The Design of an Efficient Hardware Subroutine Protocol for FPGAs

by

Trevor Joseph Bauer

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

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Trevor J. Bauer, 1994
All rights reserved

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part, and to grant others the right to do so.

Signature of Author ...

Department of Electrical Engineering and Computer Science
May 16, 1994

Certified by ..................................................

Anant Agarwal
Thesis Supervisor

Accepted by ..........................................

F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses
The Design of an Efficient Hardware Subroutine Protocol
for FPGAs

by
Trevor Joseph Bauer

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract
Reconfigurable logic devices allow for great flexibility in the ways computing tasks
are accomplished. With them, operations which require the flexibility of software at
the high speeds of custom hardware can be performed. A system has been developed
in which computationally intensive software routines are identified and implemented
on dedicated, reconfigurable hardware increasing processing speeds without the cost
and permanence of custom ASICs. Hardware subroutines maintain the abstraction,
portability, and ease of implementation of software. Several algorithms have been
demonstrated on the Virtual Wires board, illustrating the benefits and possibilities
of such a system.

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

The Virtual Wires Group, Jonathan Babb, Russ Tessier, Matthew Dahl, Silvina Hanono, and David Hoki, contributed the underlying system for this project. Without their assistance and the substantial Virtual Wires Emulation System, I would not have had the opportunity to explore FPGA computing so freely. Thanks to Anant Agarwal, who was tremendous in encouraging me and providing me with inspiration and enthusiasm.

George Hadjiyiannis, Kirk Johnson, John Kubiatowicz, Gino Maa, Anne McCarthy, and Ed Ouellette provided support and insight when disks crashed, hardware failed, or I became simply confused.

Thanks also to Bits, who encouraged me stay cool and smile. Ali kept the free food messages coming. Mark’s grape-apple-lemon-nutmeg icecream is the best way to end an evening. Thanks also to Damon for never being too busy to welcome a little interruption. My friends offered encouragement, motivation, and distraction, all at the appropriate times.

And finally, many thanks to my marvelous parents and sister, Robyn, whose positive outlook and warm fuzzies are irreplaceable. Their confidence and support has helped me all along the way.
Contents

1 Introduction .................................................. 9
  1.1 Background .............................................. 9
  1.2 Motivation ............................................... 10
  1.3 What is a Hardware Subroutine? ......................... 11
  1.4 Goals and Accomplishments of This Thesis .............. 12

2 Background ..................................................... 13
  2.1 Reconfigurable Hardware ................................ 13
  2.2 Existing FPGA Systems ................................... 14
    2.2.1 Logic Emulation .................................... 14
    2.2.2 Multi-Purpose Hardware ............................. 14
    2.2.3 FPGA Computing/ Adaptable Systems ............. 15

3 Environment ..................................................... 16
  3.1 Necessary Assumptions .................................. 16
  3.2 The Virtual Wires Platform ............................. 17
  3.3 Configuration of the FPGAs .............................. 18

4 Methods .......................................................... 20
  4.1 My Experience ......................................... 20
  4.2 Design Strategy ......................................... 21

5 The Hardware Subroutine Platform ......................... 23
  5.1 Protocol ................................................. 23
5.2 The Hardware Implementation ........................................... 24
5.3 Taking Advantage of Hardware Subroutines ......................... 25

6 Results ................................................................. 27
6.1 Example Algorithms .................................................. 27
   6.1.1 Bitwise-Or ...................................................... 28
   6.1.2 Vector Operations .............................................. 28
   6.1.3 Floating Point Operations .................................... 30
   6.1.4 Sorting Arrays .................................................. 31
6.2 System Bottlenecks .................................................. 33

7 Conclusions and Future Research .................................... 34
7.1 Conclusions .......................................................... 34
7.2 Future Research ..................................................... 34
   7.2.1 Automating the Process ...................................... 35
   7.2.2 Alternative approaches to input and output ................. 35

A Interface Code ....................................................... 37
   A.1 Code for run_board.c ............................................ 37
   A.2 Code for reset_board.c .......................................... 40

B Example Bubble Sort Routine ....................................... 41
   B.1 Shell for Routine ................................................ 41

C Three Element Cross Product ........................................ 46
   C.1 Shell For Routine ................................................ 46
   C.2 Cross-Product Software Subroutine ............................ 47
   C.3 Cross-Product Hardware Subroutine ............................ 48

D Eight Element Inner Product ........................................ 49
   D.1 Shell For Routine ................................................ 49
   D.2 Inner-Product Software Subroutine ............................ 50
   D.3 Inner-Product Hardware Subroutine ............................ 51
List of Figures

1-1 interpolate_data() is implemented in hardware. .................. 11
3-1 The Virtual Wires Emulation Board ............................... 18
6-1 Bitwise-Or function ............................................. 29
6-2 Three-dimensional cross-product ................................. 30
6-3 Inner product of eight-element arrays ......................... 31
6-4 Performing a parallel bubble sort. .............................. 32
List of Tables

6.1 Software vs. Hardware Performance Comparison .............. 28
Chapter 1

Introduction

1.1 Background

The introduction of reconfigurable logic to the computing community during the past decade has expanded computing possibilities by allowing hardware to be customized with increased flexibility, shorter development times, and lower cost. In the past, the distinction between hardware and software domains was more easily identifiable, and the difference was extreme. To execute a task with maximum efficiency, custom hardware needed to be designed, and flexibility was sacrificed. When architecture was designed to run many general purpose computing tasks, performance for any particular function was sacrificed. Certainly, the trend in recent years has been to generalize microprocessors and increase compiler responsibility while shrinking the instruction set. This trend towards reduced instruction set computing, RISC, was appropriate considering several technological advances: the increasing numbers of clock cycles required for memory accesses and the benefits of performing complex instructions as a series of pipelined simple instructions. However, to perform larger, computational functions such as digital signal processing, floating point operations, graphics acceleration, and countless others, designers today are incorporating customized co-processors for improved results. While these application specific integrated circuits, ASICs, conveniently and successfully improve performance, they are used only by a small subset of operations.
It would be convenient to have a reconfigurable coprocessor which serves as a floating point unit some of the time, a signal processing ASIC at another time, and a compression/decompression or encryption/decryption ASIC a short period later. Such a reconfigurable system would allow the microprocessor or compiler an opportunity to identify highly parallel key routines which are more suited for customized hardware. Field-programmable gate arrays, FPGAs, are the reconfigurable ICs required for such a system. Used effectively in custom computing environments, they run software procedures at speeds until now available only with custom hardware.

