ARCHITECTURAL VERIFICATION OF FOUR-INSTRUCTION SUPERSCALAR PROCESSOR FOR MIPS I INSTRUCTION SET

By

ANSHUMAN ANAND

Bachelor of Engineering

Nagpur University

INDIA

1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2003

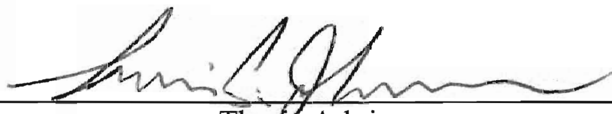
ARCHITECTURAL VERIFICATION OF

FOUR-INSTRUCTION SUPERSCALAR

PROCESSOR FOR MIPS I

INSTRUCTION SET

Thesis Approved:



Thesis Advisor

Dean of the Graduate College

ACKNOWLEDGEMENTS

Although it has just been just two and a half years studying in Oklahoma State University, it is an experience that will stay with me forever. My stay in Stillwater has given me a lot to cherish, good friendships, an excellent studying environment to name a few.

First of all I would like to thank my parents, Sushma Anand and Arun Kumar Anand. They have inspired me throughout my life and have taught me never to give up. I would also like to thank my aunt and uncle, Malika Chaudhary and Ravi Chaudhary, for helping me realize my long-standing dream of studying in an esteemed university. I would also like to thank my brother, Ayush Anand for his help and support.

I consider myself fortunate enough to have made a lot of good friends. I would like to thank all of then for the help and support they extended towards me. I would especially like to thank Ritesh, Srinivasan and Nitin for making my stay in Stillwater an experience of home away from home.

I was fortunate enough to work under Fritz Writ at the Daily O'Collegian. I would like to thank him for believing in me and providing me with financial support.

I wish to extend my sincere appreciation to my adviser Dr. Louis G. Johnson for his able guidance, intelligent supervision and inspiration that helped me finish this thesis. My sincere appreciation also extends to my committee members Dr. Yumin Zhang and Dr. Weili Zhang whose assistance and encouragement are also invaluable.

Finally, I would like to thank the Department of Electrical and Computer Engineering for supporting me during these two and a half years of study.

TABLE OF CONTENTS

Chapter Page

1. Introduction and Thesis organization 1

1.0 Thesis Introduction 1
1.1 Background Study 4
1.1.0 Preliminary Studies 4
1.1.1 Understanding branches & branch prediction alternatives 5
1.1.2 Branch target buffer indexing 8
1.1.3 Register file issues 9
1.2 Thesis Organization 11

2. Architectural Overview 13

2.0 Introduction 13
2.1 Overall design & dataflow 13
2.1.0 Architectural Features 13
2.1.1 Handling Branch and Jump Instructions 18
2.1.2 Handling Speculated Data 19
2.1.3 Handling out of order execution 20
2.1.4 Handling dynamic scheduling of instructions 22
2.1.5 Handling Multiply and Divide Instruction 23
2.1.6 Handling Memory Stores 23
2.1.7 Handling Data Consistency 24
2.2 Pipeline Stages 25
2.2.0 Introduction 25
2.2.1 Instruction Fetch Stage 26
2.2.2 Instruction Decode/Issue Stage 27
2.2.3 Instruction Dispatch 28
2.2.4 Instruction Execution 29
2.2.5 Instruction Write Back 30
2.2.6 Instruction Commit I 31
2.2.7 Instruction Commit II 32

3. Design of Branch Prediction Mechanism 33

3.0 Introduction 33
3.1 Branch Target Buffer 34
3.2 Branch Prediction Buffer 37

Chapter Page

4. Instruction Set Architecture Decoder & Prioritizer 40

4.0 Introduction 40
4.1 Instruction set architecture decoder design 43
4.2 Prioritizer design 46

5. Register file design 50

5.0 Introduction 50
5.1 Issue pointer buffer/Commit pointer buffer design 52
5.2 Value Buffer Design 56

6. Maintaining Data Consistency & Instruction Execution 59

6.0 Introduction 59
6.1 Source Overwrite Logic 60
6.2 Destination Overwrite Logic 63
6.3 Instruction Execution 65

7. Design Verification Methodology & Simulation 71

7.0 Introduction 71
7.1 Memory Model 72
7.3 Simulation 76
7.4 Results 77
7.4.0 Instruction per cycle (IPC) 77
7.4.1 Misprediction Rate 78
7.4.2 Dispatch Stalls 78
7.4.3 Percentage utilization of BPB and BTB. 78

