A VHDL MODEL OF A SUPERSCALAR IMPLEMENTATION OF THE DLX INSTRUCTION SET ARCHITECTURE

by

Paul A. Ferno

A thesis submitted in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Engineering

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York

October, 1996

Approved by _____

Dr. Kevin Shank, Assistant Professor

_____

Dr. Tony Chang, Professor

_____

Dr. Roy Czernikowski, Dept. Head and Professor

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title: A VHDL Model of a Superscalar Implementation of the DLX Instruction Set Architecture

I, Paul A. Ferno, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Signature: _____

Date: 11-9-96

Abstract

The complexity of today's microprocessors demands that designers have an extensive knowledge of superscalar design techniques; this knowledge is difficult to acquire outside of a professional design team. Presently, there are a limited number of adequate resources available for the student, both in textual and model form. The limited number of options available emphasizes the need for more models and simulators, allowing students the opportunity to learn more about superscalar designs prior to entering the work force. This thesis details the design and implementation of a superscalar version of the DLX instruction set architecture in behavioral VHDL. The branch prediction strategy, instruction issue model, and hazard avoidance techniques are all issues critical to superscalar processor design and are studied in this thesis. Preliminary test results demonstrate that the performance advantage of the superscalar processor is applicable even to short test sequences. Initial findings have shown a performance improvement of 26% to 57% for instruction sequences under 150 instructions.

Acknowledgments

There are a few people who deserve recognition for their support and assistance of my efforts towards the completion of this thesis. First, I would like to thank my graduate committee members, Dr. Roy S. Czernikowski, Dr. Kevin Shank, and Dr. Tony Chang, who put in the extra time to help get this thesis completed. I would also like to thank Frank Casilio, for providing me with the equipment that I needed, when I needed it. Rick Brink deserves special thanks for his willingness to proof read and edit this document, in an effort to make this thesis more readable.

Finally, I would like to sincerely thank my parents, Richard and Sandra Ferno, who through their time, unwavering support, and unending love, allowed this thesis to occur. It is to them that this milestone in my educational experience is dedicated.

Trademarks

ALPHA, DEC are trademarks of Digital Equipment Corporation.

AMD, AMD-K5 are trademarks of Advanced Micro Devices, Inc.

HP, PA-RISC, PA-8000 are trademarks of Hewlett-Packard Company

Intel, Pentium are trademarks of Intel Corporation

MC68000 is a trademark of Motorola, Inc.

MIPS, R10000 are trademarks of MIPS Technologies, Inc.

PowerPC is a trademark of International Business Machines, Inc.

SPARC is a trademark of SPARC International, Inc.

Table of Contents

List of Figures

List of Tables

Glossary

ALU	Arithmetic Logic Unit
Big Endian Mode	A method of storing data in memory such that the most significant byte of a word is stored at a lower address.
BIST	Built In Self Test: a method of testing which allows internal nodes of the integrated circuit to be tested and verified after fabrication.
BTB	Branch Target Buffer: a memory array that uses the instruction address to access the next instruction's address prior to the decoding of the instruction.
Cache	A high speed buffer storage that contains frequently accessed instructions and/or data; it is used to reduce access times.
CAM	Content Addressable Memory: a form of memory which is accessed via the data it holds rather than an address to a location.
CISC	Complex Instruction Set Computer
Commitment	The commitment of an instruction, when discussing reorder buffers and register files, occurs when it has been completed successfully and all the instructions that preceded it in instruction order have safely completed, and have been committed.
Completion	When discussing reorder buffers and register files, the completion of an instruction occurs when the instruction reaches the point where it will alter the state of the processor, such as a branch instruction changing the program counter, or the store instruction writing to memory.
CPI	Clocks Per Instruction: a quantifiable measurement of processor performance (Hennessy, 1996).

Delay Slot	The instruction that immediately follows the branch, trap, or jump instruction that must always be executed. This is done to minimize the penalty for determining where the branch, jump, or trap is going to take the PC.
DIS	DISpatch stage: this stage holds the decoded instructions until they are able to execute. This stage does not appear in the standard DLX pipeline.
DLX	An academic architecture used to demonstrate computer design alternatives, first introduced in (Hennessy, 1996).
EX	EXecution stage: the stage in the DLX pipeline that contains the ALUs and performs the actual operations specified by the instructions.
FPR	Floating Point Register: a register that is used to hold floating point values while they are in the processor.
FPU	Floating Point Unit: the section of the processor that handles the floating point operations.
F*x*	Specifies the FPR number x, or in double precision instructions registers numbered x and $x+1$.
GPR	General Purpose Register: a register that is used to hold addresses, integers and other assorted data that is not in floating point format.
GUI	Graphical User Interface: a way of presenting the data in a graphical manner with windows and graphics as opposed to straight ASCII text.
ID	Instruction Decode Stage: the stage of the DLX pipeline that separates the instructions into sources, destination, immediate value and instruction type.
IF	Instruction Fetch Stage: the stage of the pipeline that is responsible for reading the instruction from memory.
ILP	Instruction Level Parallelism: the parallelism that is found at the instruction level and is exploited by superscalar and VLIW processors.

ISA	Instruction Set Architecture: the microprocessor as seen from the programmers point of view, i.e. the registers, instruction formats, and interrupt handling.
KByte	Kilobyte: 2^{10} or 1,024 bytes.
LRU	Least Recently Used: a protocol for the replacement of entries in caches, specifically the instruction that has not been used in the longest amount of time.
LSB	Least Significant Bit: the end bit of a binary number that has the lowest numerical significance, i.e. in the 4-bit binary representation of the number 7 (0111), the right most '1' is the LSB since it represents 2^0, or 1.
MByte	Megabyte: 2^{20} or 1,048,576 bytes.
MEM	MEMory stage: this stage of the DLX pipeline handles accesses to memory for loads and stores. This stage is moved into the execution stage in the superscalar DLX processor.
MSB	Most Significant Bit: the end bit of a number that has the highest numerical significance, i.e. in the 4-bit binary representation of the number 7 (0111) the zero is the most significant bit since it represents 2^3, or 8.
Out-of-Order Execution	The execution of instructions in an order that differs from the sequential occurrence of the instructions in the compiled code.
PC	Program Counter: the register that keeps track of the address of the next instruction fetch from memory.
RAM	Random Access Memory
RAW	Read After Write: a hazard occurring in pipelined processors when one instruction tries to read a register before the previous instruction has written the data.
RISC	Reduced Instruction Set Computer

RTL	Register-Transfer Level: the level of specification in a VHDL file that describes the individual components such as registers and adders.
R*x*	Specifies the GPR number *x*
Significand	The part of a floating point number which contains the significant digits of the number. This excludes the exponent and sign bits.
SPEC	System Performance and Evaluation Cooperative: an organization founded in the late 1980's whose purpose was to create a standard set of benchmarks that would provide a more neutral way to compare processor performances.
SPECfp95	The 1995 version of the Floating Point test suite created and maintained by SPEC.
Sub-normal	A floating point number that is too small to have an implicit 1 in its representation but is not yet unrepresentable.
Superscalar	A multiple issue microprocessor design that uses dynamic scheduling along with compiler optimizations to take advantage of ILP
Tomasulo's Algorithm	An algorithm created by Robert Tomasulo for the IBM 360/91, which provided a solution for avoiding WAW and WAR hazards that occur during out-of-order execution (Hennessy, 1996).
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word: a multiple issue microprocessor that takes advantage of ILP solely through the use of advanced compilers.
WAR	Write After Read: a hazard that only occurs in out-of-order execution processors. Occurs when one instruction writes to a register prior to an earlier instruction reading the previous value.

WAW Write After Write: a hazard that only occurs in out-of-order execution processors. Occurs when one instruction writes to a register out of sequence, i.e. instruction 5 writes to R1 and then instruction 2 writes to R1.

WB WriteBack: the final stage of the DLX pipeline that handles the updating of the register file with the newly calculated or loaded data.

1.0 Introduction

The complexity of today's microprocessors demands that designers have an extensive knowledge of superscalar design techniques; this knowledge is difficult to acquire outside of a professional design team. Presently, there are a limited number adequate resources available for the student, with (Hennessy, 1996) being the best when looking for written explanations. The SuperDLX simulator, written by Cecile Moura, is currently the only tool available to students for experimenting with different superscalar processor configurations. The limited number of options available points to the need for more models and simulators, allowing students the opportunity to learn more about superscalar designs prior to entering the work force. This paper describes the superscalar processor model created, in VHDL, during this thesis work at Rochester Institute of Technology.

1.1 DLX

The instruction set architecture (ISA) chosen for this superscalar model was the DLX. The DLX ISA that first appeared in (Hennessy, 1996), contains only the most common RISC instructions, totaling a mere 92 instructions. This is far less than the 130 plus instructions found in many other RISC ISAs. Below is a summary of the DLX architecture, with a more detailed description available in (Hennessy, 1996) and (Kaeli, 1996).

The DLX architecture contains 32 general purpose registers and 32 single precision floating point registers, along with various other registers for interrupt and

exception handling. The general purpose and floating point registers are each 32 bits wide. The floating point registers can be combined in even-odd pairs to create double precision registers of 64 bits. Loads and stores to the general purpose registers (GPR) can occur in byte, half-word, and word sizes. Data less than 32 bits in length is sign extended or padded as required by the instruction. The only registers that can not be freely read and written are GPR zero (R0) and GPR thirty-one (R31). R0 is wired with the value of zero and writing to the register has no effect. R31 is used to store the return address for jump and link instructions, so the use of this register needs to be watched carefully.

DLX instructions occur in three formats, displayed in Figures 1.1 through 1.3 and described in Table 1.1. The numbers above the instruction representation in the following figures are the bit numbers and the numbers below are the size of those fields in bits.

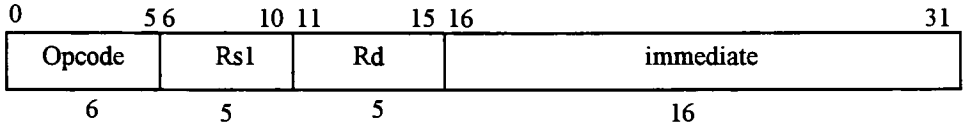

Figure 1.1 - DLX I-Type Instruction Format (Kaeli, 1996)

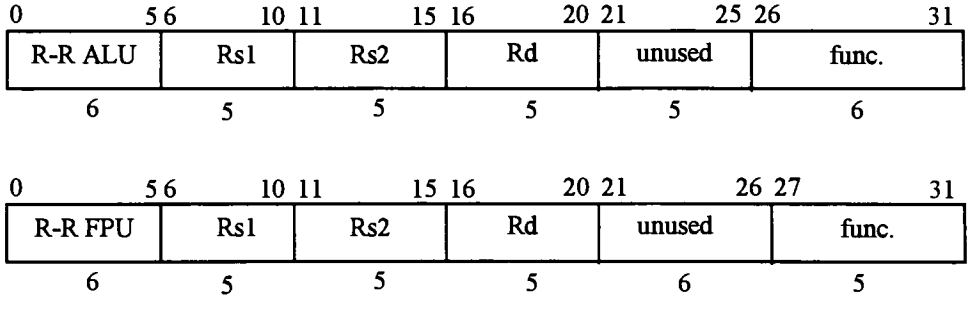

Figure 1.2 - DLX R-Type Instruction Formats (Kaeli, 1996)

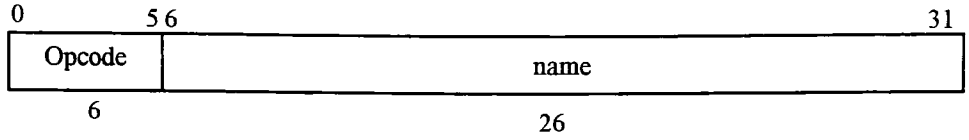

Figure 1.3 - DLX J-Type Instruction Format (Kaeli, 1996)

Instruction Type	Instructions Supported
I-Type (immediate)	load, store, and all immediate instructions
R-Type (register)	moves, all non-immediate ALU functions, and FPU functions
J-Type (jump)	jump, jump and link, trap, and return from exception

Table 1.1 - Supported Instruction Types

The DLX ISA specifies that memory accesses are directly mapped, with no translation or virtual addresses. This simplifies memory accesses. Data Loads and stores must follow the rules listed in Table 1.2. All instruction fetches must also be on word boundaries, which requires the least significant 3 bits to be zeros.

Type of Access	Size of Data (bits)	Least Significant Byte of the Address
Byte	8	XXXX XXXX
Halfword	16	XXXX XXX0
Word	32	XXXX XX00
Single Precision FP	32	XXXX XX00
Double Precision FP	64	XXXX XX00

Table 1.2 - Memory Access Alignments of the DLX Architecture

The DLX pipeline, as defined by Hennessy (Hennessy, 1996), consists of five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and writeback (WB). The stages are shown in Figure 1.4 as they appeared in Hennessy (Hennessy, 1996). Another architectural feature of the DLX ISA is the delay slot after branches and jumps. The delay slot is the instruction position after the branch or

jump which is always executed independent of the branch's result. Using this technique allows the architecture to eliminate some of the delay associated with branch instructions.

	Clock Number							
Instuction Number	1	2	3	4	5	6	7	8
Instruction i	IF	ID	EX	MEM	WB			
Instruction i + 1		IF	ID	EX	MEM	WB		
Instruction i + 2			IF	ID	EX	MEM	WB	
Instruction i + 3				IF	ID	EX	MEM	WB

IF	= instruction fetch
ID	= instruction decode
EX	= execution
MEM	= memory access
WB	= write back

Figure 1.4 - The DLX Pipeline (Hennessy, 1996)

The DLX architecture works well as an academic architecture since it is uncomplicated and well understood. Yet this architecture still encompasses all the major features used in many of today's more complex microprocessors.

1.2 Superscalar Processors

A superscalar processor issues varying numbers of instructions per clock and may be either statically scheduled by the compiler or dynamically scheduled using hardware techniques. A superscalar processor has performance advantages over a scalar processor, but also introduces many design tradeoffs and difficulties.

The major advantage of a superscalar design is that it reduces the clocks per instruction (CPI). In a scalar pipelined processor the best achievable CPI is 1.00, since only one instruction is issued per clock. A superscalar processor, however, can issue as many as four or five instructions per clock, with plans for 32 issue machines by the year 2003 (Computergram, 1996). Consider this example. A scalar processor would have to be

clocked at 800 MHz to achieve the same number of instructions per second as a 4 issue 200 MHz superscalar processor, in the ideal case. In most cases it is easier to create a design that runs with a longer clock period.

However, this advantage introduces numerous design constraints that must be satisfied. Instruction dispatching, data hazards, and control hazards are all problems that arise with a superscalar design. Pipelined processors also exhibit these problems. However, the number of hazards increases in a superscalar design, because there are multiple pipelines in the processor. In a superscalar design, hazards not only occur within the pipeline, but between pipelines as well.

These hazards occur as a result of issuing multiple instructions per clock. Currently there are three choices for the issuing of instructions. They are in-order issue and in-order completion, in-order issue and out-of-order completion, and out-of-order issue and out-of-order completion. The easiest to implement is in-order issue and completion, but this method also results in the most stalls due to data and control dependencies. An example is in the following code taken from (Hennessy, 1996). The data dependencies in this code segment are between the LW and ADD instructions. The ADD instruction requires the value in register R1 before it can perform the addition, but the LW instruction reads the value to place into register R1 first. With this dependency, the ADD instruction is forced to wait until the result of the LW instruction can be forwarded to it.

```
LW    R1, 45(R2)    Load word from R2 + 45 into R1
ADD   R5, R1, R7    Add R1 and R7 placing result in R5
SUB   R8, R6, R7    Subtract R7 from R6 placing result in R8
OR    R9, R6, R7    Or R6 and R7 placing result in R9
```

In this example, with in-order issue and in-order completion, the ADD instruction will stall until the LW instruction has completed. The stall impedes the progress of the SUB and OR instructions that do not have dependencies with the LW or ADD instructions. Handling long latency events, such as divides, also causes degraded performance since the instructions following the divide would not be able to finish until the divide finishes, which could take tens of clock cycles. Figure 1.5 graphically portrays the execution sequence of the above instructions on a dual issue processor for the in-order issue and in-order completion case.