1.2 Motivation

There are multiple uses of FPGAs. They have been successfully utilized for prototyping ASICs, since they have a very short development time and the cost of modifying a design is minimal. They comprise logic emulation systems [3] [13] and are sold in commercial packages to ASIC developers for this purpose. Programmable gate arrays have been used as reconfigurable processors and preprocessors in customized computing systems. And they can replace small quantities of simple ASICs. All of these possibilities and more illustrate the success of field-programmable gate arrays. The Virtual Wires Group at the Massachusetts Institute of Technology has focused on logic emulation with FPGAs. By successfully eliminating the physical pin limitations of partitioned netlists, virtual wires have improved gate utilization and thus dramatically lowered cost and increased speed.

An alternative application for reconfigurable hardware is FPGA computing: utilizing FPGAs to increase performance and functionality in a general computing environment. In the future, computing systems will likely comprise both general-purpose processors and configurable hardware. [11] The motivation behind this is two-fold. First, many researchers have noticed that ten percent of written code takes ninety percent of the execution time. Hence, if that ten percent could be executed on an ASIC customized for just that functionality, large performance gains could be realized. The second motivating factor behind this research is that no processor is perfect.
Figure 1-1: `interpolate_data()` is implemented in hardware.

for every task. In a conscious effort to make processors handle all tasks well, designers have limited performance of specific tasks.

1.3 What is a Hardware Subroutine?

When writing software, a designer implements many subroutines which perform various functions. An interface forms an abstraction for the underlying complexity and implementation. A hardware subroutine is not much different. To the software engineer, it is a standard procedure call which accepts several arguments and returns upon completion. The difference lies in the implementation. The hardware subroutine is customized on reconfigurable hardware, and the task will execute on the FPGAs rather than as a series of micro-instructions. Figure 1-1 illustrates the concept of a hardware subroutine among software.
1.4 Goals and Accomplishments of This Thesis

This research is an effort to realize possibilities of FPGA computing with gate utilization benefits of Virtual Wires. A system has been developed in which computationally intensive software, identified as suitable for customized hardware and configured onto an FPGA board, is referenced as a hardware subroutine. A collection of algorithms has been implemented using the Verilog hardware description language and placed on the Virtual Wires board demonstrating design issues involved with such an environment. Additionally, a controller on the FPGA board reduces bandwidth between the hardware subroutine and the host computer. It also effectively provides a means through which the hardware subroutine can include iterations and loops of unspecified length.

Due to the nature of the hardware subroutine, many aspects of FPGA computing can be leveraged upon. For instance, the inherent parallelism present in any circuit can be utilized by an innovative designer. Additionally, a number of subroutines can be placed simultaneously on one array of FPGAs and accessed in parallel via a very long instruction word. A final goal of this hardware subroutine system is to provide a convenient means for experimentation with many architectural and algorithmic concepts and quickly acquiring results.
Chapter 2

Background

2.1 Reconfigurable Hardware

There are a variety of reconfigurable hardware alternatives. Xilinx, Altera, AT&T Microelectronics, Motorola, and several other companies are heavily involved in the field. While some FPGAs are nor-gate based, the ones used by the Virtual Wires group for the purpose of this research are SRAM based. Specifically, they are the Xilinx 4005 SRAM based FPGAs. The nor-gate based field-programmable gate arrays are self-described; they consist of arrays of nor gates with customized routing to implement various logic functions. Xilinx SRAM based FPGAs' fundamental elements are configurable logic blocks (CLBs). Other FPGA elements include input/output blocks (IOBs) and interconnect [9].

The CLBs take at least four bits of address input and effectively perform a lookup of a corresponding one bit value. In this way, the CLB serves as a four-input combinatorial logic function. Larger modules are created by routing CLBs together. Between them, the interconnect, and the on-chip flip-flops (the IOBs), the skeleton of any ASIC is possible. The reconfigurability of CLBs opens the door to FPGA computing; this research represents one such opportunity.
2.2 Existing FPGA Systems

FPGAs have been used in a variety of systems. Three categories of research which predominate the field are built-in diagnostics, multi-purpose hardware, and adaptable system design[5]. A number of sponsors encourage research and development in each of these areas. IEEE workshops have been held for field-programmable gate arrays, custom computing machines, and programmable logic. In addition, FPGAs have appeared in journals and workshops for computer-aided design and VLSI. All of these combine the complexity and performance of ASICs with the flexibility and simplicity of a user-programmable device.

2.2.1 Logic Emulation

The Virtual Wires Project at the Massachusetts Institute of Technology is grounded in logic emulation. This group invented and implemented a system through which pin limitations, normally imposed when large netlists are partitioned, can be avoided. Thus, large netlists are simulated on a small number of FPGAs [1]. Complete systems which perform this task are marketed by several companies. Quickturn, Virtual Machine Works, Zycad, and Aptix are among designers of such ASIC emulation units, achieving four to six orders of magnitude in speed over software simulations.

2.2.2 Multi-Purpose Hardware

Customizations can be made to computers for performance gains; for instance, a math co-processor is often added to systems, increasing the time required for floating point operations. High-speed compression boards help to increase the data transfer rate on hard disks as well as data storage potential. Advanced video capabilities require a video card. And for modern communications, a fax/modem or network access card is essential. These cards are costly in both space and cash. Additionally, none of these special-purpose hardware pieces is used continuously. If a convenient way of interchanging them were available, it might be possible to get by with one of them at a time. Field-programmable gate arrays provide such an opportunity.
The intention of multi-purpose hardware design is to supply a board which can be configured to perform special hardware functions. While masked ICs are three times cheaper and contain ten times the array capacity of FPGAs, they lack the flexibility of a multi-purpose hardware system.

The AnyBoard reconfigurable hardware system, designed by a group of researchers at North Carolina State University [12] is one such project. Although AnyBoard is a rapid-prototyping system used for designing special-purpose boards, a faster, cheaper system similar to the AnyBoard could be practically used as multi-purpose hardware.

More closely resembling a true reconfigurable system is the Splash Board [6]. Splash effectively uses FPGAs as a reconfigurable parallel computation resource. Splash 2 (the successor of Splash) is a special purpose parallel processor which primarily uses FPGAs for computation; the microprocessor serves only for support [2].

