This is to certify that the

thesis entitled

A VERY LONG INSTRUCTION WORD ARCHITECTURE
IMPLEMENTED ON THE SPLASH 2 FPGA ARRAY

presented by

Roy C. Wang

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng.

Diane J Rover
Major professor

Date 3/27/96

A Very Long Instruction Word Architecture Implemented on the Splash 2 FPGA Array

By

Roy C. Wang

A THESIS

Submitted to

Michigan State University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1995

ABSTRACT

A Very Long Instruction Word Architecture Implemented on the Splash 2 FPGA Array

By

Roy C Wang

The need for faster computing systems constantly pushes the limits of technology. While special purpose and custom computing systems can provide very high levels of performance, the cost associated with these systems limits their application. Recently, there has been significant improvements in the performance of general purpose processors which can be used in a wide variety of applications. The speed of these general purpose processors is limited by how many instructions they can perform per clock cycle. To increase that number, machines need to be able to execute instructions in parallel. Machines capable of such parallel execution are known as superscalar processors. Unlike other proposed superscalar models, the VLIW (Very Long Instruction Word) model extends the RISC paradigm of simplified hardware.

This thesis examines the feasibility of using Splash 2, an FPGA-based processing array, configured as a VLIW processor. The VLIW architecture implemented in this study is capable of performing two operations concurrently. This study demonstrates the use of a new platform to prototype and test architectural and compilation theories and highlights its capabilities and limitations.

Dedicated to my parents,

Maria Wang and Tung S. Wang,

whose sacrifice and perseverance has given me a better life.

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Dr. Diane Thiede Rover, for all of her help and guidance in the course of this research. Her insight provided much needed direction and encouragement.

I would also like to thank the other members of my committee, Dr. Elias Strangas and Dr. Michael Shanblatt for their efforts on my behalf. Dr. Strangas has provided support throughout my graduate education. His generosity and faith proved to be an invaluable source of strength and support.

I wish to thank my family and friends for always being there for me. Last, but hardly least, I would also like to thank Lisa for her editing efforts and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

New technology invariably alters the landscape of design and implementation of computer systems. Current trends of processor design have moved towards a RISC (Reduced Instruction Set Computer) architecture. Lower cost memory, automated design, and the need for a modular approach due to VLSI (Very Large Scale Integration) and WSI (Wafer Scale Integration) have all played a significant role in this shift of ideology. As we push the upper limits of performance with pipelining, it becomes inevitable to exploit the fine grained parallelism of programs. This is known as instruction level parallelism (ILP). Superscalar processors, which can execute multiple instructions concurrently, have already been designed and implemented to take advantage of existing software. They promise even more speed-up in the future as compiler technologies and operating systems are written to take advantage of their added capabilities.

Processor design is not only changing, but changing rapidly. The development of computer technology is one of the fastest moving industries in the world. Introducing a new product a month or two before competitors can capture a majority of the market share. With this motivation, it is not difficult to see why shrinking the design cycle is imperative. FPGA (Field Programmable Gate Array) technology is increasing in both array size and speed. These chips offer a short development cycle by implementing and testing designs without the need for expensive wafer fabrication. Individually, they are

used today for ASIC (Application Specific Integrated Circuit) and small prototyping applications, however, larger ensembles offer new capabilities in rapid prototyping and custom computing. Splash 2 is one of the first computing systems to explore these possibilities. Its design allows for SIMD (Single Instruction stream, Multiple Data stream) type architectures as well as highly pipelined and systolic applications.

This thesis explores the issues associated with implementing an instruction set processor with the FPGA-based system, Splash 2. The Spyder architecture [17] has shown that it is possible to utilize FPGAs in a general purpose processor. The Spyder, however, dedicates the FPGAs as reprogrammable execution units to provide some flexibility in its functionality. Splash 2 and other FPGA arrays represent a totally new class of machines. The size and resources of the Splash 2 allows an entire architecture to be prototyped and implemented with FPGAs. This would allow the computer designer to test new features of an ISP (Instruction Set Processor) by reconfiguring the Splash 2. For example, programs that may benefit from very specific hardware routines can reprogram the data path for better performance, and still retain the functionality of the remaining system.

FPGA technology, however, is still in a relatively young stage. The limitations imposed by the Splash 2 constrain designs. The limited logic on a single FPGA forces us to partition our design across several FPGAs. The architecture of the Splash 2 also limits the off chip resources available to each FPGA, such as memory and busses. These restrictions present a significant obstacle to implementing a general purpose instruction set processor.

Our approach to studying ILP and superscalar processors is to prototype a VLIW (Very Long Instruction Word) processor on the Splash 2. The simple hardware needed to implement a VLIW architecture is well suited for the Splash 2. A VLIW architecture will also exploit the fine grained parallelism of programs that conventional processors have not. The problems associated with parallel processing motivate the need for a platform

that can rapidly prototype and test new architectural and compilation theories. While conventional computing machines are built to process either general purpose code or highly customized instructions, FPGA-based arrays offer the true general purpose machine that can be configured as both.

New architectures are developed from studies of past processors and applications. In chapter 2, we will discuss some of the issues and approaches of implementing instruction set processors. These issues lay the foundation to our new architecture, the Dex-II. Chapter 3 provides a detailed description of our design methodology and implementation strategies. It also provides an architectural specification for Dex-II, the VLIW architecture implemented on the Splash 2 array. Chapter 4 presents the design environment and provides the simulation results and synthesis statistics of our implementation. In chapter 5, an analysis of two test programs will help us evaluate how successful the VLIW architecture extracts ILP. Chapter 6 will summarize the findings and discuss how future implementation of FPGA arrays may be improved.

CHAPTER 2

Background Information

This chapter provides the background information and motivation for this thesis. It presents a brief history of architectures and the concepts applied to improve their performance. We discuss the challenges of parallel computing as well as introduce the concept of instruction level parallelism. Finally, we consider reconfigurable systems and how they can contribute to innovations in computer architectures.

2.1 Scalar Processors

First generation computers were typically load/store machines. They were basically single accumulator processors not unlike todays programmable calculators. If multiple variables were needed in the computation, they had to be stored into memory and loaded at a later time. Memory address calculations and access were time consuming. Therefore, more complex methods of manipulating data gave birth to the first CISC (Complex Instruction Set Computer) machines. These machines are characterized by a larger set of registers and complex memory addressing modes [15]. For example, the Motorola 68040 microprocessor implements 113 instructions with 16 general purpose registers and supports 18 different addressing modes. The popular Intel 80486 microprocessor implements 157 instructions with 12 addressing modes. The concept of the CISC is to put commonly executed sequences of code into one single instruction and add special hardware to execute it as quickly as possible. Each instruction

is performed by executing subprograms written in microcode. Microcode represents the physical bits of signals that control the data path. The advantage to this design is the compatibility of code between machines and the reduced cost for program memory. If the hardware is changed, all that was needed to compensate is a rewrite of the microcode to support the same instructions. Applications are able to run without the need to recompile. These processors, however, have become very elaborate and complex to design. Transistor counts have rocketed into the millions, and the speed-up of each successive generation has dwindled.

The search for more speed led to techniques used in mainframes and supercomputers. Pipelining [12] was introduced to exploit temporal parallelism. Pipelining allows instructions to be overlapped in stages. Each stage completes part of the instruction so several instructions can be executing at the same time. To make pipelining efficient, each stage should take about the same time to complete its tasks. CISC techniques are not very well adapted to handle pipelining. Each individual instruction has its own time frame and data paths are littered with special hardware to implement those special instructions. RISC processors, however, are designed to balance these stages. RISC instructions use very basic functions to simplify the hardware as much as possible. The functionality of the processor is preserved, but the number of instructions needed to complete the same task was increased. The increase in memory size required to run programs have become less of an issue as memory sizes were quadrupling about every three years [12].

The RISC machine is characterized by a smaller set of instructions and a larger supply of registers. Addressing is kept as simple as possible and memory operations are usually limited to simple load/store commands. A typical RISC machine is the Sun SPARC CY7C601 with 69 instructions. There are 136 registers divided with a technique called *register windowing*. This is basically a paged register file to supply clean registers for subroutines. The RISC model tries to partition instructions into well-defined stages to

make pipelining as efficient as possible. The results are higher clock rates and faster design times. Chips such as the DEC Alpha and the PowerPC have already broken the 100MHz mark. Now the search for even more speed has begun to focus on spatial parallelism. Increasing the number of instructions that can execute at the same time brings us into the realm of parallel and superscalar computing.

2.2 Parallel vs. Superscalar Processors

The idea of parallel and superscalar processors is to allow instructions to be executed concurrently [18]. This is not a new idea, and there have been many successful approaches to parallel computing. Parallel processors can execute several instruction streams, while a superscalar processor can execute several instructions of a single instruction stream. Parallel execution of instructions increases the throughput of the

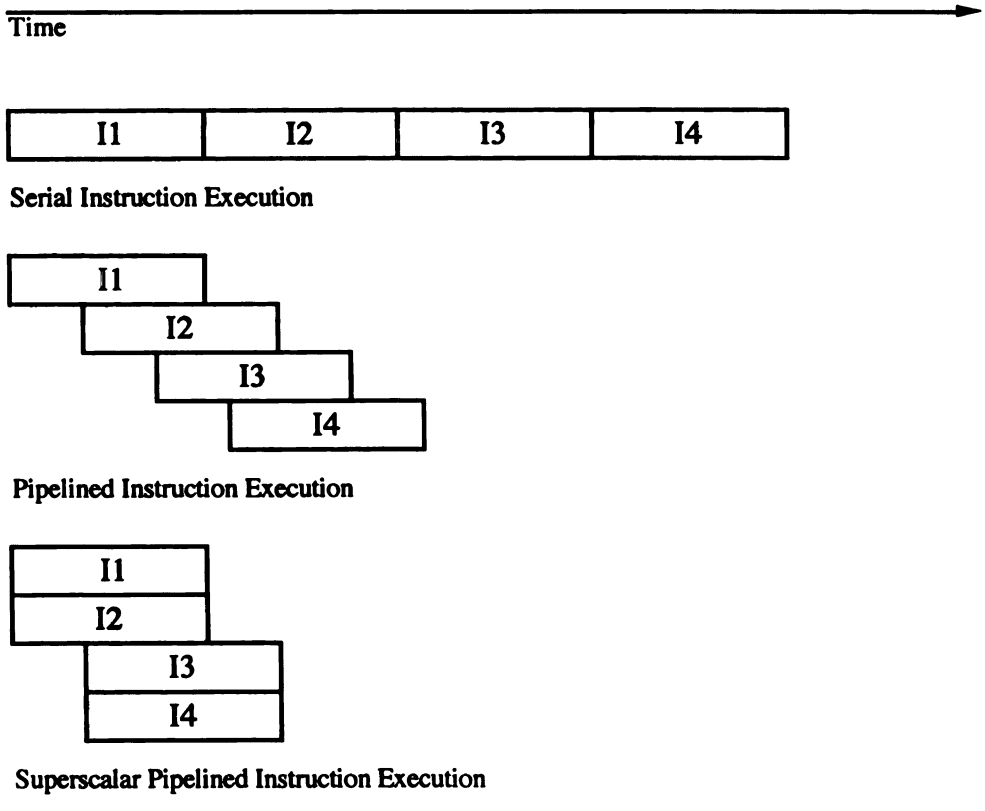


Figure 2.1. Instruction execution models

processor and decreases the time needed to finish a task as can be seen in Figure 2.1. This figure depicts the relative time needed to finish four tasks (I1-I4) and shows how pipelined and superscalar execution times compare to serial execution. Many of the traditional parallel processors need highly specialized hardware and special algorithms to achieve their goals. The Flynn classification [12] proposes four different models of computers. The SISD (Single Instruction stream, Single Data stream) architecture is the standard general purpose scalar computer. The SIMD architecture is represented by machines such as the MasPar MP-1 and Thinking Machines CM-2. This type of architecture has many identical functional units and a single instruction unit. The instruction unit broadcasts a command, and all the functional units perform the same function. This is extremely efficient for many different matrix operations where the same task is performed on large amounts of data. Unfortunately, only very specific applications can take full advantage of this setup. The MISD (Multiple Instruction stream, Single Data stream) computers are a bit more difficult to classify. Certain systolic arrays may be considered as MISD computers. The data flows through stages of the array and is operated on by whatever instruction executes at each stage of the array [7]. The Multiple Instruction, Multiple Data stream MIMD processors are the most common parallel computers. Mainframes, supercomputers, and high-end workstations all employ some form of multiprocessing capability. Employing the MIMD model, they can exploit data parallelism and specialized parallel code to expedite the completion of a single program. However, these computers are primarily used to serve many users and programs simultaneously. While they can complete a multitude of tasks, each individual program or task itself is not being executed any faster [12]. To complete a single task using multiple processors requires either a regular data structure, such as matrix calculations, or complex synchronization and message passing mechanisms.

Several issues have been studied to overcome the barriers of parallel processing. Parallel programming requires a good understanding of how the hardware is organized in

order to take full advantage of the scalable hardware. Variations of high level programming languages have been used to overcome some of these issues. A language called dbC (Data-parallel Bit C) was developed with parallel data structures for SIMD type execution. By using the new constructs supplied by the language, the compiler can effectively distribute the computation among the available processing elements. dbC has been used for parallel machines such as the Cray and Terasys. Even more interesting is the possibility of using a dbC compiler to generate hardware on an FPGA-based array to implement each program with its own hardware [8][9].

Parallel machines also suffer from synchronization problems. There are basically two methods of sharing data and keeping data coherent. The message passing model uses a distributed memory system. Each processor keeps its own data and sends data via messages when needed. The MasPar MP-1 and Ncube2 are two examples of a message passing computer. The other method is a shared memory model where each processor communicates through a common memory space. The shared memory space makes programming much easier since the programmer does not need to worry about the location of data. Although the shared memory model is theoretically sound, the scalability of such a system is limited by the bandwidth of the memory and the complexity of the caching scheme [30] used in the system. The DASH multiprocessor [19] developed at Stanford uses a distributed directory based cache coherence scheme [19][13] to scale the single memory space. However, these schemes need both software and hardware support in order to be utilized. Still, others argue that a highly scalable memory system is not necessary due to the inherently low level of parallelism that can be extracted from programs [12].

2.3 Instruction Level Parallelism

It becomes quite obvious from Figure 2.1 that the number of instructions that can be executed concurrently will greatly affect the performance of superscalar architectures.

This number is limited by the ILP of the program. ILP is the amount of parallelism that can be extracted from a program written for a sequential processor [5]. This is limited by true data dependencies, procedural dependencies, and resource conflicts [18]. Data dependencies are instructions whose operands depend on the results of previous instructions. Instructions in general cannot execute until their operands are available. Several methods to resolve data dependencies have been used. The Intel i960CA, HP PA-RISC7100, HyperSPARC, and IBM's RS6000 all use a runtime technique known as scoreboarding to resolve data dependency [15][12][28][33]. This technique requires keeping a table or "scoreboard" of what stage of execution each register is in. Other dynamic techniques use reservation stations and a version of the Tomasulo algorithm [28] implemented on the IBM 360 floating point unit. The major disadvantages of these implementations is the hardware cost and the complexity associated with them. Dependencies can also be eliminated during compile time [5]. VLIW processors depend on the compiler to generate compact code in order to exploit ILP. The biggest disadvantage to this approach is the lack of good compilers and the incompatibility of code from machine to machine.

Procedural dependencies are instructions that are affected by branching. At conditional branches, there are two sets of instructions that can be executed. The processor often has to await resolution of the branch to continue execution. Several different approaches have been used to decrease the impact of branching. Speculative execution [20][12] can be used to eliminate some of these branch latencies. Compiler techniques utilizing trace scheduling methods [4] can also move instructions across blocks of code to fill up unused cycles.

Resource dependency involves the available number of functional units to execute instructions on. The decision of how many functional units to implement depends heavily on the type of programs being run and the level of parallelism that can be

extracted. Too many units would leave costly hardware idle, while too few would create a bottleneck for the performance of the processor.

2.4 Design Issues of Superscalar Processors

The physical hardware where operations are performed are called functional units. In a scalar architecture, only one function can be executed at any given time. In a superscalar architecture, functions are differentiated in hardware to allow multiple instruction execution among all of them. The most common functional units are integer units, floating point units, memory units, and control units. The number and types of units vary from architecture to architecture. The Motorola 88110, for example, has ten functional units while the DEC Alpha has only four [33]. More units, however, do not necessarily indicate better performance.

There are typically four stages in executing an instruction as shown in Figure 2.2: Fetch, Decode, Execute, and Writeback [12]. The Execution stage can be easily scaled with the addition of hardware. The Fetch, Decode and Writeback stages are the critical points where bottlenecks can impede performance gains. The Fetch stage is responsible for getting instructions out of the instruction cache. In a superscalar processor where more than one instruction can be issued, this becomes a challenging task. Different architectures adopt many different approaches to this problem. The number of functional

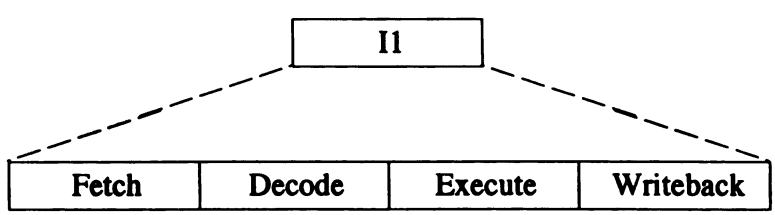


Figure 2.2. Instruction stages

units and the dependencies on the instruction dictate how many and which instructions can begin to execute. For instance, the DEC Alpha can issue up to two instructions per

clock cycle as long as they are operating on different functional units [33]. The Intel i960CA can fetch four instructions and issue up to three of them if they are on separate functional units [33]. The sophistication of issuing instructions greatly affects the ability of keeping the hardware busy. Therefore, having a multitude of functional units does not automatically make a better system. It becomes apparent that the order of instructions plays a critical role in exploiting the maximum level of parallelism.

Instructions can be *issued* in-order or out-of-order [18]. All of the architectures discussed issue instructions in-order of the program. Since there are a limited number of functional units, certain combinations of operations cannot be issued together. Two integer operations, for example, cannot be issued on the same clock cycle if there is only one integer unit. In order to execute correctly, a stall or NOP (No Operation) must be injected into the code. Out-of-order issues can eliminate this and use up the wasted stall time. This is achieved by using an instruction window to look ahead at several instructions to determine if there are any data dependencies. Unfortunately, the complexity involved in the hardware has deterred an implementation of this scheme. Instructions can also be completed in-order or out-of-order [18]. The latencies of floating point units tend to be much longer than those of integer units. With in-order completion, instructions may be needlessly stalled while one very long instruction is executing. With out-of-order completion, this can be avoided as long as data dependencies are observed. Some of the problems with out-of-order completion are the added hardware for dependency checks and the difficulties dealing with precise interrupts and exceptions. Since instructions may be completed out of order, the restart point may cause the instruction to be incorrectly executed twice. Again, extra hardware is required to support precise interrupts.

The dynamic techniques of achieving parallel instruction execution all require extra hardware. A VLIW processor tries to adhere to the RISC philosophy. Instead of adding complex hardware schemes to resolve the problems of multiple instruction issues

during runtime, VLIW processors resolve dependencies during compile time. The instruction word contains all the information for each functional unit. This makes for very long instructions as more and more functional units are added. By scheduling and analyzing programs off-line, we can get rid of the hardware overhead and ultimately achieve better performance than a dynamic solution. This approach also allows for very simple instruction issuing and decoding. The Multiflow TRACE series were the first commercially available processors implementing the VLIW architecture [29]. Unfortunately, the performance that was achieved was severely limited by the compiler technology. It was unable to extract enough ILP at the time [29]. There are basically two methods to extract ILP for a VLIW architecture: code movement and guarded execution. Code movement is much easier and cheaper to implement than guarded execution. Guarded execution requires the addition of hardware registers and additional logic along with semaphore semantics in software. The basic idea is to be able to execute speculatively along both branches when a conditional branch is encountered and to invalidate the path not taken. This method usually requires many shadow registers and memories to be effectively implemented.

2.5 Code Motion

Code motion, otherwise known as *list scheduling*, is an effective method to expose ILP. A human compiler can write very efficient assembly level code, but today's compilers are still not as smart as we would like them to be. The two most popular forms of list scheduling are known as *percolation scheduling* [22], and *trace scheduling* [4]. Early efforts such as the Bulldog [3] compiler and the Multiflow TRACE [28] implement these techniques with varying degrees of success.

Percolation gets its name from trying to "percolate" instructions up as far as possible to fill in wasted NOP space. There are several key points that must be observed when moving code around. The primary challenge is keeping the semantics the same.

Examples of code will be in the format as follows

```
I1:     OP1          * Comments and explanations
        OP2

I2:     OP1
```

where I1 is a single instruction consisting of two concurrent operations, and I2 is a single instruction with only one operation.

