val = WFireGetInfo( hBoard, WF_GET_INFO_BOARD_TYPE, &dBoardType );
   RETURN_FAILURE( ret_val );

   ret_val = WFireGetInfo( hBoard, WF_GET_INFO_BOARD_VERSION, &dBoardVersion );
   RETURN_FAILURE( ret_val );

   switch (dBoardType)
     {
     case WF_GET_INFO_BOARD_TYPE_WILDCHILD:
     fprintf( stdout,"Board is a WILDCHILD(tm) version %c\n",
              (int) (dBoardVersion-1)+'A');
     break;
     default:
     fprintf( stdout,"Board type is UNKNOWN\n");
     break;
```

```
  }


/* get PE 0 type */
ret_val = WFireGetInfoIndexed( hBoard, WF_GET_INFO_PE_XILINX_TYPE, 0,
                               &pe0_type );
RETURN_FAILURE( ret_val );

fprintf( stdout,"PE 0 is type %s.\n",
             WFireGetInfoXilinxTypeName( (WORD) pe0_type ) );
if (
     (pe0_type != WF_GET_INFO_XILINX_4028EX ))
  {
   fprintf( stdout, "fifoxmpl cannot handle PE0 type %ld\n", pe0_type);
   return(  ERR_FIFOXMPL );
  }

/* get PE x type */
ret_val = WFireGetInfoIndexed( hBoard, WF_GET_INFO_PE_XILINX_TYPE, 1,
                               &pex_type );
RETURN_FAILURE( ret_val );

fprintf( stdout,"Systolic PEs are type %s.\n",
             WFireGetInfoXilinxTypeName( (WORD) pex_type ) );
if (
     (pex_type != WF_GET_INFO_XILINX_4010E )
     )
  {
   fprintf( stdout, "fifoxmpl cannot handle PE type %ld", pex_type);
   return(  ERR_FIFOXMPL );
  }

fprintf ( stdout, "Enabling PE reset for all PEs ...\n");
ret_val = WFireResetPE ( hBoard, WF_ALL_PEs, WF_ENABLE );
RETURN_FAILURE( ret_val );

/* allow PE's to use LEDs */
fprintf ( stdout, "Turning off green LEDs ...\n" );
ret_val = WFireSetGrnLEDs( hBoard, WF_DISABLE, WF_ALL_PEs );
RETURN_FAILURE( ret_val );

fprintf ( stdout, "Turning off red LEDs ...\n" );
ret_val = WFireSetRedLEDs( hBoard, WF_DISABLE, WF_ALL_PEs );
RETURN_FAILURE( ret_val );

fprintf ( stdout, "Enabling PEs to drive LEDs ...\n" );
ret_val = WFireSpinPE( hBoard, WF_ALL_PEs, WF_ENABLE );
RETURN_FAILURE( ret_val );

/*
 * Mask all PE interrupts so that no spurious asynchronous events are
```

```
     * scheduled during PE programming.
     */
    fprintf ( stdout, "Masking all PE interrupts ...\n" );
    ret_val = WFireMaskPEInts( hBoard, WF_ALL_PEs );
    RETURN_FAILURE( ret_val );

    /*
     * Program the PEs with the FIFO example binary LCA prom files.
     * These binary prom files are created by the lcafcon program from a
     * MCS prom file.The MCS prom file is a result of VHDL synthesis and
     * Xilinx place androute.
     */
    iPe0Prefix = PREFIX_INVALID;

    if (  pe0_type == WF_GET_INFO_XILINX_4028EX )
      {
        iPe0Prefix = PREFIX_4028X304;
      }

    if (iPe0Prefix == PREFIX_INVALID )
      {
        fprintf( stdout, "Unable to configure for PE 0, type %s\n",
WFireGetInfoXilinxTypeName( (WORD) pe0_type ) );
        RETURN_FAILURE( ERR_FIFOXMPL );
      }

    p_filename = strcpy( filename, pathPrefix[ iPe0Prefix ] );
    p_filename = strcat( filename, file_slash );
    p_filename = strcat( filename, pexfilename[0]);
    p_filename = strcat( filename, file_extension );