8. Conclusion & Future Work 81

8.0 Conclusion 81
8.1 Future Work 83

References 85

LIST OF TABLES

Table Page

4.1: Functional Unit Codes for the Processor 44
7.1: Simulation Results 79

LIST OF FIGURES

Figure Page

2.1: Overall Architecture. 17
2.2: Sample code. 22
3.1: States in a two-bit prediction scheme. 33
3.2: Memory Address Division to access Branch Target Buffer. 35
3.3 Branch target buffer architecture 36
3.4: Branch prediction buffer architecture 38
4.1: I-Type instruction format. 41
4.2: J-Type instruction format 42
4.3: R-Type instruction format. 42
4.4: Instruction Set Architecture Decoder. 43
4.5: Decoder output format for instructions. 43
4.6: Prioritizer. 47
4.7: First zero logic 48
5.1: Block Diagram of the Register-file. 51
5.2: Issue Pointer Buffer/Commit Pointer Buffer Implementation. 53
5.3: Issue Pointer Buffer/Commit Pointer Buffer Timing 55
5.4: Value Buffer Implementation 56
5.5: Value Buffer timing 58
6.1: Sample Source code and Issue Pointer Buffer timing. 60
6.2: Source overwrite logic Implementation. 62
6.3: Sample Source code and Issue Pointer Buffer timing. 63
6.4: Destination overwrite logic Implementation. 64
6.5: Functional units block diagram. 66
6.6: Block diagram of Arithmetic and logic unit. 67
6.7: ALU result encoding 67
6.8: Block diagram of Branch and jump unit. 68
6.9: Branch bus encoding 68
7.1: Address-Data memory array pair 72
7.2: Flowchart of design verification methodology 73
7.3: Stack Memory dump before simulation. 76
7.4: Stack Memory dump after simulation 76

Chapter 1

Introduction and Thesis organization

1.0 Thesis Introduction

Computer applications continue to require higher and higher processing speed. This has led researchers and designers to come up with new ideas to enhance the speed of the processors. A common approach is to exploit Instruction Level Parallelism inherent in the codes. Instruction level parallelism can be defined as the potential overlap of the instructions, so that they can be executed in parallel. Identifying instruction level parallelism contributes to increasing the number of execution units in the processor that operate concurrently to enhance the performance of the processor. Multi-linear processors have more than one execution or functional unit to evaluate parallel instructions. Two common RISC (Reduced instruction set computer)-type multi-linear processor alternatives are the Super-scalar architecture and Very Large Instruction Word (VLIW) architecture. The difference between these two architectures stems from the way they utilize instruction level parallelism. VLIW processors require static scheduling of parallel instructions that can be fetched as one instruction word. Alternatively, super-scalar processors do dynamic scheduling of instruction as they are fetched from a linear instruction sequence. Static scheduling of the instructions is done at compile time. On the other hand, dynamic scheduling is done at run time when a particular sequence of instructions is being executed. The compiler plays a critical roll in optimizing the

performance of a multi-linear processor since the compiler has to identify instructions that could be evaluated in parallel.

A typical multi-linear processor fetches and decodes several instructions at a time. To ensure a steady stream of instructions, control instructions are predicted at fetch time. Decoded instructions are evaluated for data dependencies, dynamically scheduled and issued to the execution units for evaluation. Each execution unit can have a different latency causing the instructions to finish execution out of program order. Latency can be defined as the number of clock cycles between issuing an instruction to a particular pipeline and completion of execution of that instruction. At the end of execution, the results are reordered to update the machine state in program order.

Since on an average there is a branch or a jump instruction every five instructions [1], accurate branch prediction plays a critical roll in improving the efficiency of the processor, since the control flow of the program after a branch instruction is predicted, the instruction fetched after a predicted branch is referred to as a *speculated instruction* until the branch is evaluated in the pipeline. These instructions should not be allowed to alter the machine state until the branch conditions are evaluated. Hence for this intermediate period, results produced by the speculated instruction must be stored in registers other than the architectural registers to avoid alteration of the machine state. Once the branch is evaluated and the branch action matches the predicted action i.e. *taken* or *not taken*, these results should be allowed to write into the architectural registers to alter the machine state in accordance with the program. It is seen from the above

discussion that for a speculated instruction, two 32-bit data transfers are required to complete it. This is highly inefficient and takes up a lot of silicon area.