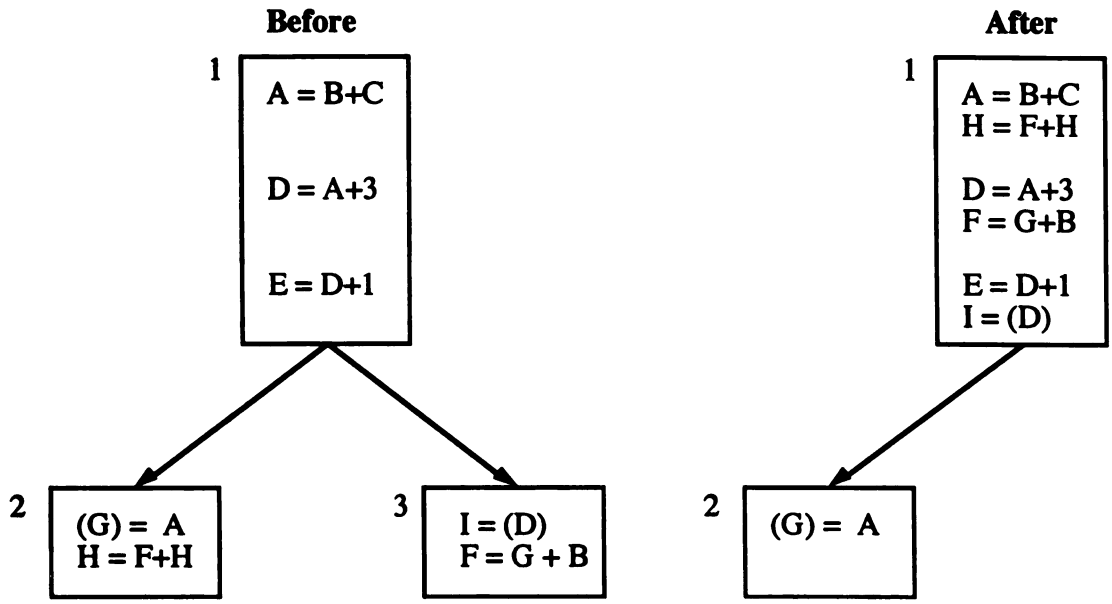


Figure 2.3. Percolation scheduling

From the example in Figure 2.3, we assume that there are only three basic blocks of code (1-3). If there were successors following blocks 2 and 3, we would also have to take those into account. The key to preserving the semantic meaning is to avoid overwriting registers that are "live" and memory locations. This restriction on memory basically dictates that store instructions cannot be moved. Ensuring that executing instructions do not affect live registers can be achieved in several ways. We can use register renaming to replace variables or to compute speculated variables. In Figure 2.4, we could not move

the instructions in block 3 without affecting the code in block 2. This is due to the two variables, I and F, in block 3 that are also used in block 2. If we rename the I and F variables in block 3 to H and J, we see that it is still possible to move the code from block 3 into block 1. This is an inexpensive and powerful performance booster.

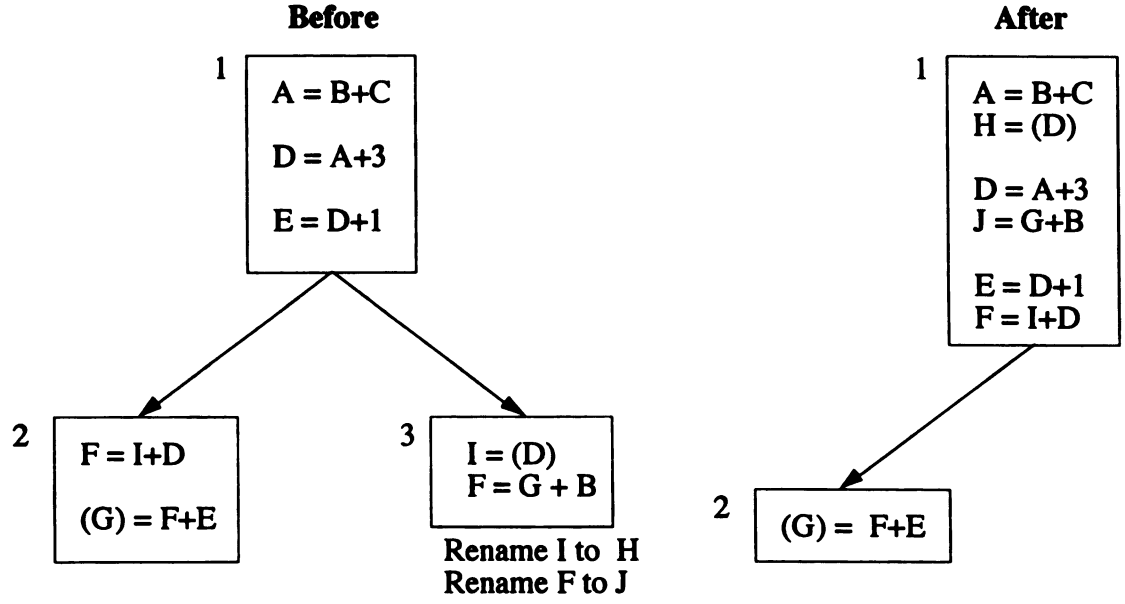


Figure 2.4. Register renaming

Another semantic preserving method is adding compensation code. If we restrict our movements to instructions that can be undone, we can shorten one path possibly at the expense of another. This is the idea behind trace scheduling [4]. If we have a good profile of how programs normally operate, then by reducing the path taken most often, the added cost of compensation code is negligible. In Figure 2.5, we reduced the left path from four to three instructions at the expense of the right path which went from four to five instructions. If we assume this is a loop with 100 iterations, then the total number of cycles needed to execute this program would be 90*3 + 10*5 = 320 cycles. The original code would have taken 90*4 + 10*4 = 400 cycles. This is a substantial improvement for

our very small example. The difficulty in such a scheme, however, is to have an accurate runtime profile of programs. If for some reason, the program switched over to executing the right path ninety percent of the time, we would experience a penalty rather than the intended performance boost.

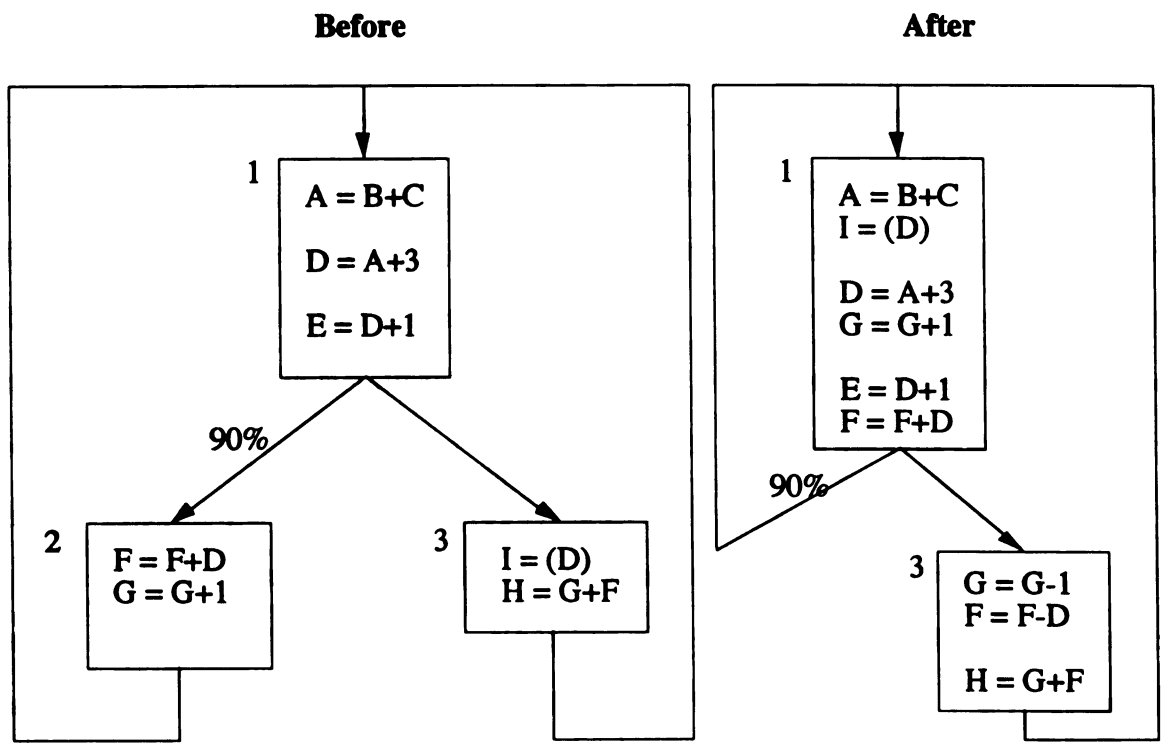


Figure 2.5. Compensation code

The other major drawback of rescheduling is exception handling. Note that in block 3 of Figure 2.5, there is a memory read into variable I. On a real system, this may cause a page fault and require the system to stop and load up a new page of memory. The increased memory reference can cause the system to start thrashing if it is scheduled up to block 1. This further limits our ability to move code around. A technique called *sentinel scheduling* [20] combines the ideas of guarded execution with list scheduling to avoid these problems.

2.6 Reconfigurable Systems

The parameters and problems associated with parallel and superscalar computing present many possible solutions. Reconfigurable systems allow the designer to implement and test various designs quickly and efficiently.

A VLIW architecture called the VIPER [10] claims a 100 MIPS peak throughput. At 25 MHz, it is capable of performing a branch, two load/store operations, and up to four ALU operations per clock cycle. A study done with the VIPER on pipelining and bypassing [1] also showed that the added cost of a fully interconnected network for operand forwarding is significant. The additional bus lines increased cycle time and silicon area. This analysis shows that the dynamics of large systems can vary widely with different implementations and applications.

By using a reconfigurable system, we can study a greater number of applications to find the best solution. A reconfigurable system gives us the ability to tailor an architecture through software. This allows the designer to optimize the amount of hardware for each application. Architectures that would benefit from additional forwarding paths, for example, could be implemented on the same system as an architecture whose main purpose is to get the fastest cycle time. Reconfigurable systems can provide computer designers with important data on the impact of architectural changes and features.

The Splash 2 is a reconfigurable system composed of 17 FPGAs. The Field Programmable Gate Array is a matrix of programmable cells called CLBs (Configurable Logic Blocks) [34]. The Xilinx 4000 series uses two function generators per CLB that can accept up to four inputs and perform any Boolean function on them. The functions are implemented with look-up tables, so the delay associated with each generator is

independent of the function being implemented. A simplified block diagram [34] of the CLB is shown in Figure 2.6. The multiplexer controls and logic functions are determined by the configuration.

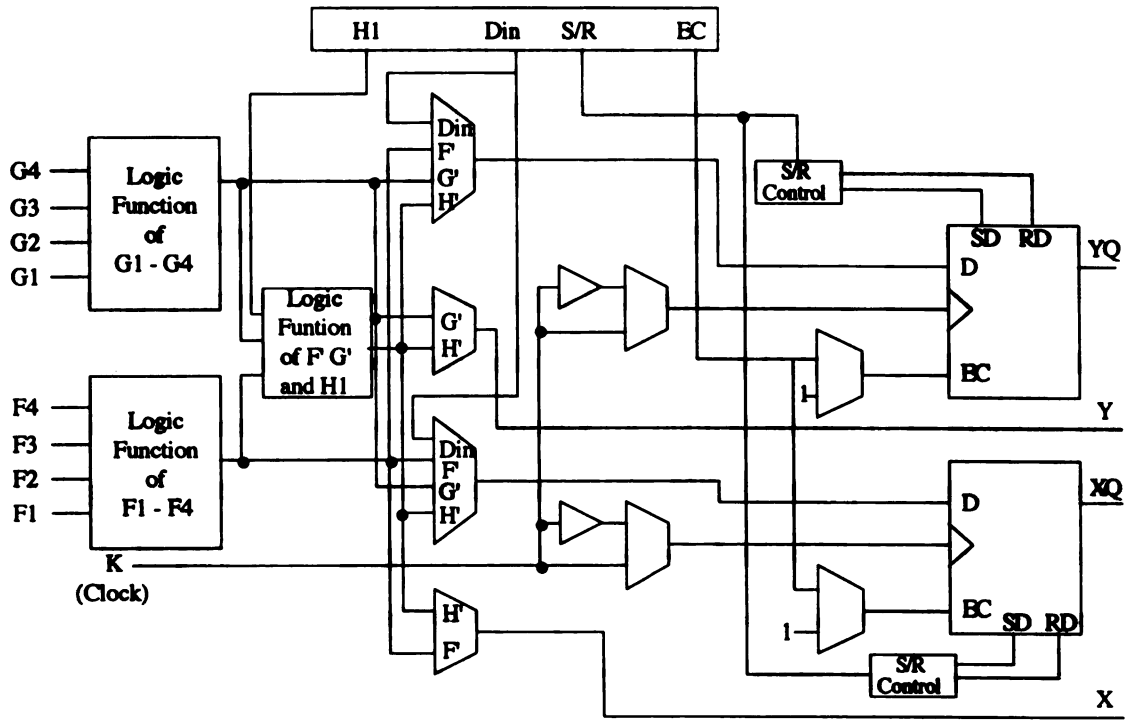


Figure 2.6. Simplified block diagram of the XC4000-Family CLB

The design can be synchronous by using the two flip-flops to latch values, or combinational using the unlatched outputs. Each CLB is surrounded by an interconnection network that allows the CLBs to communicate with each other. Delay times governed by routing are a key issue in developing fast designs with FPGAs. Signals are brought off chip through IOBs (Input Output Blocks). These pins can be configured as latched, tristated, or combinatorial signals. The third generation Xilinx 4025 has 1024 CLBs in a 32x32 array with 256 IOBs. This is large enough to implement simple algorithms and state machines. We will examine a few of these efforts, each with a unique strategy on how to utilize the FPGA for custom computing.

FPGAs have the ability to be reprogrammed an unlimited number of times. This allows for fast design and development of ASICs and prototypes at relatively low cost. Applications such as image processing benefit from the reconfigurable hardware as well. In general, these applications have parameters that may change from execution to execution, but possess potential for improved speed from hardware implemented algorithms. Image classification [25] and convolution [24] are just a few examples of the successful applications of this technology. Other applications utilize the FPGAs as a reprogrammable co-processor or as execution units [16]. A self-timed floating point co-processor [23] can be added without affecting the system clock of an existing system.

Several reconfigurable processors have also begun to appear [17][6][32][14][2]. The ability to generate hardware specifically for the instructions to be executed is an exciting prospect. Self-reconfiguring processors [6][32] utilize the feature of some FPGAs to be partially reconfigured. This allows for a "virtual hardware" setup where a processor may support several operations, but only have room for one or two actual configurations to be running on the FPGA. This also provides the ability to create as many functional units as necessary to extract the maximum level of parallelism. However, work done with the RRANN2 [11] shows that the overhead for reconfiguring systems is quite high. In a neural net application, 80% of the total time was spent in reconfiguration. This overhead still limits these systems to highly parallel applications.

Systems such as the SPYDER [17] and the DISC architecture [32] were built with the idea of executing instruction level programs. They have hardwired data paths and a small array of FPGAs for reconfigurable execution units. The larger more general purpose boards such as the Teramac [2], the Enable++ [14] and the Splash 2 [7] all have more FPGA chips and programmable crossbars to implement different communication topologies. These boards have been shown to perform well for programs with large amounts of data parallelism, however, they need to be programmed specifically to take advantage of their hardware.

With the advent of good hardware descriptor compilers and tools, the concept of using an implementation independent model [27] to explore new instruction set architectures was proposed. With these large arrays and VHDL as a tool, it becomes possible to create an instruction set architecture on top of these general purpose custom computing arrays. This gives us the ability to create a custom computing system flexible enough to implement the four models of computers that Flynn proposed.

2.7 Summary

Exploring the different ways we can enhance single processor performance, we note that there are basically two methods of increasing throughput: (1)decreasing cycle time, and (2)executing multiple instructions. Cycle time ultimately depends on the technology available. Multiple instruction execution however deals with the organization of the architecture and the programs executing on the processor.

The key for successfully executing multiple instructions lies in the ability to schedule them to avoid hazards. Instruction scheduling techniques can be static or dynamic. Although initially more attractive, dynamic scheduling techniques cost more in hardware and add complexity to the machine. This will set an upper limit of performance when compared to a statically scheduled machine. Statically scheduled machines do not need to examine instructions on the fly before execution. While it is too early to say which approach will ultimately perform best, both approaches need to be implemented and studied. Splash 2 offers a cost-effective environment to realize and test these architectures quickly.

CHAPTER 3

Design Methodology & Architecture Specification

With an understanding of some of the problems and issues of superscalar processors, we now consider implementing a superscalar processor on an FPGA-based array. The design of our superscalar processor is driven by a set of architectural specifications. This chapter first introduces the RISC architecture as the foundation of our new VLIW machine. The details and limitations imposed by the Splash 2 are discussed followed by the final architectural specification of the Dex-II that was implemented. Finally, we examine the various hazards that exist and how they are resolved.

3.1 Architecture Driven Design Methodology

Two distinct architectures form the new VLIW architecture, Dex-II. The Dex JR. provided the framework for the RISC instruction set, and the Splash 2 architecture drove the implementation strategies. This architectural driven approach is distinct from application driven designs.

Application driven architectures are targeted towards very specific tasks. The Splash 2 was developed as a method to implement application driven architectures. Application driven architectures become obsolete when the application is no longer needed or outdated. Using the Splash 2 allows the user to replace the architecture when

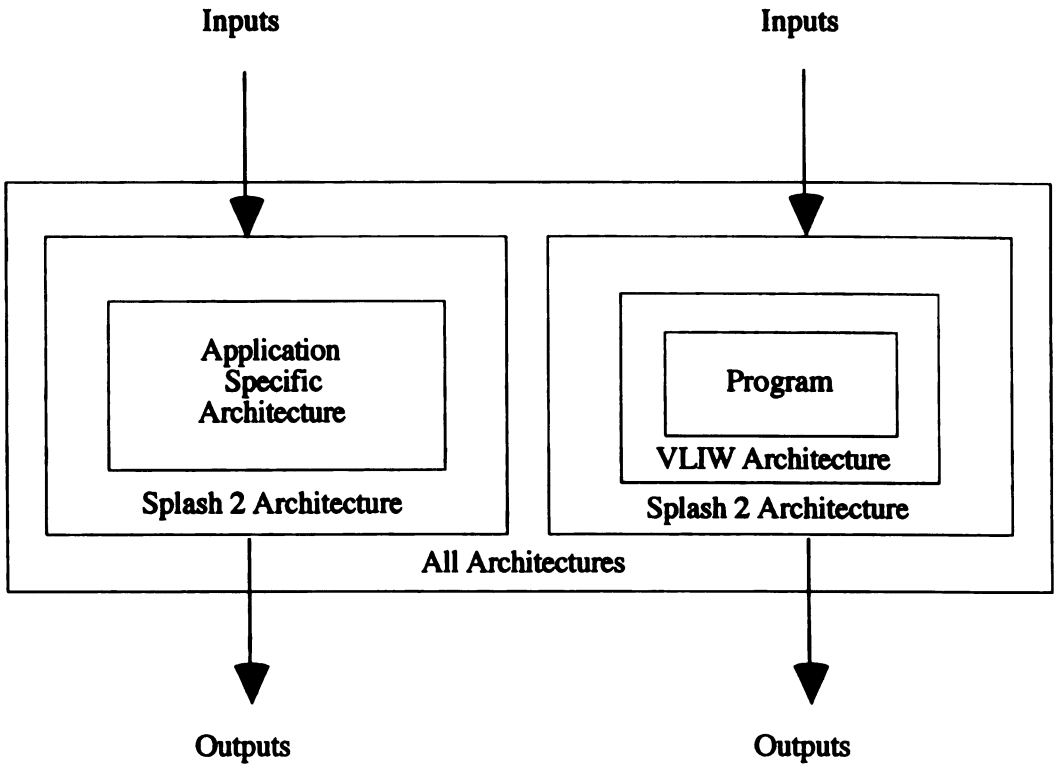


Figure 3.1. Architecture hierarchy

this occurs. Although less costly than building a new system each time a new task arises, using the Splash 2 does incur some overhead. Architectures are now limited by the resources and capabilities of the Splash 2 system. Figure 3.1 shows how an application specific architecture is constrained by the Splash 2 architecture. Application driven architectures are determined by the specific task and the resources of the Splash 2 architecture.

Figure 3.1 also shows how the VLIW architecture is implemented within the constraints of the Splash 2 architecture. It is similar to an application specific architecture, however the purpose of the VLIW is not to perform a specific task. The VLIW architecture can be programmed to solve several different tasks by replacing the program block in Figure 3.1. Instruction set architectures change over time as new hardware is developed and new instructions are added. Splash 2 offers the ability to

prototype and alter the architecture as needed. The architecture of the VLIW machine is driven by the RISC instruction set, the issues of pipelining, and the resources of the Splash 2 architecture.

3.1.1 Evaluation Criteria

The architecture of the Dex-II is based upon the RISC instruction set. This instruction set provides all the primitives necessary to perform more complex instructions. The Dex-II, therefore, must successfully implement these instructions. In doing so, we can compare the performance of our implementation to the base RISC architecture.

We will also evaluate how the VLIW instruction set affects the extraction of ILP. A comparison of the VLIW and RISC code in Chapter 5 will establish a quantitative measure of the performance of the new architecture. This measure of performance allows computer designers to better understand and improve architectures.

A successful implementation of the Dex-II will also allow us to measure the utilization of the FPGAs. This allows us to evaluate the use of the Splash 2 resources in prototyping instruction set architectures. This data, presented in Chapter 4, is important to improving future generations of FPGA-based arrays.