    fprintf ( stdout,
      "Programming PE0 with Data_Flow_Controller design from %s...\n",
      p_filename );
    fflush( stdout );

    ret_val = WFireProgramPE (hBoard, WF_PE0_BIT, p_filename);
    RETURN_FAILURE( ret_val );

    if ( (pex_type == WF_GET_INFO_XILINX_4010 ) ||
(pex_type == WF_GET_INFO_XILINX_4010E ))
      {
        iPexPrefix = PREFIX_4010_208;
      }
    else
      {
        fprintf( stdout, "Unable to configure for systolic PE type\n");
        return(  ERR_FIFOXMPL );
      }

    p_filename = strcpy( filename, pathPrefix[ iPexPrefix ] );
```

```
p_filename = strcat( filename, file_slash );
p_filename = strcat( filename, "pexblank");
p_filename = strcat( filename, file_extension );

fprintf ( stdout, "Programming all PEs with Blank design from %s ...\n", p_filename );
ret_val = WFireProgramPE (hBoard, WF_ALL_PE1_16, p_filename);
RETURN_FAILURE( ret_val );

for (i = 1; i < 9; i++ ) {
  if (pexfilename[i][0] == '\0') { continue; };

  p_filename = strcpy( filename, pathPrefix[ iPexPrefix ] );
  p_filename = strcat( filename, file_slash );
  p_filename = strcat( filename, pexfilename[i]);
  p_filename = strcat( filename, file_extension );

  fprintf ( stdout, "Programming PE%d with design from %s ...\n",
    i, p_filename );

  ret_val = WFireProgramPE (hBoard, 1<<i, p_filename);
  RETURN_FAILURE( ret_val );
}

for (j=1024;j<=32*1024;j*=2) {
  fprintf ( stdout, "Enabling PE reset for all PEs ...\n");
  ret_val = WFireResetPE ( hBoard, WF_ALL_PEs, WF_ENABLE );
  RETURN_FAILURE( ret_val );

  /* Write to the Interrupt Status Register in order to clear all PE      */
  /* interrupts that might still be pending on the board.  This is        */
  /* done to eliminate the possibility that a spurious interrupt from a   */
  /* previous session will trigger an asynchronous event for the process. */

  fprintf ( stdout, "Clearing PE interrupts ...\n" );
  ret_val = WFireClearPEInts ( hBoard, WF_ALL_PEs );
  RETURN_FAILURE( ret_val );

  /*
   * Mask all PE interrupts so that no spurious asynchronous events are
   * scheduled during PE programming.
   */
  fprintf ( stdout, "Masking all PE interrupts ...\n" );
  ret_val = WFireMaskPEInts( hBoard, WF_ALL_PEs );
  RETURN_FAILURE( ret_val );

  /* Flush any pending PE0 interrupt events so no spurrious           */
  /* asynchronous events will be generated.  Note: Here we are waiting */
  /* for this function to timeout.  If the function returns with      */
  /* WF_SUCCESS, that means an interrupt was pending from PE0         */
  /* immediately after the flush.  Since this is abnormal behavior, we */
  /* should return after such an event.                               */
```

```
    fprintf ( stdout, "Flushing PE asynchronous interrupt events ...\n" );
    ret_val = WFireWaitForPEInt( hBoard, WF_PE0_BIT, WF_FLUSH, 0 );
    if ( ret_val == WF_SUCCESS )
      {
fprintf( stdout, "command failed.\n");
return(  ERR_FIFOXMPL );
      }

    fprintf ( stdout, "Flushing PE asynchronous interrupt events ...\n" );
    ret_val = WFireWaitForPEInt( hBoard, WF_PE1_BIT, WF_FLUSH, 0 );
    if ( ret_val == WF_SUCCESS )
      {
fprintf( stdout, "command failed.\n");
return(  ERR_FIFOXMPL );
      }

    /* Unmask interrupts for PE0.  Since this application only expects to */
    /* see interrupts from PE0, all other interrupts remain masked off.   */

    fprintf ( stdout, "Unmasking PE0 and PE1 interrupts ...\n" );
    ret_val = WFireUnMaskPEInts( hBoard, WF_PE0_BIT );
    ret_val = WFireUnMaskPEInts( hBoard, WF_PE1_BIT );
    RETURN_FAILURE( ret_val );