	Clock Cycle Number							
Instruction	1	2	3	4	5	6	7	8
Load r1,45(r2)	IF	ID	EXE	MEM	WB			
Add r5,r1,r7	IF	ID	STALL	STALL	EXE	MEM	WB	
Sub r8,r6,r7		IF	STALL	STALL	ID	EXE	MEM	WB
Or r9,r6,r7		IF	STALL	STALL	ID	EXE	MEM	WB

Figure 1.5 - Dual Issue, In-Order Issue and In-Order Completion Example

The next choice for the issuing of instructions is in-order issue and out-of-order completion. This alternative requires that all instructions are issued in the correct order, but an instruction can be bypassed while in the execution stage and finish out-of-order. The previous code sequence would still have a stall in this issue protocol, due to the true data dependency between the load and the add instructions. The advantage of out-of-order completion arises when a long latency floating point instruction is issued. This is because the next instruction can be issued and need not wait until the slow instruction is completed.

The final strategy for the issuing of instructions is out-of-order issue and out-of-order completion. This is the most difficult strategy to implement, but results in the least amount of stalls. With this issue strategy, and the example code above, there would be no stalls at all. Once the ADD instruction is determined to be dependent on the LW instruction, the processor will look ahead to see that the SUB and OR instructions can still be executed. Figure 1.6 illustrates the execution of the previous code segment on a dual issue, out-of-order issue and out-of-order completion processor. Having an instruction pool between the decode and execute stages is key to performing out-of-order issue and completion. This pool of instructions allows the processor to have a larger number of instructions to choose from, and therefore increases the probability that there will always be an instruction that is able to be executed.

	Clock Cycle Number							
Instruction	1	2	3	4	5	6	7	8
Load r1,45(r2)	IF	ID	EXE	MEM	WB			
Add r5,r1,r7	IF	ID	WAIT	WAIT	EXE	MEM	WB	
Sub r8,r6,r7		IF	ID	EXE	MEM	WB		
Or r9,r6,r7		IF	ID	EXE	MEM	WB		

Figure 1.6 - Dual Issue, Out-of-Order Issue and Out-of-Order Completion Example

The choice of issue strategy determines the number and types of data hazards that must be dealt with. In a completely in-order processor, only one type of data hazard is present (RAW). However, in a processor that supports out-of-order completion, two more types of data hazards occur. These hazards are placed into three categories: read after write (RAW), write after read (WAR), and write after write (WAW).

RAW data hazards occur when instruction *j* reads a register before the preceding instruction *i* has written the new value, so *j* gets the incorrect old value. The solution to this problem is to use data forwarding on the operands. Data forwarding routes the output of the ALU and the memory stage back to the input of the ALU so that subsequent instructions get the new correct data. The selection process between forwarded data and the standard input is determined at the start of the execution stage based on the source and destination register values.

The next two hazards, WAW and WAR, occur when an instruction finishes before a preceding instruction. In the case of a WAW hazard the data written by instruction *j* is subsequently over written by instruction *i* that finishes later. The value that is left in the register is from instruction *i*, rather than *j* as it should be. A WAR hazard is similar but instead of instruction *i* writing over the result of instruction *j*, it reads the new result of instruction *j* rather than the old correct value. The techniques used to remove these hazards include reorder buffers and register renaming, which both use virtual registers.

Control hazards are a problem since they interrupt the flow of instructions into the processor. An example of this is the branch instruction. If the branch is taken then the program counter (PC) is changed by the increment specified in the instruction, otherwise the PC is incremented normally and execution continues. The result of a branch is not known until at least the end of the decode stage and therefore causes the pipeline to stall. A way around the stall is to predict the branch and speculatively execute the next instructions. Recovering from a mispredicted branch creates its own problems since

changes made by the wrongly executed instructions must be undone. A solution is to use the reorder buffers and register files that handle out-of-order completions. By not committing the speculative instructions until the branch is determined removes that problem.

The preceding hardware constructs eliminate the hazards that occur in superscalar designs, allowing the architecture to provide performance benefits.

1.3 Works of Interest

In a time when the majority of the microprocessor designs being manufactured are superscalar RISC or RISC hybrids, there are a number of VDHL models and simulators specific to the DLX ISA. A reason for the number of simulators and models of the DLX ISA is its ability to explore design concepts on a simple architecture that focuses on the basics of RISC, rather than fancy features found only in one architecture. The following three sections discuss VHDL models, simulators, and existing superscalar architectures that provide insight into the topic of superscalar designs and the DLX ISA.

1.3.1 DLX VHDL Models

There are few Hardware Description Language (HDL) models of the DLX ISA in existence today. The ones that are available range from non-pipelined to scalar pipelined in nature. One example is the DLXS, created as part of a VLSI design course in Stuttgart Germany, which is a non-pipelined scalar implementation of the DLX ISA. A second example is the DLX pipelined model that was created by Peter Ashenden and included in his book The Designer's Guide to VHDL (Ashenden, 1996).

The DLXS is a non-pipelined version of the DLX architecture. The instruction set implemented does not contain any of the floating point instructions, nor the integer multiply or divide instructions. It does add, however, a different addressing mode, three operating modes, and interrupt and exception handling from a SPARC™ design. This processor model reads in one instruction at a time and each instruction is executed completely before the next instruction is started. DLXS serves as a good starting point for architectural study since its design is similar to the MC68000™ and other earlier designs that are used in academia. Another aspect of this model is its creation in synthesizable VHDL, which is beneficial since the design can then be mapped to whatever process technology is available, highlighting the benefits of each technology (Gumm, 1995).

Peter Ashenden's model is a behavioral/RTL level representation of the DLX ISA that is used as an example in his book as he transforms the model from behavioral to register transfer language (RTL). This model does not implement the floating point instructions, and is implemented as a single flow of instructions through a non-pipelined core (Ashenden, 1996).

There exist a few more models of the DLX ISA, ranging from partial implementations, such as Ashenden's, to other implementations that include the pipeline. These models were not investigated in depth for this paper. Many of the other designs are either not completed or not readily available. As these designs near completion the information on them will hopefully be more accessible and can be included in future work. Another reason some of these models were not explored is that the time it takes to install

and evaluate these tools is non-trivial, since most of these tool were developed on a UNIX system and the development environment is never the same as the target environment.

1.3.2 DLX Simulators

There are three major DLX simulators available, and they are the DLX Interactive Simulation Composite (DISC), DLXsim, and SuperDLX. All of these simulators allow for some reconfiguration by the user, and have some user interface to depict the internal workings of the processor.

DLXsim was the first DLX ISA simulator. It was created by Larry B. Hostetler and Brian Mirtich in 1990-1991. This simulator interfaces with the user via text commands and allows the user to specify the number and latency of FPU multiply, add, and divide units. This creates a superscalar design since floating point operations can occur simultaneously with other floating point and integer instructions. The scalar integer pipeline is split into to the five stages defined by Hennessy and Patterson. These stages are fetch, decode, execute, memory, and writeback. DLXsim reads in a DLX assembly file and then internally converts the instructions to machine code. The simulator also provides many useful statistics during and after execution, such as the number of occurrences of each instruction, register values, and percentages of branches taken and not taken, to highlight a few. With these attributes DLXsim works well as a tool to explore the design constructs of pipelined processors in a step by step fashion. The next two simulators were designed as extensions from the base provided by DLXsim (Hostetler, 1996).

DISC, or DLX Interactive Simulation Composite, is based on DLXsim, and is being developed at Purdue University's School of Electrical and Computer Engineering. It adds one major extension, which is a GUI. It has also been enhanced to include scoreboarding and the Tomasulo algorithm. Both scoreboarding and the Tomasulo algorithm are methods used to resolve the WAR and WAW hazards associated with out-of-order execution of instructions. As with DLXsim, the number of FPU units is re-configurable. In addition, the DISC simulator adds the choice of scoreboarding or Tomasulo's algorithm, which allows out-of-order completion. The designers of this tool plan to extend this simulator to handle both VLIW and superscalar architectures using example architectures such as the ALPHA™ and the PowerPC™ microprocessors ("DLX Interactive .. ", 1995).

The SuperDLX was created by Cecile Moura at McGill University as a Master's project. It is also an extension of DLXsim, adding out-of-order issue and completion and speculative execution based on branch prediction. As with DISC, the SuperDLX has an Xwindow based GUI. It also allows the user to see many of the inner details of the processor, as well as specify the number and delay of many of the components such as buffers and functional units. The SuperDLX simulator appears to be a better superscalar simulator since all parts of the design can be replicated, rather than just the floating point units. This allows the SuperDLX to provide experimentation on a larger range of architectures in order to determine the appropriate architectural mix for a particular problem (Moura, 1993).

New DLX simulators and revisions of existing simulators are continually appearing. Many of these other simulators do not have a wealth of information about them and therefore were not heavily explored during this research. All of the models, explored seem to have one major drawback: there is no easy method to change the underlying algorithms. The reason for this problem is the close coupling of the model to the simulator, rather then the de-coupled style that exists with HDL and their simulators. For example, the DLXsim source code does not have a separate file for the simulator and the architecture to be modeled. This high coupling between the simulator and the design make it harder to think solely about the architecture without worrying about how changes to the architecture will effect the simulator.

1.3.3 Superscalar Architectures

With numerous superscalar RISC designs in production, there is much information available on different techniques and design choices when creating a superscalar processor. Areas of interest are branch prediction, hazard prevention, instruction fetching, and pipeline depth. In each of these areas there exist many choices to consider. At one end there is the DEC™ ALPHA with its clock rate of 500 MHz and a pipeline depth of seven to nine stages. At the other end there is a CISC/RISC hybrid, AMD's AMD-K5™, with a clock speed of 100 MHz and a five stage pipeline. The performance standards for today's processors is currently set by Hewlett-Packard and Digital Equipment with their newest processors, yet their design philosophies are much different. Digital's designs have extreme simplicity in each stage, at the cost of more stages, so that the clock period is

very small. HP's designs have stages which are more complex, and therefore a slower clock (about 180 MHz) is required. On their top processors, the PA-8000™ for HP™ and the 21164a for DEC, the SPECfp95 numbers are separated by less then one SPEC unit. With this diversity in the marketplace and research arena, there are many design choices to be made and a good tool will allow for a simple exploration of a number of alternatives. The MIPS R10000™ is another recent design, which, while not setting any performance records, provides an interesting blend of architectural choices that serves as a guideline for this superscalar DLX model. The five execution units that are in the R10000 are two integer units, an FP adder, an FP multiply/divider, and a load and store unit, which matches what was originally chosen for the superscalar DLX model (Ahi, 1995).

The MIPS R10000 is serving as the initial template for this VHDL model but all of the other existing architectures are being examined for ideas concerning the branch prediction, out-of-order execution methods, and exception handling. In this manner, the base of knowledge and design choices that are in production today can be incorporated into this design when appropriate.

1.4 Document Organization

Chapter 2 covers the structure and design components of the superscalar DLX model. The chapter is broken down by pipeline stage of the processor, with sections including instruction fetch, instruction decode, dispatch, execute, and writeback. Chapter 3 discusses the simulation methods and results. Chapter 4 evaluates the performance of the model and the effects of the assumptions made in the design process. The future of

this processor model and thesis work are discussed in chapter 5. Finally in chapter 6, the concluding remarks and discussion are made.

2.0 Overall Structure

The overall structure of this processor is shown in Figure 2.1. The pipeline is split into five stages: instruction fetch (IF), instruction decode (ID), dispatch (DIS), execute (EX), and writeback (WB). Each of these stages will be described in detail in the following sections.

Instructions are first received by the IF stage from the Instruction Cache, then forwarded on to the ID stage. As the instructions pass through the decode stage, each of the 4 decoders fetches the operands for its instruction from the register files. Once the instructions are decoded they are sent to one of four dispatch queues: ALU, floating point, load/store, or branch. While in the dispatch queues the instruction's operands are updated as needed by the data being forwarded from the execution units. Once an instruction is determined to be ready for execution, it is sent to one of the six execution units in the superscalar DLX processor. Upon completing the execution stage the instruction results are then passed on to the writeback stage which handles the reordering of instructions and determines when instructions can be committed. From the writeback unit the data is finally written back to the register file for other instructions to use.

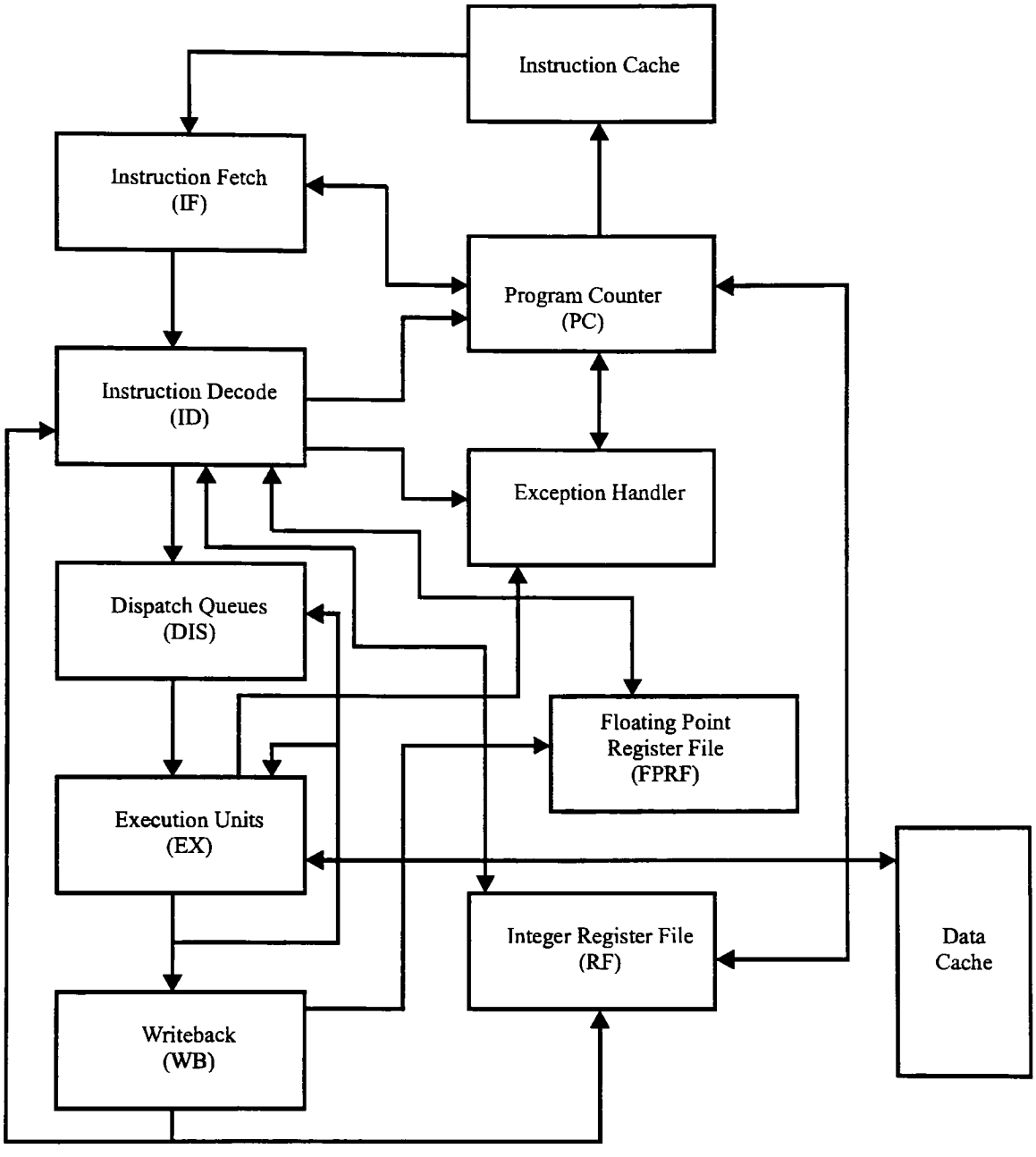

Figure 2.1 - General Block Diagram of the Superscalar DLX Microprocessor

The goal of this superscalar DLX processor is to achieve a CPI ratio of less than 1.00, so that it will exceed the maximum performance of a processor with a single pipeline. There are a number of superscalar techniques that this processor utilizes in order to achieve this goal. First, the design incorporates out-of-order issue and out-of-order completion of instructions with the use of a register file and a reorder buffer. The reorder buffer differs from a standard reorder buffer in that it does not actually contain the uncommitted instructions, the register file does. The reorder buffer does track finished instructions and signals the register file when a register can be released. So, working in combination with the register file this reorder buffer provides the same functionality as a reorder buffer or Tomasulo's algorithm. The processor also employs algorithms for speculative execution and branch prediction, and has a data forwarding mechanism, all of which aid in achieving the goal of a low CPI ratio.