3.2 Architectural Specifications

The architectural specifications set forth by the Splash 2 and Dex JR. sets the expectations and implementation of the Dex-II. Our goal is to implement a processor that has the same functionality of the Dex JR. on a platform as flexible as the Splash 2. We will then extend the original RISC architecture to a VLIW implementation called the Dex-II. The architectural specifications provide the level of detail necessary to achieve that goal.

3.2.1 Splash 2

Implementing any design on the Splash 2 requires a good understanding of the underlying hardware. The Splash 2 system is based on the Xilinx 4010 FPGA. Each chip represents a processing element that consists of 400 CLBs. Each CLB can perform two independent Boolean functions of up to four variables, a function with five variables and a four variable function in certain instances, or a single function up to nine variables. The resulting signal can be combinatorial or registered by the two flip-flops. A careful study of the simplified block diagram in Figure 2.6 should clarify the functionality of the CLBs. The resources of the CLBs will determine the final size needed to implement a design.

The XC4010 also has 160 IOBs to control and condition signals entering and leaving each chip. These IOBs are used by arranging the chips in a linear array using two edges of the chip to communicate with each other and a third edge to a central crossbar as can be seen in Figure 3.2. The left and right edges can talk only to the adjacent chips through a 36-bit data path while the third edge can communicate with up to five different chips through the crossbar.

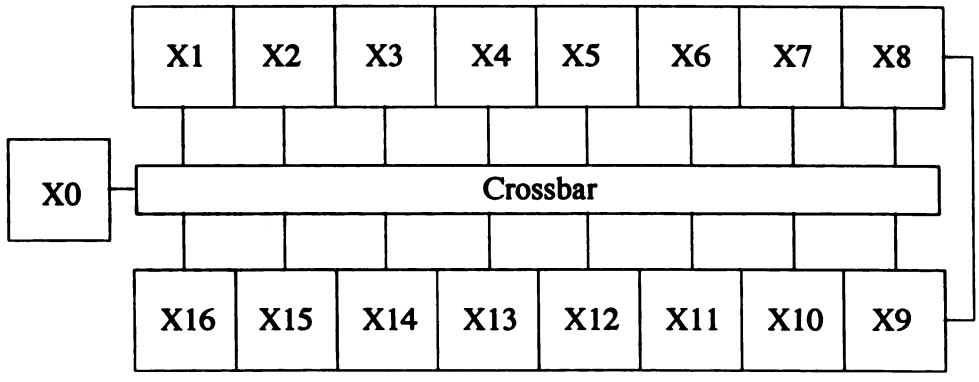


Figure 3.2. Splash board

The crossbar is a 36-bit data path arranged in 8-bit segments for a total of four 8-bit slices and one 4-bit slice. Each slice can be configured to transmit data in one direction. It can store and switch among eight different configurations connecting any of the bit slices from one chip to another. Control of the crossbar is handled by the seventeenth FPGA, X0.

Each PE (Processing Element) also has a 256K by 16-bit memory associated with it. This is accessed through the fourth edge of the chip with 18 address lines and 16 data lines. Timing of the memory is handled separately by the Splash 2 system. A read operation requires two global clock cycles, one to latch the address and the following cycle to read the data. Although it is possible to pipeline back to back reads, a read followed by a write must be separated by at least one clock cycle. Since a write is initiated by placing an address and the data to be written, a read followed by a write would require the used data lines.

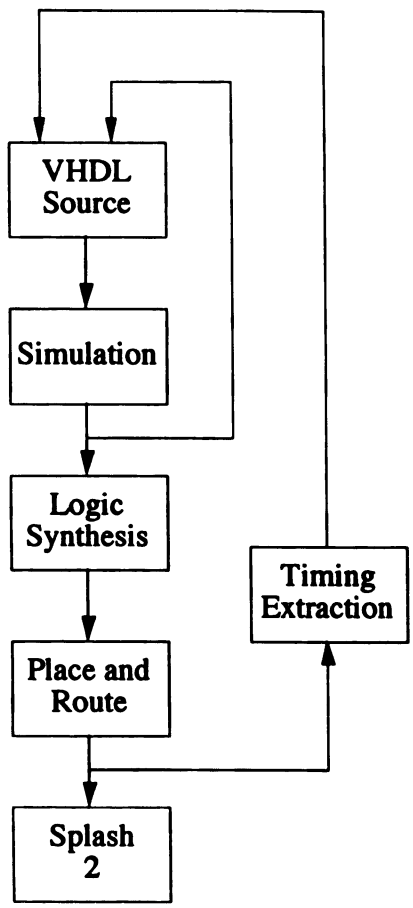


Figure 3.3. Design flow

Control of the Splash 2 system is handled by a SPARC host attached to the board through a VME backplane. The Splash 2 is scalable by adding up to 255 more Splash boards to this backplane. Each board will have its X1 chip attached to the previous boards X16 chip and so forth. The host can be interfaced with C libraries or a symbolic debugger known as T2. These software interfaces allows the user to map bitstreams to specific FPGAs, control the clock, and provides many other debugging and I/O functions.

The design flow for programming the Splash 2 system is depicted in Figure 3.3. Programming of each PE is done with a hardware description language, VHDL. Physical pins of each PE are mapped into logical signals using a combination of Splash 2 and vendor libraries. Simulation can be done to ensure that the algorithm is correct using Synopsys design tools. After simulation, the user synthesizes the VHDL code into a gate level description using Synopsys compilers. At this point, the Xilinx software is invoked to automatically place and route the desired hardware. The final result is a timing analysis of the design and a bitstream that can be downloaded to the Splash 2 system.

3.2.2 Dex JR.

The framework of our new processor, the Dex-II, comes from a simple, clean RISC architecture called the Dex JR [31]. The instruction set consists of load/store instructions, three mathematical operations, and conditional branching. The design of the Dex JR. called for a four stage pipeline: Fetch, Decode, Execute, and Writeback. This balanced our pipeline to provide the best possible cycle time with the components that were available.

A four port, 32x32-bit register file provides operands for mathematical operations as well as absolute addressing for memory. Four ports were necessary in order to write a single result and supply two operands per cycle. The fourth port was exploited by implementing a double load from memory. This special instruction proved to be a significant factor in performance by increasing the bandwidth to memory. Since all the mathematical operations take two operands, this reduced the time to load the operands from two cycles to one. Programs which require intensive memory cycles such as array and matrix operations also benefited from the ability to load multiple values.

Data dependencies of consecutive instructions were eliminated by forwarding paths. Scheduling of the correct forwarding path for execution is done statically during compile time. Only one delay slot is necessary for the branch commands as the execution

of the branch was done in the Fetch stage. The delay slot comes from waiting for condition codes to be resolved in the Execute stage.

The global clock is broken down into a four phase clock to perform memory operations. This ensures that the setup and hold time requirements are met. The instruction length is 32-bits non-encoded and the data and instruction caches are independent of each other. The data path, designed as a 32-bit architecture, is shown in Figure 3.4.

3.2.3 Dex-II

Since FPGA technology is more complex than dedicated chips, the top speed and performance will always lag the most current ASIC technologies. In order for an FPGA-based architecture to compete, it must be able to provide unique features through its programmability. This feature allows Splash 2 to prototype an ISP with the added performance of the latest architectural advances. The goal, therefore, was to increase processor throughput with a redesign of the Dex JR. by doubling the processor power. This is accomplished by designing the architecture to execute two instructions per clock cycle instead of one.

The Dex-II is a VLIW version of the Dex JR. A VLIW architecture was chosen for its relative simplicity of hardware. This was a major concern as the size of designs for each PE was limited by the physical FPGA itself. The architecture implemented on the Splash 2 is also scaled from the original 32-bit architecture down to a 16-bit data path due to the limited communication resources between PEs. The functionality of the instruction set was retained.

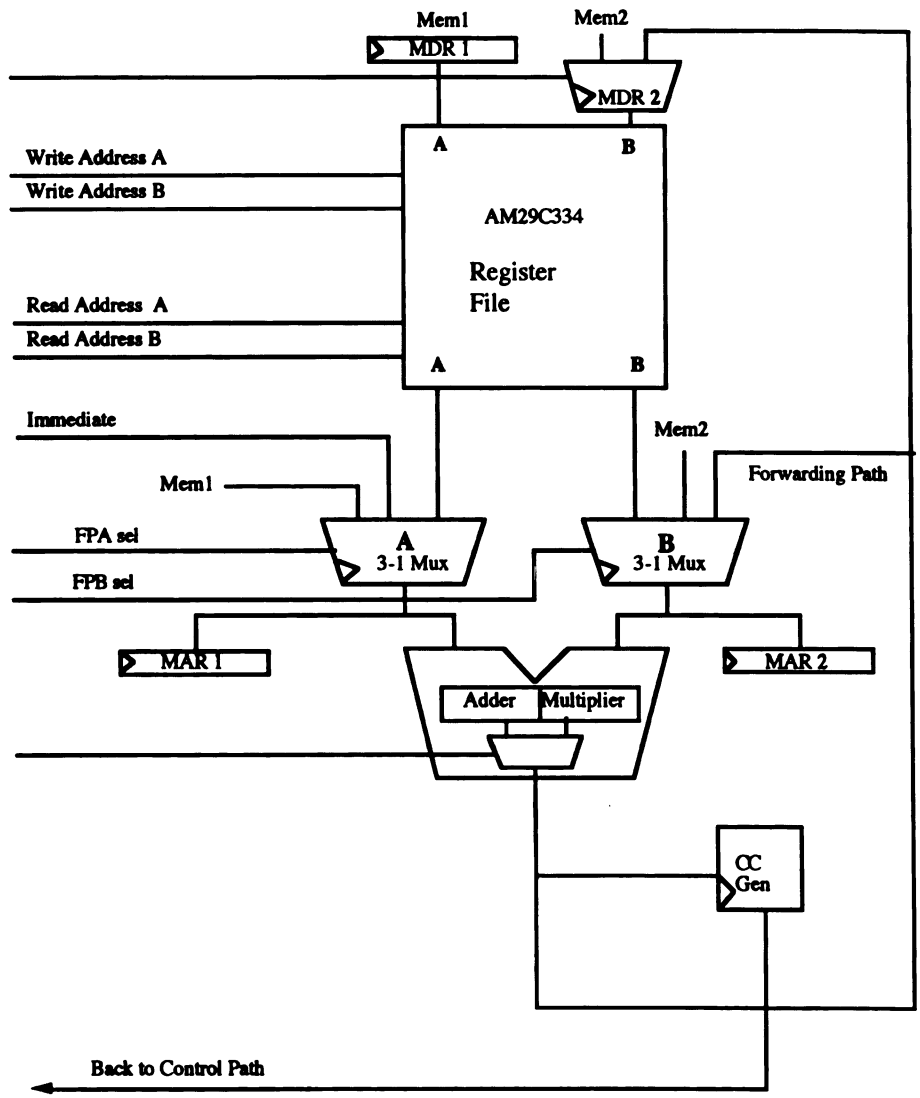


Figure 3.4. The Dex JR.

The pipeline depth was also altered to three stages due to the increase from a four phase clock to a six phase clock. This was to accommodate the need for more communication between stages and implementation across multiple PEs of the Splash 2 board. This reduction in pipeline stages also eliminates several forwarding paths. With communication resources so highly limited, this tradeoff was necessary to retain data coherency.

The instruction length is doubled to 64 bits to execute two instructions. Extra bits are used to pass register destinations of both processors determined at compile time to retain data coherency. The redesigned data path is shown in Figure 3.5. The box enclosing the functional units allow any combination of two instructions to be executed with its own operands. Both memory and registers are kept coherent with a combination of hardware and compile time methods.

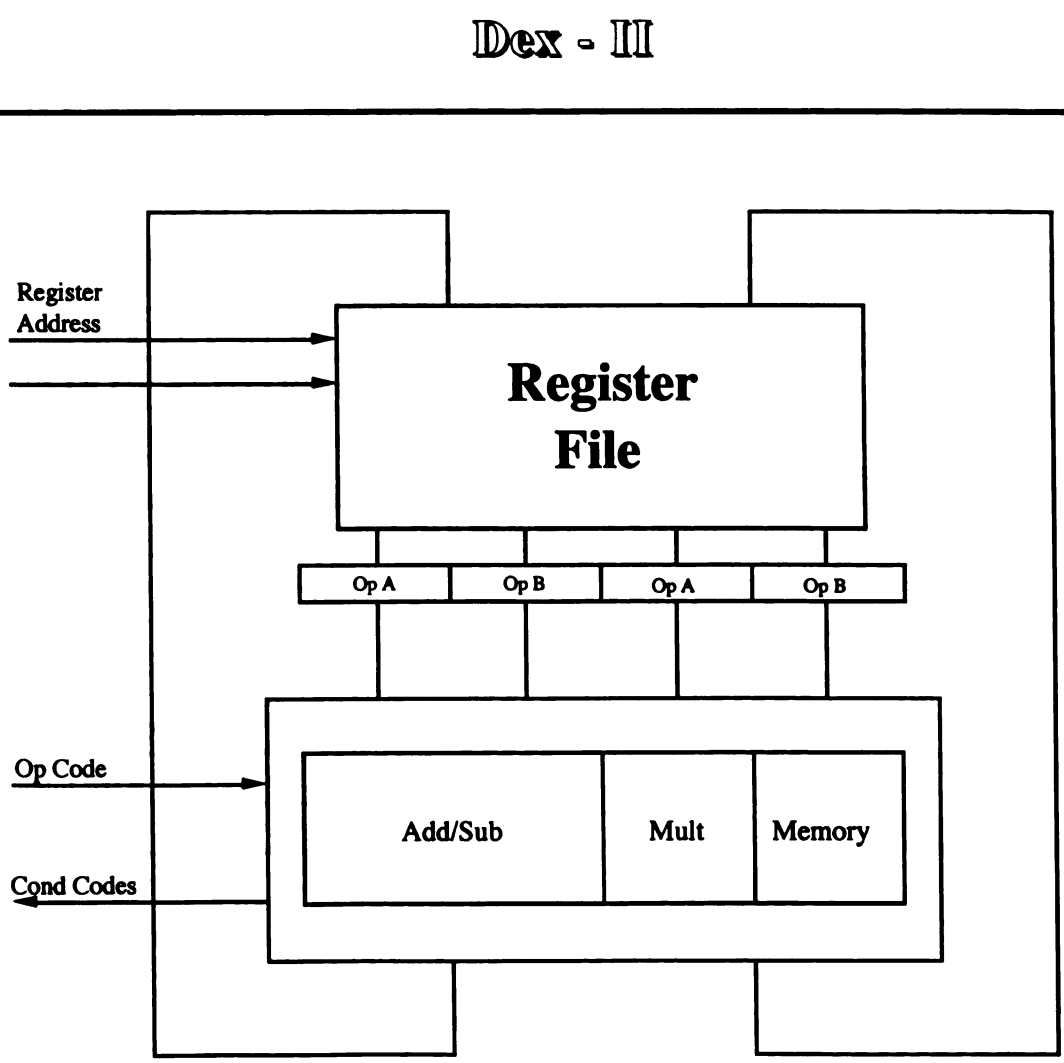


Figure 3.5. The Dex-II

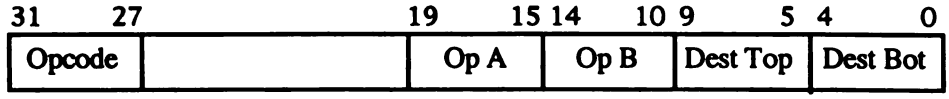
3.2.4 The Instruction Set

The instruction set consists of 14 different operations, as listed in Table 1. This reduced instruction set allows more complex functions and memory addressing by the combination of these simpler instructions. The Dex-II allows any two of these operations to be executed at the same time except for branching.

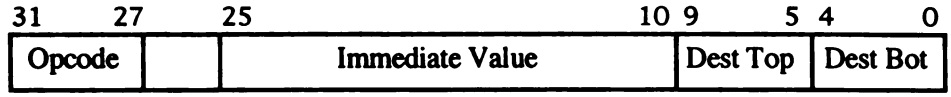
Table 1. Instruction set

Instruction	Description	Register Transfer Level Rx, Ry, Rz
LD	Load from memory	Rx <- (Ry)
LDI	Load immediate	Rx <- immediate
ST	Store to memory	(Rx) <- Ry
ST	Copy Register	Rx <- Ry
BRA	Branch	PC <- Address
BZ	Branch if Zero	PC <- Address if Z
BN	Branch if Neg. or Zero	PC <- Address if N
BNZ	Branch if Neg. or Zero	PC <- Address if NZ
BNV	Branch if Neg. or oVfl.	PC <- Address if NV
ADD	Add	Rx <- Ry + Rz
SUB	Subtract	Rx <- Ry - Rz
SFTL	Shift Left	Rx <- Ry shifted left 1
SFTR	Shift Right	Rx <- Ry shifted right 1
MULT	8-bit Multiply	Rx <- Ry * Rz
NOP	No Operation	

ALU Operations, Load, Store



Load Immediate



Branch

Operation	Opcode
LD	00100
LDI	00101
ST	00110
ST	00111
ADD	10000
SUB	10001
SFTL	10010
SFTR	10011
BRA	01000
BZ	01001
BN	01010
BNZ	01011
BNV	01100
NOP	00000

Registers are addressed from R1-R31. R0 will always return a zero and writing to it will not affect it.

Figure 3.6. Instruction format

Figure 3.6 depicts the three different instruction formats. The OpA and OpB fields identify the source registers for the execution units. The Dest Top and Dest Bot fields identify the destination register for the top half of the processor and the bottom half of the processor. Both halves will have identical destination registers, however, their Opcode and source registers will differ. Registers are addressed from 0 through 31 represented in standard binary form. By making the destination registers the same on the top half and bottom half of the processor, we encode information in the instruction rather than passing that information during runtime. This ensures that all instructions affecting registers will update all four PEs in the decode stages concurrently with a minimal exchange of information. There are also extra bits and room left in the opcode for future expansion and addition of instructions. Two possible additions are register windowing instructions and paged memory to exploit unused memory.

3.3 Design Constraints

The Dex-II was designed to keep wasted NOPs at a minimum. However, spatial and temporal parallelism introduces certain restrictions that must be observed. These constraints are classified as data, memory, and control hazards.

3.3.1 Data Hazards

Data hazards are constraints caused by the data path. Two operands that have data dependency between them cannot be issued on the same clock cycle. Figure 3.7 shows an example of data dependent instructions. The first example shows two instructions that cannot be executed together since the second instruction is dependent upon the first instruction. This is executed correctly by the insertion of NOPs in the second instruction slot to delay execution until the result has been computed and saved to register 3. True data dependencies can not be avoided, however, certain architectures implement special hardware to reduce their impact.

Example.

The following instructions will produce erroneous results

```
ADD  R3, R1, R2          * R3 <- R2 + R2
ADD  R5, R3, R4          * R5 <- R3 + R4
```

This is due to the fact that R3 will not be computed in time for the second instruction. In order to execute correctly, either another instruction needs to be scheduled in its place, or a NOP must be injected

```
ADD R3, R1, R2
NOP

ADD R5, R3, R4
NOP
```

Figure 3.7. Data dependency

The ALU of the Dex-II, unlike the SuperSPARC, are completely separate from one another. The cascaded ALU scheme shown in Figure 3.8 is used in the SuperSPARC to eliminate this restriction. If the second instruction depends on the first, than they are executed sequentially with ALUC, otherwise they are executed independently on ALU2 and ALU0. This scheme was implemented at the expense of another pipeline stage. For the highly piped design, this solution is feasible, however, for the Dex-II this would effectively give the same performance as the example with two instructions. Even worse, the addition of another pipeline stage would affect the performance of other instructions.

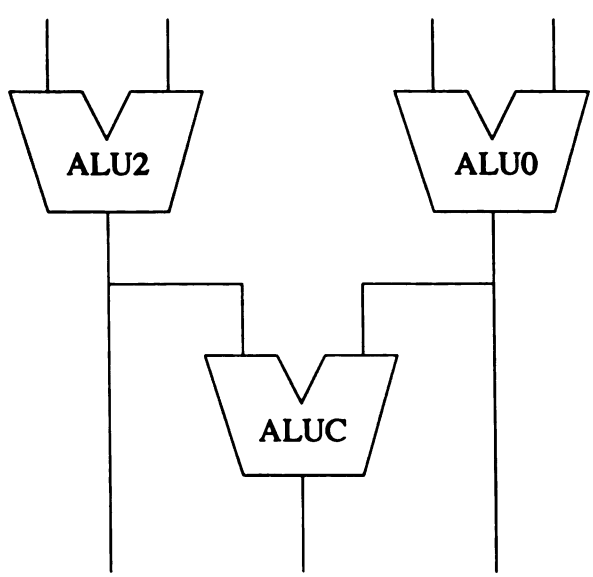


Figure 3.8. Cascaded ALU

Read after Write (RAW) and Write after Read (WAR) data dependencies are eliminated by the decode/writeback scheme employed. This scheme ensures that registers are updated prior to broadcasting the operands for the next instruction. Details of the implementation can be found in the Synthesis chapter.

3.3.2 Memory Hazards

Care must be taken to avoid assigning the same register or memory location with different values on the same cycle. Figure 3.9 shows code which tries to write two different results into the same register location.

```
Example.

ADD  R3, R1, R2          * R3 <- R1 + R2
ADD  R3, R4, R4          * R3 <- R4 + R4

This will corrupt the register file.

ADD  R3, R1, R2
NOP                      *Or reschedule another instruction

ADD  R3, R4, R4
NOP
```

Figure 3.9. Memory hazard

Writing different values to the same memory address or register in the Dex-II will cause the top and bottom processor to have conflicting memory and registers. This may force programs that dynamically address memory to insert NOPs to ensure data coherency.