    /* Set the clock configuration, frequency and set the clock to */
    /* free run mode. This is done before the resetting of the PE0 */
    /* input FIFO since the clock must be running in order for     */
    /* WFireResetFIFO to work properly.              */

    fprintf ( stdout, "Configuring Pclk for local mode ...\n" );
    ret_val = WFireClkConfig( hBoard, WF_LOC, bpc_mode );
    RETURN_FAILURE( ret_val );

    fprintf ( stdout, "Setting the frequency to %f MHz...\n", *clk_freq );
    ret_val = WFireClkSetFreq( hBoard, clk_freq );
    RETURN_FAILURE( ret_val );

    fprintf ( stdout, "Free running Pclk ...\n" );
    ret_val = WFireClkFreeRun( hBoard );
    RETURN_FAILURE( ret_val );

    /* Reset the FIFOs for PE0 and PE1.  Note that Pclk must be running */
    /* in order for this to work.  The third and fourth arguments of    */
    /* WFireResetFIFO are the almost-full and almost-empty flag offset  */
    /* values, respectively. These values should match the flag offset  */
    /* values used in the WILDFIRE VHDL simulation model.               */

    fprintf ( stdout, "Resetting PE0 FIFO ...\n" );
    ret_val = WFireResetFIFO( hBoard, WF_PE0_OUT_FIFO, AF_OFFSET, AE_OFFSET );
    RETURN_FAILURE( ret_val );
```

```
   fprintf ( stdout, "Resetting PE1 FIFO ...\n" );
   ret_val = WFireResetFIFO( hBoard, WF_PE1_IN_FIFO, AF_OFFSET, AE_OFFSET );
   RETURN_FAILURE( ret_val );

   /* Take all of the the PEs out of reset mode now that everything is */
   /*  configured properly.                                           */

   pbuffer = ( DWORD * ) WildfireAlloc ( 128*1024 * sizeof ( DWORD ));

   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_BLOCKED);

   for (i=0; i < 128*1024;i++) pbuffer[i] = 0L;
   /*   for (i=0; i < 128*256; i++) pbuffer[i] = i;
for (i=128*256; i <128*512;i++) pbuffer[i] = i-128*512;
for (i=128*512; i <128*1024;i++) pbuffer[i] = 0;
   */

   for (i=256;i<512;i++) { pbuffer[i] = i-256; }
   for (i=512;i<768;i++) { pbuffer[i] = i-512; }

   pbuffer[0] = 0L;
   pbuffer[1] = j;
   pbuffer[2] = 1L;
   pbuffer[3] = 256+j;
   pbuffer[4] = 256+j*2;
   pbuffer[5] = 256+j*3;

   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_BLOCKED);
   ret_val = WFireWrite(hBoard, 1, 0, 128*1024, pbuffer);

   gettimeofday(&first, &tzp);

   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_SW_ARBITRATE);

   fprintf ( stdout, "Disabling reset for all PEs ...\n" );
   ret_val = WFireResetPE( hBoard, WF_ALL_PEs, WF_DISABLE );
   RETURN_FAILURE( ret_val );

   fprintf ( stdout, "Waiting for the interrupt from PE1 ...\n" );
   ret_val = WFireWaitForPEInt (hBoard, WF_PE1_BIT, WF_ALL, 6000);
   RETURN_FAILURE( ret_val );

   gettimeofday(&second, &tzp);
   if (first.tv_usec > second.tv_usec) {
     second.tv_usec += 1000000;
     second.tv_sec--;
   }
   elapsed.tv_usec = second.tv_usec - first.tv_usec;
   elapsed.tv_sec  = second.tv_sec  - first.tv_sec;
```

```
    fprintf (stdout,
             "%d.%.6d seconds to execute %d\n",
             elapsed.tv_sec,elapsed.tv_usec,j);
   }