### 2.2.3 FPGA Computing/Adaptable Systems

This study is categorized as FPGA computing, specifically hardware subroutines. A great deal of research has been done in the past regarding similar investigations. At the University of Hawaii, for instance, a Xilinx board was used to execute user programs, either partially or completely [10]. A group of researchers at Plessey Semiconductors investigated potentials for swapping hardware subroutines in and out of FPGAs in the form of Electrically Reconfigurable Arrays [8]. The PRISM-II system [14] improved performance of many tasks by "augmenting the functionality of the core processor with new instructions that match the characteristics of targeted applications."
Chapter 3

Environment

The hardware subroutine platform consists of a set of software utilities and an FPGA board. A printed circuit board designed by Russ Tessier for the Virtual Wires project is used. The software utilities serve several functions: generating the Xilinx configuration program, partitioning logic (when the logic exceeds the size of one gate array), and configuring FPGAs. Other required software for incorporating hardware subroutines includes both the tools for communicating with the board across ethernet and the run_board interface bridging hardware and software.

3.1 Necessary Assumptions

Standard computing environments today are not quite ready for the system proposed by this research. In fact, many assumptions are made, and there are many tasks which must be automated to make this platform effective. Most of this automation will be included in compilers. For instance, a key element needed to maximize the benefit from hardware subroutines is in the identification of the task to be implemented on the configurable system. Researchers have discussed characteristics of such routines [11] and compilers could use these traits to identify these appropriate tasks, interpret software descriptions of these routines, and convert them into logic gates for hardware implementation. The software designer identifies some of these routines and is able to make notes to the compiler regarding intelligent decisions. There are ongoing
projects working to tackle both of these opportunities. Special compilers exist, at Brown University, for instance, which translate subsets of C code to verilog. Verilog is then compiled with synthesis tools to a gate level representation suitable for hardware configuration. Currently, these steps must be performed manually. Measures taken to overcome these difficulties are documented in Chapter 4, Methods.

3.2 The Virtual Wires Platform

The software designed for the Virtual Wires Emulation System [3] conveniently assists in the implementation of algorithms on the FPGA array. Once a routine is designed in verilog, an LSI netlist can be synthesized. The Virtual Wires software incorporates necessary elements of virtual wires, including cell controllers, the shift registers, and the various logic partitions. Occasionally a larger circuit merits hand partitioning, but in general this system effectively translates gate level designs into executables for loading onto the FPGA array without designer intervention.

The array itself (Figure 3-1) consists of sixteen Xilinx 4005 FPGAs connected in a nearest-neighbor mesh. The board maintains expansion capabilities, with ports for additional boards on each of its four sides, north, west, south, and east. The hardware subroutines are generally small circuits compared with full scale microprocessors; for the algorithms demonstrated herein, a single board proved more than adequate. The FPGA board connects to a Sun SPARC station via a serial port, about one kilobyte/second, interfacing with a Motorola HC11 [7]. The serial port is slow, but it successfully controls data input and output between the board and host. The modified virtual wires controller, which enables multiple computation cycles to be run on the board between transfers across the serial interface, improves bandwidth dramatically and reduces the cost of I/O. System improvements which reduce I/O costs include modified on-board controllers and an SBUS interface. The SBUS has three orders of magnitude more bandwidth than the serial port. Although the penalty due to I/O is great enough to outweigh the performance gains of hardware subroutines, it is not relevant to this investigation. FPGAs must eventually be included on the board with
the microprocessor to create a viable solution for performance enhancement. Once this is accomplished, the I/O problem will be significantly reduced.

### 3.3 Configuration of the FPGAs

The Xilinx FPGAs are configured via a serial xchecker (Xilinx proprietary) cable. Future improvements in the FPGA architecture will aid in the success of FPGA computing. Some of these modifications are directly related to the process of programming gate arrays for their logic function. One of these, which will be achieved through improved fabrication processes and architectures, is a reduction of the time required for configuration. Currently, if a single Xilinx 4005 FPGA is on the same
board as the microprocessor the loading can be achieved in 32ms [5]. At 66Mhz this is equivalent to approximately two million clock cycles – an expensive task if performed frequently. Perhaps it might be worthwhile to designate a second processor in charge of FPGA configuration. In any case, the overhead required for performing a hardware subroutine will drop as this time decreases. Another capability which would add to the flexibility of FPGA computing includes a feature for configuring one portion of the FPGA while letting the rest continue running. Atmel Corp. currently has an architecture available with this capability. This allows faster reconfiguration and self-modification of FPGAs. With these advancements, methods for accessing hardware subroutines become both beneficial and imperative.
Chapter 4

Methods

4.1 My Experience

The members of the Virtual Wires team did a superb job of setting up a convenient, robust environment which served as a foundation for this system. In order to design a hardware subroutine platform, I first became familiar with the Virtual Wires environment, described in Chapter 3. My first goals included taking relatively simple functions, writing them in verilog, and understanding the procedure for development on the Virtual Wires board. The communications between the workstation and the board were through verilog; that system was designed by Silvina Hanono [7].

Once I felt comfortable in this environment, my next goal was to modify the communications to drive the board without the verilog simulator. To run a hardware subroutine, I needed a way to access the board via a C interface. Routines for passing data over the LCS network and through the host’s port had already been written for similar tasks [K. Johnson, 1991] and I was able to reuse much of it in my own interface. My first test procedure was a very simple one which returned the bit-wise-or of two eight bit characters (Figure 6-1). My plan was to get a bare bones demonstration up and working. Once the circuit was running on the virtual wires board and I could access it from a software routine, without the use of the verilog interface, many opportunities became available.
4.2 Design Strategy

This strategy of getting something simple up and running first, and incrementally improving it later is one that I try to practice consistently in my work. Thus my first implementation shows very poor performance and has few capabilities, but once the shell has been formed, it becomes an easier task to add features than it might have been to implement the entirety from scratch. This technique especially helps in the debugging process, because it ensures modularity and testability. Performing a bit-wise-or via excessive amounts of messy software was my first run prototype. It enabled me to proceed by adding features, optimizing code, and cleaning up.

The next step was thus abstracting the software involved in calling a hardware subroutine from the SPARC station. By condensing functionality and consolidating various pieces of code, I successfully hid the details from the programmer. From then on, the subroutine call need only include the proper files and reference the run_board procedure. Later this became a pair of procedures, run_board and reset_board, allowing the designer multiple calls to the hardware subroutine without having to reset the Virtual Wires Emulation system.

Next, I was able to advance to more complicated algorithms, sorting and vector operations, for instance, which gave me hopes of achieving performance gains, rather than the significant losses which were present with the simpler routines. However, problems in this system lay in the bandwidth. Since the virtual wires system was originally designed for logic emulation, it made sense that the outputs should be driven to the workstation (via the serial interface) after every emulation cycle. Outputs are read and the next test vector could be sent to the board. For the purpose of FPGA computing, however, this does not make sense.