In order to accurately portray the limits of this processor design, the memory model has been implemented with a latency of zero, however bandwidth limits are enforced.

2.1 Instruction Fetch

The instruction fetch (IF) stage of the processor, shown in Figure 2.2, is made up of three logic blocks: the instruction fetch, the branch target buffer (BTB), and the program counter (PC). The purpose of these three blocks is to read instructions from the Instruction Cache, via a 128 bit bus, at the rate of four instructions per clock cycle, and to determine the address of the next instructions.

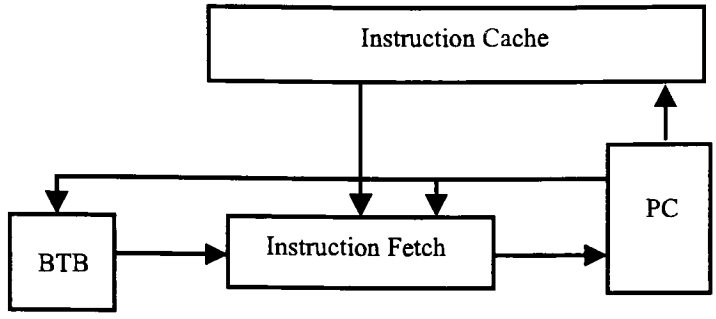

Figure 2.2 - Instruction Fetch Stage with Instruction Cache

After the instruction is fetched, it is compared to the PC to see if it is a needed instruction. An example of this is shown in Figure 2.3, where the address ends in "0100"

128 bit Input Line

	Instruction 0	Instruction 1	Instruction 2	Instruction 3
	Invalid	Valid	Valid	Valid
Least Significant 4 bits	X"0"	X"4"	X"8"	X"C"

Program Counter = X"00028AD4"

Figure 2.3 - Instruction Invalidation Example

This implies that instruction 0 is not to be executed and is therefore marked as invalid. During the invalidation process, instruction addresses are also sent to the BTB to determine if any of these instructions are branches. The use of the BTB allows the processor to determine the next instruction on a branch immediately rather than having to wait until the branch instruction is resolved.

If a branch is detected, the instructions that follow the delay slot instruction are marked as invalid. The new address for the PC is then sent to the PC where exceptions,

mispredictions, and special instructions are all taken into account before the new address is sent to the instruction cache.

During this stage, instruction numbers are assigned to each instruction. This is done to assist in the reordering process that must occur during the writeback stage of execution. The instruction number is a ten bit binary number which is incremented for each valid instruction fetched. The choice of 10 bits was made based on the maximum possible number of instructions that could be in the processor at once, which is approximately 80. To eliminate the possibility that the same instruction number would appear twice in the processor at the same time, a range of 1024 was chosen, but ranges of 256 or 512 could also have been chosen. These instruction numbers are used extensively during the execution of instructions to determine precedence and inter-operation relationships.

The following sections describe in detail the three major components of the IF stage: memory, the program counter, and the branch target buffer.

2.1.1 Memory

The memory system used in this processor model is a hybrid of a three level random access memory system. The typical three level RAM model would consist of on-chip cache, 2nd level cache, and main memory. For this processor a hybrid 2 level memory was created for simplicity, since memory architecture was not the main focus of this work. The memory was divided into a main memory section and a cache section. The main memory is similar to conventional main memory in that it holds the entire program once it

is read from "disk" For simulation purposes, the main memory was limited to 2^{20} bytes or 1 MByte of memory, with the test programs limited to this size as well. Using this 1 Mbyte limit also allows the processor to be freed from the complexity of page faults.

The significant area of difference between the two models is in the interface. The main memory is dual ported and therefore allows both data and instruction caches to be satisfied at once. When there is a conflict between a read and a write at the same memory location, the read is delayed. This is not true for conventional main memory which is almost always single ported and uses a bus to pass data to the two different caches.

The second level of memory is split into separate instruction and data caches, as is found in a Harvard architecture. Both caches interface with the main memory via 128 bit data busses. The 128 bit data width requires the address sent to the main memory to be only 28 bits, rather than the full 32. Table 2.1 lists the addresses of the sixteen bytes that are in a 128 bit memory line and shows that only the upper 28 bits are required to access the memory. Another reason that only the most significant 28 bits of the address are used is that it forces memory access alignment, so that only 128 bit rows can be read from memory from addresses that have "0000" for the four least significant bits.

Memory location address (32-bits) in hexadecimal
03020300
03020301
03020302
03020303
03020304
03020305
03020306
03020307
03020308
03020309
0302030A
0302030B
0302030C
0302030D
0302030E
0302030F

Table 2.1 - Addresses of 16 Bytes of Memory in One 128 Bit Main Memory Access

The use of the Harvard architecture removes the structural hazards that occur when one instruction is being fetched and another is loading an operand.

The 8 KByte instruction cache is direct mapped and passes 128 bits at a time to the processor. This allows the processor to receive four instructions per cycle to support an issue rate of up to four instructions per clock. The choice to direct map the instruction cache was supported by the physical memory addresses. Virtual memory, while allowing multiple programs to run at the same time with the full address space, complicates the addressing. This being an academic architecture, virtual memory was ruled out since it adds more complexity to an already complex design.

The data cache is 8Kbytes in size and configured to be byte accessible. The data cache is byte, half-word, and word accessible with the access type being determined by four select lines, which are asserted based on the least significant two bits of the address, shown in Table 2.2. To accommodate the spatial variety in data, the data cache is two way set associative. The replacement strategy is least recently used (LRU), based on a 4 bit shift register. The data cache is also designed to follow the writeback protocol. In this manner the number of writes to main memory are reduced. For simulation purposes there is a flush data cache line added which forces all the dirty cache lines to be written back to main memory. Since this design is for simulation only, the flush occurs in zero time. On a write miss, the cache line is allocated from main memory and placed in the cache, then written to and marked as dirty.

Least Significant 2 bits of Address	Byte Access Byte Select Lines	Half-word Access Byte Select Lines	Word Access Byte Select Lines
00	1000	1100	1111
01	0100	Illegal	Illegal
10	0010	0011	Illegal
11	0001	Illegal	Illegal

Table 2.2 - Byte Select Lines for Different Addresses and Memory Access Types

The access speed of this memory is not specified in the VHDL model. The memory access time for this project is assumed to be instantaneous, which, while not realistic, helps to keep the memory interface simple. This simplification allows the focus to be placed on the limits of the processor rather than the limits imposed by the memory.

2.1.2 Program Counter

The program counter (PC) in this processor determines the next instruction address. There are four inputs to this block: the next instruction address from IF, the address from the BTB on a branch instruction, the correct address from a resolved branch that was mispredicted, and the address of the exception handler if an exception has occurred. The order of priority is based on the instruction's position in the program. For example, if instruction number thirty-five causes an exception and at the same time instruction thirty-two resolves a mispredicted branch, the exception will be ignored since the mispredicted branch occurred prior to instruction thirty-five's execution. Essentially, instruction thirty-five never happened, so there is no exception to resolve.

Along with selecting the next address to send to the instruction cache, the program counter also determines when instructions should be invalidated, such as on a misprediction, exception, or trap instruction. When such an event occurs, the PC sends out two signals. One is a bit signaling that an invalidation must occur, the other is the instruction number or delay slot instruction number of the instruction that caused the exception or branch. The other units will invalidate all instructions that follow that instruction number.

The new program counter and instruction number are sent to the IF for instruction numbering and instruction validation. Therefore, in this design the PC not only increments the current program counter register to step through straight line code, but also handles the control switches associated with exceptions and branches.

2.1.3 Branch Target Buffer

A Branch Target Buffer (BTB) is content addressable memory (CAM) that contains the address of branch instructions and the destination of the taken branch. The BTB works in conjunction with the branch prediction unit, which is described in section 2.4.5, to determine the next address based on the result of the branch instruction.

In the scalar DLX pipeline the BTB supplies the next address to the PC on branches that are predicted taken, then in the next clock the branch prediction unit determines the result of the branch. On a correctly predicted branch there is no delay, but on a mispredicted branch, the most recently fetched instruction is invalidated, unless delay slots are being used, and the correct address is sent to the PC. After the branch is resolved the prediction table and the BTB are updated with the new result. This involves updating the branch prediction history bits and changing the BTB. The BTB will add the branch and its target address on taken branches that were not present before. Also, if a branch was in the BTB but was not taken then the entry is removed.

The design of the BTB in the superscalar DLX model functions in a similar manner, however the out-of-order execution found in the superscalar processor complicates the process. When a branch instruction is decoded the register to be tested might not be calculated yet and therefore the branch cannot be resolved. To alleviate this problem, the superscalar DLX has a separate unit, the branch dispatch unit, that holds the branches that are unresolved until the register is available to be evaluated. As the registers are written with their correct values the branch is executed, and, based on the taken or not

taken result, the branch prediction history bits are then updated and passed on to the PC. Once the branch prediction bits are updated, the prediction criteria are applied to them and if the prediction result changes, the BTB is altered to reflect this change.

The BTB in the superscalar DLX processor accepts the address of the first instruction of the four just fetched from the PC, at the same time the IF is getting the address. This address, plus the three subsequent addresses, are then checked against the stored addresses to see if any are a branch instruction. On a hit, the predicted address is sent to the IF stage. If all four instructions are branches, then all four next addresses would be sent to the IF stage, where the decision would be made as to which was the correct next address to send to the PC. The other function of the BTB is to update the look-up table, which occurs when updates are received from the branch unit. The data from the branch unit has a bit that selects whether it is an update or removal and the entry is added or removed from the table as required. When the BTB is full and a new branch is encountered, then the least recently used branch entry is removed and replaced with the update. The entry removed is determined by a 16 bit shift register that is updated with a '1' for every access, and a '0' for every clock that no access occurred.

Branch instructions are handled in this manner, allowing for speculative execution across multiple branches prior to the resolution of the actual branch instruction. This architectural choice requires either the branch prediction to be more accurate, or a very fast recovery method to limit the misprediction penalties in order to achieve a significant gain in performance.

2.2 Instruction Decode

The instruction decode stage consists of two blocks: the decoders, and the register files, which are also part of the writeback stage. The purpose of this pipeline stage is to decode the instructions and fetch the source and destination registers for the four instructions. This must all happen within a single clock cycle. The data extracted and passed on to the dispatch units during the decode stage is shown in Table 2.3.

2.2.1 Decoding

Decoding an instruction involves partitioning the instruction into the sources, destination and function. The way an instruction is broken down depends on the instruction type and function. For instance, two R-type instructions such as ADDF (single-precision floating point add) and MOVF (single-precision floating point move) have the same fields, but the MOVF does not use the second source field.

The information that is extracted and passed on to the subsequent stages is shown in Table 2.3. While this information may seem excessive, it allows each decoder to decode any instruction. Although this is practical for simulation, it is area prohibitive in fabricated designs. Much of this information could be extracted locally via case statements on the function, but for simulation it was easier to pass the information along already broken out, rather than creating a large case statement in every logic block. A real design would benefit from the local extraction via multiplexers, because there would be a reduced number of traces that must be routed across the chip. Since the actual details of modern processors are highly proprietary, it is not possible to determine if this logic holds,

or to tell if it is practiced in production designs. In future versions, if this design is to be synthesized or placed into a VLSI layout, the number of lines would need to be reduced for simplicity and performance reasons.

Name	Size	Description
Valid Bit	1 bits	Signifies that this instruction is still valid
Instruction Number	10 bits	This is the number of the instruction in the execution sequence, used for reordering and invalidations caused by control changes.
Source1	32 bits	The 32 bits of data for integer operations and the lower 32 bits of data for double precision floating point operations of operand one.
Source1a	32 bits	The upper, most significant, 32 bits of a floating point value and used to hold the offsets for load and store operations, otherwise it is unused.
Source1 Register or Data	1 bits	A '1' signifies that the value contained in the lowest 7 bits of the Source1 field is the register number of where the source will be, and a '0' signifies that the data in the Source1 field is the actual data.
Source1 Single or Double	1 bits	Used for floating point operations to signify when source one is a single precision (0) or a double precision (1) value.
Source1 Type	2 bits	These two bits indicate where the data is coming from: '00' Integer register file, '01' floating point register file, '10' special registers, and '11' immediate value.
Source2	32 bits	The 32 bits of data for integer operations or the lower 32 bits of data of operand two for double precision floating point operations.
Source2a	32 bits	The upper, most significant, 32 bits of a floating point value and used to hold the offsets for load and store operations, otherwise it is unused.
Source2 Register or Data	1 bits	A '1' signifies that the value contained in the lowest 7 bits of the Source1 field is the register number of where the source will be, and a '0' signifies that the value in the Source1 field is the actual data.

Source2 Single or Double	1 bits	Used for floating point operations to signify when source one is a single precision (0) or a double precision (1) value.
Source2 Type	2 bits	These two bits indicate where the data is coming from: '00' Integer register file, '01' floating point register file, '10' special registers, and '11' immediate value.
Destination	5 bits	These 5 bits specify which virtual register the result should be written to.
Destination Single or Double	1 bits	Specifies whether the result is single or double precision during floating point operations.
Destination Type	2 bits	Specifies where the result should go '00' Integer Register File '01' Floating Point Register File '10' Special Registers '11' No result is written
Function	6 bits	Determines which function will be performed in the appropriate execution unit.
Dispatch Queue	2 bits	Selects which dispatch queue the instruction goes to: '00' ALU dispatch '01' Floating Point dispatch '10' Branch dispatch '11' Load/Store dispatch
Instruction Address	32 bits	The address of the instruction, used to create the return from exception address when an exception occurs.

Table 2.3 - Information Extracted from Instruction During Decode Stage

The operand information extracted from the instruction is then passed on to the register file. The source values that are returned from the register file could either be the actual data or the physical register that will eventually contain the required data. This is denoted by the value of the register_or_data signal, with a value of '1' signifying a register number and a '0' representing the actual data. The destination is also sent to the register file and a new register is allocated with the virtual register number assigned to it. This is

done so that subsequent instructions will reference the most recent mapping of that register number.

While the operands are retrieved from the register file, the decoder determines to which dispatch queue the instruction should be sent. The possible queues are integer, floating point, load/store, or branch queue. The correct function number is also determined for the given dispatch unit.

2.2.2 Register File

The register file is a block of very fast associative memory that holds the register values during program execution. The DLX ISA specifies 32 general purpose registers and 32 floating point registers. This is adequate in a scalar pipelined processor, but data and structural hazards arise when a design is made superscalar. This superscalar DLX design contains 128 32-bit general purpose registers and 128 32-bit floating-point registers, but the programmer can only reference registers R0 to R31 as specified by the DLX ISA. The reason for this large number of registers is to support out-of-order execution. The additional registers are used for a register renaming scheme which serves to reduce the additional WAW and WAR hazards caused by out-of-order execution.

Register renaming is a process that maps one of the ISA specified 32 virtual registers to one of the 128 physical registers. This process was first introduced in the IBM 360/91 and was invented by Robert Tomasulo (Hennessy, 1996). This implementation differs from the reservation stations specified in Tomasulo's algorithm in a few ways. First, one large pool of registers is used, rather than the smaller distributed

blocks specified by Tomasulo. The benefits of having one large pool is that registers can be reallocated dynamically, as in a unified cache. This allows the registers to be assigned to the pipeline or virtual register that is being used the most, rather than having a fixed number per pipeline or virtual register.