3.3.3 Control Hazards

Keeping both processors synchronized is a challenging task. Since the top and bottom processors have independent program counters, they need the same information in

order to branch correctly. Since most conditional branching occurs on comparisons, condition codes are only generated by the Add/Sub unit. This means that both instructions setting the condition codes must be the same and that the conditional branch must follow with a one instruction delay. Figure 3.10 illustrates how control hazards are resolved in the Dex-II by this strategy.

```
Example.

SUB  R3, R1, R2          * Sets up top CC
SUB  R3, R1, R2          * Sets up bottom CC

NOP                      * Or filled with some useful instruction
NOP                      * Or filled with some useful instruction

BEQ 1000                 *Branch to $1000 if zero flag set
BEQ 1000                 *Branch to $1000 if zero flag set
```

Figure 3.10. Control hazard

3.4 RTL Specifications of Dex-II

To implement the Dex-II on Splash 2, we begin by developing a simple RTL (Register Transfer Level) description of the Dex-II. This allows us to partition our design according to resources and communication paths available on the Splash 2. It became very apparent that the communication paths offered by the crossbar and linear array would be the determining factor of how our design would be implemented. The number of clock cycles needed to implement an instruction cycle was determined by the data paths between PEs. After this was established, a detailed RTL that is easily translated into VHDL code was simulated and synthesized.

3.4.1 PE Partitions

Mapping a design over to the Splash system is a challenging task. Each processing element is limited by the number of CLBs, number of I/O pins, and the amount of memory available per PE. Even with larger FPGA chips, the amount of

hardware that can be assembled onto them is still somewhat limited. With this in mind, we begin by splitting the resources stage by stage.

The partitioning of functions is shown in Figure 3.11. Processing elements PE1 - PE8 represent half of the complete VLIW processor and can stand alone to execute scalar RISC code. This half of the processor will be referred to as the *top processor* hereafter. Elements PE9 - PE16 are the mirror hardware to implement the second processor. This half will be referred to as the *bottom processor*.

The control path is implemented across the first two PEs and supplies a 32-bit instruction word to its half of the processor. Since the memory is only 16-bits wide, two elements are needed to generate the 32-bit word. Elements PE1 and PE2 also house the program counter, the instruction memory, and the conditional branching logic. The decode stage contains the register file and circuitry to forward data. We chose to use two elements in this case to provide extra bandwidth to allow dual reads from memory. This allows us to shorten the number of clock cycles needed to complete a single instruction cycle. The execute stage consists of four functional units: Two adder/subtractor units, a multiplier, and the memory manager. This setup is cloned on the bottom to provide equal

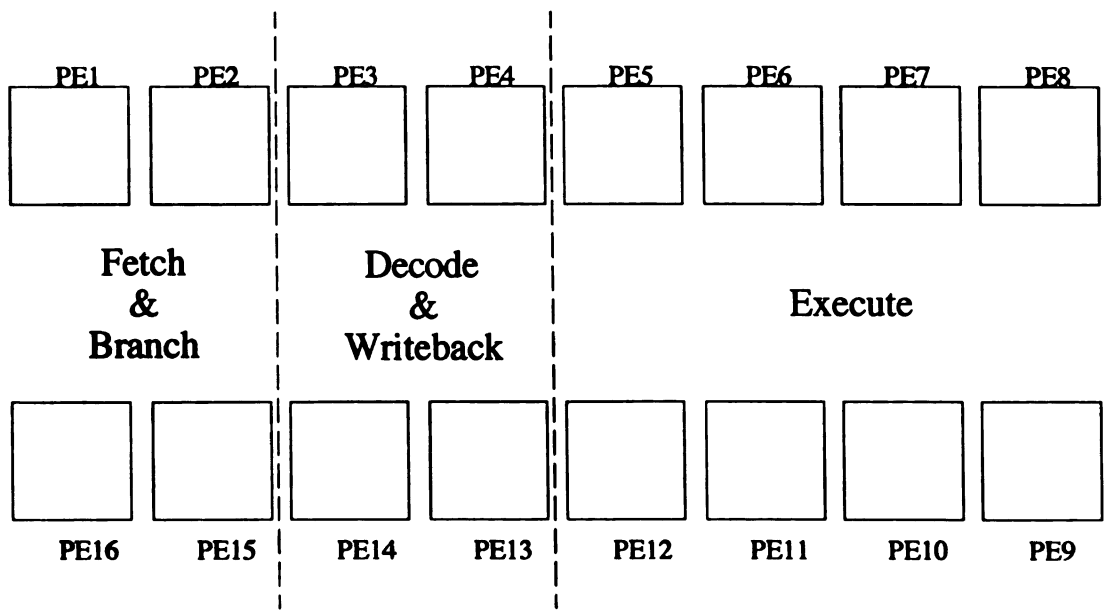


Figure 3.11. PE partitioning

functionality on both the top and bottom halves of the processor. The memory units of PE8 and PE9 utilize the extra left-right communication lines between them to keep the memory coherent.

3.4.2 Communication Paths

The amount of communication between PEs was critical in this partitioning scheme. As noted before, the major problem encountered here is the limited number of communication paths to implement bus structures. While the crossbar is capable of handling multiple connections, we cannot, for example, selectively broadcast a 16-bit operand from the decode stage to each of the four execute stages.

The left/right communications of each chip are dedicated to the control path. Instructions issued from PE1 and PE2 flow to the right and supply each pipeline stage with the next instruction. The data path, therefore, is limited to using the crossbar for passing operands and results around. Using the programmable crossbar, each cycle could send and receive data from different PEs.

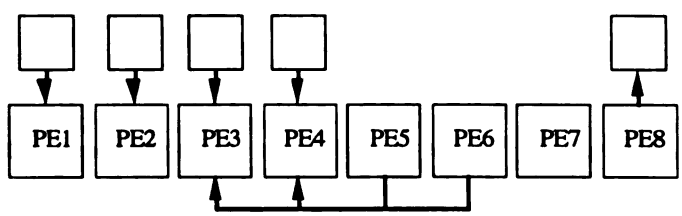


Figure 3.12. Cycle 1

Figure 3.12 shows the first of six cycles. PE1 and PE2 fetch the next instruction from their respective memories while PE3 and PE4, the Decode stage, fetch the operands from their register file memory. The 16-bit results from the Execute stage from PE5 and PE6 are sent back to the Decode PEs through the crossbar. If the instruction in the Execute stage is a memory store, then PE8 performs the store operation.

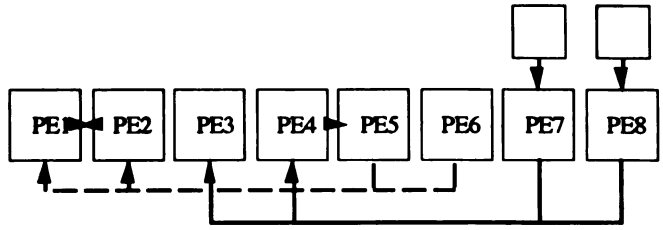


Figure 3.13. Cycle 2

Figure 3.13 depicts the communication paths for the second cycle. The Fetch stage swaps information about the new instruction and forms the full 32-bit instruction word. The Decode stage begins to pass the next instruction along to the Execute PEs and receives the final two results from PE7 and PE8. PE5 and PE6 now send the results of the condition codes to the Fetch stage while PE7 and PE8 complete their memory reads.

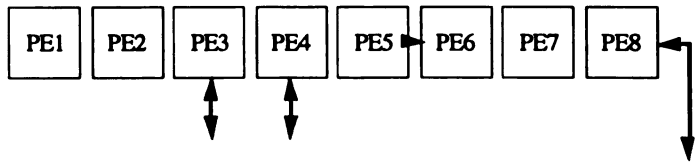


Figure 3.14. Cycle 3

The third cycle is where synchronization with the second processor occurs. Information about the destination register of the second RISC processor is encoded in the instruction word, therefore, only the data needs to be communicated. Figure 3.14 shows the Decode stage sending and receiving the 16-bit results from the second processor. Meanwhile, the execute stage continues to pass along the next instruction and PE8, the memory manager, exchanges information on what kind of memory operation is performed. The Fetch stage at this point also updates its PC (program counter). If a branch is indicated, the PC loads the absolute address from the instruction word, otherwise, the PC is incremented by one.

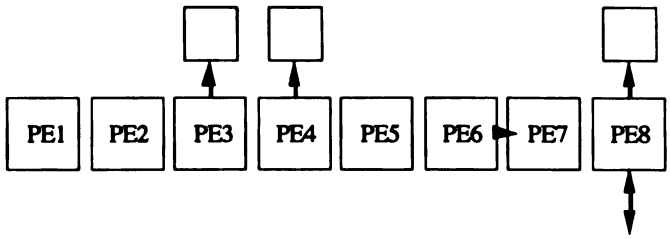


Figure 3.15. Cycle 4

Cycle four shown in Figure 3.15 begins the writeback sequence. PE3 and PE4 writes the result from the Execute stage into the register file. Since only one operand can be written at a time, the result from the second processor must wait. The memory manager swaps address and operand information. If the second processor performed a store operation, then we must also perform the same store operation to keep the two memories coherent.

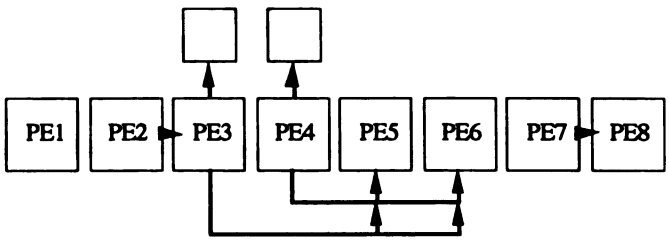


Figure 3.16. Cycle 5

Figure 3.16 shows the two decode stage PEs writing the results from the second processor and sending the operands for the next instruction to PE5 and PE6. The next instruction for the Decode stage is passed from the Fetch stage and the final execution PE gets the next instruction as well.

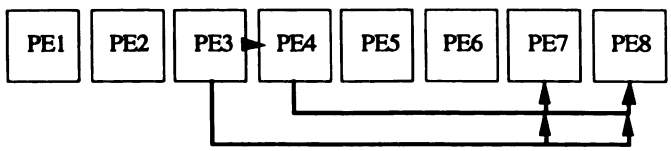


Figure 3.17. Cycle 6

The final cycle shown in Figure 3.17 illustrates the Decode stage passing operands to the last two PEs. The second Decode PE receives the next instruction. Each PE updates its respective instruction buffer to begin the next instruction.

3.4.3 Fetch Stage

The Fetch stage, whose operations are described with RTL statements in Figure 3.18 keeps track of the current program counter (PC). Using the PC as an address, it reads instructions from PE1 and PE2 in phases 1 and 2 to form a complete 32-bit instruction. Since each PE only has a 16-bit memory chip, to form the complete instruction, PE1 must send the most significant word to the right and PE2 concatenates the two to send to the Decode stage. Notice that the branch instructions are also executed in this stage. In order to get the correct branch address, PE1 must have portions of the lowest significant word. Therefore, PE2 must also send its values to the left. This occurs from phase 2 to phase 3 through the XP_Left and XP_Right data path. Cycle 4 is where the actual branch takes place, otherwise PC merely increments by one.

3.4.4 Decode Stage

The Decode stage is the most complex of all the stages. It needs to supply all four execution units with two operands each and receive the results from all four execution units. A multiplexer selects the correct result and the information is swapped between the top and bottom processor to ensure the register files are identical. This is the stage that dictated the six-phase clocking scheme.

Register Transfer Level

Fetch

Phase	PE1	PE2
1	MAR <- PC	MAR <- PC
2	IFetch [31 -16] <- MDR XP_Right [31 - 16] <-MDR	IFetch [15 -0] <- MDR XP_Left [15 - 0] <-MDR
3	CC <- XBar [35-32] IFetch [15-0] <- XP_Right	CC <- XBar [35-32] IFetch [31-16] <- XP_Left
4	PC <- PC+1 if not a branch PC <- IFetch [25-10] if IFetch [29-27] = CC	PC <- PC+1 if not a branch PC <- IFetch [25-10] if IFetch [29-27] = CC
5		
6		

Figure 3.18. Fetch RTL Description

The Decode stage begins by assuming that the operands are in memory and starts the read process. The 5-bit address is taken directly from the instruction word Id. The only difference between the PE3 and PE4 is which operand they read. In the second phase, the value of the register is read from memory. The results from the first execution unit (PE5) is also read into both decode stages. Since the Xbar can be configured in four 8-bit segments, each decode stage can accept up to two 16-bit results each phase. The third phase latches the results from execution units 3 and 4 (PE7 and PE8).

The results from each execution unit are funneled into a 4-1 mux and selected by the opcode of the executing instruction, Ix. The output of the mux will contain the correct result from the instruction in the execute stage of the pipeline. Note that the mux is not latched and thus provides the correct value by the end of the third phase, even though the inputs were latched at the beginning. The execution result is latched from the output of the mux at the beginning of phase 4 on the top and bottom halves of the processor. Likewise, the top latches the bottom result in phase 4. With both results available, the decision to forward operands can be made.

If the source registers in the decode stage match the destination register in the execute stage, than the operand is replaced with the new result. The correct operand begins to broadcast its values to execution units 1 and 2 (PE5 and PE6). Cycle 4 also physically writes the results back into the register file if the destination is not R0. Cycle 5 writes the bottom half of the processor result if the destination is not R0. Both operands are sent to execution units 3 and 4 (PE7 and PE8). The last phase latches the next instruction and moves the current Id to Ix.

Register Transfer Level

Decode and Writeback

Phase	PE3	PE4
1	MAR <- RegA	MAR <- RegB
2	OpA <- MDR Mux2 <- XBar[31 - 16]	OpB <- MDR Mux2 <- XBar[15 - 0]]
3	Mux3 <- XBar[31 - 16] Mux4 <- XBar[15 - 0] XBar [31 - 16] = MuxResult *	Mux3 <- XBar[15 - 0] Mux4 <- XBar[31 - 16] XBar [31 - 16] = MuxResult *
4	OpA <- MuxResult if DestTop = RegA OpA <- XBar [15 - 0] if DestBot = RegA Temp <- XBar [15 - 0] MAR <- Dest. Top MDR <- MuxResult XBar [31 - 0] <- OpA	OpB <- MuxResult if DestTop = RegB OpB <- XBar [15 - 0] if DestBot = RegB Temp <- XBar [15 - 0] MAR <- Dest. Top MDR <- MuxResult XBar [31 - 0] <- OpB
5	MAR <- Dest. Bot MDR <- Temp XBar [31 - 0] <- OpA	MAR <- Dest. Bot MDR <- Temp XBar [31 - 0] <- OpB
6	Ix <- Id Id <- XP_Left XP_Right = Id *	Ix <- Id Id = XP_Left *

* These values are actually non-latched. They are shown for ease of understanding.

Figure 3.19. Decode RTL Description

3.4.5 Execution Stage

The Execution stage of the pipeline consists of four PEs. Two PEs are configured to act as an integer Adder/Subtractor unit, one Multiplication unit, and the last PE as a Memory Management unit. Each unit can be reconfigured with relative ease to add special instructions. In addition, there is no need to keep the top half and the bottom half symmetric either. Therefore, it is possible to have up to eight different functional units. Instructions, however, would be limited to executing on the half which has the hardware to support them.

The Adder/Subtractor unit is found on PE5 and PE6. This configuration also has the right and left shift functions programmed onto it. Operands for this unit are actually latched at the end of the previous instruction. After the first phase is completed, the result is broadcasted on the crossbar to the decode stage. Condition codes are calculated during the second phase. Codes supported by the Dex-II are Negative, oVerflow, and Zero. The negative flag is set if the most significant bit of the result is a 1. An overflow occurs if the add/sub unit indicates a carry out condition, and the zero flag is set if the result is zero. The second Adder/Subtractor unit function is merely to generate the condition codes.

The Multiplication unit utilizes the PE memory as a look-up table to achieve its purpose. The reason for an 8-bit multiply is straightforward. The 16-bit data path can only support answers from two 8-bit words. Rather than design an elaborate plan for multiple register destinations, we restrict the operands to eight bits. Higher bits are ignored. The memory table is generated externally and loaded into the PE memory prior to execution of the program. By taking the two operands and using them as an address, the Multiplication unit can generate its result by the second phase. This works out well as we can only send two results back during each phase and phase 1 is being used by both add/sub units. The last phase is used to latch operands for the next instruction.

The Memory Management unit is located at the ends of the processor halves to utilize the extra communication lines. The right side of PE8 is connected to the left side of PE9 in the linear chain. This allows us to synchronize the efforts of PE8 and PE9 to keep a uniform main memory. The first phase latches the memory addresses and performs a write if the instruction is a store command. Otherwise, it assumes a read from the address. The second phase will return a result to the decode stage. This result is either the value of another register, the value of a memory location, or the 16-bit immediate value in the instruction word Ix. The third phase begins the synchronization of the two memories by exchanging results and instruction words. If the bottom instruction was a store, then the top writes the value into memory and vice versa. Since only stores can affect memory, only stores need to be considered. As before, the last phase is used to latch the new operands for the next instruction.

Register Transfer Level

Execute

Phase	PE5 & PE6	PE7	PE8
1	If SHFTL Xbar <- OpA << 1 If SHFTR Xbar <- OpA >> 1 If Add Xbar <- OpA + OpB If Sub Xbar <- OpA - OpB	MAR [15 - 8] <- OpA MAR [8 - 0] <- OpB	MAR <- OpB MDR <- OpA If ST, enable Write else Read
2	Set CC based on result Xbar [35 - 32] <- CC	Xbar <- MDR	If MV, Xbar <- OpA If LD, Xbar <- MDR If LDI, Xbar <- Immediate value.
3			Bot Ix <- XP_Right [15 - 0] XP_Right [31 downto 16] <- Ix [31 - 0]
4			Xbar [31 - 16] <- OpB Xbar [15 - 0] <- OpA If BotIx = ST then {MAR <- Xbar [31 - 16] MDR <- Xbar [15 - 0]}
5	OpA <- Xbar [31 - 16] OpB <- Xbar [15 - 0]		
6	Ix <- XP_Left XP_Right = Id *	OpA <- Xbar [31 - 16] OpB <- Xbar [15 - 0] Ix = XP_Left *	OpA <- Xbar [15 - 0] OpB <- Xbar [31 - 16] Ix = XP_Left *

* These values are actually non-latched. They are shown for ease of understanding.

Figure3.20. Execute RTL Description

CHAPTER 4

Simulation and Synthesis Environment & Results

With the RTL and architectural specifications completed, the design can be coded with VHDL and functionally simulated using Synopsys tools. This chapter presents the simulation tools used to implement the Dex-II. When the simulation correctly implements the architectural specification, the VHDL code can be synthesized and tested on the Splash 2. The results of the synthesis are also presented in this chapter.

4.1 Simulation & Synthesis Process

The register transfer level description was coded in VHDL for both simulation, and synthesis. Figure 4.1 shows how the VHDL hierarchy is arranged. To program each PE, we must write VHDL code for each XPARTS module. A sample description of the VHDL code is shown in Figure 4.2. The VHDL code is very similar to other high level languages. A header defines the name of the PE that is being programmed, followed by a list of signal declarations. These are followed by constant declarations and logic assignments. A process statement marks the start of the description of the architecture's logic. The wait statement defines the signal sensitivity of the process. The process will be evaluated only when a signal on the sensitivity list changes. In this case, all of the logic is synchronous on the rising edge of the clock. The code following the wait statement describes the functionality of our architecture taken from the RTL descriptions.