More often than not, hardware subroutines require multiple iterations; they can run for many cycles without warranting host interaction. This requires a controller which intelligently handles communication. The virtual wires system is configured with a controller to manipulate signals between the HC11 and FPGAs, determining when each emulation cycle begins and when the communication between the FPGA
board and the workstation should be initiated. By intercepting the signals from the functional logic before they reached the controller, I forged the necessary sequence to the virtual wires logic causing additional emulation cycles to occur. The modified controller can also reroute inputs; it can feed back outputs or, potentially, access the on-board SRAMs. To inform the controller of its status, the subroutine manages its state so that it properly generates a done signal. This signal, when active, signifies the end of the procedure and the subroutine’s return. With a controller that effectively eliminates unnecessary communication between the workstation and the FPGA board, both the required bandwidth and the time to complete hardware subroutines were successfully reduced.
Chapter 5

The Hardware Subroutine Platform

5.1 Protocol

The protocol consists of several control paths and data constraints. Certainly, large amounts of flexibility are available to any designer wishing to use the hardware subroutine system. The basic scheme consists of two software subroutines, reset_board and run_board.

- void reset_board( int in_array.size,
                     int out_array.size)

- void run_board( unsigned char * in_array,
                  int in_array.size,
                  unsigned char * out_array,
                  int out_array.size)

These routines are called from within a C procedure. reset_board initiates communication between the workstation and the FPGA board, and resets the virtual wires system encapsulating the hardware subroutine. It is called once, before run_board is called. run_board is then called as many times as appropriate. Each time it is called, it properly formats arrays and appends the write enable signal, \texttt{weN}, and done signal,
DoneN. The elements of these arrays can be of any data type; subroutines have been implemented using floats, ints, and chars. Currently, run_board is designed to run on a SPARC station, which is a big endian machine. When run_board returns, the output array contains the output vector from the FPGA board. These functions are listed in sections A.1 and A.2.

The routine reset_board establishes the proper port as defined in the interface.ctl file. Once this connection is established, code written by Kirk Johnson in 1991 is used to pass data across the network, from any LCS station, through the host SPARC, and to the board. On board, the virtual wires circuitry is reset.

The arguments for run_board are pointers to arrays. This is not to say that a designer may not chose to implement a subroutine more suited for a collection of independent arguments, but in my practice most hardware suitable subroutines have included significant amounts of data easily represented with arrays. If a designer wishes to implement a task with many data types, it should be conformed into an array of one type to be passed. Though this may seem awkward, to do otherwise would be to destroy the generality of hardware subroutines.

5.2 The Hardware Implementation

The hardware module may be written in a hardware description language such as verilog, and it must include an input for the active low write_enable signal, fed from the controller, and an output for the done signal fed to the controller. The write_enable signal must cause the inputs to the subroutine to be read from the serial interface (via the HC11). Once the write_enable signal becomes inactive, the routine should begin execution. If, at the end of an emulation cycle, the subroutine is finished, the done signal should be active, causing the output signals to be sent through the HC11, back to the host.

There is another requirement of the hardware subroutine. In the verilog description of the function, the inputs and outputs should be ordered:

write_enable, input_elements..., done_signal, output_elements..., clk
The reason for this order is two-fold. First, the virtual wires system requires that inputs be listed first, outputs be listed last, and the emulation clock can arbitrarily fall anywhere. Second, the controller which specifically works with write-enable and done uses the order to identify those wires. Any bit-wide status symbols or operands should be placed immediately after the write-enable for inputs or after the done-signal for outputs.

5.3 Taking Advantage of Hardware Subroutines

It is appropriate that eventually many of the steps involved with taking advantage of hardware subroutines be automated, tackled by compilers. For now, however, the user must perform much of the development manually. The first step in this process is writing the software. Namely, the procedures that will eventually be run on the board should be written in C. This code will serve as a functional specification and a platform from which behavior of the hardware can be compared, evaluated and debugged. Then the critical section of this software must be identified. As long as these steps are performed manually, only procedures which are obviously appropriate may find their way to becoming a reconfigurable hardware subroutine; however, a clever compiler may be able to more efficiently identify elusive, computationally intensive portions of the software suitable for hardware.

Once the subroutine has been identified, it must be converted to a logic function suitable for hardware implementation. A gate-level representation can be generated from any level, behavioral, functional, or structural, of verilog. The debugging of the subroutine can be performed with the verilog simulation tool and then compiled to gates after its functionality has been verified. In this research, Synopsis tools were used for this compilation. This netlist is then run through the virtual wires software platform to generate a test.exe file which can be loaded directly onto the Xilinx FPGAs, in this case via the Xilinx xchecker cable. Once again, this phase of the process should eventually be automated as compilers become more adept at identifying potential parallelism in software interpretation from C to gates become
possible. Example code, both C and Verilog, for a bubble-sort algorithm is listed in Appendix B.

The subroutine present on the FPGAs can be debugged using tools developed by Silvina Hanono [7]. These tools, \texttt{vw\_run\_serial\_debug} and \texttt{vw\_run\_verilog}, often proved convenient for tracing inputs and outputs to and from the FPGA board. However, for debugging the internal circuitry within the virtual wires environment and between functional partitions, a new tool would be useful. As it stands, tracing internal signals on the board requires propagating them to outputs and tracing them off the board. To simulate blocks of logic including the virtual wires encapsulation, each functional module had to be individually simulated and test vectors had to be hand generated. When bugs were discovered, they were more often in the test vectors themselves than in the logic. While Synopsis is consistent in converting verilog to gates and the virtual wires shell does little to disturb the overall functionality, it takes as much as three hours to remake the circuit.

With the subroutine debugged, the merge between software and hardware can be made in one step. The designated function can be replaced by the call to the \texttt{run\_board} procedure, passing pointers and sizes for the input and output arrays. The arrays can be of any data type, but, of course, the hardware subroutine must be expecting the proper data structures or havoc will arise. The design steps are summarized below.

- Design high level functionality in C code. Debug this using your favorite C debugger.

- Design a hardware subroutine to replace the computationally intensive software subroutines. Debug the verilog code with verilog tools. Debug entire hardware subroutine on the board with \texttt{vw\_run\_verilog}.

- Replace call to software algorithm with calls to \texttt{reset\_board} and \texttt{run\_board}.

26
Chapter 6

Results

6.1 Example Algorithms

Using this platform, a number of algorithms have been implemented. Algorithms were chosen for their inherent parallelism. They include a bit-wise-or, several vector operations, floating-point operations, and sorting tasks. Each function had unique advantages and drawbacks which determined its suitability as a hardware subroutine.