The study undertaken in this thesis tries to tackle this inefficiency by having extra register locations other than the architectural registers called pseudo-registers, and a pointer scheme is followed to reference both architectural and pseudo registers. This scheme renames each logical destination register of an incoming instruction, to a pseudo register referenced by pointers called pseudo-pointers. Two separate lists of these pointers are maintained, one for all types of instructions and the other for only un-speculated instructions. When a branch instruction preceding the speculated instruction is evaluated and it is established that the prediction was correct, the machine state is altered by updating the pointer lists instead of moving the data. As the pointes are only 6-bits, the inefficiency is considerably reduced.

This processor scheme is implemented using the Verilog hardware description language (HDL). The following study provides architectural details of each component used in the processor, stressing issues involved in the implementation and methods used to overcome these issues. This study also discusses verification methodology, documenting steps involved in compiling a 'C' program and loading it onto the simulated instructions cache and data cache for simulation. Finally, simulation results are presented for a sample 'C' program verifying the design.

1.1 Background Study

1.1.0 Preliminary Studies

Two important implementation alternatives for multi-linear processors are the VLIW approach and the Superscalar approach. A discussion on the performance comparison of both approaches is presented at length in [9]. It is shown in the paper that both the approaches have comparable performance for selected benchmarks. It is further shown that a relatively small instruction window is required for a superscalar processor as compared to a VLIW processor to exploit the same amount of parallelism. The overall micro-architecture of a superscalar processor is well elaborated in [10]. This paper discusses the concept of instruction level parallelism. It also discusses the technique involved in the implementation of the basic phases of a superscalar processor including instruction fetch and conditional branch processing, identifying data dependencies, issuing instructions for execution, memory interaction and committing the processor state in the correct order. A prototype implementation of a superscalar processor with a fetch size of two following the million instructions per second (MIPS) instruction set is presented in [11]. This paper brings out different design issues involved in a superscalar processor including a detailed study of the data-path, multi-ported register file, data and control hazards. These papers give an overall understanding of the micro-architecture and the issues involved in the design of a superscalar architecture without going into details.

1.1.1 Understanding branches & branch prediction alternatives

An efficient branch prediction mechanism is gaining more and more importance because processors are getting wider and wider and the functional units are getting deeper and deeper [2]. Reference [2] talks about different types of branches and their behavior patterns, which is important to understand. According to [2], branches can be conditional or unconditional. Conditional branches are further divided into immediate branch, indirect branch and return [2]. The branch target address is encoded in the instruction for an immediate branch, whereas indirect branch reads the branch target address from a register and return gets its target address from the stack memory. It further points out that for the SPECint95 benchmark, 72% of branches are conditional, 17% are unconditional, 10% are return and 1% are indirect. Reference [2] further shows the distribution of execution frequencies of static conditional branches. On an average 53% of all the branches were executed 99 times or fewer and 11% of all the branches were executed 10,000 times or more. It also points out that 53% of the branches that were executed 99 times or fewer make up 0.2% of the branches in the dynamic instruction stream while 11% of the branches that were executed 10,000 times or more make up 87% of the branches in the dynamic instruction stream. This proves that 10% of the code is responsible for 90% of the execution [2]. This proves that a good branch prediction scheme is an imperative for a highly efficient processor. Different schemes employed for predicting branches are as follows:

1. *Two-bit* branch predictor [7].
2. *GAg* branch predictor [1].
3. *gshare* branch predictor [1].

4. *LGshare* branch predictor [1].
5. *Block-level* prediction scheme [8] [3].
6. *Hybrid* prediction scheme.

The two-bit branch predictor is the most simple prediction scheme. This scheme employs a two-bit saturating counter for each predicted branch and the most significant bit is the prediction of branches. The counter is updated at the time of commit. If the prediction was correct, the counter is incremented or else decremented. This prediction scheme is capable of ignoring a single divergent branch.