2.2.2.1 Register Renaming

Register renaming allows for instructions to execute out-of-order and still avoid data dependency hazards. The following code segment will be used to illustrate how register renaming works.

mov	R_v1, R_v4, R_v2
add	R_v5, R_v1, R_v2
sub	R_v6, R_v1, R_v5
mult	R_v5, R_v12, R_v13
div	R_v10, R_v5, R_v7

Note: R_v# *represents virtual register # of 32*
R# *represents physical register # of 128*

For the purposes of describing how register renaming eliminates WAW and WAR hazards, assume that these instructions are being executed on a dual issue processor, which yields the execution sequence below. For this example, all the instructions listed above are single cycle instructions, and the processor contains full data forwarding.

Pipeline 1	*Pipeline 2*
mov	*mult*
add	*div*
sub	

The preceding text represents how the instructions are dispatched into the execution units. Without register renaming, once the 'mult' instruction completes, the

'sub' instruction will improperly use the newly created R5 value as one of its sources. By mapping the virtual registers to physical registers as shown below, this hazard is eliminated. Also, the WAW hazard of the 'add' instruction writing its results to R5 after the 'mult' instruction has already updated the register, is also avoided.

mov	R_v1	→	*R43*
	R_v4	→	*R31*
	R_v2	→	*R24*
add	R_v5	→	*R44*
	R_v1	→	*R43*
	R_v2	→	*R24*
sub	R_v6	→	*R45*
	R_v1	→	*R43*
	R_v5	→	*R44*
mult	R_v5	→	*R46*
	R_v12	→	*R25*
	R_v13	→	*R8*
div	R_v10	→	*R47*
	R_v5	→	*R46*
	R_v7	→	*R15*

With these mappings in place the instructions now look like the following.

mov R43, R31, R24
add R44, R43, R24
sub R45, R43, R44
mult R46, R25, R8
div R47, R46, R15

Now the true dependencies can be seen between the instructions, and the execution sequence shown previously can occur without a hazard. The only problem remaining is to determine when a register mapping can be released, which is discussed later in section 2.5.2.

2.2.2.2 Integer Register File

The integer register file (RF) is made up of 128 records. Each of these records is composed of thirty-two bits of data, a 5 bit virtual register number, an in-use bit, and a data valid bit. To satisfy all of the structural requirements that this superscalar design imposes, the interface to the registers has 8 read ports and 5 write ports. The problem of reading and writing to the same register location is one that is usually handled by staggering the events on different phases of the system clock in production designs (i.e. writes during phase 1, and reads during phase 2). In this design, the reads and the updates are all done during the first phase of the clock. To eliminate the problem of decoded instructions missing the register updates, the data lines from the writeback stage are also routed to the decoders. If any of the source registers have been updated, the newly decoded instructions are updated.

When there are no longer any registers available to assign to the destinations of new instructions, the register file must force the IF, PC, and decoders to stall until the destinations become available. Stalling the IF and PC essentially stops the clock to these units until the decoder is able to finish with the current instructions.

When registers are released from a mapping they are marked as invalid. Depending on the exception recovery model enacted, the exact criteria for releasing the registers varies, but the general idea is that once there are no longer any references to this register, it can be released. As stated earlier, the exception model and release criteria will be discussed in more depth in later sections.

In addition to the 128 regular entries, the RF also contains 32 other registers. These registers are to insure the special registers, such as the interrupt address register (IAR), floating point status register (FPSR), and any others that may be defined in later versions of the ISA, are accounted for. When instructions are decoded, the destination and sources are sent to the RF along with a separate line which, when asserted, signifies that the source or destination is a special register. At this point, the processor will cease operation if there are no available special registers. Since the current ISA has only two instructions that can write to the IAR and only the floating point arithmetic writes to the FPSR, it will be unlikely that there will be more than 32 live references to the FPSR and IAR at any one time. In future versions of the processor, if the number or frequency of special register accesses increases, then stall capability will have to be added to this section of the register file as well.

2.2.2.3 Floating Point Register File

The floating point register file (FPRF) is very similar to the RF. The FPRF contains 128 elements that consist of a 32-bit data element, a 5 bit virtual register number, a valid bit, an in-use bit, and a single or double precision bit. The main difference between the RF and the FPRF is that the FPRF has the ability to combine two registers into one virtual 64-bit double precision register. The restriction on this option is that the virtual register number must be even, i.e. R0, R2,..,R30. In this model an assert statement will alert the user that they have made an illegal assignment.

In double precision form, the most significant 32 bits are stored in the lowest addressed register, so that when addressing a double precision element in R2, R3 contains bits 32 to 64 and R2 has bits 0 to 31 with bit 0 being the most significant bit (MSB).

As with the RF, the FPRF also has the ability to stall the decoders, IF, and PC should there be no available registers for the destinations.

2.3 Dispatch

The dispatch stage in this superscalar DLX processor is responsible for determining when an instruction can be executed. The dispatch stage is split into four separate dispatch queues, shown in Figure 2.4. Each of the dispatch queues acts independent of the others, however information is shared between queues to determine when dependent data will be available. Figure 2.4 also illustrates the interconnects that allow this information to be shared.

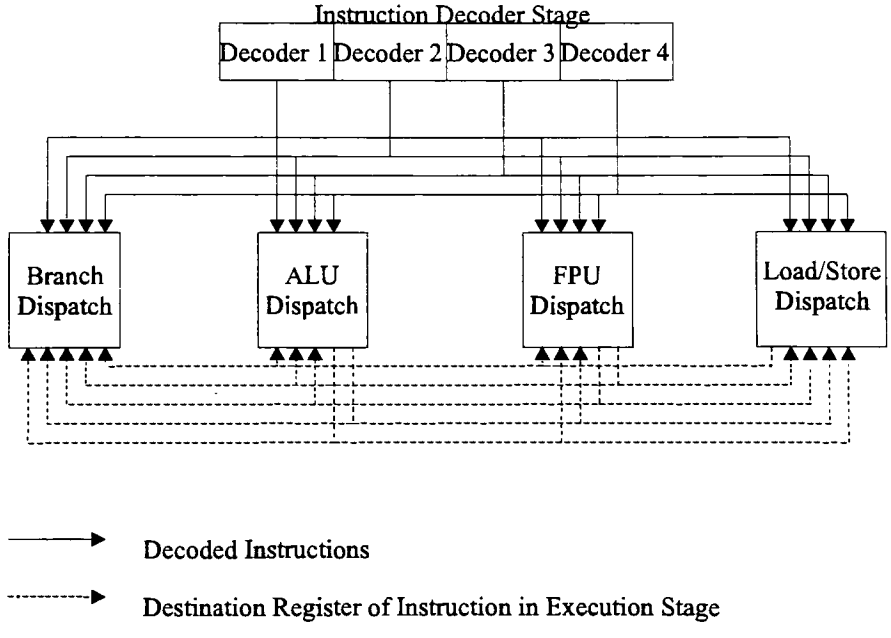

Figure 2.4 - Data Paths from Decoders to Dispatch Units

When designing a dispatch queue or reservation station, which a distributed dispatch queue is called, the major issue is the size of the queue. The larger the queue, the more instructions available for execution, which reduces the possibility of a stall. The problem with a larger queue is speed. The larger the queue, the longer it takes to determine the next instruction to send. The factors used to determine the size of queue include the process technology, circuit design and processor speed.

The dispatch queues implemented in this superscalar DLX processor are each 16 entries in size. This allows up to 64 pending instructions. The operation of the dispatch queues is split into two phases, one for each phase of the clock, which are called phase_one and phase_two. The main role of phase_one of the dispatch queue is to allocate space for the new instructions and to update the entries with the new data that is forwarded from the execution stage. Phase_two is responsible for issuing the oldest ready instruction that it currently contains.

Phase_one of the dispatch queue is split into three parts. These are invalidate, add instructions, and add data. On the rising edge of phase one of the clock, the invalidate section is activated if there has been an invalidate signal from the PC. When an invalidate has occurred, the dispatch queue is searched for any instructions that occur after the invalidated instruction. If such an instruction is found, the valid bit in that record is cleared and the slot is now available for new instructions.

Once the invalidations are complete, if they were required, the four instructions that were just decoded are checked to see if they are intended for this pipeline.

Instructions destined for this dispatch queue are then placed into available slots in the queue. In the case that there are not enough slots available for all of the new instructions, the dispatch queue issues a stall to the decoder until enough slots have been freed to store the new instructions. The slots are freed either by invalidations or by issuing instructions to the execution units. If one dispatch unit stalls, the decoders and then the other dispatch queues must also stop accepting new instructions in order to prevent multiple copies of the same instructions from being stored. Once the new instructions are placed into the queue, the forwarded data from the execution units must be checked.

The process of checking the forwarded data is done by stepping through the instructions in the queue. Any instruction that has a source that is marked as a register is checked against the new data coming form the execution units. When there is a match between one of the forwarded result's register number and type and the stored instruction's source register number and type the data is moved into the dispatch queue entry and the source is marked as valid. When a source of an instruction is marked as valid, the other source is checked for validity as well. If both sources are now valid, then the entire instruction is marked as 'ready to issue'

With phase_one complete, when phase two of the clock transitions to high, the phase_two part of the dispatch queue must determine which instruction to send to the execution stage. Not all dispatch queues access all the execution units, Figure 2.5 below illustrates which dispatch queues connect to which execution units. The ALU dispatch queue and the FPU dispatch queue are both capable of dispatching two instructions per

clock. The branch queue and load/store queue, on the other hand, can only issue one instruction per clock. The decision making process for each queue is the same. To determine the next instruction to send, each instruction in the queue is checked to see if it is ready to execute. To be classified as 'ready to execute', the instruction must either have its 'inst_ready' field set, or have its sources available via data forwarding from an instruction that is currently in one of the execution stages. After the next instruction has been chosen, it is sent to the execution unit and its entry in the queue is removed. Upon removing the entry, phase two is complete and phase one is ready to start again.

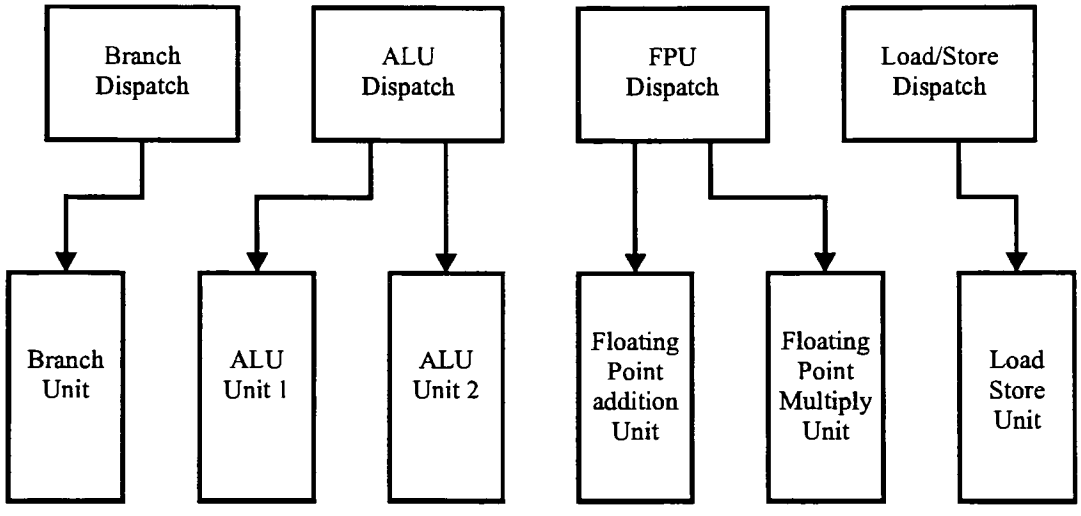

Figure 2.5 - Data Paths from Dispatch Queues to Execution Units

The load/store dispatch queue has a slightly different mode of operation. Along with operating as above, there is also another mode in which nothing happens during phase two. This mode occurs when a double load or store occurs, since the execution unit requires two clocks to perform the double loads or stores via its 32 bit data bus.

The nature of this design places the responsibility of managing the registers on the register file, freeing all other processor components from these details. By the time the instruction reaches the dispatch queue, the register file has made the mappings required to handle the WAW and WAR hazards. All execution units, with the exception of the load/store unit, are single cycle units, which allows the dispatch queues to send out a new instruction on every clock. This removes any possible structural dependencies. Finally, data forwarding removes the RAW hazards that can occur when data dependencies exist.

2.3.1 Out-of-Order Issue

Out-of-Order issue is the main stay of modern processor performance. This issue strategy allows the processor to keep the execution units full for a larger percentage of time with respect to an in-order-issue policy. The dispatch queue, register file, data forwarding, and writeback buffer all work together to allow out-of-order execution. The register file removes many data dependencies via register renaming. The other data dependencies are handled via the data forwarding of results back to the execution and dispatch units. Until the instructions get to the dispatch queue they are still in order. It is only after they leave the dispatch queues that their order has been changed. When an instruction is ready to issue, it is sent to the execution stage whether the instructions before it have gone or not. The reorder buffer in the writeback stage has the responsibility of guaranteeing correctness. The instructions are not allowed to make permanent changes to the register file, or commit, until all prior instructions have completed and have been committed. While complex, the out-of-order issue and out-of-order completion strategy

provides a justifiable performance increase and is used in many of today's high performance processors from HP, AMD, Intel, and others.

2.4 Execution

The execution stage of the superscalar DLX processor is composed of six parallel units. These execution units include two ALUs (ALU1, ALU2), a floating point adder (FPadd), a floating point multiplier (FPmult), a load/store unit, and a branch prediction unit. The organization of these units can be seen in Figures 2.2 and 2.6. The execution stage of this superscalar DLX processor differs from that of the scalar processor in that it includes additional arithmetic units, a memory unit, and a branch resolution unit. This allows the superscalar DLX to issue up to six instructions on every clock and pass six results on to the writeback unit on every clock. However, this is accomplished by modeling the floating point multiplier and load/store units as single cycle units, which is not realistic. The load/store unit should have at least a one cycle delay but as was stated before, the memory model for this design was created with no time delay. This discrepancy allows the focus of design issues to be placed on the processor design rather than the memory architecture which, while important, is not the focus of this thesis. The simplification of the floating point multiply unit was due to time limitations. The accurate portrayal of a floating point multiplier and divider would be a sizable undertaking in itself, so the choice to use the VHDL library constructs was made. It is in these areas that future revisions of this thesis may include further research and attention.

By simplifying the multiplier and divider designs, additional time was provided to implement the entire DLX instruction set. This complete implementation allows for more programs to be run on the model, and for the different interactions between the functional units to more closely simulate a production superscalar processor.

In the next few sections, more detail is provided on each of the functional units. In addition, detail is provided on the data forwarding mechanism which allows many of the data dependency delays between instructions and functional units to be minimized, if not removed completely.

2.4.1 Data Forwarding

Data forwarding was created to help remove pipeline stalls that would occur when two sequential instructions shared a piece of data. In the standard DLX pipeline, once the result was calculated by the ALU it would not be available to subsequent instructions in the registers until three clock cycles later. Data forwarding allowed the result to be available to the immediately following operation on the next clock cycle. The superscalar DLX processor does not have the MEM stage between the execution and the writeback stages, which would allow data to be used after a two clock cycle delay. This is still not acceptable since it would force many more instructions to be delayed since there are six execution units instead of one.

The increased number of execution units also demands that there be a larger number of data forwards. There are a total of five sources for data forwards: the two ALU units, the two FPU units, and the load/store unit. If the data forwards from the writeback

stage were also implemented, as the scalar pipelined DLX does, there would be ten sources for data forwards, which is quite a large number. The implementation of this processor avoids this large number of data forwards by designing the five feedback lines to the dispatch queues, as well as the execution units. This way the operands receive the new data prior to it being written back to the register file, the same as if the scalar data forwarding had been implemented. Figure 2.6 shows the data forwarding lines in relation to the execution units and the dispatch queues.