The presented data indicates that Virtual Wires should perhaps not be used with hardware subroutines. While proving successful for logic emulation, they are not be appropriate or cost-effective for hardware subroutines.

Computation times for each algorithm are determined for the both software subroutine and its hardware replacement. In determining the software execution times, the code is run on a DEC5000/25. The DECstation is clocked at 25Mhz. Unix tools prof and pixie provide numbers of cycles required to complete a subroutine, and thus fairly accurate execution times are attainable. The hardware subroutine performance times can be computed by multiplying the number of Virtual Wires Phases with the number of Virtual Wires Cycles in each phase. This product represents the number of clock cycles required for one iteration of the hardware subroutine; the Virtual Wires board is clocked at 20 Mhz which can be used to determining the total time of all iterations for a hardware subroutine call. Note that there are many 20 Mhz cycles in each emulation cycle, and it is the emulation cycle which clocks the
Table 6.1: Software vs. Hardware Performance Comparison

<table>
<thead>
<tr>
<th>function name</th>
<th>software time in microseconds</th>
<th>hardware time in microseconds</th>
<th>% of time faster (slower)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit-wise-or</td>
<td>&lt; 1</td>
<td>4.96</td>
<td>N/A</td>
</tr>
<tr>
<td>floating point multiply</td>
<td>&lt; 1</td>
<td>9.5</td>
<td>N/A</td>
</tr>
<tr>
<td>vector multiply</td>
<td>2.6</td>
<td>6.3</td>
<td>(146%)</td>
</tr>
<tr>
<td>sort of 8 elements</td>
<td>18</td>
<td>18</td>
<td>0%</td>
</tr>
<tr>
<td>8 element inner product</td>
<td>1.9</td>
<td>1.8</td>
<td>6.3%</td>
</tr>
<tr>
<td>sort of 16 elements</td>
<td>118</td>
<td>79</td>
<td>33%</td>
</tr>
</tbody>
</table>

hardware subroutine. The 20 Mhz clock merely triggers the bit-shifters. The time spent in I/O between the host and the Virtual Wires board is not included with the hardware time.

The outcome of these results are presented in Table 6.1. It is interesting to note that there is not a significant difference in performance. This is due to a number of bottlenecks in the hardware subroutine platform, addressed in Section 6.2.

6.1.1 Bitwise-Or

Figure 6-1 is a circuit diagram of the eight-bit-or function. This subroutine was used as a simple introduction to the system. It fits well within one FPGA, and the controller and Virtual Wires serial I/O logic sit on the other. The bitwise-or function illustrates some of the bottlenecks this hardware subroutine faces. Namely, this is a subroutine which should take a single clock cycle to complete. Instead, due to Virtual Wires overhead, cycles used for piping inputs and outputs from one FPGA to the other take forty-five cycles – significant overhead. This is an extreme example of a hardware subroutine which did not gain from the use of Virtual Wires, but rather paid a stiff penalty.

6.1.2 Vector Operations

Vector operations are much more conducive to parallelism which is not handled adequately by a serial processor. By taking advantage of this in both a cross product
Figure 6-1: Bitwise-Or function

of three dimensions and an inner product of two eight-element arrays, small gains in performance were achieved. This is with the Virtual Wires costing thirty-six 20 Mhz cycles per emulation cycle for the inner product and one hundred and twenty-six ticks per emulation cycle in the cross product.

Because the emulation cycles are so slow, critical paths in the hardware subroutine are not an issue. It would be difficult to violate the timing, and none of these algorithms had any timing troubles. However, the length of the paths for these circuits, illustrated in Figures 6-2 and 6-3 raise possibilities for pipelining data through the subroutine. One advantage of hardware subroutines is that architectural alterations such as pipelining are transparent. One must only remember to control the done signal appropriately so the controller can properly return output data.
A logic diagram of the cross product is presented in Figure 6-2. Figure 6-3 presents the inner product’s logic diagram. Each of these figures illustrate both the long paths that signals must travel through and the parallelism which can be exploited.

6.1.3 Floating Point Operations

A floating point multiplier was implemented using a hardware procedure call. Verilog code for it is listed in Appendix E. It consists of a multiplier for the mantissas, an adder for the exponents, and a finite state machine for normalizing the result. Because most modern sequential chips contain floating point units of their own, no performance gain was expected. In fact, due to the high level of inter-dependency in
Figure 6-3: Inner product of eight-element arrays

this circuit, twenty-one Virtual Wires phases are required for each emulation cycle. The multiplier also consumes all sixteen FPGAs. Packed CLB utilizations range from 7% to 88% (the average was 54%). Thus, performing a floating point multiply with a hardware subroutine is extremely inefficient and expensive; however, in this case the hardware subroutine provides a convenient means for exploring various hardware multiplication algorithms. By simply modifying the test.v file, numerous modifications and improvements can be made to the hardware.

6.1.4 Sorting Arrays

Verilog code for a parallel bubble sort algorithm is listed in Appendix B, and Figure 6-4 is an illustration of how a parallel bubble sort is accomplished. Performance
gains can be achieved via hardware subroutines, as confirmed by this demonstration. The time required to sort $n$ elements on a serial machine grows as $\text{Order}(n^2)$, while time required in the parallel algorithm is $\text{Order}(n)$. With this algorithmic difference, performance gain can always be achieved by a high enough value of $n$. The difference between sorting eight and sixteen elements is significant. Unfortunately, more than sixteen elements becomes a bit more difficult, simply due to the FPGA register limitations. By utilizing on-board SRAM, larger arrays can be stored, but this is at the cost of some of the parallelism.
6.2 System Bottlenecks

There are several bottlenecks in this system which prevent hardware subroutines from demonstrating spectacular performance. One is a shortage of registers per FPGA. Another is an excessive amount of time spent multiplexing virtual signals between FPGAs. A third limitation on hardware subroutines, though neglected for the most part in this research, is in the I/O between the host workstation and the Virtual Wires board. Perhaps Virtual Wires are inappropriate for hardware subroutines.

The register shortage is most critical in the north-west FPGA. All inputs and outputs to the hardware subroutine must pass through that first FPGA in order to reach the HC11 serial port controller. Also, a Virtual Wires constraint requires that each of these inputs and outputs have a register allocated on that particular FPGA; thus, for a "16 characters in, 16 characters out" routine, the first FPGA must have 16 chars * 8 bits/char * 2 (input and output) = 256 registers. There are only a few more than that available (the XC4005 has 392 clb flip-flops), and Virtual Wires requires those for bit-shifting. In fact, every shiftgroup is implemented with thirty-two registers, though often only nine are used (wasting twenty-three registers or more).