The *GAg*, *gshare* and *LGshare* branch predictor is a two-level adaptive branch prediction scheme [2]. GAg branch prediction maintains two levels of branch history. The first level is maintained in a buffer called the global history register (GHR) which stores the history of all most recently executed branches. The second level of history, called the pattern history, stores the information about the most likely branch outcome if a particular pattern matches in the GHR. *Gshare* branch prediction indexes the pattern history by doing a logical XOR of the global history and branch address. This scheme is more accurate than the GA share scheme as the branch interference is reduced due to the XORed branch address [1]. The LGshare scheme instead of using a global history of the branches uses the hybrid history by concatenating global history to the local history of a particular branch. This local history is stored in the branch target buffer. This hybrid history is XORed with the branch address to index the pattern history table. For a wider superscalar processor with an instruction block size of eight or more, the probability of

having more than one branch in the fetch group is high [8]. Hence multiple branch prediction is required to ensure a continuous flow of instructions. Instead of predicting individual instructions the block prediction scheme predicts the outcome of the entire block. In this scheme two pattern tables are maintained, the first is the block history table (BHT) which stores the outcome of the recently executed branches and is indexed using the block address. The second level is the pattern history which consists of counters for each possible target of the block and is incremented and decremented depending on the prediction. Reference [3] claims 15% increase in the fetch size for an 8-wide superscalar processor and improvement of 25% for a 12-wide superscalar processor.

According to [5], a higher hardware budget for the branch history tables tends to yield a more accurate branch predictor. Recent studies have shown that feature size reduction and a shorter clock cycle will lead to multi-cycle accesses to large on-chip structures. Hence, the access delay of large branch history tables will reduce the instruction per cycle (IPC) count. For a gshare prediction scheme with a pattern history size of 2 KB implemented for a clock rate of 2 GHz, memory accesses time is doubled causing the IPC to drop by 40% [5]. A Hybrid prediction scheme is proposed by [5] to counter this effect. A Hybrid predictor is similar to the two-level adaptive predictor but has smaller branch history tables.

The implementation in this thesis uses the most simple prediction scheme of two-bit prediction. This scheme was preferred over others because it is less complex than the two-level prediction scheme. The block prediction scheme and hybrid prediction scheme

was not considered as the fetch size of the processor implemented is four; hence the probability of having more that one branch in a fetch group is small.

1.1.2 Branch target buffer indexing

To avoid a pipeline stall the branch prediction mechanism has to predict the branch target address no later than the first pipeline stage, since the branch address is not known prior to instruction decode. Hence some indirect technique is required to index the branch target buffer. Reference [13] highlights this problem and identifies the scheme used in the Intel Pentium processor for a fetch size of two. The Intel Pentium processor indexes the branch target buffer with the address of the instruction in the first decode stage. If there is a hit in the branch target buffer, the branch instruction is predicted without incurring any penalties. Hence, in this case a branch instruction is identified by the address of the instruction fetched one cycle before the branch instruction. This paper argues that the fetch width of a future generation superscalar processor will be much greater that two. To identify a branch instruction, instruction in the decode stage (whose address is used to index the branch prediction information) should always be the same. This is difficult to achieve for a larger fetch size. Reference [13] proposes two indexing schemes for indexing branch target buffers:

1. Basic block based indexing
2. Fetch address based indexing.

According to this paper, the time at which a branch instruction is predicted depends on which address is used to identify it. In order to predict a branch sufficiently early a branch instruction should be identified by the address of an instruction that

dominates the branch [13]. The dominant instruction is the one that is executed prior to a branch whenever the branch is executed. The basic block indexing scheme identifies the branch with the basic block address, whereas the fetch address based scheme identifies the branch with the instruction cache line address. There may be more than one branch instruction in a cache line. In this case, to avoid all the branch instructions in the same cache line mapping to the same branch target buffer entry, all the other branches are identified by their corresponding basic block starting address. These schemes assume delayed branches are not used.

The branch target buffer indexing scheme used in our processor is similar to the basic block indexing scheme. The branch target buffer in our cases fetches four consecutive locations starting from the basic block address. If there is a hit, branch target address corresponding to the first hit is used.

1.1.3 Register file issues

Superscalar processors exploit instruction level parallelism by dynamically scheduling instructions for out-of-order execution. These processors evaluate a large window of in flight instructions to find multiple ready and independent instructions for parallel execution every clock cycle. Supporting a larger instruction window requires larger components within the processor like the register file, reorder buffer etc. Typically, a register file of a superscalar processor is required to be multi-ported to handle more than one instruction every cycle. These large multi-ported register files can potentially

compromise clock cycle time. Reference [12] identifies these problems and proposes two schemes to handle them.