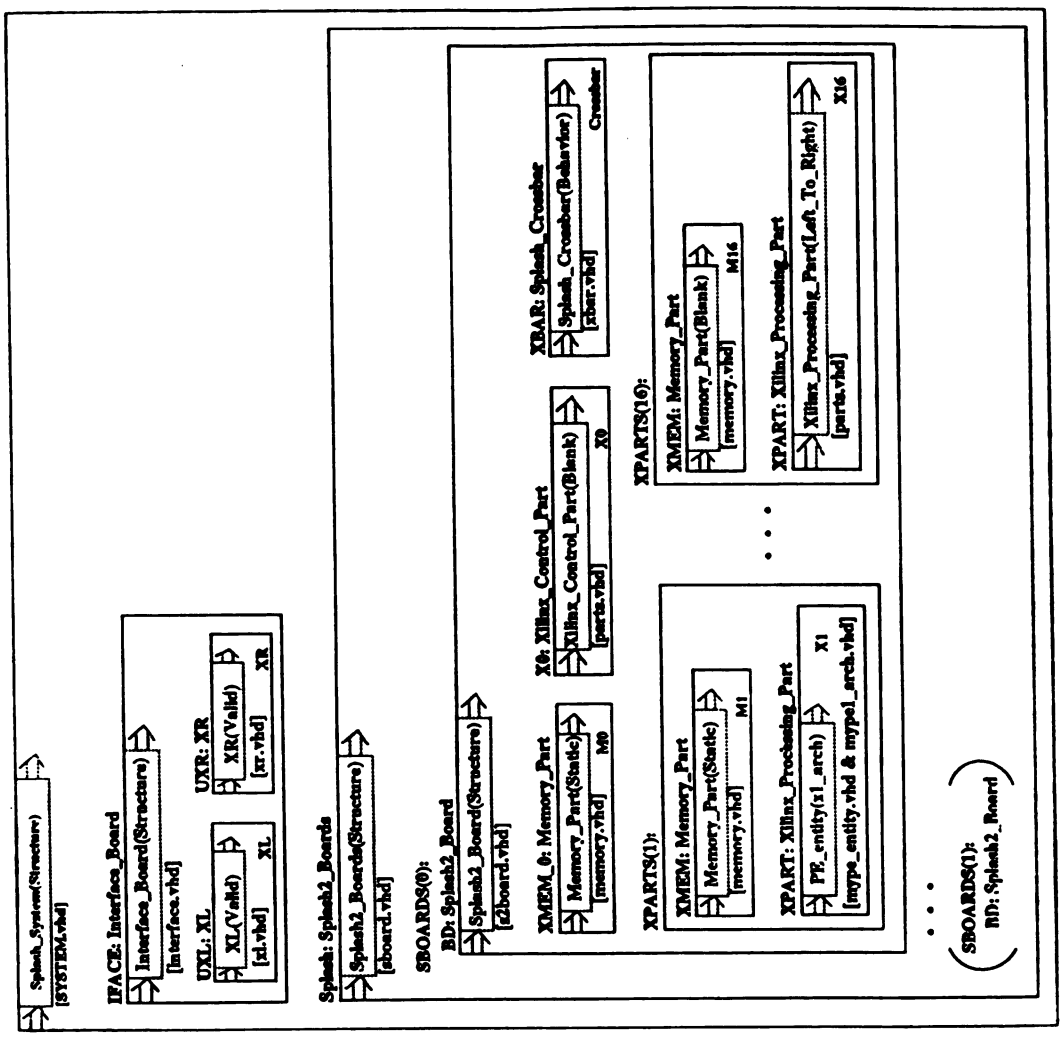


Figure 4.1. VHDL hierarchy

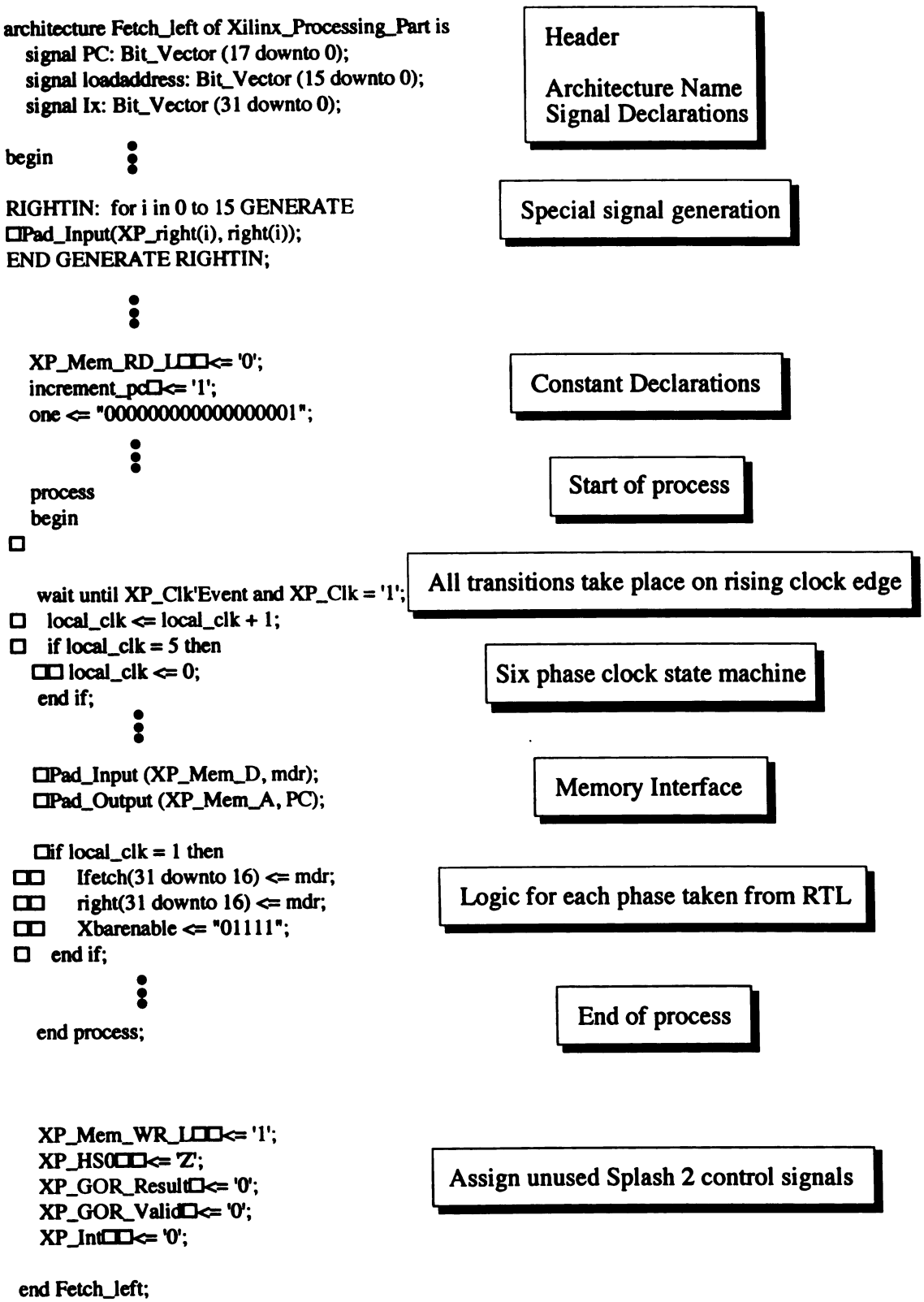


Figure 4.2. Code description

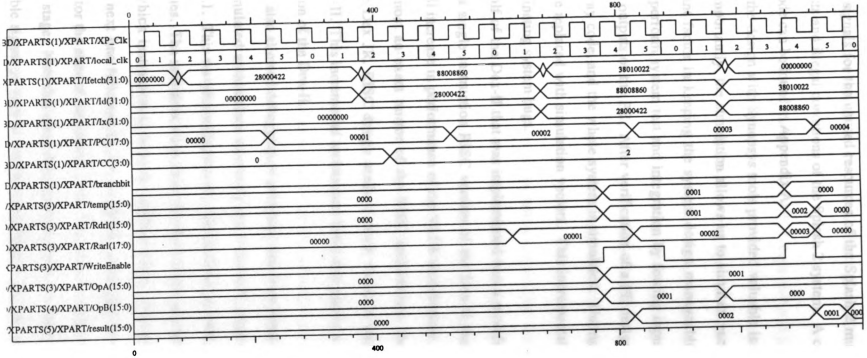


Figure 4.3. Simulator output

Lastly, the control signals for the unused resources of the Splash 2 must be defined to avoid signal conflicts and illegal conditions of the Splash 2 system. A complete copy of the implemented code can be found in Appendix A.

Simulating the design with Synopsys tools provided valuable insight into timing problems. More importantly, the simulation allows us to simulate each PE separately. This is an important factor in keeping the system design manageable. By using the simulator, we can perform system test and integration by forcing inputs and observing internal nodes and outputs of the PE. After verification of a PE design, it is integrated with the Splash array code, and the whole system is simulated. Unimplemented PEs in the system could be replaced with simulation vectors. This incremental, PE by PE build was crucial in eliminating system bugs.

The first half of the Dex-II that was implemented was a modified version of the Dex JR. This is a single instruction RISC version of the Dex-II that will be used to evaluate the overall increase in performance of the VLIW architecture. Since the VLIW architecture is almost an exact mirror of the RISC architecture, the simulation and verification of the Dex JR. RISC design made it much easier to implement the final version of the Dex-II. This incremental architecture build also played a crucial role in the final implementation of the Dex-II.

Simulation also allowed debugging of programs that were hand-compiled. Figure 4.3 shows the simulated architecture executing the Fibonacci program which can be found in Figure 5.1. This waveform displays the global clock as well as the local clock on the first two lines. The next three lines show the state of the instruction pipeline. We can clearly see which instruction is in each pipeline stage. The program counter is displayed on the next line followed by the condition codes and the branch bit. This allows us to monitor the state of the control path. The next six lines display information about the Decode stage. Note that when the first instruction is actually executed, we can see the WriteEnable signal go high as the two immediate loads are executed. The next

instruction is the addition function and the result of the Execute stage is shown on the last line of the simulation output.

The objective of our simulation is to write VHDL code that will generate a correct implementation of the Dex-II in hardware on the first pass. Unfortunately, the compilers, synthesis tools, and debugger are still relatively young and full of idiosyncrasies. For example, the interactive command tool, T2, interfaces the host to the Splash 2. T2 has the ability to scan the flip-flop states of the IOBs. While the design may utilize these pins going off-chip, the compiler will use a flip-flop from an internal CLB and configure the IOB without a registered value. This gives misleading feedback to the user as the pin appears to never change its value.

While simulation did provide information regarding timing, the actual implementation of our design code did not simulate correctly. As a result, the time taken to synthesize and debug our implementation was dominated by repeated compilation and synthesis. Using VHDL as a synthesis tool requires the designer to follow a certain style and rules of coding to achieve the desired implementation.

4.2 Synthesis Results

Upon verification of our design, the automated design tools were invoked to produce a design specifically for the XC4010 FPGA. The UNIX script, VHDL2XNF, invokes the Synopsys VHDL design compiler to generate an XNF (Xilinx Netlist Format) wirelist. This XNF file is used by the XNF2BIT script which invokes the PPR (Partition, Place and Route) software. These tools automatically place, partition, and route the XNF description of each PE to the physical chip architecture. The placement and routing of CLBs are iteratively improved upon automatically. The design flow is shown in Figure 4.4 along with the tools used in each step.

At the highest degree of optimization, our design yielded the synthesis results summarized in Table 2. This table shows us the percentage of total CLBs utilized in each PE of our design and the timing associated with the synthesis results. The percentage of CLBs utilized gives us a metric to estimate how much more logic can be implemented in each PE. This is important for future design considerations.

As we can see from the table, our critical time is dictated by PE5. The 90.0 ns result gives us an effective global clock speed of 11 MHz. With the division of the six-phase clock, this would give us a peak 1.85 MIPS (Millions of Instructions per Second) rating. Assuming that our VLIW machine can execute two independent instructions per cycle, that would effectively double our peak to a 3.70 MIPS rating. Current workstations have a typical peak MIPS rating in the hundreds. Thus, while slow

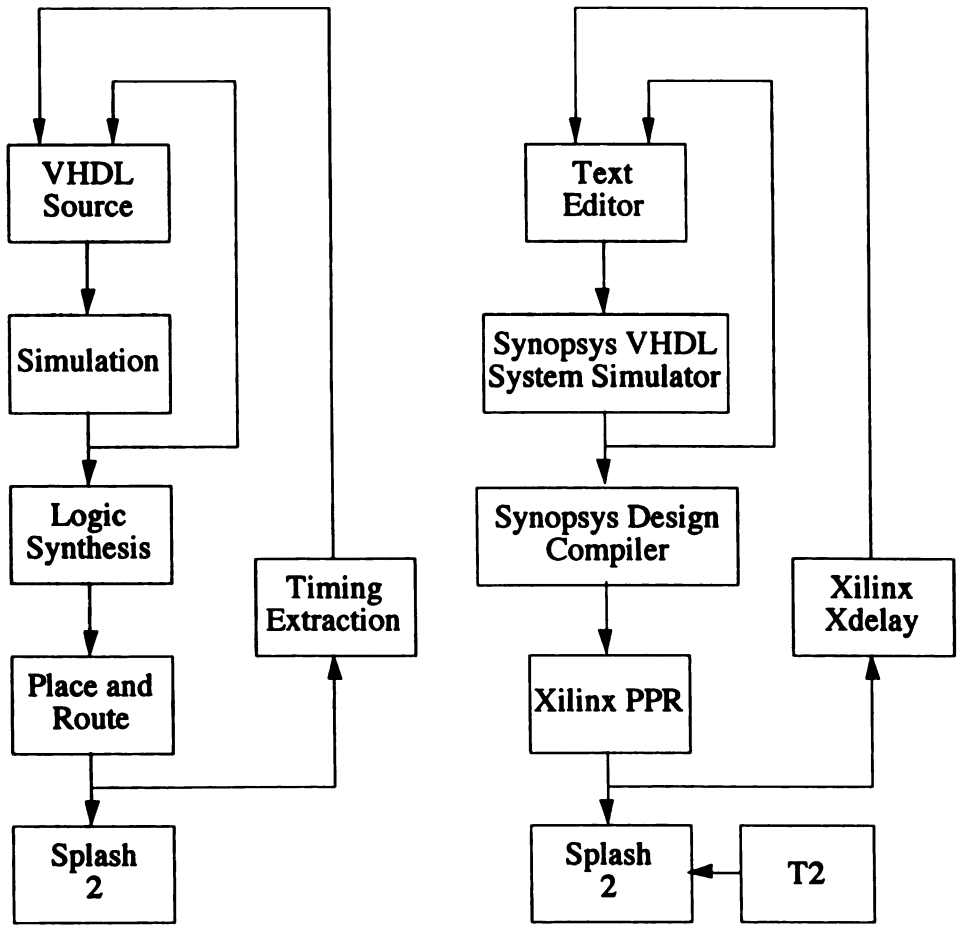


Figure 4.4. Design flow and tools

compared to workstations, this does represent a speed capable of running applications.

Another limitation of the Splash 2 involves incorporating memory subsystems and additional peripherals, features found in any workstation. These functions must be emulated on the Splash 2 using the Sun host. While slow, this would allow different schemes of virtual memory and caching techniques to be explored with the prototyped architecture. This is important as different architectures will utilize resources uniquely.

In order for the Dex-II to execute real applications, the instruction set would need to be enhanced to support additional instructions. The indicator for expandability in our synthesized design is the percentage of CLBs used. The percentages used in each PE are all under 50%. Xilinx predicts that the usage of CLBs can approach 75% before routing becomes impossible to achieve using automated design software. Thus, enhancements on this initial design are possible.

VHDL offers a powerful tool to realize architectures. Its high level language features allows designers to quickly design and integrate new systems. This high level language also introduces certain ambiguities. Just as high level language compilers may not generate efficient machine code, VHDL synthesizers have difficulties generating efficient logic. VHDL favors certain coding styles for different synthesis targets. In order to efficiently support systems that are dependent on synthesis results, such as Splash 2, these tools will need to be improved considerably.

This chapter has presented the results of mapping and implementing the Dex-II architectural specifications on the Splash 2 system using a series of simulation and synthesis tools. We now have an operational version of a modified Dex JR. and Dex-II on Splash 2. Various aspects of the process pose limitations, including support provided by the tools. Beyond the process, the implementation has provided timing and utilization results that help characterize this prototyping platform. Real applications are possible, however limitation due to FPGA logic and Splash resources remain.

Table 2. Summary of synthesis results

PE	Function	%CLB Used	Critical Time (ns)
0	Crossbar Control	1	14.9
1	Fetch	16	41.1
2	Fetch	19	44.4
3	Decode	42	54.8
4	Decode	46	55.7
5	Add/Sub Shifter	32	90.0
6	Add/Sub Shifter	31	80.8
7	Multiply	33	34.9
8	Memory	43	53.2
9	Memory	49	55.9
10	Multiply	33	33.9
11	Add/Sub Shifter	30	81.7
12	Add/Sub Shifter	31	84.8
13	Decode	45	44.9
14	Decode	43	53.8
15	Fetch	20	44
16	Fetch	16	41.3

CHAPTER 5

Test Programs

The ultimate goal of a processor is to execute programs. We can test different processors by running programs and comparing their execution times. In this study, we will compare the VLIW version, Dex-II, versus the original RISC design, Dex JR. A Fibonacci test program was implemented to verify operation of the Dex-II on the Splash 2 system. This program is hand compiled and loaded into memory using T2. Using memory dumps of the register file and memory, we can ensure that the processor and programs are executing correctly. A bubble sort program is also analyzed to better characterize the performance of our processor with a larger program. This program provides a better basis to compare the Dex-II versus the Dex JR. than the simple Fibonacci program.

5.1 The Runtime Environment

The runtime environment was controlled through the T2 debugger. T2 provides the interface to control all of the Splash 2 hardware. The interface allows us to download the bitstream pattern generated by the synthesis tools for the Dex-II design, program and read the memory for each PE, and control the system clock. This control allows us to step through the execution of a program clock by clock and examine the results.

The comparison of the Dex-II, our VLIW machine, to the original Dex JR., our RISC machine, will show the merits and shortcomings of the VLIW architecture. This

kind of comparison is necessary to evaluate the merits of architectural improvements. To evaluate the performance the Dex-II, we count the number of useful instructions it can execute per clock cycle. Useful instructions exclude NOPs. The repeated branch and condition code generation instruction, typically a SUB instruction, is counted as a single instruction. This metric is called IPC (Instructions Per Clock cycle). The inverse of the IPC gives us CPI (Clock cycles Per Instruction). Using the IPC or CPI, we can calculate how many MIPS (Millions of Instructions Per Second) the machine can execute if we know what the clock cycle time is. Although the MIPS rating can misrepresent the performance of the machine, it is still a useful tool in evaluating architectures with similar instruction sets. In order to improve the performance of the architecture, it is essential to increase the MIPS the machine can execute. This can be done by decreasing the cycle time, or increasing the IPC.

5.2 Fibonacci Sequence

The Fibonacci sequence is computed by adding the two previous members in the sequence to generate the current member of the sequence. The first two members of the sequence start with two ones. The Fibonacci program, which computes the Fibonacci sequence, is used to test the Dex-II. This program will initialize the register file by loading initial values into R1 and R2. R3 is then computed by adding R1 and R2 together. The registers are then shifted by loading the value of R2 into R1 and R3 into R2. The process is then repeated to compute the infinite series.

5.2.1 Dex-II Verification

The Fibonacci program exercises all of the hardware necessary for the Dex-II to be implemented on Splash 2. It shows correct utilization of the PE memory, both read and write, by successfully updating the register file and fetching of instructions. Synchronization of the four separate register files is achieved through the crossbar, as is

operand passing. The results of the Fibonacci calculation also verifies the correct mathematical execution in the adder unit.

Figure 5.1 shows an instruction by instruction register dump of the Fibonacci program used to test the Dex-II. After every six ticks of the global clock dumps the first five memory locations of PE3 and PE13. This represents the register files from the two processors. Note that we start examining the register file at clock tick 18, or the 3rd instruction cycle. This is due to the first instruction filling the pipeline in the Fetch and Decode stages. Figure 5.1 shows the results from the initial loading of the first two registers, and through a single loop of the Fibonacci program. We can see how each instruction affects the register file. This program execution clearly demonstrates the functional correctness of the Dex-II.

5.2.2 Program Implementation and Performance

The Fibonacci sequence is a relatively easy algorithm to implement. The code for the single RISC processor was used first to verify the synthesis steps. The VLIW version was then execute to verify the additional features of the Dex-II. Both of these codes are shown in Figure 5.2. The complete runtime results from both versions of the code are located in Appendix B.

There are six useful instructions being performed in the RISC model in six clock cycles giving us a IPC (Instructions Per Clock) of 1. The VLIW, however, performs the same six instructions in four clock cycles for an IPC rating of 1.5, or 2.775 MIPS. In the VLIW case, if we could eliminate the need for either processor to perform the BRA command, we could compact the code into three cycles for the theoretical IPC of 2. The first two instructions, however, are only performed once each time the program is executed. To get a better comparison, we must analyze the calculation loop.

Results read back from Splash 2		Instruction being executed			
T2:4> !Added RB 3 at tick 18 Board 0 PE 3 Offset 0 000000: 0000 0001 0001 000A 000A Board 0 PE 13 Offset 0 Words 32 000000: 0000 0001 0001 000A 000A		LDI	R1, 1	LDI	R2, 1
T2:5> ! Added RB 4 at tick 24 Board 0 PE 3 Offset 0 000000: 0000 0001 0001 0002 000A Board 0 PE 13 Offset 0 Words 32 000000: 0000 0001 0001 0002 000A	Loop	ADD	R3, R1, R2	NOP	
T2:6> ! Added RB 5 at tick 30 Board 0 PE 3 Offset 0 000000: 0000 0001 0002 0002 000A Board 0 PE 13 Offset 0 Words 32 000000: 0000 0001 0002 0002 000A		ST	R1, R2	ST	R2, R3
T2:7> ! Added RB 6 at tick 36 Board 0 PE 3 Offset 0 000000: 0000 0001 0002 0002 000A Board 0 PE 13 Offset 0 Words 32 000000: 0000 0001 0002 0002 000A		BRA	Loop	BRA	Loop
T2:8> ! Added RB 7 at tick 42 Board 0 PE 3 Offset 0 000000: 0000 0001 0002 0003 000A Board 0 PE 13 Offset 0 Words 32 000000: 0000 0001 0002 0003 000A	Loop	ADD	R3, R1, R2	NOP	

Figure 5.1. Fibonacci Execution Results

The VLIW still outperforms the RISC processor by reducing the number of cycles needed in the calculation loop. Note in Figure 5.2 that while the RISC machine uses four instructions, the VLIW is capable of producing the same results using only three. This represents a 33% increase in the performance of our new machine compared to the RISC by increasing the IPC to 1.3, or 2.47 MIPS. While lower then the overall comparison, this represents a more realistic gain in performance.