While virtual wires are well suited for logic emulation, they may not be for certain hardware subroutines. Virtual wires are appropriate for large netlists which require much partitioning and can afford to improve CLB utilization at the cost of performance. Hardware subroutines, on the other hand, have performance (rather than FPGA utilization) as the major goal. If a circuit will fit on eight FPGAs and require only one cycle to transfer data from one partition to the next, it may win over the same circuit partitioned onto only three FPGAs with each data transfer requiring nine clock cycles. Hardware subroutines, though there is no restriction, tend to be simpler and smaller in nature than ASIC designs for logic emulation.
Chapter 7

Conclusions and Future Research

7.1 Conclusions

A hardware subroutine platform has been implemented, and several algorithms have been successfully performed on it. These algorithms include vector and floating-point operations as well as parallel bubble sorts. On the Virtual Wires board, a controller has been configured which allows for an unspecified number of iterations to be performed without extra data transfer over the serial interface. System limitations and performance bottlenecks have been identified, and directions for future work have been specified.

7.2 Future Research

There are a number of investigations which could improve hardware subroutine performance dramatically. These involve locating the bottlenecks and searching for alternatives. It would be a worthwhile experiment to implement a collection of hardware subroutines without the encapsulation of Virtual Wires. It would also be productive to investigate modifications to Virtual Wires. For instance, fewer registers in shift-groups might be wasted, and unnecessary cycles per phase could be avoided. Additionally, algorithms which require many iterations, have implicit parallelism, and have minimum data storage requirements will produce maximum performance. Cur-
rently, a hardware subroutine for the Newton-Raphson Method, which approximates the roots to a given function, is being assembled. It can take an arbitrary number of iterations and minimal I/O.

7.2.1 Automating the Process

Much of the process involved with of hardware subroutines will be automated. The identification of computationally intensive, hardware-suitable subroutines is one such process. The compilation from software to logic gates is another. Eventually the hardware subroutine should be abstracted completely from the engineer. When software is compiled, some of it will be executed as instructions via the microprocessor and other parts may be configured on FPGAs and accessed appropriately. To the end user, the functionality remains unchanged; only the underlying implementation has been modified for increased performance.

7.2.2 Alternative approaches to input and output

Another issue which warrants investigation is that of input/output. The current system has communication between the FPGAs and the microprocessor flowing through a serial port. This kills the performance. As FPGAs gain gate capacities, one FPGA should eventually prove adequate, and it could be placed on the same board as the microprocessor, either sharing a bus or networked. However, other means of communication should be considered as well. For instance, it may be that no arguments need be physically passed at all; perhaps the hardware subroutine and microprocessor could instead share memory, and communication could take place through use of a well-protected domain. Once a robust and convenient memory interface has been developed for the FPGA computing system, shared memory is a possible next step towards increased functionality and performance.

No constraints demand that FPGAs and the microprocessor(s) remain distinct. Integrated reconfigurable logic arrays may soon become a key component in future microprocessors [4]. Once this is achieved, the hardware subroutine will become
commonplace, and it will become advantageous to designate smaller and simpler tasks for the FPGA. As the i/o overhead shrinks, the hardware configuration time becomes shorter, and the programmable hardware becomes faster, this ultimate goal of FPGA computing will approach its potential.

The demonstrated hardware subroutine protocol creates many more opportunities for development and investigation than it satisfies. Improvements will be made to increase performance and flexibility of the system. While many of the improvements excite interest and challenge architectural aspects of computing, the one which would initiate a fundamental change in the future of FPGAs is in the automation. The largest drawbacks with such a system currently lie in the extra design time required to utilize hardware subroutines and the demand on a designer to understand both the software and the logic circuit design aspects of computing. When compilers have assumed this task, the most significant thorn in FPGA computing will have been removed.
Appendix A

Interface Code

A.1 Code for run_board.c

#define NUM_BITS_IN_ELEMENT 8 /* size of an ascii character */
#define not_done(a) (0x01&(a))
#define wen_low(a) ((a) = (a) & 0xFE)
#define wen_high(a) ((a) = (a) | 0x1)
#ifdef debug
#define use_board() \ 
    send_msg(rec_len, send_len, send_buf); \ 
    receive_msg((long) rec_len, send_len, rec_buf); \ 
    display_status(send_buf, send_len, rec_buf, rec_buf, rec_len)
#else
#define use_board() \ 
    send_msg(rec_len, send_len, send_buf); \ 
    receive_msg((long) rec_len, send_len, rec_buf)
#endif

void display_status(unsigned char * send_buf, 
            int send_len, 
            unsigned char * rec_buf, 
            int rec_len);

void run_board(unsigned char * indata, 
        int in_array_size, 
        unsigned char * outdata, 
        int out_array_size) { 
    unsigned char * send_buf; 
    unsigned char * rec_buf;
char rec_len, send_len;
int i;

send_len = (NUM_BITS_IN_ELEMENT * in_array_size + 2 + 7)/8;
/* one extra bit for the vv_reset and one for WriteEnable
   the +7 is to round up */
rec_len = (NUM_BITS_IN_ELEMENT * out_array_size + 1 + 7)/8;
/* one extra bit for the done signal */
send_len++; /* for rec_len */

/* allocate memory */
send_buf = (unsigned char *) malloc(send_len * sizeof(char));
rec_buf = (unsigned char *) malloc(rec_len * sizeof(char));
send_buf[0] = (unsigned char) rec_len;

/* fill data array ; really just reversing the order */
for(i = 1; i <= in_array_size; i++)
    send_buf[i] = indata[in_array_size-i];
/* The low bit is WEN, which is doesn't matter (see next line) */

wen_low(send_buf[send_len-1]);
send_buf[in_array_size+1] = 0;
use_board();
use_board();
wen_high(send_buf[send_len-1]);
use_board();

/* reverse the order; the offset of 1 is due to the done bit */
for(i = 1; i < out_array_size+1; i++)
    *outdata++ = *(++rec_buf);
}

void display_status(unsigned char * send_buf,
    int send_len,
    unsigned char * rec_buf,
    int rec_len) {
    int i;

    printf("Sent:\n");
    for(i=1; i<send_len-1; i++)
        printf("...%x.. ",(send_buf[i]&0x000ff));
    printf("\n-------\n");
    printf("Recieved:\n");
    for(i=0; i<rec_len; i++)
printf("...%x... ", rec_buf[i]);
printf("\nwen \t= %d\n", send_buf[send_len-1]&1);
printf("#onen \t= %d\n", not_done(rec_buf[0]));
printf("-------------
");
/* To be called only once, before run_board */

void reset_board(int in_array_size, int out_array_size) {
    unsigned char * send_buf;
    unsigned char * rec_buf;
    char rec_len, send_len;
    int i;