Another measure was also taken to insure data is available for future instructions. The result data that is being sent to the register file, from the writeback unit, is also sent to the decoders. This allows instructions which were just decoded to have access to the data that was just put into the register file, but was not available when their operands were fetched.

In the scalar DLX processor, the reads and writes to the register file are staggered so that the writes occur before the reads. This option was not implemented in the superscalar DLX due to a few design choices made with respect to the decoders and the writeback stages. Changing the register file access to more closely resemble that of the scalar DLX would be an improvement for later versions that are implemented in a more structural VHDL description. This change would improve the design since it would reduce the number of lines on the chip and provide a better solution for fabrication.

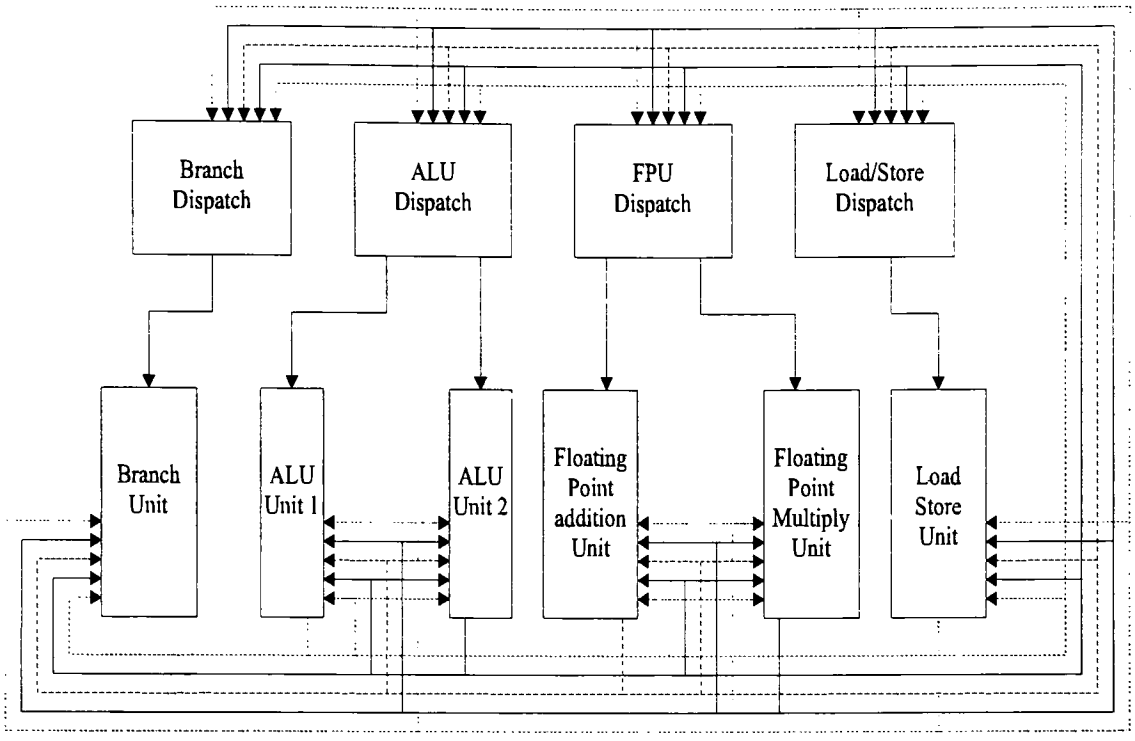

Figure 2.6 - Data Forwarding Paths from Execution Units

The data passed to the ALU via data forwarding is shown in Table 2.4. This data includes the physical register number, data, type, and validity. If the src_data bit is set for either of the source operands, then forwarding is used. The addresses of the destination registers for the forwarded data is compared with the lower seven bits of the source fields in the input data. When there is a match, that forwarded source is used for the operand in the operation. All of the execution units in the superscalar DLX processor use data forwarding, which alleviates some of the data dependency delays.

Forwarded Information	Size	Used By
Data	32 bits: alu1, alu2 64 bits: Floating Point Add, Floating Point Multiply, Load/Store unit	All Units
Register	7 bits	All Units
Single or Double	1 bit	Floating Point Units Load/Store Unit
Type	2 bits	All Units
Valid	1 bit	All Units

Table 2.4 - Description of Forwarded Information

2.4.2 Load/Store Unit

The load/store unit is responsible for bringing data into the processor and placing it back into memory. This block of the processor is one of the bottle necks. The reason for this is that there can be two floating point instructions, two integer instructions, and a branch instruction all executing at the same time, but there can only be one load or store happening to get the data for the five other instructions. A way to alleviate the bottleneck is to have multiple ported RAM. While this would seem like the easiest method to solve the problem the multiple ports create a more complex memory structure which leads to a greater expense, and possibly reduced performance.

An alternative to multiple read/write ports, which is used in many processors today, is load bypassing. Load bypassing occurs when a load and a store are both ready to execute, and the load is sent first even if the store instruction is older. This method allows the data to get into the processor as fast as it can. If the load attempts to retrieve a data

element that has not been stored yet then the data is forwarded from the pending store instruction.

The load/store strategy that was used for this processor is based around the idea that if the load or store is ready, then execute it. The main reason behind this was the assumption of no memory delay. The one restriction placed on the store instructions is that all instructions prior to the store instruction must have committed prior to the store being allowed to execute. This restriction enforces the requirement that any instruction that does not commit does not place the processor into an unrecoverable state. The writing to memory by an invalid instruction would be such an event.

During the execution of the instruction the address alignment is checked. Table 1.2 illustrates the required alignments for specific data access modes. When an instruction tries to fetch from or store data to a mis-aligned address, an exception occurs. The destination of the exception is shown in Table 2.7, which is discussed in section 2.5.2. After the alignment is verified the byte select lines are set according to Table 2.2. In the Pentium™ processor the byte select lines determine which bytes of the data are accepted into the processor, where the data is aligned and sent on to the writeback unit. The method used in this superscalar DLX processor is to have the data cache parse the data based on the byte select lines and send the right-justified data to the processor. One example of this is to read a byte from hexadecimal address X"0000234D" The address's two least significant bits are '01', therefore the byte select lines are set to '0100'. The data received from the date cache would have the upper 24 bits zeroed out and the

information would be in the lowest 8 bits. This data can then be written directly to the register file via the writeback stage. To simplify the data cache, the task of justifying the data could be brought into the chip in future versions.

The specific design of the DLX load/store unit is split into two sections, one active during phase one of the clock and the other active during phase two. Information is passed between the two processes via signals.

The phase one process is responsible for receiving the instruction from the dispatch queue and determining which instruction it is. Once the instruction has been identified, the byte select lines are set. If the instruction is a read the data is put onto the data bus, otherwise the data bus is placed in high impedance mode. At this point the data address is placed on the bus and the address strobe is asserted. The only other signals affected are those concerning double precision loads or stores, which will be discussed later.

During phase two of the processor clock the data bus is read if the data ready line is asserted, otherwise nothing happens. If the instruction is either a load half (LH) or load byte (LB) instruction then the data is sign extended to word length. The final step of phase two is the assignment of values to the signals going to the writeback queue.

The previous description is valid for all load and store instructions except for the double load (LD), double store (SD) and load high immediate (LHI). The load high immediate instruction is handled by the ALU since memory access is not required. The double precision loads are handled in a similar fashion, except that they take two cycles. To signify that the instruction is double precision, a signal labeled 'double' is set during

the first cycle and cleared during the second cycle. When the signal 'double' is asserted, another signal going to the load-store dispatch is also asserted. This extra signal lets the dispatch queue know that a double load or store is taking place and that it should not dispatch another instruction on this cycle.

2.4.3 ALU

The arithmetic logic unit of this model handles sixteen different functions including add, sub, shift, and all of the integer set-on-comparison functions. During phase one of the clock, the ALU accepts data from the dispatch unit. The pipelined implementation of the processor mandates that the ALU support data forwarding to help eliminate data dependency stalls.

In deciding on the type of adders to implement for the core of the ALU, there are three major varieties. These include Ripple Carry, Carry Look Ahead (CLA), and the Carry Select Adder (CSA). The Ripple Carry adder is the easiest to design since only one bit is calculated at a time and the carry-out is then propagated on to the next bit. While simple, this design creates the longest delays. A 32 bit adder would incur 32 full adder delays, which even at half a nanosecond each, would require the clock period be no shorter than sixteen nanoseconds. The Ripple Carry adder is no longer a feasible design option for today's high speed designs, which leads to the second design choice. The CLA adder combines multiple bits into larger groups and calculates the results for all the bits at once, by the equations in Figure 2.7. Equations E1 and E2 are the standard equations for the sum and carry-out for the simple Ripple Carry Adder. From these equations, the carry

generate equation, E3, and the carry propagate equation, E4, are derived. Using equations E3 and E4 in conjunction with the carry-in from the previous stage, equation E5, the current stage carry-out can be calculated at the same time as the previous and next stages. A more in depth description of the CLA and CSA can be found in Appendix A of (Hennessy, 1996).

$$E1: s_i = a_i \overline{b_i c_i} + \overline{a_i} b_i \overline{c_i} + \overline{a_i b_i} c_i + a_i b_i c_i$$
$$E2: c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$
$$E3: g_{i-1} = a_{i-1} b_{i-1}$$
$$E4: p_{i-1} = a_{i-1} + b_{i-1}$$
$$E5: c_i = g_{i-1} + p_{i-1} c_{i-1}$$

Figure 2.7 - Equations for Standard Adder and CLA Carry

Using this adder design requires more logic and transistors, which is allowable today since transistors are basically 'free' in today's process technologies when compared to the area required by interconnect wiring. CLA circuitry allows a 32 bit add to be completed in $\log_n X$ steps, where n is the group size and X is the total number of bits in the operation. So, for a 32 bit add instruction, the result would be available after three full adder delays, when using four bit groups, a vast improvement over that of the Ripple Carry adder.

The third design choice is not really a stand alone design choice since it is mostly used along with either of the first two designs. With the CSA, the result for all but the n lowest order bits is calculated in parallel for a carry-in of one and a carry-in of zero. Then, once the carry-in is known, the correct value is multiplexed out.

Hewlett Packard has taken these general concepts and expanded on them in their PA-8000 microprocessor. The adder, which is the core of their design, is based on an alteration of the standard CLA and CSA adders using a set of equations to reduce the number of inputs to the carry function. The specifics of the alterations were not researched but are available from the sources listed in (Naffziger, 1996). All the changes that HP made to the adder design resulted in an adder that provides the result to a 64 bit addition in less than one nanosecond. This superior performance allows the ALU and floating point adder to provide the results within the 10 nanosecond clock period of a 100 MHz design.

2.4.4 Floating Point Unit

The floating point unit, along with the ALU, are the heart of most high performance processor designs. The FPU implemented for this design is built around the IEEE754 standard for both single and double precision floating point numbers. The IEEE754 floating point formats are shown in Figure 2.8 and the specifications are shown in Table 2.5. A quad-precision notation is also available, but is not standardized, and rarely implemented in modern day processors. The lack of acceptance of a standard for the quad-precision numbers mandated that they be left out of this design.

IEEE 754 Floating-Point Standard

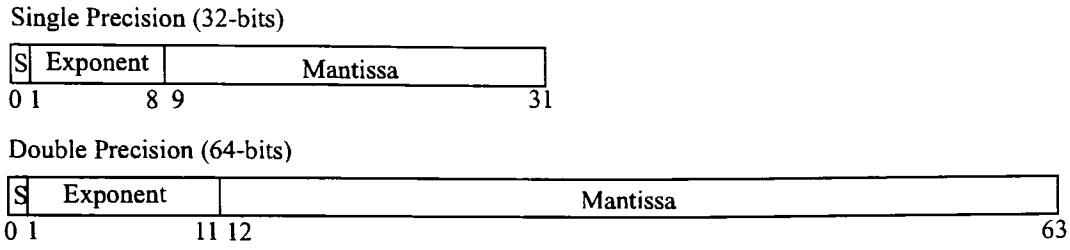

Figure 2.8 - IEEE754 Floating Point Standard

Precision	Total Length (bits)	Sign (bits)	Exponent Length (bits)	Exponent Range	Mantissa Length (bits)	Mantissa Range (base 10)
Single	32	1	8	+127, -126	23	10^{-38}, 10^{+38}
Double	64	1	11	+1023, -1022	52	10^{-308}, 10^{+308}

Table 2.5 - IEEE754 Floating Point Characteristics

There are a few options to consider when dealing with floating point arithmetic and precision. The first choice is to implement the circuitry so that both single and double precision operations can be properly performed. This method would create a slightly more complex design throughout the FPU since almost every data bus and functional unit would have to have two modes of operation, single and double precision. A simpler method would be to convert the single precision numbers into double precision, then perform all operations in double precision lengths. The tradeoff involved with having simpler hardware is the added latency caused by the larger number of bits. Instead of having a 24 bit addition for the single precision add, a slower 53 bit addition of double precision numbers would occur. The implementation of this superscalar processor kept

the functions separate as an arbitrary choice, but in the pure behavioral model there is no penalty for a larger number of data paths.

A few floating point instructions can cause exceptions to occur, specifically overflow and underflow exceptions. A floating point overflow occurs when a number becomes too large to be expressed in the given number of bits. When this occurs, the result of the operation is altered to represent positive or negative infinity, as shown in Table 2.6 (Stallings, 1990).

Exponent, e	Significand	Value
Single-Precision		
255	≠ 0	Not a Number
255	0	± Infinity
0<e<255	Any Number	Normal Numbers
0	≠0	Sub-normal
0	0	± Zero
Double Precision		
2047	≠0	Not a Number
2047	0	± Infinity
0<e<2047	Any Number	Normal Numbers
0	≠0	Sub-normal
0	0	± Zero

Table 2.6 - Values of IEEE754 Floating Point Numbers

The other area of concern is when a number becomes too small to be represented correctly. There are two methods to handle this event. The first method is to round down to zero and signal an underflow, which creates a sharp end to the numbers that can be represented. This can lead to incorrect results if the underflow is not properly handled by the operating system or programmer. A decimal example of this follows, with the lower exponential limit of 10^{-38}.

$$\begin{array}{r} 4.531593920 \times 10^{-38} \\ \underline{-\ 4.121582303 \times 10^{-38}} \\ 0.410011617 \times 10^{-38} \end{array}$$

The result would be rounded to zero. This is because in order to store a normal floating point number, the first bit or 'digit' must be non-zero. To round this result to an appropriate form would require the exponent to reach -39 which is out of the valid range. Therefore, the result would be zero, implying that $4.531593920 \times 10^{-38}$ equals $4.121582303 \times 10^{-38}$, which is not true.

The solution to this is to allow sub-normals, which are different from normal floating point numbers in a couple of ways. The first way is that the exponent is set to zero even though when the number is unpacked the exponent goes to -38 (or what ever the lowest allowable exponent is). The second way that they differ is in the implicit, or unpacked bit, which for standard floating point numbers is a '1', but for a sub-normal number it is a zero, which allows for results such as in the example above to be represented. The use of sub-normals allows the rounding to a zero value to be much more gradual and even. Sub-normal floating point numbers are supported in this superscalar DLX processor model.

In all floating point arithmetic operations, guard bits are used to retain the most precision possible. The potential loss of precision occurs when one of the operands is shifted. An example of guard bits is shown in Figure 2.9. The precision that is preserved by the guard bits is reclaimed during the normalization process. Normalizing a floating

point number is the process of shifting the number until the most significant bit (MSB) is a one, shown in Figure 2.10. During this process the exponent is decremented unless this would take it below the minimum exponential value. In the case where the lowest exponent has been exceeded the number becomes a subnormal and is stored with a zero value for its exponent.