The first scheme handles the problem of increased register file size by a two-level register file, where the number of registers in the critical data path is reduced. The other scheme handles the multi-porting problems by using a minimally ported banked register file.

The two-level register file uses an allocation policy that leaves values that have potential readers in the level one register file and transfer other values to the level two register file. A special hardware component, named *Usage table* [12], keeps track of the potential readers of each register in the level one register file. *Usage table* [12] consists of a counter, called *Pending Consumers* [2], which is incremented during rename if an instruction sources that particular register value. The same instruction decrements the counter when it is dispatched. If a counter for a particular register reads zero, it is copied to the level two register file. Information of this transfer is maintained in a buffer called the *Copy List* [2] to restore the machine state after misprediction.

Even though the port requirement on a register file for a processor of fetch size of eight is at least 24 ports, the average port requirement is fewer for several reasons:

1. Many operands are read from the forwarding network and not from the register file.
2. Many instructions have single register operands.

3. Many instructions are not register write instructions.

An IPC degradation of 2% was observed if the ports on the register file above were restricted to four read and four write ports [12]. This three fold reduction in the ports complicates the issue logic and the restricted write bandwidth must be handled with an added complexity in the commit logic. Hence, [12] suggests a minimally-ported banked register file. The register file bank with a single read port incurs an IPC degradation of 1% because of read conflicts [12]. If the write ports are also restricted to one per bank, IPC degradation increases to 5% [12].

The register file designed for our processor is designed based on s pointer scheme, where pointers to the data are transferred from the reorder buffer to the commit pointer buffer which keeps data at one location during commit.

Background studies put forward in the above discussion helped us make design decisions for a lot of components of the processor design under test.

1.2 Thesis Organization

This documentation is divided into eight chapters. Chapter 2 talks about the overall design of the processor, identifying different components used to realize the design. Chapter 3 talks about the design of the branch prediction mechanism. Chapter 4 describes the design of the MIPS I Instruction set architecture decoder and priorotizer logic for renaming. Chapter 5 presents the design of the register file explaining the

concept of pseudo-registers. Chapter 6 identifies the data inconsistency problem and presents a solution employed in this design. Chapter 7 presents design verification methodology and simulation results. Chapter 8 concludes the discussion by proposing future work beyond this study.

Chapter 2

Architectural Overview

2.0 Introduction

The design under verification here is a four-instruction speculative super-scalar processor, which can fetch four instructions from a liner instruction sequence every clock cycle and dynamically schedule them. Speculative prediction of the outcome of branch or jump instruction is done to reduce control stalls in the processor. This chapter talks about the overall design of the processor, how different components interact with each other, overall data flow and different pipeline stages of the processor.

2.1 Overall design & dataflow

2.1.0 Architectural Features

The processor design is based on the MIPS Instruction Set Architecture; this instruction set architecture is one of the industry standard architectures. The MIPS instruction set comes in four different versions MIPS I, II, III & IV. This processor is an implementation of the MIPS I Instruction Set Architecture. Basic instruction types in the MIPS I instruction set are:

1. Arithmetic & logic instructions for basic arithmetic operation such as addition, subtraction and logical operations like AND, OR etc.,

2. Branch & Jump instructions, they can be conditional or unconditional. These instructions decide the control flow of the program.
3. Multiply & Divide instructions are special instruction for multiplication and division operations, as these instructions take longer than the other instructions they are grouped into one type.
4. Load & Store instructions, these instructions perform data transfers from and to the data memory.
5. Floating-point instructions, these instructions operate on floating-point numbers.
6. Co-Processor 0 Instruction, co-processor is the control functional unit described by the MIPS Instruction Set. The co-processor 0 handles interrupt's, configuration options and controls on-chip functions like the cache and timer. A set of instructions described in the MIPS instruction set interacts with the co-processor 0.
7. BREAK and SYSCALL instructions fall in a special category called miscellaneous instructions.

The SDE-MIPS compiler is used to generate memory dumps of the 'C' programs. This compiler does not handle interrupts on its own; hence interrupt handling capabilities are not build in this processor design. As the processor does not handle interrupts the co-processor 0 is designed as a static element that always has logical '0' at the output. It is further assumed that the compiler does not produce floating-point instructions. This assumption is true as a flag in the compiler make file restricts the compiler from generating these instructions. Hence, the processor does not need capability to execute

floating-point instructions. As the main focus of the study is on the processor architecture and not on the design of the peripherals it is also assumed that the instruction and data cache is infinite.