    /* establish contact with the port designated in interface.ctl */
    initial_talk();

    /* run_board explains these next 3 lines */
    send_len = (NUM_BITS_IN_ELEMENT * in_array_size + 2 + 7)/8;
    rec_len = (NUM_BITS_IN_ELEMENT * out_array_size + 1 + 7)/8;
    send_len++;

    /* allocate memory */
    send_buf = (unsigned char *) malloc(send_len * sizeof(char));
    rec_buf = (unsigned char *) malloc(rec_len * sizeof(char));
    send_buf[0] = (unsigned char) rec_len;

    /* Reset VW */
    for(i = 1; i < send_len; i++)
        send_buf[i] = 0xff;
    use_board();
    /* Done with reset */
}
Appendix B

Example Bubble Sort Routine

B.1 Shell for Routine

/* Shell for calling bubble sort routine, whether hardware or software */

#define NUM_ELEMENTS 8
#define INPUT_FILE "unsorted.dat"
#define OUTPUT_FILE "sorted.dat"
#define USE_HARDWARE_SUBROUTINE

void bsort1(unsigned char * data, int array_size);

void main() {
    unsigned char indata[NUM_ELEMENTS];
    unsigned char outdata[NUM_ELEMENTS];
    FILE * input_stream = fopen(INPUT_FILE, "r");
    FILE * output_stream = fopen(OUTPUT_FILE, "w");
    int i;

    /* read array of data from file */
    fread(indata, sizeof(char), NUM_ELEMENTS, input_stream);

    #ifdef USE_HARDWARE_SUBROUTINE
    /* sort data, using the parallel sort hardware subroutine */
    reset_board(NUM_ELEMENTS, NUM_ELEMENTS);
    run_board(indata, NUM_ELEMENTS, outdata, NUM_ELEMENTS);
    #else
    bsort(indata, NUM_ELEMENTS); outdata = indata;
    #endif

    /* write array of data to file */
    fwrite(outdata, sizeof(char), NUM_ELEMENTS, output_stream);
    fclose(input_stream);
    fclose(output_stream);
    return 0;
}

41
/* write sorted array out to a file */
fwrite(outdata, sizeof(char), NUM_ELEMENTS, output_stream);

/* tie up lose ends */
fclose(input_stream);
fclose(output_stream);
}

/end{verbatim}
/section{Bubble Sort Software Subroutine}
/begin{verbatim}

/* Bubble Sort algorithm in C, when not using hardware subroutine */

#define YES 1
#define NO 0
#define swap(a,b,c) (c)=(a);(a)=(b);(b)=(c)

void bsort1(unsigned char * data, int array_size) {
    int i;
    int done = NO;
    int changes = YES;
    int offset = 0;
    unsigned char tmp;

    while(!done) {
        done = !changes;
        changes = NO; /* No swaps were made yet this iteration */
        for(i = offset%2; i < array_size - 1; i += 2) {
            if(data[i] > data[i+1]) {
                swap(data[i], data[i+1], tmp);
                changes = YES;
            }
        }
        /* really done if no changes were made this iteration or last */
        done = done && !changes;
        offset++;
    }
}
/sectionBubble Sort Hardware Subroutine

/* Hardware subroutine written in verilog and compiled to gates
to be run on the virtual wires board*/

module bsort(weN, datain0, datain1, datain2, datain3, chngs, changes1,
            changes2, state, doneN, data0, data1, data2, data3, data4, data5,
data6, data7, clk);

input [15:0] datain0, datain1, datain2, datain3;
input weN;
input clk;

output [7:0] data0, data1, data2, data3, data4, data5, data6, data7;
output doneN;
output [1:0] state;
output changes1, changes2;
output [3:0] chngs;

// state names
parameter WRITE=0, SORT1=1, SORT2=2, DONE=3;

reg [7:0] data0, data1, data2, data3, data4, data5, data6, data7;
wire doneN;
reg changes1, changes2;
reg [3:0] chngs;
reg [1:0] state, next_state;

// doneN is active (low) if we are in the DONE state
nand u1 (doneN, state[1], state[0]);

always @(posedge clk )
begin
  // if weN is active (low) state = WRITE.
  state = next_state & {weN, weN};
  case (state)
    WRITE: begin
      // start state: set up for write
      data0 = datain0[15:8]; data1 = datain0[7:0];
data2 = datain1[15:8]; data3 = datain1[7:0];
data4 = datain2[15:8]; data5 = datain2[7:0];
data6 = datain3[15:8]; data7 = datain3[7:0];
      // initialize change signals
changes1 = 1;
changes2 = 1;
chngs[3:0] = 4'b1111;
if (!weN)
    next_state = WRITE;
else
    next_state = SORT1;
end
// SORT1 corresponds to an offset of 0.
SORT1: begin
    sort(data0, data1, chngs[0]);
    sort(data2, data3, chngs[1]);
    sort(data4, data5, chngs[2]);
    sort(data6, data7, chngs[3]);
    if (changes2 | changes1)
        next_state = SORT2;
    else
        next_state = DONE;
end
// SORT2 corresponds to an offset of 1.
SORT2: begin
    sort(data1, data2, chngs[0]);
    sort(data3, data4, chngs[1]);
    sort(data5, data6, chngs[2]);
    sort(data0, data7, chngs[3]);
    if (changes2 | changes1)
        next_state = SORT1;
    else
        next_state = DONE;
end
DONE: begin
    if (!weN) // This is not really needed because
        // state = nextstate & weN
        next_state = WRITE;
    else
        next_state = DONE;
end
endcase
end

task sort;
inout [7:0] sort_data1;
inout [7:0] sort_data2;
inout chng_bit;

if(sort_data1 > sort_data2) begin
  chng_bit = 1;
  {sort_data1, sort_data2} = {sort_data2, sort_data1};
end
else begin
  chng_bit = 0;
  {sort_data1, sort_data2} = {sort_data1, sort_data2};
end
endtask

endmodule
Appendix C

Three Element Cross Product

C.1 Shell For Routine

#include "./vmult3.h"