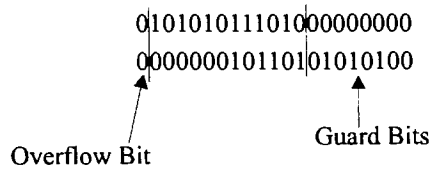

Figure 2.9 - Floating Point Guard Bits

Prior to normalization 0.00010111011010001 Exponent = 34

After normalization 1.01110110100010000 Exponent = 30

Figure 2.10 - Floating Point Normalization

There are a number of alternatives when rounding floating point numbers, such as truncation, negative infinity, positive infinity, and nearest. When dealing with rounding techniques there are a few terms that must be defined such as sticky bits, round bit and guard bits. The guard bits are the same as shown in Figure 2.9. The round bit is the guard bit immediately to the right of the last significand digit, and the sticky bits are the guard bits to the right of the round bit.

The simplest rounding technique is that of truncation. With truncation, the digits beyond the end of the significand are ignored, so for a significand of five digits the number

4.35249 would round to 4.3524. While simplest, truncation is not the most precise method of rounding.

The next two methods, negative infinity and positive infinity, try to round the result to the infinities. If a result is less than zero and either the round or sticky bits are equal to one, then the result is incremented by one under the negative infinity rule. If the result is positive and the sticky or round bits are equal to one, then the result is incremented by one under the positive infinity rule. The rounding method used in this superscalar DLX processor model is the nearest method. The nearest method specifies that if the round bit is equal to '1' then add one to the result significand, otherwise leave the result significant as is. The nearest rounding method is closest to what humans do with decimal numbers and remains neutral in its overall effect, in that it will not bias the results one way or the other.

2.4.4.1 Floating Point Adder

The floating point adder handles all of the floating point instructions except for the multiplies and the divides, which are handled by the floating point multiplier. The instructions handled by this floating point adder unit include all of the test instructions, floating point move instructions, conversion functions, as well as the addition and subtraction instructions.

The latency of these floating point instructions, with the exception of the multiply and divide instructions, has reached the level of a single cycle in many modern processors through the use of an increased number of transistors. An example of an extremely fast

core design is the PA-RISC PA8000 adder, mentioned in the previous section. This design allows a floating point operation to complete in one clock period at 100 or even 200 MHz. With an adder delay of less than 1 ns, there is time to do a couple of adds and shifts, and still get the data to the register before the end of the clock cycle, even at 200 MHz. Based on this information, the latency and issue rate for the DLX floating point adder unit is one clock cycle.

The process of floating point addition is actually more complex than that of floating point multiplication. The major steps in floating point addition and subtraction are: check for zeros, add guard bits, equalize the exponents, add the significands, check for significand overflow, round the result, normalize, final check for under or overflow, and then pack the result. The subtraction function is implemented in the same manner as the addition except for the inversion of the second operand which is required in twos complement subtraction. Each of the steps in the addition and subtraction process are shown in Figure 2.11, which was taken from (Stallings, 1990).

The other complex functions that are executed in the floating point adder unit are the convert instructions. These six operations handle the conversion between double precision, single precision and integer formats, in all the combinations. One of the issues that needed to be handled was when the data to be converted exceeded the capacity of the destination format, for instance trying to convert the single precision value 1.29384×10^{24} into an integer value where the maximum value in signed mode is 2,147,483,647. In cases like this a floating point overflow or underflow is signaled and the result is set to the

maximum or minimum value that it can be, so for the example above the integer value would be set to 2,147,483,647.

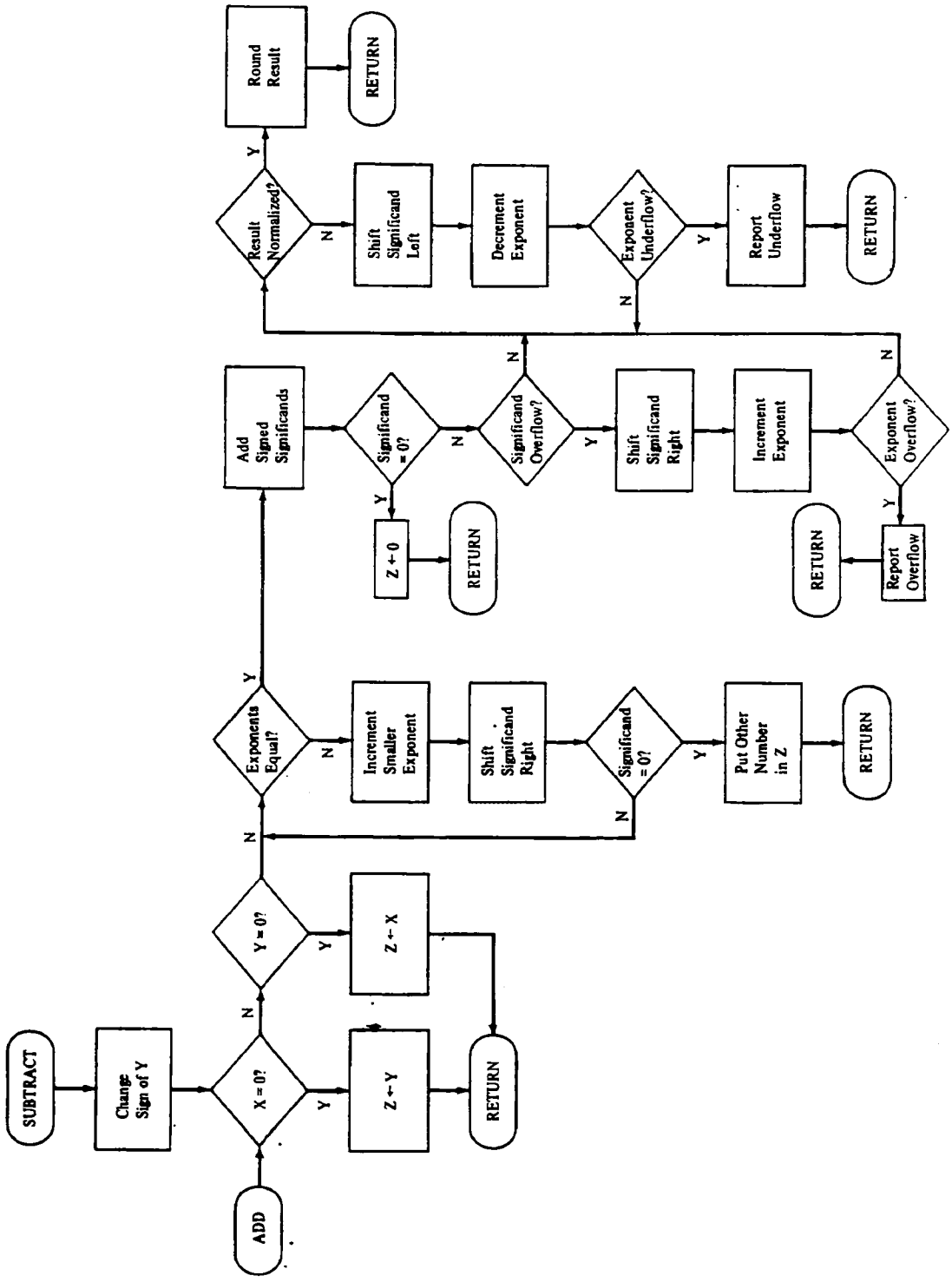

Figure 2.11 - Floating Point Addition and Subtraction, x ± y = z (Stallings, 1990)

2.4.4.2 FPU Multiply

The multiplication of two floating point or fixed point numbers is fairly simple but very time consuming. The simplest way to implement an n-bit multiplication or division is to execute a series of n shift and adds or n shift and subtracts. While simple, this takes the most time. The amount of time is on the order of n full adder delays. So even with the PA-8000 adder, the 53 bit multiplication required in double precision would still take approximately 50 ns.

This superscalar DLX processor uses the multiplication and division functions provided in the ARITHMETIC library of VHDL that was available in the Rochester Institute of Technology (RIT) computing environment. The multiply and divide functions accepted std_logic_vectors and performed the multiply or divide via the shift and add or shift and subtract method. Time constraints did not allow further development of a more precise and realistic model, i.e. one that takes multiple cycles as in a production processor. Future work with this design could focus on the implementation of the multiplier and divider to improve their performance, when represented in a more realistic manner.

Due to the similarities between computer multiplication and division, only multiplication is discussed here. The major difference between multiplication and division is in the shift and add stage. For multiplication it is right shift and add, and for division it is left shift and subtract. The first step in floating point multiplication is the addition of the exponents. Once this has been done and it is determined that the result did not exceed the limits for the exponent's value, the significand multiplication can occur. If the exponent's

maximum value is exceeded a floating point overflow occurs. The resulting length of the significand multiplication will be the sum of the lengths of the two sources. These extra bits are removed during the normalization and rounding of the result, as is done with the addition and subtraction instructions. The result of a division will be shorter in length than the dividend, and must also undergo normalization and rounding.

Time limitations prevented further and more in-depth research into this area, but this area would be an excellent area for future work.

2.4.5 Branch Prediction Unit

The accurate prediction of branches determines much of the performance benefits that can be gained by superscalar designs. It is obvious that the greater the accuracy of the prediction algorithm, the greater the performance increase the superscalar design will provide. If the accuracy of the prediction algorithm falls below a certain level, then the superscalar design may even have a reduced performance with respect to a scalar design because of overhead associated with removing the invalid instructions from the processor. If the prediction algorithm is correct 90% of the time, then the majority of the processor's clock cycles are being used to execute valid instructions. However, if the branch prediction algorithm is correct only 50% of the time then half the time the processor is cleaning up after missed branch predictions and not doing 'productive' work. Of the many branch prediction methods that exist, the strategy used in this model is the n-bit saturating counter. If the counter is greater than or equal to one half of the maximum counter value, the branch will be predicted to be taken, otherwise it will be predicted not taken. A

saturating counter is used because once the counter has reached the maximum or minimum value it will 'saturate' at that value and not wrap around. An example of this is when the 3-bit counter has a value of "111" and the branch is taken. This causes an increment of the counter, resulting in a counter value of "111" rather than "000" if a wrap around occurred.

Other techniques to predict branches include history bits in a shifter to determine patterns in the branch history, and learning algorithms which run on sample data sets and provide cues to the compiler which are then passed on to the processor. Many of the branch prediction algorithms require considerable extra logic, or considerable extra time to characterize data sets. One strategy used 24 history bits that were compared against a look up table to determine a taken or not taken result. The benefit of this method was that with a 24 bit history, smaller loops can be detected and predicted almost 100% of the time. Storing 24 bits for 256 or 1024 branches can get area intensive, however, often requiring between 750 bytes to 3 Kbytes of memory for just one table. Another alternative was to keep a global history of branches and base the prediction on what the last x number of branches had done. The look up table for this scheme contains 2^n bits of data for each branch entry, where n is the number of previous branches that are used to determine the prediction for the current branch.

The branch prediction unit in this superscalar DLX processor contains an array of 1024 3 bit saturating counters. During the first phase of the clock the instruction type is determined and the register is compared to determine if it equals zero or not, or against

whatever the test condition is. Once the comparison is completed and the correct address is determined, the address and instruction number are sent to the PC. The counter value in the branch history table is also incremented or decremented depending on whether the branch was taken or not taken, respectively. Once the data gets to the PC it is checked to see if the BTB predicted address matches what the branch prediction unit just calculated. When there is a match the processor continues along the current stream of instructions. When a misprediction occurs, the PC is changed to the newly calculated destination and all instructions executed after the mispredicted branch are invalidated.

In the branch prediction unit during phase two of the processor clock, the newly updated counter is compared with the branch prediction criteria. In this version of the processor, if the counter is greater than or equal to "100", in binary, then the branch is predicted taken, otherwise it is predicted not taken. If the prediction is different from the previous branch prediction, the BTB is notified. So if the branch was previously predicted taken, but now is predicted not taken, then its entry in the BTB would be removed. The converse is also true.

Research has shown that for a 4096 entry 2 bit prediction buffer the prediction accuracy ranged from 82% to 99%, depending on the application (Hennessy, 1996). With the smaller 1024 entry 3 bit prediction buffer the prediction accuracy will not be as high but can safely be assumed to average right around the 80% mark, which is fine for the initial design of this processor. However, improving this prediction accuracy may be an area for future work.

2.5 Writeback

The writeback stage of this superscalar DLX processor is responsible for reordering the instructions and passing their results onto the register file. The writeback unit can accept up to six completions and send five result values back to the register files per clock cycle. The values written back can all go to either the floating point register file, the integer register file, or be split in any combination between the two. The major function of the writeback stage is to reorder the instructions and determine when instructions can be committed and still maintain a recoverable state should an exception occur.

There are a few key terms that need to be defined before further presentation of the retirement methods for instructions. Three of the terms concern the virtual to physical mapping of registers, and they are *creation*, *retiring*, and *removal*. A virtual to physical mapping is *created* when an instruction specifies a virtual register as a destination. For example, instruction I_1 specifies register R_v as its destination register, which is mapped to the physical register R^1_p. Future instructions that use register R_v as a source will be given the address of register R^1_p. This mapping will stay active until a later instruction, let's say I_2, specifies register R_v as its destination register and a mapping is created to physical register R^2_p. At this point the virtual to physical mapping of R_v to R^1_p has been *retired*. At a later time, depending on the exception model, the retired mappings will be *removed*, and the physical registers from those mapping will be free for reuse (Farkas, 1995).

The other two terms that need to be defined concern the final stages of execution for an instruction, namely the *completion* and *commitment* of the instruction. When an instruction has reached the point where it will alter the state of the machine, it is said to have *completed*. For example, an 'add' instruction has completed when it writes back to its destination register, a 'branch' instruction completes when the PC has been changed, and a 'store' instruction completes when the cache has been written to. Once the results of the instruction can be used by subsequent instructions, the instruction has *completed*. For *commitment*, all the preceding instructions must have *completed*. Once an instruction has been committed it will not be re-issued due to a mispredicted branch or exception, since all prior instructions safely completed. A completed instruction can be re-executed if a preceding uncompleted instruction causes an exception or mispredicted branch (Farkas, 1995). With the previous terms explained the discussion of exception models and the writeback unit can continue.

2.5.1 Structure

The structure of the writeback unit is split into two sections, one active on phase one of the processor clock and one active on phase two. During phase one, the results are gathered from the execution units and sent to the appropriate register files. The reorder buffer, which is an array of bits, is also marked with the completed instructions. If the invalidate signal is asserted, the retired instructions are checked against the invalidated instruction number and all instructions that follow that instruction number are voided. At the same time, the reorder buffer is checked and all entries after the invalid instruction, and

before the oldest committed instruction, are cleared. Those instructions should have never actually executed.

During phase two of the system clock the reorder buffer is checked. This checking consists of starting at the last committed instruction and going forward in the instruction number sequence until an instruction that is not completed is encountered. At this point the oldest instruction pointer is updated to point to the previous entry, and the last_committed signal is assigned the value of oldest instruction pointer. In any given cycle, up to the full array minus 1 instruction, can be committed. Upon receiving the last_committed value, the register files will release those registers for reuse, allowed by the exception model implemented.

2.5.2 Exceptions

Exceptions can occur for a number of reasons, including divide by zero, mis-aligned address, or invalid opcode. When an exception occurs the PC is changed to a specific memory address, shown in Table 2.7, where there should be a jump instruction to the start of the instructions that make up the exception handling routine. If one does not exist, the processor will start fetching and executing instructions from that point, causing unpredictable results.

Exception Type	Originating Unit	Destination Address
Divide by Zero	FPU Multiply Unit	X"00000004"
Integer Overflow	ALU1 and ALU2	X"00000008"
Floating Point Underflow	FPU Add Unit and FPU Multiply Unit	X"0000000C"
Floating Point Overflow	FPU Add Unit and FPU Multiply Unit	X"00000010"
Mis-aligned Instruction Address	Program Counter	X"00000014"
Mis-aligned Data Access	Load/Store Unit	X"00000018"
Illegal Opcode	Any of the Decoders	X"0000001C"
not implemented exceptions	No Units at this time	X"00000020"

Table 2.7 - Exception Types and Destination Address

The other side effect of exceptions is that all instructions that happen after the instruction that caused the exception must be invalidated along with any results that might have been produced. So at the time of an exception, the PC asserts the invalidate line and all the units check the instructions that they currently hold to make sure that they are still valid instructions. The instructions that are determined to be invalid are removed from the processor if they have not yet reached the execution stage, otherwise they are treated as NOPs.