There are six functional units in the processor to execute up to six instructions in parallel. They are:

1. Two Arithmetic Logic Units (ALU).
2. One Branch and Jump Unit (BJU).
3. One Multiply and Divide Unit (MDU).
4. One Load and Store Unit (LSU).
5. Co-processor 0 (CP0).

The probability of two out of four instructions in a block to be arithmetic is high as the number of arithmetic instructions is nearly twice that of any other types. Hence having two arithmetic logic units will reduce a lot of structural stalls.

The processor is clocked at 10 nsec. and is divided into seven pipeline stages which are.

1. Instruction Fetch.
2. Instruction Decode/Issue.
3. Instruction Dispatch.
4. Instruction Execution.
5. Instruction Write Back.
6. Instruction Commit I.

7. Instruction Commit II.

Each pipeline stage is of the duration of one clock cycle. The diagram on the next page shows the different pipeline stages divided by pipeline registers. The first pipeline register is the program counter. Each component in the processor is divided in time to show how it interacts with other components in each pipeline stage.

This processor has 32 architectural registers, as described in the MISP I instruction set. In addition to these architectural registers, there are 32 extra registers called pseudo-registers. These registers store data temporarily before the machine state is changed. These 64 registers are implemented as a single register file called the Value Buffer. Results of the instruction executions are written back on two 32-bit buses called Common Data Bus I and Common Data Bus II respectively. In addition an independent write back bus is provided on the branch and jump functional unit to write the branch target address after evaluating a particular branch instruction.

The next half of the chapter talks about design issues involved in the super-scalar architecture and explains how these issues are handled with different components in the processor. The design details of each of these components will be discussed at length in the next few chapters.