#define NUM_ELEMENTS 3
#define swap(a,b,c) (c)=(a);(a)=(b);(b)=(c)
#define USE_HARDWARE_SUBROUTINE

void vmult3(unsigned char * indata,
        int in_array_size,
        unsigned char * outdata,
        int out_array_size);

void main()
{
char indata[NUM_ELEMENTS*2];
char outdata[NUM_ELEMENTS];
int i=0;
int j;

while(i < NUM_ELEMENTS*2) indata[i++] = i;

i = 0;
while(i < NUM_ELEMENTS*2)
        printf("%d ", indata[i++]);
        printf("\n");

#ifdef USE_HARDWARE_SUBROUTINE
run_board(indata, NUM_ELEMENTS*2, outdata, NUM_ELEMENTS);

```
#else
    vmult3(indata, NUM_ELEMENTS*2, outdata, NUM_ELEMENTS);
#endif

/* output data to screen */
    i = 0;
    while(i < NUM_ELEMENTS)
        printf("%d ",outdata[i++]);
        printf("\n");
}

C.2 Cross-Product Software Subroutine

void vmult3(unsigned char * indata,
           int in_array_size,
           unsigned char * outdata,
           int out_array_size) {

    char a,b,c,x,y,z;
    x = *indata++;
    y = *indata++;
    z = *indata++;
    a = *indata++;
    b = *indata++;
    c = *indata;

    *outdata++ = y*c-z*b;
    *outdata++ = z*a-x*c;
    *outdata = x*b-y*a;
}

47
C.3 Cross-Product Hardware Subroutine

module test(datain0, datain1, datain2, weN, data0, data1, data2, doneN, clk);

input [15:0] datain0;
input [15:0] datain1;
input [15:0] datain2;
input weN;
input clk;

output [7:0] data0;
output [7:0] data1;
output [7:0] data2;
output doneN;

reg [7:0] data0;
reg [7:0] data1;
reg [7:0] data2;
reg doneN;

always @(posedge clk)
begin
  if (!weN) begin
    data0 = (datain0[7:0] * datain2[7:0] - datain1[15:8] * datain2[15:8]);
    data1 = (datain1[15:8] * datain1[7:0] - datain0[15:8] * datain2[7:0]);
    data2 = (datain0[15:8] * datain2[15:8] - datain0[7:0] * datain1[7:0]);
    doneN = 0; // Note: This subroutine requires NO iterations.
  end
end

endmodule
Appendix D

Eight Element Inner Product

D.1 Shell For Routine

#include "./bsortc.h"

#define NUM_ELEMENTS 16
#define INPUT_FILE "input.dat"
#define OUTPUT_FILE "output.dat"
#define USE_HARDWARE_SUBROUTINE

void innerp(unsigned char * data, int array_size);

void main() {
    unsigned char indata[NUM_ELEMENTS];
    unsigned char outdata[NUM_ELEMENTS];
    FILE * input_stream = fopen(INPUT_FILE, "r");
    FILE * output_stream = fopen(OUTPUT_FILE, "w");
    int i, tmp;

    /* read array of data from file */
    for(i = 0; i < NUM_ELEMENTS; i++) {
        fscanf(input_stream, "%d", &tmp); indata[i] = (unsigned char) tmp;
    }

    #ifdef USE_HARDWARE_SUBROUTINE
    reset_board(NUM_ELEMENTS, 1);
    run_board(indata, NUM_ELEMENTS, outdata, 1);
    #else
    innerp(indata, NUM_ELEMENTS);
    #endif
}
outdata[1] = indata[0];
#endif

/* write sorted array out to a file */
printf("The answer is \%d.\n", outdata[1]);

/* tie up lose ends */
fclose(input_stream);
fclose(output_stream);
}

D.2 Inner-Product Software Subroutine

void innerp(unsigned char * data, int array_size) {
    data[0]=(data[0] * data[8] +
data[1] * data[9] +
data[6] * data[14] +
data[7] * data[15]);
}

D.3 Inner-Product Hardware Subroutine

module test (weN, dataina0, dataina1, dataina2, dataina3, dataina4, dataina5, dataina6, dataina7, datainb0, datainb1, datainb2, datainb3, datainb4, datainb5, datainb6, datainb7, doneN, dataout, clk);

input [7:0] dataina0, dataina1, dataina2, dataina3, dataina4, dataina5;
input [7:0] dataina6, dataina7, datainb0, datainb1, datainb2, datainb3;
input [7:0] datainb4, datainb5, datainb6, datainb7;
input weN;
input clk;

output [7:0] dataout;
output doneN;

reg [7:0] tmp0;
reg [7:0] dataout;
reg doneN;

always @(posedge clk) begin
  if(!weN) begin
    tmp0 = ((dataina0 * datainb0 +
             dataina1 * datainb1) +
             (dataina2 * datainb2 +
             dataina3 * datainb3)) +
             ((dataina4 * datainb4 +
               dataina5 * datainb5) +
              (dataina6 * datainb6 +
               dataina7 * datainb7));
    doneN = 1;
  end
  else begin
    doneN = 0;
    dataout = tmp0;
  end
end
endmodule
Appendix E

32-bit Floating-Point Multiplier in Verilog

module test(weN, datain0, datain1, doneN, dataout, clk);

input [31:0] datain0, datain1;
input  weN, clk;

output  [31:0] dataout;
output   doneN;

reg  [31:0] dataout;
reg   doneN;
reg [47:0] tmp;
wire sign0, sign1, signout;
wire    [7:0] oldexpout;
wires zeroN, zero1N, zero0N;

assign sign0 = datain0[31];
assign sign1 = datain1[31];

xor u1 (signout, sign0, sign1);
and  u2 (zeroN, zero0N, zero1N);
or u3 (zero0N, datain0[30], datain0[29], datain0[28], datain0[27],
datain0[26], datain0[25], datain0[24], datain0[23]);
or u4 (zero1N, datain1[30], datain1[29], datain1[28], datain1[27],
datain1[26], datain1[25], datain1[24], datain1[23]);

always @(posedge clk)
begin
if (!weN) begin /* multiply mantissas */
doneN = 1;
tmp[47:0] = ({1'h1, datain0[22:0]} * {1'h1, datain1[22:0]});
dataout[30:23] = (datain0[30:23] + datain1[30:23] - 8'h7e);
dataout[31] = signout & zeroN;
end
else if (!tmp[47] & zeroN) begin /* normalize result */
doneN = 1;
tmp[47:0] = {tmp[46:0], 1'h1};
end
else if (!zeroN) begin /* in case we are multiplying by zero */
dataout[30:23] = 8'h00;
dataout[22:0] = 0;
doneN = 0;
end
else begin /* Done! */
doneN = 0;
dataout[22:0] = tmp[46:24];
end
end
endmodule
Bibliography