There are two prominent forms of exception implementation, precise and imprecise. This superscalar DLX processor implements precise exceptions, since this method allows for easier exception recovery. In the following two sections the two exception models are described.

2.5.2.1 Imprecise Exceptions

The imprecise exception model is based on three rules. To accurately portray these rules let us consider this example, where instruction I_1 names register R_v as a destination, which has been mapped to physical register R_p. The three rules for freeing this register are as follows:

1. Instruction I_1 must have *completed*
2. Instructions that use R_p have *completed*
3. The mapping can be killed when any instruction I_x, that has register R_v as a destination, has *completed*, assuming instruction I_x follows instruction I_1 in code order. The other stipulation is that all instructions between I_x and I_1 that are branches must also have *completed* before the R_v to R_p mapping can be killed.

The major point of this model is that completion is good enough for killing virtual to physical mappings. By only requiring completion of instructions and not commitment allows the imprecise exception model to free registers sooner than the precise exception model. Recovery from exceptions is not as simple as with precise exceptions since the instructions that allow the release of registers have not committed and therefore could still cause an exception (Farkas, 1995). Further comparisons between the two exception models follow the explanation of the precise exception model.

2.5.2.2 Precise Exceptions

The precise exception model has a more stringent set of rules for the freeing of registers. The main criteria is the commitment of all preceding instructions. Once again, an example will be used to help illustrate the concepts. Instruction I_1 specifies virtual register R_v as a destination, which is mapped to the physical register R_p. The rules of

precise exceptions specify that before the virtual to physical mapping between R_v and R_p can be killed, the following three conditions must be satisfied:

1. Instruction I_1 must have *committed.*
2. Instruction I_2, which is the first instruction after I_1 that has R_v as a destination register, has *committed.*
3. All instructions, that occur after I_1 and before I_2 in program order, that use R_v must have *committed.*

The precise exception model, with its commit requirement, assures that the exact state of the processor can be recovered no matter when the exception or misprediction occurs. When an exception or misprediction occurs, all of the mappings that occurred after the faulting instruction are removed and the correct state is achieved. The imprecise model, while allowing registers to be freed sooner and being able to recover the processor state, does not accomplish the resetting of the processor as easily as that of the precise model. During a study done at Digital's Western Research laboratory, it was shown that the imprecise exception model only reduces the number of registers required for a four issue machine by 20%. The savings only increases to 37% for an eight issue architecture. The Digital researchers concluded that since the register file cycle time is more dependent on the number of access ports, rather than the number of registers, that the few extra registers required by the precise exception model were a small cost for the benefit (Farkas, 1995).

3.0 Simulation and Test

The simulation of this superscalar DLX processor was performed on an HP Model 715/80 workstation with 128 Mbytes of RAM. The VHDL source code for this model was compiled using Mentor Graphics Corporation's qhvcom compiler. The compiled architectures were simulated using Mentor's qhsim simulator. The qhsim simulator has options to trace signals in graphical waveforms and text based listings, both of which were very useful during the debugging of the processor. The programs that were run on this superscalar DLX processor model were assembled using the DLXasm program. The DLXasm program is the work of Peter Ashenden, who extracted the assembler from the DLXsim program that was created at the University of California at Berkeley. For the larger programs, a version of the Gnu gcc compiler was used to convert the C code to DLX assembly code where is was translated to machine language via DLXasm.

The initial testing plan was to test the individual components such as the memory, decoder, etc. separately to verify functionality. After a certain point, the number of signals that needed to be driven by the test bench became unmanageable and the testing methodology switched to entire chip testing. Even though the test philosophy changed to include the entire processor, the area under test was limited to smaller sections, like the integer instruction path, loads and stores, etc. The other advantage, apart from having almost all of the signals internally driven, is the ability to see immediately if the corrections just made caused any adverse side effects in the other sections of the processor.

3.1 Integer Unit

The testing of the integer unit was split into two stages: functionality and edge cases. The initial testing phase of the ALU and integer pipeline was focused on functionality for the normal execution state. This means that overflows and other exception causing events were avoided. Testing began with the execution of two instructions at once. Next, forwarding of data to the next instructions was tested. Then, it was verified that instructions were committing correctly. Finally, instructions causing exceptions were tested. At this point the integer unit was considered functioning and the testing proceeded to the load and store unit.

3.2 Load/Store Unit

Once the ALUs and standard pipeline structures were proven to work, testing moved on to the load/store unit so that data could be moved on to and off of the processor. The testing started with pure functional testing and then moved onto some of the edge cases. The general tests were unsigned and signed byte loads, half-word load, and word loads along with the similar stores. The floating point loads and stores were also tested during this phase, including the two cycle double precision loads and stores. Upon satisfying general functionality, the test cases involving the exception causing events, such as mis-aligned memory accesses, were used. Once most of the edge conditions were proven functional, the testing shifted focus to the floating point unit.

3.3 Floating Point Unit

The simulation and testing of the floating point unit was split into four areas. These were the functionality and edge cases for both the adder and the multiplier. First to be

tested were the floating point add and convert instructions, both in single and double precision. As the data paths and functionality of these functions were verified, the multiply and divide functions were tested. The first goal of testing was functionality in the standard case. As the standard functionality was proven, the edge cases were tested, such as divide by zero, overflow, and underflow. At this point, the core functional sections of the superscalar DLX processor were functionally tested. The testing then began to focus on one of the more complex parts of the processor, the branch unit.

3.4 Branch Unit

Testing of the branch unit also included the other PC modifying instructions such as J, JR, JAL, JALR, RFE, and TRAP. The testing of these instructions is the most difficult since they change the instruction flow in the processor. When a branch instruction occurs, every block is altered by the invalidate signal, which makes it very difficult to determine exactly where an error is originating. As with the other functional units, the first tests were to determine that this unit and these instructions worked properly in non-edge cases. After correct normal execution was verified, the edge case conditions were explored, such as jumps or branches to illegal addresses. As the final bugs were being resolved with the branch unit, the next and final testing phase was starting, which was to test the entire processor with larger programs that included a variety of instructions.

3.5 Overall

The testing of the complete processor was accomplished through the use of programs that have larger loops which created thousands of instructions to execute at run

time. A good mix of instructions were also provided so that all of the execution units would be kept busy rather than using a program that only exercises one execution unit. During this overall testing, the test bench was modified to record the number of instructions that were dispatched from the dispatch queues, and also the number of instructions that were received by the write back unit. These numbers were used to determine the CPI rating for this processor over a given set of programs. The results of these tests are discussed in the next section.

4.0 Performance and Discussion

In addition to providing an excellent learning vehicle, the goal for designing this superscalar DLX processor was to create a model of a high performance processor core. There were two performance goals set for this processor. The first goal was to exceed the maximum CPI possible in a scalar DLX pipeline (1.00). The second goal was to achieve a CPI below 0.50 when presented with an instruction mix that had enough instruction level parallelism (ILP) to support it.

In theory, when ignoring the latencies of memory and stalls, a four issue processor is capable of attaining a CPI value of 0.25, as shown in Figure 4.1. The optimum CPI value of 1.00 for scalar pipelined processors is derived in the same manner and is shown in Figure 4.2. The superscalar DLX processor designed for this thesis has the capability to issue six instructions per cycle, but can only fetch four instructions per cycle. This limitation on the number of instructions that can be fetched in a clock cycle limits this processor to an optimum CPI value of 0.25.

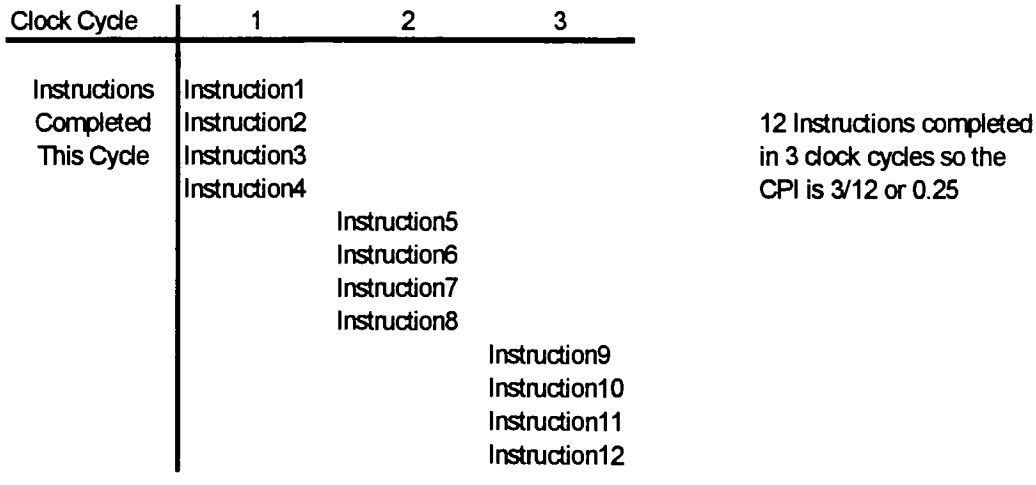

Figure 4.1 - The CPI Calculation for a 4-Issue Processor

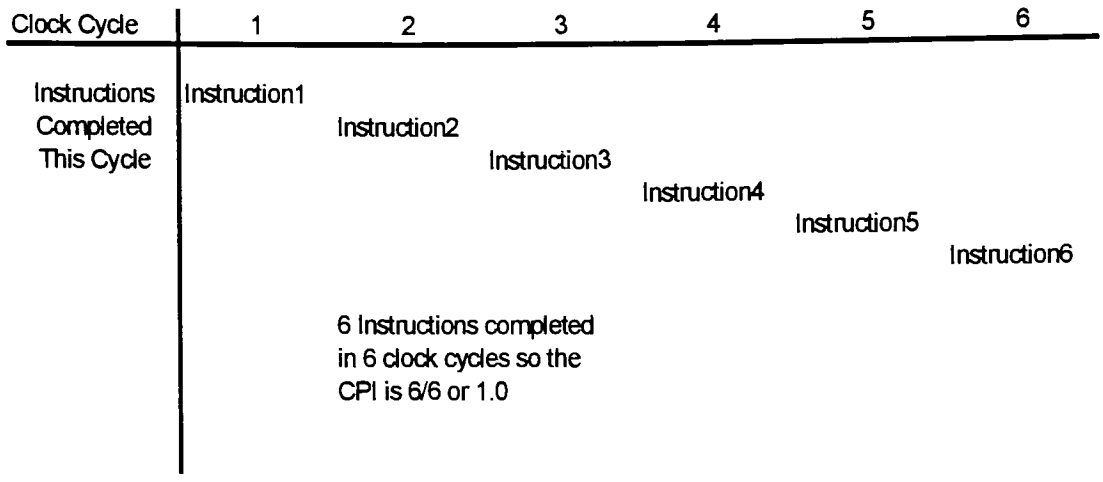

Figure 4.2 - The Optimum CPI Calculation for a Scalar Pipelined Processor

All of the tests run to determine the functionality of the processor were also used to gather examples of CPI values. During testing, the CPI values that were observed ranged from 1.00 down to 0.35. For comparisons between the superscalar DLX processor and a scalar DLX processor, some assumptions were applied to both. Specifically, memory was assumed to have no delay and floating point multiplies and divides were completed in a single clock cycle. Based on those assumptions, the results obtained during the testing of the superscalar DLX processor are compared with those of a scalar DLX processor, as shown in Table 4.1. The number of cycles specified for the scalar processor are based on its optimum CPI of 1.00 and the assumption that there are no stalls. While this assumption is not obtainable in practice, it allows the superscalar DLX processor to be compared against the ideal single pipeline processor. The initial penalty of four clocks to fill the pipeline is included in the scalar numbers.

Number of Instructions	Scalar Processor Clocks	Scalar Processor CPI	Superscalar Processor Clocks	Superscalar Processor CPI	Superscalar Processor Improvement
15	19	1.267	14	0.933	26%
33	37	1.121	24	0.606	46%
59	63	1.085	34	0.576	47%
61	65	1.066	35	0.574	46%
63	67	1.063	42	0.667	37%
119	123	1.034	70	0.588	43%
137	141	1.029	61	0.445	57%
221	225	1.018	91	0.412	59%
683	687	1.006	256	0.375	63%
763	767	1.005	266	0.349	65%

Table 4.1 - Results Comparison, Superscalar DLX vs. the Optimum Scalar DLX

The data displayed in Table 4.1 represents a very limited sample of the ongoing testing process. While the last CPI value for the superscalar DLX is below 0.50, it is still over 40% higher than the theoretical minimum. There are a number of contributors to this result. The first, and major reason why the performance is not optimal is due to the size of the test program. As the programs get larger, more and more of the initial pipeline fills and other latencies are hidden and averaged away. An example of this trend can be seen in the data for the scalar processor. The only penalty attached to the CPI of 1.00 for the scalar processor is the initial pipeline load penalty of four clock cycles for a five stage pipeline. As the number of instructions increased from 15 to 763 the actual CPI went from 1.267 to 1.005, improving by almost 21%.

Another reason the superscalar CPI value is not optimal is the invalidation of instructions on jumps, traps, and mispredicted branches. For each trap, jump, and RFE instruction, there are at least 3 fetched instructions that are invalidated and wasted. When a mispredicted branch occurs, there are a minimum of twelve instructions that must be invalidated, wasting three clock cycles of fetches. These penalties are the major reasons that a premium is placed on the accuracy of the branch prediction algorithm.

The final major component contributing to a less than optimal CPI value is the instruction mix. If a set of instructions does not inherently have any ILP, then the superscalar processor is only going to show the smallest, if any, amount of improvement over the scalar processor. An example of a set of instructions that is very difficult, if not impossible to execute in parallel, is shown below. The clock cycle of execution is based on the execution pattern of the superscalar DLX processor model.

Instruction		**Clock Cycle of Execution**
lw	*r1,#0x500(r0)*	1
movi2fp	*f1,r1*	*2*
addf	*f3,f2,f1*	*3*
movfp2i	*r4,f3*	*4*
addi	*r5,r4,#0x10*	*5*
sw	*#0x504(r0),r5*	6

The problem with this code is that every instruction has an operand that depends on the previous instruction. With data dependencies that are this tight there will be very little, if any, performance gain by any superscalar processor. The only way that this inherent serialness can be overcome is to make the pool of ready instructions larger, which can be done by making the dispatch queues larger and the fetch rate higher.

The next segment of code has a very large amount of ILP and would perform very well when executed in a superscalar processor. The clock cycle of execution shown is based on the superscalar DLX processor model. The reason that the 'sd' instruction does not execute during clock cycle 3, when it is technically able to, is that in the precise exception model, 'stores' are not allowed to write to memory until all previous instructions have committed.

Instruction		Clock Cycle of Execution				
lw	*r1,#0x500(r0)*	1				
ld	*f2,#0x504(r0)*		2			
addi	*r3,r1,#0x24*		2			
subi	*r4,r2,#0x346*			3		
addd	*f4,f2,f2*		2			
cvti2d	*f6,r1*			3		
add	*r6,r3,r4*			3		
multd	*f8,f2,f2*		2			
sw	*#0x510(r0),r6*				4	
sd	*#0x512(r0),f8*					5

The performance demonstrated by the superscalar DLX processor is a good sign that the design effectively takes advantage of the ILP that is found. While the optimum value for the CPI of 0.25 has not been achieved yet, the ability of the processor to reach the CPI value of 0.35 is a promising sign. Further testing and refinement of the architecture should bring the CPI closer to the 0.25 CPI goal.

5.0 Future work

There are a number of areas that are open to future improvement or exploration. Some specific areas include the branch prediction strategy, memory model, and design of the floating point multiplier. Along with altering existing structures in this processor model, some future work could be spent adding new functionality to the design such as built in self test hardware (BIST). One of the major goals of this thesis was to provide a base architecture on which new techniques and designs could be explored. As the processor designs being produced in industry change and research papers published, these new concepts and designs can be tested. By incorporating new designs into the appropriate parts of this architecture, the impact on the performance of the entire superscalar architecture can be observed, as well as the effect one design improvement might have on other parts of the processor.