To demonstrate the flexibility of the architecture, we also attempted to tailor the architecture to our problem and get a faster design. Since the Fibonacci program only required the add function to be implemented as an execution unit, we removed the excess

RISC

```
        LDI   R1, 1
        LDI   R2, 1
Loop    ADD   R3, R1, R2
        ST    R1, R2
        ST    R2, R3
        BRA   Loop
```

VLIW

```
        LDI   R1, 1          LDI   R2, 1
Loop    ADD   R3, R1, R2     NOP
        ST    R1, R2         ST    R2, R3
        BRA   Loop           BRA   Loop
```

Figure 5.2. Two Fibonacci programs

functionality which improves the performance of the most critical PE. As a result, our cycle time was reduced to let our system clock run at 12.7 MHz with a peak RISC MIPS of 2.11 and a peak VLIW MIPS of 4.23. This represents a 14% increase in performance of both processors.

5.3 Bubble Sort

The bubble sort algorithm represents code that performs many comparisons, memory accesses, and several conditional branches. This kind of code introduces many NOP cycles that decrease the performance of the processor. The initial program is first introduced followed by an optimized program utilizing some of the techniques introduced previously. The results will give a better understanding of the necessity of compiler optimizations.

Figure 5.3 shows a non-optimized version of the bubble sort program for our RISC machine and Figure 5.4 is the same program on our VLIW version of the machine. As we can see, we initially reduce the total number of instructions from 23 to 19. Out of

```
1                   LDI     R1, 0
2                   LDI     R2, 0
3                   LDI     R3, Length
4                   LDI     R8, 1
5         Loop1     SUB     R4, R3, R1
6                   NOP
7                   BZ      End
8                   LD      R2, R1
9                   LD      R5, (R1)
10        Loop2     LD      R6, R2
11                  SUB     R4, R5, R6
12                  NOP
13                  BN      Swap
14        Check     ADD     R2, R2, R8
15                  SUB     R4, R2, R3
16                  NOP
17                  BN      Loop2
18                  ADD     R1, R1, R8
19                  BRA     Loop1
20        Swap      ST      (R1), R6
21                  ST      (R2), R5
22                  LD      R5, R6
23                  BRA     Check
24        End
```

Figure 5.3. RISC bubble sort program

these four, only two represent a significant gain in performance. The initialization routine was shortened from 4 to 2 instructions. While this is significant, the program spends a minimal amount of time in this routine. The loops, however, are where the bulk of the program execution takes place. The results here were marginal. The only improvements in performance is observed in Loop1 and the Swap routine, both reducing the number of instructions by one. This represents only an 11% increase in performance.

To optimize the VLIW machine, we begin by trying to move instructions into the NOP cycles. Instruction #6 can be moved up to instruction #4 since we either perform the instruction or end the program. Instruction #15 now can moved up to the NOP of instruction #13, however, we must take into account what happens if the conditional branch is true. In order to maintain the functionality, we must add compensation code at instruction #9 to undo our speculative execution. We can also move instruction #11 up to instruction #90, however, we would need to add two instructions to compensate in the

1		LDI	R1, 0	LDI	R2, 0
2		LDI	R3, Length	LDI	R8, 1
3	Loop1	SUB	R4, R3, R1	SUB	R4, R3, R1
4		NOP		NOP	
5		BZ	End	BZ	End
6		LD	R2, R1	LD	R5, (R1)
7	Loop2	LD	R6, R2	NOP	
8		SUB	R4, R5, R6	SUB	R4, R5, R6
9		NOP		NOP	
10		BN	Swap	BN	Swap
11	Check	ADD	R2, R2, R8	NOP	
12		SUB	R4, R2, R3	SUB	R4, R2, R3
13		NOP		NOP	
14		BN	Loop2	BN	Loop2
15		ADD	R1, R1, R8	NOP	
16		BRA	Loop1	BRA	Loop1
17	Swap	ST	(R1), R6	ST	(R2), R5
18		LD	R5, R6	NOP	
19		BRA	Check	BRA	Check
20	End				

Figure 5.4. VLIW bubble sort program

Swap routine. This would shorten one path and lengthen another. Since the number of times the Swap routine is called is dependent on the data set, making this substitution would not be beneficial without prior knowledge of the data.

The optimized version of the VLIW code shown in Figure 5.6 manages to shorten Loop1 to 3 instructions and Loop2 from 10 to 9 instructions. To be fair, we perform the same optimizations on our RISC processor. This time, we can only move instruction #8 to instruction #6. The RISC code is shown in Figure 5.5. The final comparison shows a total of 22 instructions for the RISC and 17 for the VLIW. Code size of the VLIW was reduced to 74% of the RISC code. If we consider the performance of the loops, we managed to reduce the total number of instructions from 18 down to 15.

The optimized VLIW code executes 21 useful instructions in 17 clock cycles for a IPC of 1.23. The RISC performs 20 useful instructions in 22 clock cycles for a IPC of .90. In this case, the high number of conditional branches with the lack of computation between sections would not benefit from the ability to schedule operations into the redundant control instructions. This penalty for conditional branches occurs whenever a

program executes several branches in succession. The lack of useful instructions between each section makes moving code across branches difficult to impossible.

It becomes evident that smart compilers are necessary if we want to see an effective VLIW processor utilized in the future. While smart compilers are needed for scheduling instructions in any superscalar processor, methods of scanning larger blocks of instructions during run time can be used to reduce the impact of bad compilers. However, run-time methods will require extra cycle time that may not be practical in a super-pipelined, superscalar architecture. Static scheduling will still be faster ultimately.

The programs used to characterize our architecture are simple examples. These programs allow us to visualize the effectiveness of our instruction set. They also allow us to see the shortcomings in how we deal with the various hazards. The bubble sort program, for instance, clearly shows that we need a better way to handle control instructions. The Splash 2 also does not support functions such as floating point arithmetic. To execute real applications and benchmarks, these functions would need to be emulated on the Sun host, or additional hardware must be added to the Splash 2.

```
1                LDI     R1, 0
2                LDI     R2, 0
3                LDI     R3, Length
4                LDI     R8, 1
5        Loop1   SUB     R4, R3, R1
6                LD      R2, R1
7                BZ      End
8                LD      R5, (R1)
9        Loop2   LD      R6, R2
10               SUB     R4, R5, R6
11               NOP
12               BN      Swap
13       Check   ADD     R2, R2, R8
14               SUB     R4, R2, R3
15               NOP
16               BN      Loop2
17               ADD     R1, R1, R8
18               BRA     Loop1
19       Swap    ST      (R1), R6
20               ST      (R2), R5
21               LD      R5, R6
22               BRA     Check
23       End
```

Figure 5.5. Optimized RISC bubble sort program

```
1                LDI     R1, 0                   LDI     R2, 0
2                LDI     R3, Length              LDI     R8, 1
3        Loop1   SUB     R4, R3, R1              SUB     R4, R3, R1
4                LD      R2, R1                  LD      R5, (R1)
5                BZ      End                     BZ      End
6        Loop2   LD      R6, R2                  NOP
7                SUB     R4, R5, R6              SUB     R4, R5, R6
8                SUB     R1, R1, R8              NOP
9                BN      Swap                    BN      Swap
10       Check   ADD     R2, R2, R8              NOP
11               SUB     R4, R2, R3              SUB     R4, R2, R3
12               ADD     R1, R1, R8              NOP
13               BN      Loop2                   BN      Loop2
14               BRA     Loop1                   BRA     Loop1
15       Swap    ST      (R1), R6                ST      (R2), R5
16               LD      R5, R6                  NOP
17               BRA     Check                   BRA     Check
18       End
```

Figure 5.6. Optimized VLIW bubble sort program

CHAPTER 6

Conclusions and Future Investigations

This chapter summarizes our findings and evaluates the success of our implementation of an ISP on the Splash 2. The possibilities of using Splash 2 as a tool in studying computer architectures and compilers is also presented. Lastly, proposals for future investigations and projects are offered as possibilities to extend the study of computer architectures on Splash 2.

6.1 The Dex-II Evaluation

The successful implementation of our design did achieve the goal of implementing a RISC and a VLIW architecture. The clock speed of 1.85 MHz allows the Dex-II to execute large programs in a reasonably short period of time. This allows for characterizing architectures and architectural features of real applications which would otherwise be prohibitive in simulation. Even so, the performance is too slow to be useful in practical applications.

The primary factor for poor cycle time stems from the fact the architecture suffers from an unbalanced pipeline. Empty slots evident in phases of each unit except the Decode stage signify a poorly balanced pipeline. This was the result of waiting for operands to be passed around on an inadequate interconnect array. The pipeline may be better balanced by adding functionality in the Fetch and Execute stages decreasing the

amount of instructions needed to complete tasks. Additional functionality, however, starts to drift away from the RISC paradigm, and other problems may arise.

Since the design is based on VHDL and automated synthesis for FPGAs, changes in the architecture can be quickly implemented as seen with the Fibonacci program. Additional hardware features can be added with relative ease. Support for register windowing can be added with the addition of some opcodes. Subroutines and jump commands can also be implemented onto the Fetch stages with relative ease. Specialized execution units can be programmed in an attempt to increase the performance of the processor for specific applications. Additional hardware to measure performance could also be added to monitor processor usage. This would provide invaluable data to support trace scheduling techniques.

6.2 VLIW vs. RISC

The Dex-II implements two RISC processors side by side on the Splash 2. This design permits two RISC instructions to execute in a single clock cycle. Although there are some data and control hazards that must be avoided, the VLIW architecture is successful in increasing the IPC. For the VLIW architecture, we achieve an IPC of 1.50 for the Fibonacci program and 1.23 for the Bubble Sort program. The RISC versions of the same code achieved an IPC of 1.00 and .90, respectively.

The prototype processor indicated problems with our control scheme. The results from our program analysis focuses attention at the double branch instruction which was required to keep both processors in synchronization. This problem did not exist for the single RISC processor model. Future designs with this architecture must successfully implement a more unified control path to control the multiple data paths.

6.3 Splash 2 Evaluation

Computers were originally designed as machines that could solve many problems by using different programs. With FPGAs, programmers and designers now have the ability to alter the machine to fit the problem. The success of these arrays will be dependent upon the resources available to each FPGA. While logic emulators built with arrays of FPGAs can be used to prototype circuits, it will be the resources such as the memory and the crossbar of the Splash 2 that will offer a new level of flexibility to prototype more complex architectures.

The bottleneck for successful design on the Splash 2 was the lack of communication between chips to implement bus structures. While we were able to achieve a respectable clock rate of 11 MHz from the FPGA, the six-cycle division forced by communication restrictions effectively reduced our cycle time to a little under 2 MHz. In order for faster speeds to be obtained, the level of interconnects needs to be enhanced through a crossbar with greater functionality as well as FPGAs with greater I/O capabilities.

The lack of floating point processing is also a factor that will limit the usefulness of FPGA-based arrays. Floating point processing is required in many applications in both science and engineering. These types of computations are usually highly repetitive and can be structured in such a way to take advantage of the reconfigurable architecture.

Although the use of synthesis tools considerably shortened the design cycle, these tools were not perfect. As the efficiency of the synthesis tools and number of FPGA libraries increase, these synthesis tools will improve the overall performance of reconfigurable computing systems. The size of our design generated by the synthesis tools utilized less than 50% of the CLB resources. This suggests that our design is close to the optimal speed as the routing tools had plenty of space to route signals within the

FPGA. By adding more functionality to our designs, these tools become even more significant as they directly impact the speed of the design.

6.4 Future Investigations

Although the practical uses are limited at this time, the Dex-II does offer substantial research opportunities. FPGA-based systems, for example, can provide an invaluable tool for compiler testing. Different architectures can be simulated on the Splash 2 to execute compiled codes for various scheduling techniques. As reconfigurable processors become a reality, new compiler technology can be rapidly adapted. Synthesis tools will also play a large role in the success of reconfigurable processors. Compilers no longer need to be written and optimized for a machine, but the machine can now be optimized to support the compiler.

Speculative execution is another method to decrease execution times. Guarded execution and complicated shadow registers can be used to let speculative instructions run their course. This method is not very feasible in our design due to the restrictions of the board itself, however each board as a whole may be configured as a shadow system. A system can be implemented where the host is used to synchronize several Splash boards executing a program. When a program branch is encountered, both routes are sent off to two boards configured as a processor. The host needs to maintain the actual state of the machine but can orchestrate hundreds of these processors.

Threaded programming is also an emerging software methodology that would support a co-processor approach. Separate threads can be run on independent boards without the communication overhead to track the actions of other threads. Memory would be managed by the host and treated as a cached memory system.

The Dex-II is an aggressive utilization of the resources available from the Splash 2. The architecture is far from perfect, however. The design is limited to a 16-bit architecture until even larger chips make it possible to compact larger designs and

provide more communications. However, the ability to change designs so quickly does make the Splash 2 and general FPGA-based arrays very attractive. A key factor in RISC development was the shortened turnaround time for each new architecture. FPGA-based systems would allow new architectures to be prototyped quickly. This provides a powerful tool for computer and compiler designers to test and implement new designs and ideas.

APPENDICES

Appendix A

Synthesis Code

A1 Control1.vhd (PE1)

```
architecture Fetch_left of Xilinx_Processing_Part is
   signal PC: Bit_Vector (17 downto 0);
   signal loadaddress: Bit_Vector (15 downto 0);
   signal Ix: Bit_Vector (31 downto 0);
   signal Id: Bit_Vector (31 downto 0);
   signal Ifetch: Bit_Vector (31 downto 0);
   signal increment_pc: Bit;
   signal branchbit: bit := '0';
   signal reset:bit := '0';
   signal local_clk: integer range 0 to 6;
   signal one: Bit_Vector (17 downto 0);
   signal CC: Bit_Vector (3 downto 0);

   signal Right: Bit_Vector (31 downto 0);

   signal mdr: Bit_Vector (15 downto 0);
   signal Xbarin, Xbarout: Bit_Vector (35 downto 0);
   signal Xbarenable: Bit_Vector (4 downto 0);


begin

RIGHTIN:  for i in 0 to 15 GENERATE
          Pad_Input(XP_right(i), right(i));
END GENERATE RIGHTIN;

RIGHTOUT:  for i in 16 to 31 GENERATE
          Pad_Output(XP_right(i), right(i));
END GENERATE RIGHTOUT;

   XP_Mem_RD_L                  <= '0';
   increment_pc  <= '1';
   one <= "000000000000000001";

   process
   begin
```

```
        wait until XP_Clk'Event and XP_Clk = '1';
        local_clk <= local_clk + 1;
        if local_clk = 5 then
                local_clk <= 0;
        end if;

        Pad_Input (XP_Mem_D, mdr);
        Pad_Output (XP_Mem_A, PC);

        if local_clk = 1 then
                Ifetch(31 downto 16) <= mdr;
                right(31 downto 16) <= mdr;
                Xbarenable <= "01111";
        end if;

        if local_clk = 2 then
                Ifetch (15 downto 0) <= right (15 downto 0);
        end if;

        if local_clk = 3 then
                PC <= PC + one;
                if (Ifetch(31 downto 30) = 01) then
                        PC(15 downto 0) <= Ifetch (25 downto 10);
                end if;
        end if;


  end process;



  XP_Mem_WR_L                   <= '1';
  XP_HS0                <= 'Z';
  XP_GOR_Result         <= '0';
  XP_GOR_Valid          <= '0';
  XP_Int                <= '0';

end Fetch_left;
```

A2 Control2.vhd (PE2)

```
architecture Fetch_right of Xilinx_Processing_Part is
  signal PC: Bit_Vector (17 downto 0);
  signal one: Bit_Vector (17 downto 0);
  signal Ifetch: Bit_Vector (31 downto 0);
  signal reset:bit := '0';
  signal local_clk: integer range 0 to 6;
  signal CC: Bit_Vector (3 downto 0);

  signal Left: Bit_Vector (31 downto 0);
  signal mdr: Bit_Vector (15 downto 0);
  signal Xbarout, Xbarin: Bit_Vector (35 downto 0);
  signal Xbarenable: Bit_Vector (4 downto 0);
```

```
begin

LEFTIN:  for i in 16 to 31 GENERATE
        Pad_Input(XP_left(i), Left(i));
END GENERATE LEFTIN;

LEFTOUT:  for i in 0 to 15 GENERATE
        Pad_Output(XP_left(i), Left(i));
END GENERATE LEFTOUT;


   XP_Mem_RD_L                  <= '0';
   one <= "0000000000000000001";
   XP_Xbar_EN_L <= Xbarenable;

   process
   begin

        wait until XP_Clk'Event and XP_Clk = '1';

        Pad_Output(XP_Right(31 downto 0), Ifetch);
        local_clk <= local_clk + 1;
        if local_clk = 5 then
                local_clk <= 0;
        end if;

        Pad_Input (XP_Mem_D, mdr);
        Pad_Output (XP_Mem_A, PC);
        Xbarenable <= "11111";

        if local_clk = 0 then
                XP_LED <= '1';
        end if;

        if local_clk = 1 then
                Ifetch(15 downto 0) <= mdr;
                left(15 downto 0) <= mdr;
                Xbarenable <= "01111";
                XP_LED <= '0';
        end if;

        if local_clk = 2 then
                Ifetch (31 downto 16) <= left (31 downto 16);
                CC <= Xbarin (35 downto 32);
                XP_LED <= '1';
        end if;

        if local_clk = 3 then
                PC <= PC + one;
                if Ifetch(31 downto 30) = 01 then
                        if ((Ifetch(29 downto 27)=000) or (CC(3 downto 1)) = Ifetch (29 downto 27))
                        then PC(15 downto 0) <= Ifetch (25 downto 10);
                        end if;
                end if;
                XP_LED <= '0';
        end if;

   end process;
```

```
   XP_Mem_WR_L              <= '1';
   XP_HS0           <= 'Z';
   XP_GOR_Result    <= '0';
   XP_GOR_Valid     <= '0';
   XP_Int           <= '0';

end Fetch_right;
```