   /* read back the memory elements */

   fprintf ( stdout, "Reading PE1 memory ...\n" );

   ret_val = WFireSetMemoryMode(hBoard, WF_PE1_BIT, WF_BLOCKED);

   ret_val = WFireRead (hBoard, 1, 0, 128*1024, pbuffer);

   RETURN_FAILURE( ret_val );

   /*   for (i=0; i < 1024; i+=8) {
     fprintf (stdout, "Address %6x: ",i);
     for (j=i;j<i+8;j++) {
      fprintf (stdout, "%4x ",pbuffer[j]);
     }
     fprintf (stdout, "\n");
   }
   */

   return ret_val;
}

/****************************** Local Function   **************************
 * Function     : WF_ShutdownWildfire
 * Parameters  : unit     : Bit3 unit (including VME backplane)
 *               hBoard     : Slot in the unit
 * Returns     : WF_ERROR
 * Description : Shuts down a WILDFIRE system specified by unit and slot.
 ***********************************************************************/
#ifdef PROTO_DEF
WF_ERROR WF_ShutdownWildfire ( WORD unit, WF_BOARD hBoard )
#else
WF_ERROR WF_ShutdownWildfire ( unit, hBoard )
     WORD unit;
     WF_BOARD hBoard;
#endif
{
WF_ERROR ret_val = WF_SUCCESS;
WF_CLOCK_STATUS pclk_status;

/* Mask and clear any pending PE interrupts */

   fprintf ( stdout, "Masking all PE interrupts ...\n" );
   ret_val = WFireMaskPEInts( hBoard, WF_ALL_PEs );
   RETURN_FAILURE( ret_val );
```

```
   fprintf ( stdout, "Clearing all PE interrupts ...\n" );
   ret_val = WFireClearPEInts( hBoard, WF_ALL_PEs );
   RETURN_FAILURE( ret_val );

   /* Reset the PE0 and PE1 FIFOs */
   fprintf ( stdout, "Resetting PE0 FIFO ...\n" );
   ret_val = WFireResetFIFO( hBoard, WF_PE0_OUT_FIFO, AF_OFFSET, AE_OFFSET );
   RETURN_FAILURE( ret_val );

   fprintf ( stdout, "Resetting PE1 FIFO ...\n" );
   ret_val = WFireResetFIFO( hBoard, WF_PE1_IN_FIFO, AF_OFFSET, AE_OFFSET );
   RETURN_FAILURE( ret_val );

   fprintf ( stdout, "Get clock status ...\n" );
   ret_val = WFireClkGetStatus ( hBoard, &pclk_status, 5 );
   RETURN_FAILURE( ret_val );

   if ( pclk_status.run_state == WF_RUNNING )
     {
      fprintf ( stdout, "Clock is free-running. Stopping Pclk ...\n" );
      ret_val = WFireClkSuspend( hBoard );
      RETURN_FAILURE( ret_val );
     }

   /* Close the WILDFIRE unit  */
   fprintf ( stdout, "Closing WILDFIRE ...\n" );
   ret_val = WFireCloseBoard( hBoard );
   ret_val = WFireClose( unit );
   RETURN_FAILURE( ret_val );

   return ret_val;
}
```

# Appendix C

# Vector Addition: DSP Code

```
/* add two vectors together and time how
long it takes */

#include <stdio.h>
#include "/files1/match/mpi_dsp/mpi_dsp.h"

#define SIZE 1024*256

int *vector_a;
int *vector_b;
int *vector_c;

int size = SIZE;
int stime, etime;
int i,j;

void main () {

  vector_a = malloc (sizeof (int) *SIZE);

  vector_b = malloc (sizeof (int) *SIZE);

  vector_c = malloc (sizeof (int) *SIZE);

  for (size=1024; size <= SIZE; size*=2) {

    for (i=0; i<size; i++) {
     vector_a[i] = i;
     vector_b[i] = size-i;
    }

    stime = timer_now();

    for (i=0; i<size; i++) {
     vector_c[i] = vector_a[i] + vector_b[i];
    }
```

```
    etime = timer_now();
    printf ("Size: %d Time: %d\n",size,etime-stime);
  }
}
```