5.1 This Design

There are a number of areas in this superscalar DLX processor design that can be worked on in order to make the model more similar to an actual production processor. One such area is the floating point multiplier. Instead of using the VHDL '*' command that is available in RIT's computing environment, the multiplier should be expanded to more accurately represent the shift and add structure of a simple multiply circuit. Another advantage of expanding the multiplier to a more structural representation is the ability to make multiplies and divides take multiple clock cycles. This change would allow the user

to explore the hazards and difficulties of handling the multi-cycle operations along side the single cycle operations.

The assumption that the memory has no delay or cache miss penalty is fine for the first version of the processor model, however real memory does not work that way. A major focus for future work would be the implementation of delays and actual timing parameters for the memory structures.

Another option for further work is the synchronization or pipelining of the memory structures and accesses. With the inclusion of the more accurate memory model, more complicated memory access methods could be explored. One item to be wary of is expanding this model to the point that it is no longer simple to understand. Hiding the major principles of superscalar design under exotic logic would be counter-productive to the educational purpose guiding this design.

In addition to extending the existing architectural elements, new components could be added in future research. One area that would be very beneficial to explore is that of built-in test structures. The ability to test a superscalar processor while it is on a mother board without the need of a multi-million dollar tester is a tremendous advantage during the debug process. The time spent to add scan registers and other BIST structures into this design would be well spent. By including the BIST structures into the design model, design approaches that are difficult to test can be altered to allow for a simplified testing cycle. As processors get more and more complex, the amount of time it takes to test them

will also grow. It is for this reason that including the BIST into the design model would be a great benefit.

These are just a few areas in which time spent on future research is likely to produce very beneficial results. Another area that could provide some worthwhile results would be to convert the behavioral VHDL code to synthesizable VHDL code, or work on the synthesis process for behavioral hardware descriptions. The areas for improvement are not limited to only those mentioned. Virtually every logical block in a superscalar design has multiple implementation possibilities that can be explored.

5.2 Other Architectures

Further research done on this processor does not need to be limited to just peripheral sections of the chip. It can also include the pipeline. There are many areas that can be explored, the most obvious being the depth and width of the pipeline. One way to use this architecture for future explorations would be to examine the future road map for Digital's Alpha processor and explore the proposed designs. Currently the Alpha 21164 is a four issue machine running at 500 MHz. Digital has plans over the next seven years that would increase the issue rate to 8, 16 and finally 32 instructions per clock cycle, which would be around 1GHz (Computergram, 1996). The challenges associated with the handling of 32 instructions per clock cycle per stage are numerous and worthy of investigation, especially if the VHDL description was synthesizable. A study of the area requirements for a 32 issue machine would be a very interesting topic to explore as well.

These few examples show that there are many interesting topics that can be further explored as an outgrowth of this investigation.

6.0 Conclusion

The complexity of today's microprocessors demands that designers have an excellent knowledge of superscalar design techniques; this knowledge is difficult to acquire outside of a professional design team. With the limited number of non-proprietary resources available, it is difficult for a student to attain the hands-on knowledge that would be important in the professional design field. The limited number of options available points to the need for more models and simulators, allowing students the opportunity to learn more about superscalar designs prior to entering the work force.

The goal of this thesis was to create a VHDL model of a superscalar processor that implemented the DLX ISA. The processor should be robust enough so that university students looking to study in the area of advanced processor design would find the model useful in their studies. High performance was also a goal of this processor. Exceeding the scalar processor's theoretical limit of one clock per instruction was the minimum performance that would be accepted.

The first requirement of this thesis was to implement a superscalar processor that accepted all 92 of the DLX ISA's specified instructions. This VHDL processor model will accept all of the instructions specified in The DLX Instruction Set Architecture Handbook, (Sailer, 1996). Implementation of the complete instruction set including the floating point instructions allows the user to freely experiment with few constraints. The ability to take a C program and compile it to DLX machine language, and then run it, makes this model very useful in the process of learning about superscalar processors. The

subdivision of the processor into five stages and eighteen separate functional units allows the student to isolate areas of study, rather than have to observe the entire processor at one time.

The requirement of high performance was satisfied in several ways. First, out-of-order execution is supported. This allows a number of stalls to be avoided which would have decreased performance. In order to support out-of-order execution, register renaming, dispatch queues, and a reorder buffer were used. The processor is capable of fetching four instructions on every clock cycle. This helps to keep the dispatch queues full of instructions that are ready to issue.

Once the instructions are in the processor, they need to be executed in such a way as to enhance the performance of the processor. The choice to issue up to six instructions per clock cycle was determined by the natural grouping of instructions into five groups: integer ALU, loads and stores, floating point addition, floating point multiplication, and branches. The reason that six instructions are issued rather than five is the general dominance of integer instructions in non-scientific programs. This prompted the inclusion of a second integer ALU.

After the instructions have been executed they need to be efficiently written back to the register files. The writeback or retirement of the instructions is handled by the writeback unit which is capable of retiring five results to the register files during every clock. Since the branch instructions do not create a result, there are only five possible results to writeback from the six executed instructions.

The previously described implementation allows the superscalar DLX processor to have a theoretical minimum CPI of 0.25, or 4 instructions per clock. The actual minimum CPI value that was produced during testing was 0.375 or a little under three instructions per clock. The reasons for this less than optimum performance are explained in section 4.0. As the work on this processor model continues, lower CPI values are expected when the test program contains the right level of ILP

The end result of this thesis work is a high performance VHDL model implementing the complete DLX ISA. This model can be used to explore new architectural choices or to just learn about superscalar architectures. The completion of this project adds another resource to those available on the workings and principles of superscalar architectures, and provides a starting platform for future research work.

APPENDIX A

The DLX Instruction Set

R_d: General Purpose Destination Register
R_{s1}: General Purpose Source Register
R_{s2}: General Purpose Source Register
Immediate: 16 Bit Immediate Operand Value
Offset: 16 or 26 Bit Offset Added to the PC
F_d: Floating Point Destination Register
F_{s1}: Floating Point Source Register
F_{s2}: Floating Point Source Register

Instruction Name	Assembly Language Symbol	Instruction Format	
Data Transfer	**Instruction Type: Immediate (I)**		
Load Byte Signed	LB	LB	R_d, offset(R_{s1})
Load Byte Unsigned	LBU	LBU	R_d, offset(R_{s1})
Load Halfword Signed	LH	LH	R_d, offset(R_{s1})
Load Halfword Unsigned	LHU	LHU	R_d, offset(R_{s1})
Load Word	LW	LW	R_d, offset(R_{s1})
Load SP Float	LS	LS	F_d, offset(R_{s1})
Load DP Float	LD	LD	F_d, offset(R_{s1})
Store Byte	SB	SB	offset(R_{s1}), R_d
Store Halfword	SH	SH	offset(R_{s1}), R_d
Store Word	SW	SW	offset(R_{s1}), R_d
Store SP Float	SF	SF	offset(R_{s1}), F_d
Store DP Float	SD	SD	offset(R_{s1}), F_d
	Instruction Type: Register (R)		
Read Special Register	MOVS2I	MOVS2I	R_d, R_{s1}
Write Special Register	MOVI2S	MOVI2S	R_d, R_{s1}
Move Integer to FP register	MOVI2FP	MOVI2FP	F_d, R_{s1}
Move FP register to Integer	MOVFP2I	MOVFP2I	R_d, F_{s1}
Arithmetic, Logical	**Instruction Type: Immediate (I)**		
Add Unsigned Immediate	ADDUI	ADDUI	R_d, R_{s1}, Immediate
Add Signed Immediate	ADDI	ADDI	R_d, R_{s1}, Immediate
Subtract Unsigned Immediate.	SUBUI	SUBUI	R_d, R_{s1}, Immediate
Subtract Signed Immediate.	SUBI	SUBI	R_d, R_{s1}, Immediate
And Immediate	ANDI	ANDI	R_d, R_{s1}, Immediate
Or Immediate	ORI	ORI	R_d, R_{s1}, Immediate

Xor Immediate	XORI	XORI	R_d, R_{s1}, Immediate
Load High Immediate	LHI	LHI	R_d, Immediate
Shift Left Logical Immediate.	SLLI	SLLI	R_d, R_{s1}, Immediate
Shift Right Logical Immediate.	SRLI	SRLI	R_d, R_{s1}, Immediate
Shift Right Arithmetic Immediate.	SRAI	SRAI	R_d, R_{s1}, Immediate
Set Less Than Immediate.	SLTI	SLTI	R_d, R_{s1}, Immediate
Set Less Than or Equal To Immediate	SLEI	SLEI	R_d, R_{s1}, Immediate
Set Greater Than Immediate.	SGTI	SGTI	R_d, R_{s1}, Immediate
Set Greater Than or Equal To Immediate	SGEI	SGEI	R_d, R_{s1}, Immediate
Set Equal To Immediate	SEQI	SEQI	R_d, R_{s1}, Immediate
Set Not Equal To Immediate	SNEI	SNEI	R_d, R_{s1}, Immediate
Arithmetic, Logical	**Instruction Type: Register (R)**		
Add Unsigned	ADDU	ADDU	R_d, R_{s1}, R_{s2}
Add Signed	ADD	ADD	R_d, R_{s1}, R_{s2}
Subtract Unsigned	SUBU	SUBU	R_d, R_{s1}, R_{s2}
Subtract Signed	SUB	SUB	R_d, R_{s1}, R_{s2}
Multiply Unsigned	MULTU	MULTU	F_d, F_{s1}, F_{s2}
Multiply Signed	MULT	MULT	F_d, F_{s1}, F_{s2}
Divide Unsigned	DIVU	DIVU	F_d, F_{s1}, F_{s2}
Divide Signed	DIV	DIV	F_d, F_{s1}, F_{s2}
And	AND	AND	R_d, R_{s1}, R_{s2}
Or	OR	OR	R_d, R_{s1}, R_{s2}
Xor	XOR	XOR	R_d, R_{s1}, R_{s2}
Shift Left Logical	SLL	SLL	R_d, R_{s1}, R_{s2}
Shift Right Logical	SRL	SRL	R_d, R_{s1}, R_{s2}
Shift Right Arithmetic	SRA	SRA	R_d, R_{s1}, R_{s2}
Set Less Than	SLT	SLT	R_d, R_{s1}, R_{s2}
Set Less Than or Equal To	SLE	SLE	R_d, R_{s1}, R_{s2}
Set Greater Than	SGT	SGT	R_d, R_{s1}, R_{s2}
Set Greater Than or Equal To	SGE	SGE	R_d, R_{s1}, R_{s2}
Set Equal To	SEQ	SEQ	R_d, R_{s1}, R_{s2}
Set Not Equal To	SNE	SNE	R_d, R_{s1}, R_{s2}
Control	**Instruction Type: Jump (J)**		
Branch Equal To Zero	BEQZ	BEQZ	R_{s1}, Offset
Branch Not Equal To Zero	BNEZ	BNEZ	R_{s1}, Offset
Branch FP Status Register True	BFPT	BFPT	Offset
Branch FP Status Register False	BFPF	BFPF	Offset
Jump	J	J	Offset
Jump Register	JR	JR	R_{s1}
Jump And Link	JAL	JAL	Offset

Jump And Link Register	JALR	JALR	R_{s1}
Trap	TRAP	TRAP	Offset
Return From Exception	RFE	RFE	

Floating Point	**Instruction Type: Register (R)**		
Add Single Precision	ADDF	ADDF	F_d, F_{s1}, F_{s2}
Add Double Precision	ADDD	ADDD	F_d, F_{s1}, F_{s2}
Subtract Single Precision	SUBF	SUBF	F_d, F_{s1}, F_{s2}
Subtract Double Precision	SUBD	SUBD	F_d, F_{s1}, F_{s2}
Multiply Single Precision	MULTF	MULTF	F_d, F_{s1}, F_{s2}
Multiply Double Precision	MULTD	MULTD	F_d, F_{s1}, F_{s2}
Divide Single Precision	DIVF	DIVF	F_d, F_{s1}, F_{s2}
Divide Double Precision	DIVD	DIVD	F_d, F_{s1}, F_{s2}
Less Than Single Precision	LTF	LTF	F_{s1}, F_{s2}
Less Than Double Precision	LTD	LTD	F_{s1}, F_{s2}
Less Than or Equal To Single Precision	LEF	LEF	F_{s1}, F_{s2}
Less Than or Equal To Double Precision	LED	LED	F_{s1}, F_{s2}
Greater Than Single Precision	GTF	GTF	F_{s1}, F_{s2}
Greater Than Double Precision	GTD	GTD	F_{s1}, F_{s2}
Greater Than or Equal To SP	GEF	GEF	F_{s1}, F_{s2}
Greater Than or Equal To DP	GED	GED	F_{s1}, F_{s2}
Equal To Single Precision	EQF	EQF	F_{s1}, F_{s2}
Equal To Double Precision	EQD	EQD	F_{s1}, F_{s2}
Not Equal To Single Precision	NEF	NEF	F_{s1}, F_{s2}
Not Equal To Double Precision	NED	NED	F_{s1}, F_{s2}
Move Single Precision	MOVF	MOVF	F_d, F_{s1}
Move Double Precision	MOVD	MOVD	F_d, F_{s1}
Convert Single To Double Precision	CVTF2D	CVTF2D	F_d, F_{s1}
Convert Single To Integer	CVTF2I	CVTF2I	F_d, F_{s1}
Convert Double To Single Precision	CVTD2F	CVTD2F	F_d, F_{s1}
Convert Double To Integer	CVTD2I	CVTD2I	F_d, F_{s1}
Convert Integer To Single Precision	CVTI2F	CVTI2F	F_d, F_{s1}
Convert Integer To Double Precision	CVTI2D	CVTI2D	F_d, F_{s1}

Bibliography

Ahi, Ali, et al. "R10000 Superscalar Microprocessor." 1995. Online. Internet. May 1996. Available http://sgigate.sgi.com/pub/doc/R10000/hotchips/hotchips.ps

Ashenden, Peter J. The Designer's Guide to VHDL, San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Computergram International. "DEC Outlines Road Map for ALPHA from Here to 2003." (May 15, 1996) n.pag. Online. Internet. July 27, 1996. Available http://infopad.eecs.berkeley.edu/CIC/announce/alpha_roadmap

"DLX Interactive Simulation Composite." Purdue University, 1995. Online. Internet. August 8, 1996. Available http://yara.ecn.purdue.edu/~teamaaa/disc

Farkas, Keith, et al. "Register File Design Considerations in Dynamically Scheduled Processors." Western Research Laboratory, November 1995. Online. Internet. January 1996. Available http://www.research.digital.com/wrl/home.html

Gumm, Martin. "The DLXS Processor." University of Stuttgart, 1995. Online. Internet. Available http://rassp.scra.org/public/vhdl/models/processors/NEW_DLX.tar.gz

Hennessy, John L and Patterson, David A. Computer Architecture: A Quantitative Approach, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Hostetler, Larry B., and Brian Mirtich. "DLXsim - A Simulator for DLX." (January 19, 1996. 19 pages. Online. Internet. May 20, 1996. Available ftp://max.stanford.edu/pub/hennessy-patterson.software

"Industry's Most Powerful 64-Bit Microprocessor Attains Performance Level That Outperforms Competitors By Up To 260 Percent." (April 4, 1996) n.pag. Online. Internet. July 27, 1996. Available http://www.hp.com/csopress/96apr4.thml

Kaeli, David R. and Sailer, Philip M. The DLX Instruction Set Architecture Handbook, San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Moura, Cecile. "SuperDLX: A Generic Superscalar Simulator." Thesis. McGill University, 1993. Online. Internet. 8 August 1996. Available http://www-acaps.cs.mcgill.ca/superdlx

Naffziger, Samuel. "A Sub-Nanosecond 0.5mm 64b Adder Design." (June 19, 1996) n.pag. Online. Internet. July 27, 1996. Available http://wwwhp.com/wsg/strategies/subnano/subnano.html

Stallings, William. Computer Organization and Architecture: Principles of Structure and Function, New York, NY: Macmillian Publishing Company, 1990.