A3 Decode1.vhd (PE3)

```
architecture Decode_left of Xilinx_Processing_Part is

   signal Itemp: Bit_Vector (35 downto 0);
   signal Ix: Bit_Vector (35 downto 0);
   signal Id: Bit_Vector (35 downto 0);
   signal Ifetch: Bit_Vector (35 downto 0);
   signal reset:bit;
   signal local_clk: integer range 0 to 6;

   signal Left: Bit_Vector (35 downto 0);
   signal Right: Bit_Vector (35 downto 0);

   signal Xbar_en: Bit_Vector (4 downto 0);  -- enable bit for Xbar (low)
   signal zero: Bit_Vector (4 downto 0);

   signal RdrL: Bit_Vector (15 downto 0);
   signal Mdr: Bit_Vector (15 downto 0);
   signal RarL: Bit_Vector (17 downto 0);
   signal Xbarin, Xbarout: Bit_Vector (35 downto 0);

   signal OpA: Bit_Vector (15 downto 0);
   signal temp: Bit_Vector (15 downto 0);
--   signal result: Bit_Vector (15 downto 0);

   signal Muxin0, Muxin1, Muxin2, Muxin3, Muxresult: Bit_Vector (15 downto 0);

begin


   Select0 : mux4_1H port map (Muxin0(0), Muxin1(0), Muxin2(0), Muxin3(0),
                               Ix(30), Ix (31), Muxresult (0));
   Select1 : mux4_1H port map (Muxin0(1), Muxin1(1), Muxin2(1), Muxin3(1),
                               Ix(30), Ix (31), Muxresult (1));
   Select2 : mux4_1H port map (Muxin0(2), Muxin1(2), Muxin2(2), Muxin3(2),
                               Ix(30), Ix (31), Muxresult (2));
   Select3 : mux4_1H port map (Muxin0(3), Muxin1(3), Muxin2(3), Muxin3(3),
                               Ix(30), Ix (31), Muxresult (3));
   Select4 : mux4_1H port map (Muxin0(4), Muxin1(4), Muxin2(4), Muxin3(4),
                               Ix(30), Ix (31), Muxresult (4));
   Select5 : mux4_1H port map (Muxin0(5), Muxin1(5), Muxin2(5), Muxin3(5),
```

```
                                    Ix(30), Ix (31), Muxresult (5));
  Select6 : mux4_1H port map (Muxin0(6), Muxin1(6), Muxin2(6), Muxin3(6),
                                    Ix(30), Ix (31), Muxresult (6));
  Select7 : mux4_1H port map (Muxin0(7), Muxin1(7), Muxin2(7), Muxin3(7),
                                    Ix(30), Ix (31), Muxresult (7));
  Select8 : mux4_1H port map (Muxin0(8), Muxin1(8), Muxin2(8), Muxin3(8),
                                    Ix(30), Ix (31), Muxresult (8));
  Select9 : mux4_1H port map (Muxin0(9), Muxin1(9), Muxin2(9), Muxin3(9),
                                    Ix(30), Ix (31), Muxresult (9));
  Select10 : mux4_1H port map (Muxin0(10), Muxin1(10), Muxin2(10), Muxin3(10),
                                    Ix(30), Ix (31), Muxresult (10));
  Select11 : mux4_1H port map (Muxin0(11), Muxin1(11), Muxin2(11), Muxin3(11),
                                    Ix(30), Ix (31), Muxresult (11));
  Select12 : mux4_1H port map (Muxin0(12), Muxin1(12), Muxin2(12), Muxin3(12),
                                    Ix(30), Ix (31), Muxresult (12));
  Select13 : mux4_1H port map (Muxin0(13), Muxin1(13), Muxin2(13), Muxin3(13),
                                    Ix(30), Ix (31), Muxresult (13));
  Select14 : mux4_1H port map (Muxin0(14), Muxin1(14), Muxin2(14), Muxin3(14),
                                    Ix(30), Ix (31), Muxresult (14));
  Select15 : mux4_1H port map (Muxin0(15), Muxin1(15), Muxin2(15), Muxin3(15),
                                    Ix(30), Ix (31), Muxresult (15));

--Xbarin is from the xbar where xbarout is the value being sent to the xbar...
  Pad_Xbar (XP_Xbar, Xbarout,Xbarin, Xbar_en);
  XP_Xbar_EN_L          <= Xbar_en;
  Xbarout (31 downto 16) <= Muxresult;
  Xbarout (15 downto 0) <= Muxresult;

  Muxin1 <= OpA;

  zero <= "00000";

  Pad_Output (XP_right, IFetch);


   process
   begin

         wait until XP_Clk'Event and XP_Clk = '1';

         Pad_Input (XP_left, Ifetch);
         Pad_Output (XP_Mem_A, RarL);
         Pad_InOut (XP_Mem_D, Rdrl, Mdr, '0');
         XP_Mem_RD_L <= '1';
         XP_Mem_WR_L <= '1';
         temp <= Xbarin (15 downto 0);                    --Syncing bottom

         local_clk <= local_clk + 1;
         if local_clk = 5 then
                  local_clk <= 0;
         end if;


         if local_clk = 0 then
                  Pad_Output (XP_Mem_A (4 downto 0), Id (19 downto 15));
                  XP_Mem_RD_L <= '0';
                  Xbar_en <= "10000";
         end if;
```

```
        if local_clk = 1 then
                XP_Mem_RD_L <= '0';
                Pad_InOut (XP_Mem_D, Rdrl, OpA, '0');
                Muxin2 <= Xbarin(31 downto 16);
                Xbar_en <= "10000";
        end if;

        if local_clk = 2 then
                -- get data from Xbar, save the bottom half to write next cycle.
                Muxin3 <= Xbarin(31 downto 16);
                Muxin0 <= Xbarin(15 downto 0);
                Xbar_en <= "11100";
        end if;

        if local_clk = 3 then
                if Ix(9 downto 5) = Id(19 downto 15) then OpA <= Muxresult;
                end if;
                if Ix(4 downto 0) = Id(19 downto 15) then OpA <= Xbarin(15 downto 0);
                end if;
                Ix (31 downto 30) <= "01";
--              temp <= Xbarin (15 downto 0);                   --Syncing bottom
                Pad_Output (XP_Mem_A (4 downto 0), Ix(9 downto 5));
                if (Ix (9 downto 5) /= zero) then
                        Pad_InOut (XP_Mem_D, Muxresult, Mdr, '1');
                        XP_Mem_WR_L <= '0';
                end if;
                Xbar_en <= "11111";
        end if;

        if local_clk = 4 then
        -- need to write bottom Muxresult
                Xbar_en <= "11111";
                Pad_Input (XP_left, Id);
--              Id <= Ifetch;
                Itemp <= Id;
                if (Ix (4 downto 0) /= zero) then
                        Pad_Output (XP_Mem_A (4 downto 0), Ix(4 downto 0));
                        Pad_InOut (XP_Mem_D, temp, Mdr, '1');
                        XP_Mem_WR_L <= '0';
                end if;
        end if;

        if local_clk = 5 then
                Ix <= Itemp;
        end if;


    end process;


    XP_HS0              <= 'Z';
    XP_GOR_result       <= '0';
    XP_GOR_Valid        <= '0';
    XP_Int              <= '0';
    XP_LED              <= '1';

end Decode_left;
```

A4 Decode2.vhd (PE4)

```
architecture Decode_right of Xilinx_Processing_Part is

  signal Ix: Bit_Vector (35 downto 0);
  signal Id: Bit_Vector (35 downto 0);
  signal Ifetch: Bit_Vector (35 downto 0);
  signal reset:bit;
  signal local_clk: integer range 0 to 6;

  signal Left: Bit_Vector (35 downto 0);
  signal Right: Bit_Vector (35 downto 0);

  signal WriteEnable: Bit;
  signal Xbar_en: Bit_Vector (4 downto 0) ;  -- enable bit for Xbar (low)
  signal zero : Bit_Vector (4 downto 0) ;

  signal Mdr: Bit_Vector (15 downto 0);
  signal RdrL: Bit_Vector (15 downto 0);
  signal RarL: Bit_Vector (17 downto 0);
  signal Xbarin, Xbarout: Bit_Vector (35 downto 0);

  signal OpB: Bit_Vector (15 downto 0);
  signal temp: Bit_Vector (15 downto 0);

  signal Muxin0, Muxin1, Muxin2, Muxin3, Muxresult: Bit_Vector (15 downto 0);


begin

  Select0 : mux4_1H port map (Muxin0(0), Muxin1(0), Muxin2(0), Muxin3(0),
                              Ix(30), Ix (31), Muxresult (0));
  Select1 : mux4_1H port map (Muxin0(1), Muxin1(1), Muxin2(1), Muxin3(1),
                              Ix(30), Ix (31), Muxresult (1));
  Select2 : mux4_1H port map (Muxin0(2), Muxin1(2), Muxin2(2), Muxin3(2),
                              Ix(30), Ix (31), Muxresult (2));
  Select3 : mux4_1H port map (Muxin0(3), Muxin1(3), Muxin2(3), Muxin3(3),
                              Ix(30), Ix (31), Muxresult (3));
  Select4 : mux4_1H port map (Muxin0(4), Muxin1(4), Muxin2(4), Muxin3(4),
                              Ix(30), Ix (31), Muxresult (4));
  Select5 : mux4_1H port map (Muxin0(5), Muxin1(5), Muxin2(5), Muxin3(5),
                              Ix(30), Ix (31), Muxresult (5));
  Select6 : mux4_1H port map (Muxin0(6), Muxin1(6), Muxin2(6), Muxin3(6),
                              Ix(30), Ix (31), Muxresult (6));
  Select7 : mux4_1H port map (Muxin0(7), Muxin1(7), Muxin2(7), Muxin3(7),
                              Ix(30), Ix (31), Muxresult (7));
  Select8 : mux4_1H port map (Muxin0(8), Muxin1(8), Muxin2(8), Muxin3(8),
                              Ix(30), Ix (31), Muxresult (8));
  Select9 : mux4_1H port map (Muxin0(9), Muxin1(9), Muxin2(9), Muxin3(9),
                              Ix(30), Ix (31), Muxresult (9));
  Select10 : mux4_1H port map (Muxin0(10), Muxin1(10), Muxin2(10), Muxin3(10),
```

```
                                    Ix(30), Ix (31), Muxresult (10));
Select11 : mux4_1H port map (Muxin0(11), Muxin1(11), Muxin2(11), Muxin3(11),
                                    Ix(30), Ix (31), Muxresult (11));
Select12 : mux4_1H port map (Muxin0(12), Muxin1(12), Muxin2(12), Muxin3(12),
                                    Ix(30), Ix (31), Muxresult (12));
Select13 : mux4_1H port map (Muxin0(13), Muxin1(13), Muxin2(13), Muxin3(13),
                                    Ix(30), Ix (31), Muxresult (13));
Select14 : mux4_1H port map (Muxin0(14), Muxin1(14), Muxin2(14), Muxin3(14),
                                    Ix(30), Ix (31), Muxresult (14));
Select15 : mux4_1H port map (Muxin0(15), Muxin1(15), Muxin2(15), Muxin3(15),
                                    Ix(30), Ix (31), Muxresult (15));

Pad_Xbar (XP_Xbar, Xbarout, Xbarin, Xbar_en);
XP_Xbar_EN_L          <= Xbar_en;
Xbarout(31 downto 16) <= Muxresult;
Xbarout(15 downto 0) <= Muxresult;

Muxin1 <= OpB;

zero <= "00000";
Rarl <= "0000000000000000";
Rdrl <= "0000000000000000";

Pad_Output (XP_right, Id);

 process
 begin

        wait until XP_Clk'Event and XP_Clk = '1';

        Pad_Input (XP_left, Ifetch);
        Pad_Output (XP_Mem_A, RarL);
        Pad_InOut (XP_Mem_D, Rdrl, Mdr, '0');
        XP_Mem_RD_L <= '1';
        XP_Mem_WR_L <= '1';
        temp <= Xbarin (15 downto 0);
        Ix <= Ix;

        local_clk <= local_clk + 1;

        if local_clk = 5 then
                local_clk <= 0;
        end if;

        if local_clk = 0 then
                Pad_Output (XP_Mem_A (4 downto 0), Id (14 downto 10));
                XP_Mem_RD_L <= '0';
                Xbar_en <= "10000";
        end if;

        if local_clk = 1 then
                XP_Mem_RD_L <= '0';
                Pad_InOut (XP_Mem_D, Rdrl, OpB, '0');
                Muxin2 <= Xbarin(31 downto 16);
                Xbar_en <= "10000";
        end if;

        if local_clk = 2 then
```

```
        -- get data from Xbar, save the bottom half to write next cycle.
                Muxin3 <= Xbarin(15 downto 0);
                Muxin0 <= Xbarin(31 downto 16);
                Xbar_en <= "11100";
        end if;

        if local_clk = 3 then
        -- need to write secondary Muxresult
                if Ix(9 downto 5) = Id(14 downto 10) then OpB <= Muxresult;
                end if;
                if Ix(4 downto 0) = Id(14 downto 10) then OpB <= Xbarin(15 downto 0);
                end if;
                Ix(31 downto 30) <= "01";
                temp <= Xbarin (15 downto 0);
                XP_Mem_WR_L <= '1';
                if (Ix (9 downto 5) /= zero) then
                        Pad_Output (XP_Mem_A (4 downto 0), Ix(9 downto 5));
                        Pad_InOut (XP_Mem_D, Muxresult, Mdr, '1');
                        XP_Mem_WR_L <= '0';
                end if;
                Xbar_en <= "11111";
        end if;

        if local_clk = 4 then
        -- need to write secondary Muxresult
                Xbar_en <= "11111";
                XP_Mem_WR_L <= '1';
                if (Ix (4 downto 0) /= zero) then
                        Pad_Output (XP_Mem_A (4 downto 0), Ix(4 downto 0));
                        Pad_InOut (XP_Mem_D, temp, Mdr, '1');
                        XP_Mem_WR_L <= '0';
                end if;
        end if;

        if local_clk = 5 then
                Pad_Input (XP_left, Id);
                Ix <= Id;
        end if;


    end process;


    XP_HS0              <= 'Z';
    XP_GOR_result       <= '0';
    XP_GOR_Valid        <= '0';
    XP_Int              <= '0';
    XP_LED              <= '1';

end Decode_right;
```

A5 Execute1.vhd (PE5)

```
architecture Execute1 of Xilinx_Processing_Part is

   signal Id: Bit_Vector (35 downto 0);
   signal Ix: Bit_Vector (35 downto 0);
   signal reset:bit := '0';
   signal local_clk: integer range 0 to 6;

   signal Left: Bit_Vector (35 downto 0);
   signal Right: Bit_Vector (31 downto 0);

   signal OpA: Bit_Vector (15 downto 0);
   signal OpB: Bit_Vector (15 downto 0);
   signal Muxin1,Muxin2,Muxin3,Muxin4: Bit_Vector (15 downto 0);
   signal Xbarin, Xbarout: Bit_Vector(35 downto 0);
   signal Xbar_en: Bit_Vector (4 downto 0);

         -- Muxin2 is not used in this pe and should be removed later...

   signal Result: Bit_Vector (15 downto 0);
   signal zero: Bit_Vector (15 downto 0);
   signal CondCode : Bit_Vector (3 downto 0);


begin

-- Adder unit
  addsub : adsu16h port map (OpA, OpB, Ix(27), Muxin1, CondCode(3));


-- Channel one of the results from the adder or shifter onto the xbar
-- depending on opcode selected and desired function.

  Select0 : mux4_1H port map (Muxin1(0), Muxin1(0), Muxin3(0), Muxin4(0),
                              Ix(27), Ix (28), Result (0));
  Select1 : mux4_1H port map (Muxin1(1), Muxin1(1), Muxin3(1), Muxin4(1),
                              Ix(27), Ix (28), Result (1));
  Select2 : mux4_1H port map (Muxin1(2), Muxin1(2), Muxin3(2), Muxin4(2),
                              Ix(27), Ix (28), Result (2));
  Select3 : mux4_1H port map (Muxin1(3), Muxin1(3), Muxin3(3), Muxin4(3),
                              Ix(27), Ix (28), Result (3));
  Select4 : mux4_1H port map (Muxin1(4), Muxin1(4), Muxin3(4), Muxin4(4),
                              Ix(27), Ix (28), Result (4));
  Select5 : mux4_1H port map (Muxin1(5), Muxin1(5), Muxin3(5), Muxin4(5),
                              Ix(27), Ix (28), Result (5));
  Select6 : mux4_1H port map (Muxin1(6), Muxin1(6), Muxin3(6), Muxin4(6),
                              Ix(27), Ix (28), Result (6));
  Select7 : mux4_1H port map (Muxin1(7), Muxin1(7), Muxin3(7), Muxin4(7),
                              Ix(27), Ix (28), Result (7));
  Select8 : mux4_1H port map (Muxin1(8), Muxin1(8), Muxin3(8), Muxin4(8),
                              Ix(27), Ix (28), Result (8));
  Select9 : mux4_1H port map (Muxin1(9), Muxin1(9), Muxin3(9), Muxin4(9),
                              Ix(27), Ix (28), Result (9));
  Select10 : mux4_1H port map (Muxin1(10), Muxin1(10), Muxin3(10), Muxin4(10),
                              Ix(27), Ix (28), Result (10));
  Select11 : mux4_1H port map (Muxin1(11), Muxin1(11), Muxin3(11), Muxin4(11),
                              Ix(27), Ix (28), Result (11));
  Select12 : mux4_1H port map (Muxin1(12), Muxin1(12), Muxin3(12), Muxin4(12),
```

```
                                    Ix(27), Ix (28), Result (12));
Select13 : mux4_1H port map (Muxin1(13), Muxin1(13), Muxin3(13), Muxin4(13),
                                    Ix(27), Ix (28), Result (13));
Select14 : mux4_1H port map (Muxin1(14), Muxin1(14), Muxin3(14), Muxin4(14),
                                    Ix(27), Ix (28), Result (14));
Select15 : mux4_1H port map (Muxin1(15), Muxin1(15), Muxin3(15), Muxin4(15),
                                    Ix(27), Ix (28), Result (15));

zero <= "0000000000000000";

Pad_Xbar (XP_Xbar, Xbarout, Xbarin, Xbar_en);
XP_Xbar_EN_L <= Xbar_en;

Xbarout (31 downto 16) <= result;
Xbarout (15 downto 0) <= result;
Xbarout (35 downto 32) <= CondCode;
Pad_Output (XP_right, Id);

 process

 begin

        wait until XP_Clk'Event and XP_Clk = '1';
        local_clk <= local_clk + 1;
        if local_clk = 5 then
                local_clk <= 0;
        end if;

        Pad_Input (XP_left, Left);

        if local_clk = 0 then
-- Compute and broadcast results.
                -- >>>>>>> Shift left <<<<<<<<
                Muxin3 (15 downto 1) <= OpA (14 downto 0);
                Muxin3 (0) <= '0';

                -- >>>>>>>>> Shift right sign extension <<<<<<<<<<<
                Muxin4 (14 downto 0) <= OpA (15 downto 1);
                Muxin4 (15) <= OpA(15);
        end if;

        if local_clk = 1 then
                Pad_Input (XP_left, Id);
                if Result = zero then CondCode(1) <= '1';
                end if;
                if Result (15) = '1' then CondCode(2) <= '1';
                end if;
        end if;

        if local_clk = 2 then
        end if;

        if local_clk = 3 then
                Xbar_en <= "10000";
        end if;

        if local_clk = 4 then
-- Latch new ops from the decode stage!!
```

```
                Opa <= Xbarin (31 downto 16);
                Opb <= Xbarin (15 downto 0);
                Xbar_en <= "11111";
        end if;

        if local_clk = 5 then
                Ix <= Id;
        end if;


    end process;

--    XP_Left               <= TriState (XP_Left);
--    XP_Right              <= TriState (XP_Right);
    XP_Mem_A              <= TriState (XP_Mem_A);
    XP_Mem_D              <= TriState (XP_Mem_D);
    XP_Mem_RD_L                   <= '1';
    XP_Mem_WR_L                   <= '1';
    XP_HS0                <= 'Z';
    XP_GOR_Result         <= '0';
    XP_GOR_Valid          <= '0';
    XP_Int                <= '0';
--    XP_Xbar_EN_L          <= "11111";
    XP_LED                <= '1';

end Execute1;
```

A6 Execute2.vhd (PE6)

```
architecture Execute2 of Xilinx_Processing_Part is

    signal Id: Bit_Vector (35 downto 0);
    signal Ix: Bit_Vector (35 downto 0);
    signal reset:bit;
    signal local_clk: integer range 0 to 6;

    signal Left: Bit_Vector (35 downto 0);
    signal Right: Bit_Vector (31 downto 0);

    signal OpA: Bit_Vector (15 downto 0);
    signal OpB: Bit_Vector (15 downto 0);
    signal Muxin1,Muxin2,Muxin3,Muxin4: Bit_Vector (15 downto 0);
    signal Xbarin, Xbarout: Bit_Vector(35 downto 0);
    signal Xbar_en: Bit_Vector (4 downto 0);

        -- Muxin2 is not used in this pe and should be removed later...

    signal Result: Bit_Vector (15 downto 0);
    signal CondCode : Bit_Vector (3 downto 0);
    signal zero: Bit_Vector (15 downto 0);
```

```
begin

-- Adder unit
  addsub : adsu16h port map (OpA, OpB, Ix(27), Muxin1, CondCode(3));


-- Channel one of the results from the adder or shifter onto the xbar
-- depending on opcode selected and desired function.

  Select0 : mux4_1H port map (Muxin1(0), Muxin1(0), Muxin3(0), Muxin4(0),
                              Ix(27), Ix (28), Result (0));
  Select1 : mux4_1H port map (Muxin1(1), Muxin1(1), Muxin3(1), Muxin4(1),
                              Ix(27), Ix (28), Result (1));
  Select2 : mux4_1H port map (Muxin1(2), Muxin1(2), Muxin3(2), Muxin4(2),
                              Ix(27), Ix (28), Result (2));
  Select3 : mux4_1H port map (Muxin1(3), Muxin1(3), Muxin3(3), Muxin4(3),
                              Ix(27), Ix (28), Result (3));
  Select4 : mux4_1H port map (Muxin1(4), Muxin1(4), Muxin3(4), Muxin4(4),
                              Ix(27), Ix (28), Result (4));
  Select5 : mux4_1H port map (Muxin1(5), Muxin1(5), Muxin3(5), Muxin4(5),
                              Ix(27), Ix (28), Result (5));
  Select6 : mux4_1H port map (Muxin1(6), Muxin1(6), Muxin3(6), Muxin4(6),
                              Ix(27), Ix (28), Result (6));
  Select7 : mux4_1H port map (Muxin1(7), Muxin1(7), Muxin3(7), Muxin4(7),
                              Ix(27), Ix (28), Result (7));
  Select8 : mux4_1H port map (Muxin1(8), Muxin1(8), Muxin3(8), Muxin4(8),
                              Ix(27), Ix (28), Result (8));
  Select9 : mux4_1H port map (Muxin1(9), Muxin1(9), Muxin3(9), Muxin4(9),
                              Ix(27), Ix (28), Result (9));
  Select10 : mux4_1H port map (Muxin1(10), Muxin1(10), Muxin3(10), Muxin4(10),
                              Ix(27), Ix (28), Result (10));
  Select11 : mux4_1H port map (Muxin1(11), Muxin1(11), Muxin3(11), Muxin4(11),
                              Ix(27), Ix (28), Result (11));
  Select12 : mux4_1H port map (Muxin1(12), Muxin1(12), Muxin3(12), Muxin4(12),
                              Ix(27), Ix (28), Result (12));
  Select13 : mux4_1H port map (Muxin1(13), Muxin1(13), Muxin3(13), Muxin4(13),
                              Ix(27), Ix (28), Result (13));
  Select14 : mux4_1H port map (Muxin1(14), Muxin1(14), Muxin3(14), Muxin4(14),
                              Ix(27), Ix (28), Result (14));
  Select15 : mux4_1H port map (Muxin1(15), Muxin1(15), Muxin3(15), Muxin4(15),
                              Ix(27), Ix (28), Result (15));


  Pad_Output (XP_right, Id);
  Pad_Xbar (XP_Xbar, Xbarout, Xbarin, Xbar_en);
  XP_Xbar_EN_L <= Xbar_en;

  Xbarout (31 downto 16) <= result;
  Xbarout (15 downto 0) <= result;
  Xbarout (35 downto 32) <= CondCode;

zero <= "0000000000000000";

  process
  begin

        wait until XP_Clk'Event and XP_Clk = '1';
```

```
        local_clk <= local_clk + 1;
        if local_clk = 5 then
                local_clk <= 0;
        end if;

        Pad_Input (XP_left, Left);

        if local_clk = 0 then
-- Compute results.

                -- >>>>>>> Shift left <<<<<<<<
                Muxin3 (15 downto 1) <= OpA (14 downto 0);
                Muxin3 (0) <= '0';

                -- >>>>>>>> Shift right sign extension <<<<<<<<<<
                Muxin4 (14 downto 0) <= OpA (15 downto 1);
                Muxin4 (15) <= OpA(15);
        end if;

        if local_clk = 1 then
                if Result = zero then CondCode(1) <= '1';
                end if;
                if Result (15) = '1' then CondCode(2) <= '1';
                end if;
        end if;

        if local_clk = 2 then
                Pad_Input (XP_left, Id);
        end if;

        if local_clk = 3 then
                Xbar_en <= "10000";
        end if;

        if local_clk = 4 then
-- Latch new ops from the decode stage!!
                OpB <= Xbarin (31 downto 16);
                OpA <= Xbarin (15 downto 0);
                Xbar_en <= "11111";
        end if;

        if local_clk = 5 then
                Ix <= Id;
        end if;



   end process;


-- XP_Left              <= TriState (XP_Left);
-- XP_Right             <= TriState (XP_Right);
   XP_Mem_A             <= TriState (XP_Mem_A);
   XP_Mem_D             <= TriState (XP_Mem_D);
   XP_Mem_RD_L                  <= '1';
   XP_Mem_WR_L                  <= '1';
   XP_HS0               <= 'Z';
   XP_GOR_Result        <= '0';
```

```
   XP_GOR_Valid        <= '0';
   XP_Int              <= '0';
--   XP_Xbar_EN_L      <= "11111";
   XP_LED              <= '1';

end Execute2;
```

A7 Execute3.vhd (PE7)

```
architecture Execute3 of Xilinx_Processing_Part is

   signal Id: Bit_Vector (35 downto 0);
   signal Ix: Bit_Vector (35 downto 0);
   signal reset:bit;
   signal local_clk: integer range 0 to 6;

   signal Left: Bit_Vector (35 downto 0);


   signal OpA: Bit_Vector (15 downto 0);
   signal OpB: Bit_Vector (15 downto 0);
   signal Mar: Bit_Vector (17 downto 0);
   signal Mdr: Bit_Vector (15 downto 0);
   signal Xbarin, Xbarout: Bit_Vector(35 downto 0);
   signal Xbar_en: Bit_Vector (4 downto 0);

   signal Result: Bit_Vector (15 downto 0);

begin

  Pad_Output (XP_right, Id);
  Pad_Xbar (XP_Xbar, Xbarout, Xbarin, Xbar_en);
  XP_Xbar_EN_L <= Xbar_en;

  Xbarout (31 downto 16) <= Mdr;
  Xbarout (15 downto 0) <= Mdr;

--Mar(17 downto 9) <= OpA(8 downto 0);
--Mar(8 downto 0) <= OpB(8 downto 0);

   process
   begin

        wait until XP_Clk'Event and XP_Clk = '1';

        local_clk <= local_clk + 1;

        if local_clk = 5 then
              local_clk <= 0;
        end if;

        Pad_Input (XP_left, Left);
```

```
        if local_clk = 0 then
                Pad_Output (XP_Mem_A, Mar);
                XP_Mem_RD_L <= '0';
                Xbar_en <= "11111";
        end if;

        if local_clk = 1 then
                Pad_Input (XP_Mem_D, Mdr);
                XP_Mem_RD_L <= '1';
        end if;

        if local_clk = 2 then
        end if;

        if local_clk = 3 then
                Pad_Input (XP_left, Id);
        end if;

        if local_clk = 4 then
                Xbar_en <= "10000";
        end if;

        if local_clk = 5 then
-- Latch new ops from the decode stage.
                OpB <= Xbarin (31 downto 16);
                OpA <= Xbarin (15 downto 0);
                Mar(17 downto 9) <= Xbarin (8 downto 0);
                Mar(8 downto 0) <= Xbarin (24 downto 16);
                Xbar_en <= "11111";
                Ix <= Id;
        end if;



   end process;


   XP_Mem_WR_L              <= '1';
   XP_HS0           <= 'Z';
   XP_GOR_Result    <= '0';
   XP_GOR_Valid     <= '0';
   XP_Int           <= '0';
   XP_LED           <= '1';

end Execute3;
```

A8 Execute4.vhd (PE 8)

```
architecture Execute4 of Xilinx_Processing_Part is

   signal Id: Bit_Vector (35 downto 0);
   signal Ix: Bit_Vector (35 downto 0);
   signal reset:bit;
```

```
  signal local_clk: integer range 0 to 6;

  signal Left: Bit_Vector (35 downto 0);

  signal BotIx: Bit_Vector (15 downto 0);

  signal OpA: Bit_Vector (15 downto 0);
  signal OpB: Bit_Vector (15 downto 0);

  signal Mar: Bit_Vector (17 downto 0);
  signal Mdr: Bit_Vector (15 downto 0);
  signal Rdr: Bit_Vector (15 downto 0);
  signal WriteEnable: Bit;

  signal Xbarin, Xbarout: Bit_Vector(35 downto 0);
  signal Xbar_en: Bit_Vector (4 downto 0);

  signal Result: Bit_Vector (35 downto 0);
  signal Result1: Bit_Vector (15 downto 0);
  signal Result2: Bit_Vector (15 downto 0);
  signal OpcodeWrite, SrcRegister, SrcMemory, SrcImmediate: Bit_Vector (4 downto 0);

begin

RIGHTIN:  for i in 0 to 15 GENERATE
        Pad_Input(XP_right(i), BotIx(i));
END GENERATE RIGHTIN;

RIGHTOUT:  for i in 16 to 31 GENERATE
        Pad_Output(XP_right(i), Ix(i));
END GENERATE RIGHTOUT;


--  Pad_Output (XP_right(31 downto 16), Ix(31 downto 16));
--  Pad_Input (XP_right(15 downto 0), BotIx);

OpcodeWrite <= "00110";
SrcRegister <= "00111";
SrcMemory <= "00100";
SrcImmediate <= "00101";

  Pad_Xbar (XP_Xbar, Xbarout, Xbarin, Xbar_en);
  XP_Xbar_EN_L <= Xbar_en;
--  Xbarout (31 downto 16) <= result1;
--  Xbarout (15 downto 0) <= result2;


   process
   begin

        wait until XP_Clk'Event and XP_Clk = '1';

        local_clk <= local_clk + 1;

        if local_clk = 5 then
                local_clk <= 0;
        end if;
```

```
        Pad_Input (XP_left, Left);

        Pad_InOut (XP_Mem_D, Rdr, Mdr, WriteEnable);
        Pad_Output (XP_Mem_A, Mar);

        XP_Mem_WR_L <= '1';
        XP_Mem_RD_L <= '1';

        Xbarout (31 downto 16) <= OpA;
        Xbarout (15 downto 0) <= OpA;

        if local_clk = 0 then
                Pad_Output (XP_Mem_A (15 downto 0), OpB);
                if (Ix (31 downto 27) = OpcodeWrite) then
                        Pad_InOut (XP_Mem_D, OpA, Mdr, '1');
                        XP_Mem_WR_L <= '0';
                else
                        XP_Mem_RD_L <= '0';
                end if;
                Xbar_en <= "11111";
        end if;

        if local_clk = 1 then
                if Ix (31 downto 27)= SrcMemory then
                        Pad_InOut (XP_Mem_D, Rdr,Xbarout (31 downto 16), '0');
                        Pad_InOut (XP_Mem_D, Rdr,Xbarout (15 downto 0), '0');
                end if;
                if Ix (31 downto 27)= SrcImmediate then
                        Xbarout (31 downto 16) <= Ix(25 downto 10);
                        Xbarout (15 downto 0) <= Ix(25 downto 10);
                end if;
        end if;


        if local_clk = 2 then
                Xbar_en <= "10000";
        end if;

        if local_clk = 3 then
                Pad_Output (XP_Mem_A (15 downto 0), Xbarin (31 downto 16));
                Rdr <= Xbarin(15 downto 0);
                if BotIx(15 downto 11) = OpcodeWrite then
                        Pad_InOut (XP_Mem_D, Xbarin (15 downto 0), Mdr, '1');
                        XP_Mem_WR_L <= '0';
                end if;
                result1 <= OpB;
                result2 <= OpA;
                Xbar_en <= "11111";
        end if;

        if local_clk = 4 then
                Pad_Input (XP_left, Id);
                Xbar_en <= "10000";
                local_clk <= 5;
        end if;

        if local_clk = 5 then
-- Latch new ops from the decode stage.
```

```
                Opa <= Xbarin (31 downto 16);
                Opb <= Xbarin (15 downto 0);
                Xbar_en <= "11111";
                Ix <= Id;
        end if;


    end process;


    XP_HS0              <= 'Z';
    XP_GOR_Result       <= '0';
    XP_GOR_Valid        <= '0';
    XP_Int              <= '0';
    XP_LED              <= '1';

end Execute4;
```

A9 Xbarcontrol.vhd (PE0)

```
architecture Xbarcontrol of Xilinx_Control_Part is

signal count :  integer range 0 to 6;

begin

process
begin

 wait until X0_Clk'Event and X0_Clk = '1';

          count <= count + 1;
          if (count = 5) then count <= 0;
          end if;

          X0_Xbar_set <= itobv (count,3);

end process;


   X0_SIMD <= TriState(X0_SIMD);
   X0_XB_Data <= TriState(X0_XB_Data);
   X0_Mem_A <= TriState(X0_Mem_A);
   X0_Mem_D <= TriState(X0_Mem_D);
   X0_Mem_RD_L <= '1';
   X0_Mem_WR_L <= '1';
   X0_GOR_Result_In <= "ZZZZZZZZZZZZZZZZ";
   X0_GOR_Valid_In <= "ZZZZZZZZZZZZZZZZ";
   X0_GOR_Result <= '0';
   X0_GOR_Valid <= '0';
--   X0_XBar_Set <= "000";
   X0_XBar_Send <= '0';
   X0_X16_Disable <= '0';
   X0_Int <= '0';
   X0_Broadcast_Out <= '0';
   X0_HS0 <= 'Z';
   X0_XBar_EN_L <= '1';

 end Xbarcontrol;
```

A10 Xbarconfig

configuration 0

1	0				
2	0				
3	0	5	5	6	6
4	0	6	6	5	5
5	0				
6	0				
7	0				
8	0				
9	0				
10	0				
11	0				
12	0				
13	0	12	12	11	11
14	0	11	11	12	12
15	0				
16	0				

configuration 1

1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	4	4	3	3
8	0	3	3	4	4
9	0	13	13	14	14
10	0	14	14	13	13
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	14	14	13	13
15	0	13	13	14	14
16	0	0	0	0	0

configuration 2

1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	14	14
4	0	0	0	13	13
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	0	0	0
13	0	4	4	0	0
14	0	3	3	0	0
15	0	0	0	0	0
16	0	0	0	0	0

configuration 4

1	5	0	0	0	0
2	6	0	0	0	0
3	0	7	7	8	8
4	0	8	8	7	7
5	0				
6	0				
7	0				
8	0				
9	0				
10	0				
11	0				
12	0				
13	0	9	9	10	10
14	0	10	10	9	9
15	0				
16	0				

configuration 5

1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	4	4	3	3
8	0	3	3	4	4
9	0	13	13	14	14
10	0	14	14	13	13
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	14	14	13	13
15	0	13	13	14	14
16	0	0	0	0	0

configuration 6

1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	3	3	4	4
6	0	4	4	3	3
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	13	13	14	14
12	0	14	14	13	13
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0
16	0	0	0	0	0

Appendix B

Runtime Results

B1 Fibonacci Sequence Results

```
T2 Version 1.88 Created Wed Oct  5 09:31:37 EDT 1994

NEW INTERFACE BOARD (rev2)

T2:1> source pe5.init
2 boards available on Splash 2 unit 0

T2:2> source fib.step
Added RB 1 at tick 6
Board 0 PE 3 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A

T2:3> source fib.step
Added RB 2 at tick 12
Board 0 PE 3 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A

T2:4> source fib.step
Added RB 3 at tick 18
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 000A 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 000A 000A 000A 000A 000A 000A

T2:5> source fib.step
Added RB 4 at tick 24
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 0003 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 0003 000A 000A 000A 000A 000A

T2:6> source fib.step
Added RB 5 at tick 30
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A
```

```
T2:7> source fib.step
Added RB 6 at tick 36
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0003 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0003 0005 000A 000A 000A 000A

T2:8> source fib.step
Added RB 7 at tick 42
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A

T2:9> source fib.step
Added RB 8 at tick 48
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A

T2:10> source fib.step
Added RB 9 at tick 54
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A

T2:11> source fib.step
Added RB 10 at tick 60
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0005 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0005 0008 000A 000A 000A 000A

T2:12> source fib.step
Added RB 11 at tick 66
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A

T2:13> source fib.step
Added RB 12 at tick 72
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A

T2:14> source fib.step
Added RB 13 at tick 78
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A
```

```
T2:15> source fib.step
Added RB 14 at tick 84
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0008 0008 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0008 0008 000D 000A 000A 000A 000A

T2:16> source fib.step
Added RB 15 at tick 90
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A

T2:17> source fib.step
Added RB 16 at tick 96
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A

T2:18> source fib.step
Added RB 17 at tick 102
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0008 000D 0015 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0008 000D 0015 000A 000A 000A 000A
```

B2 VLIW Fibonacci Sequence Results

```
T2 Version 1.88 Created Wed Oct  5 09:31:37 EDT 1994

NEW INTERFACE BOARD (rev2)

T2:1> source all.init
2 boards available on Splash 2 unit 0

T2:2> source all1.step
Added RB 1 at tick 6
Board 0 PE 3 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A

T2:3> !
Added RB 2 at tick 12
Board 0 PE 3 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 000A 000A 000A 000A 000A 000A 000A

T2:4> !
Added RB 3 at tick 18
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0001 0001 000A 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0001 0001 000A 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0001 0001 000A 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0001 0001 000A 000A 000A 000A 000A
```

```
T2:5> !
Added RB 4 at tick 24
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0001 0001 0002 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0001 0001 0002 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0001 0001 0002 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0001 0001 0002 000A 000A 000A 000A

T2:6> !
Added RB 5 at tick 30
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A

T2:7> !
Added RB 6 at tick 36
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0001 0002 0002 000A 000A 000A 000A

T2:8> !
Added RB 7 at tick 42
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0001 0002 0003 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0001 0002 0003 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0001 0002 0003 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0001 0002 0003 000A 000A 000A 000A

T2:9> !
Added RB 8 at tick 48
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
```

```
T2:10> !
Added RB 9 at tick 54
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0002 0003 0003 000A 000A 000A 000A

T2:11> !
Added RB 10 at tick 60
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0002 0003 0005 000A 000A 000A 000A

T2:12> !
Added RB 11 at tick 66
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A

T2:13> !
Added RB 12 at tick 72
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0003 0005 0005 000A 000A 000A 000A

T2:14> !
Added RB 13 at tick 78
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0003 0005 0008 000A 000A 000A 000A
```

```
T2:15> !
Added RB 14 at tick 84
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A

T2:16> !
Added RB 15 at tick 90
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0005 0008 0008 000A 000A 000A 000A

T2:17> !
Added RB 16 at tick 96
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0005 0008 000D 000A 000A 000A 000A


T2:18> !
Added RB 17 at tick 102
Board 0 PE 3 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
Board 0 PE 4 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
Board 0 PE 14 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
Board 0 PE 13 Offset 0 Words 32
000000: 0000 0008 000D 000D 000A 000A 000A 000A
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Abnous, A. Bagherzadeh, N. "Pipelining and Bypassing in a VLIW Processor" *IEEE Transactions on Parallel and Distributed Systems. Vol. 5. No. 6.* June 1994. pp. 658 - 664

[2] Amerson, R. Carter, R.J. Culbertson, W.B. Kuekes, P. Snider, G. "Teramac - Configurable Custom Computing" *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[3] Ellis, J.R. "*Bulldog: A Compiler for VLIW Architectures*" Cambridge, Mass. MIT Press, 1985.

[4] Fisher, J. A. "Trace Scheduling: A Technique for Global Microcode Compaction" *IEEE TOC, C-30* . July 1981. pp. 478-490

[5] Fisher, J. A. Rau, B. R. "Instruction-Level Parallel Processing" *Science*, September 1991.

[6] French, P.C. Taylor R.W. "A Self-Reconfiguring Processor" *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* , 1994.

[7] Gokhale, M.B. Holmes, W. Kopser, A. Lucas, S. Minnich, R. Sweely, D. Lopresti, D. "Building and Using a Highly Parallel Programmable Logic Array" *Computer*, January 1991. pp. 81-87

[8] Gokhale, M.B. Minnich, R. "FPGA Computing in a Data Parallel C" *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993. pp. 94-101

[9] Gokhale, M.B. Schott, B. "Data Parallel C on a Reconfigurable Logic Array" Draft

[10] Gray, J. Naylor, A. Abnous, A. Bagherzadeh, N. "Viper: A 25-MHz, 100-MIPS Peak VLIW Microprocessor" *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*. May 9-12, 1993. pp. 4.1.1 - 4.1.5

[11] Hadley, J.D. Hutchings, B.L. "Design Methodologies for Partially Reconfigured Systems" *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[12] Hennessy, J. Patterson, D. "*Computer Architecture A Quantitative Approach*" 1990.

[13] Hill, M.D. Larus, J.R. Reinhardt, S.K. Wood, D.A. "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors" *ACM Transactions on Computer Systems, Vol. 11, No. 4*. November 1993. pp. 300-318

[14] Hogl, H. Kugel, A. Ludvig, J. Manner, R. Noffz, K.H., Zoz, R. "Enable++: A second generation FPGA processor" *IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[15] Hwang, K. "*Advanced Computer Architecture: Parallelism, Scalability, Programmability*" 1993.

[16] Iseli, C. Sanches, E. "A C++ Compiler for FPGA custom execution units synthesis" *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[17] Iseli, C. Sanches, E. "Spyder: A Reconfigurable VLIW Processor using FPGA's" *IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.

[18] Johnson, M. "*Superscalar Microprocessor Design*" 1991.

[19] Lenoski, D. Laudon, J. Gharachorloo, K. Weber, W. Gupta, A. Hennessy, J. Horowitz, M. Lam, M. "The Stanford Dash Multiprocessor" *Computer*, March 1992. pp. 63-79

[20] Malke S. Chen, W.Y. Bringmann, R.A. Hank, R. E. Hwu, W.W. Rau, B.R. Schlansker, M.S. "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution" *ACM Transactions on Computer Systems, Vol. 11, No. 4*. November 1993. pp. 376-408

[21] Meier, R.D. "Rapid Prototyping of a RISC Architecture for Implementation in FPGAs" *IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[22] Nicolau, A. "Percolation Scheduling: A Parallel Compilation Technique" *Dept. of Computer Science Cornell Tech. Report TR 85-678*, May 1985.

[23] Novak, J.H. Brunvand, E. "Using FPGAs to Prototype a Self-Timed Floating Point Co-Processor" *Proceedings of the 1994 IEEE Custom Integrated Circuits Conference*. pp. 85-88

[24] Ratha, N.K. Jain, A.K. Rover, D.T. "Convolution on Splash 2" *IEEE Symposium on FPGAs for Custom Computing Machines* 1995.

[25] Robert, M. Gorria, P. Miteran, J. Turgis, S. "Architectures for a Real Time Classification Processor" *Proceedings of the 1994 IEEE Custom Integrated Circuits Conference*. pp. 197-200

[26] Rover, D. Tsai, V. Chow, Y. Gustafson, J. "Signal-Processing Algorithms on Parallel Architectures: A Performance Update" *Journal of Parallel and Distributed Computing, Vol. 13*. 1991. pp. 237-245

[27] Salinas, M.H. Johnson, B.W. Aylor, J.H. "Implementation-Independent Model of an Instruction Set Architecture in VHDL" *IEEE Design & Test of Computers* 1993.

[28] Schuette, M.A. Shen, J.P. "An Instruction-Level Performance Analysis of the Multiflow TRACE 14/300" *Association for Computing Machinery* 1991.

[29] Shen, J.P. Class notes, 1992-1993.

[30] Takayanagi, T. Sawada, K. Sakurai, T. Parameswar, Y. Tanaka, S. Ikumi, N. Nagamatsu, M. Kondo, Y. Minagawa, K. Brennan, J. Hsu, P. Rodman, P. Bratt, J. Scanlon, J. Tang, M. Joshi, C. Nofal, M. "Embedded Memory Design for a Four Issue Superscalar RISC Microprocessor" *Proceedings of the 1994 IEEE Custom Integrated Circuits Conference*. pp. 585-590

[31] Wang, R. Milletary, J. Kobayashi, T. "The Dex JR." Senior project report 1993.

[32] Wirthlin, M.J. Hutchings, B.L. "A Dynamic Instruction Set Computer" *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* 1995.

[33] Wolfe, A. Shen, J.P. "Superscalar Processor Design" *Proceeding of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (Asplos V)* 1992.

[34] Xilinx "*The Programmable Logic Data Book* " Xilinx, Inc. 1994.