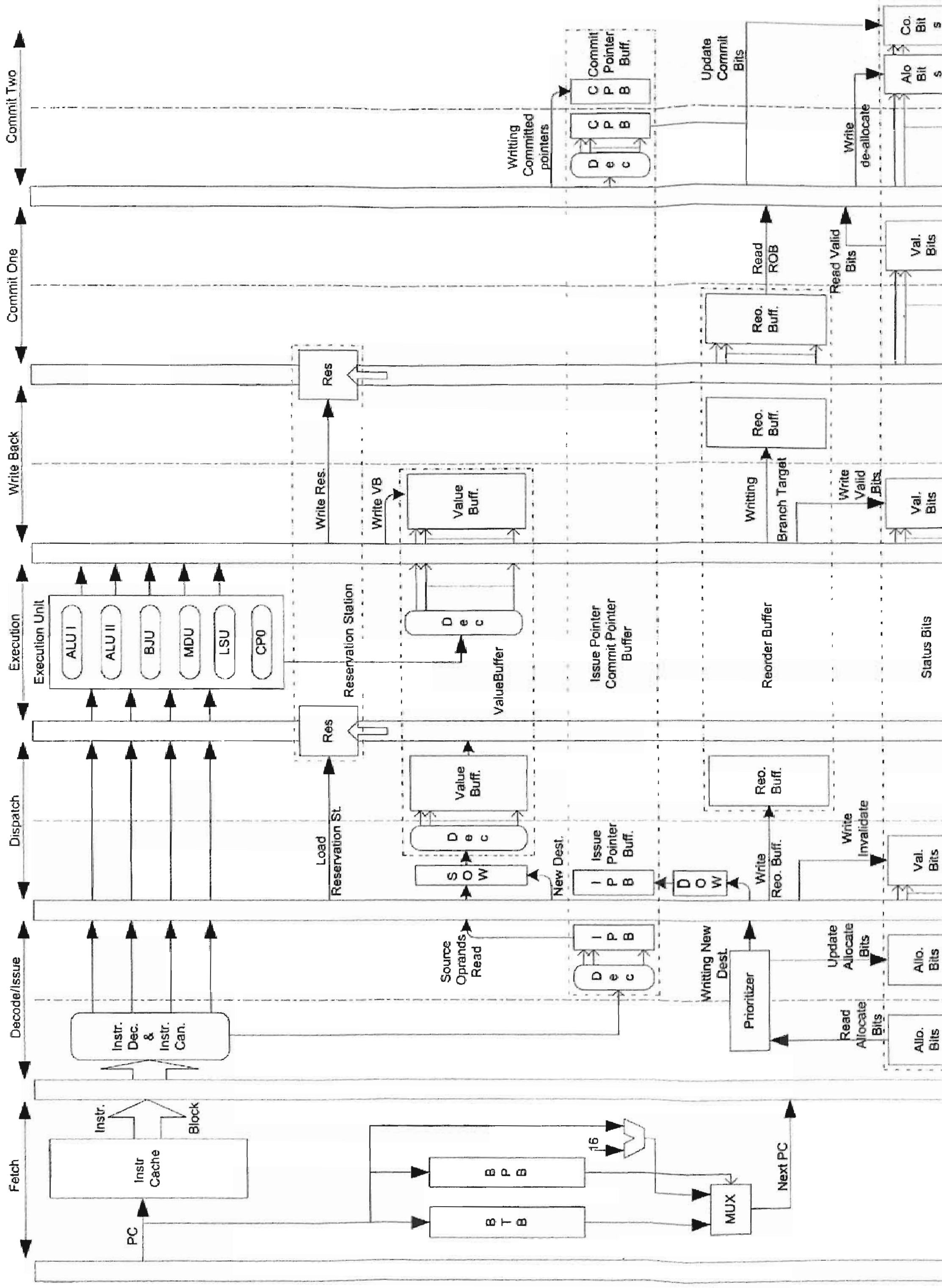


Figure 2.1: Overall Architecture.

2.1.1 Handling Branch and Jump Instructions

Control flow of the instructions in the processor is decided by the number of branch and jump instructions. On an average there is a branch or jump instruction every five instructions [1]. Whenever a branch or a jump instruction is encountered, the next instruction block to be fetched is not known until this instruction finishes execution; hence the processor has to stall. Stalling the processor every time a branch or jump instruction is encountered is highly inefficient. This inefficiency is much higher for a super-scalar processor. A Super-scalar processor looses many more potential instructions that could have been fetched during the stall time, than a linear processor, since the super-scalar processor fetches more than one instruction in a clock cycle as compared to the linear processor that fetches one instruction every clock cycle.

Speculative prediction of the branch and jump instruction is done to avoid stalling every time these instructions are encountered in this processor design. Prediction of the outcome of these instructions is based on the outcomes of their earlier runs. These earlier outcomes are stored as a two-bit saturating counter in a special cache called the Branch Prediction Buffer (BPB) and the corresponding branch target address is stored in another special cache called the Branch Target Buffer (BTB). Whenever a branch or a jump is encountered again the most significant bit of the counter predicts from where the next instruction block is fetched. If the most significant bit of the counter is set we say the branch is predicted *taken* and the next instruction block is fetched from the corresponding branch target address. If the most significant bit is reset, the branch is said to be predicted *not taken*. In this case the next instruction block is fetched from the next logical program

address. If a particular branch or jump instruction is encountered for the first time, it is always predicted *not taken* since, this new branch has never been executed.

The above branch prediction mechanism is capable of predicting one branch or jump instruction per clock cycle. Hence only the first encountered branch or jump instruction is predicted and all instructions after the delay slot of the first branch or jump instruction are cancelled.

2.1.2 Handling Speculated Data

The results produced by MIPS I instructions are 32-bit data. For a speculated instruction this data has to be stored in a temporary location until the time the branch instruction is executed in the pipeline. If the predicted outcome matches the outcome of the branch instruction after execution, these data should be allowed to be written into the architectural registers and change the machine state. Hence to complete a speculated instruction, two 32-bit data transfers are needed. This is inefficient, this inefficiency further increases because of the fact that the processor fetches four-instruction per clock cycle and statistically there is one branch or jump every five instruction [1], causing a large number of speculated instructions at any given time to be waiting to be completed in the processor.

The processor design under verification minimizes this inefficiency by using a pseudo-pointer scheme. This scheme renames logical destinations of incoming instructions by assigning pointer to the pseudo-register location and maintaining two lists

of these renamed pointers or pseudo-pointers. One list has the pseudo-pointers of all the uncompleted or speculated instructions and is called the Issue Pointer Buffer. The other has the pseudo-pointers of all the completed and un-speculated instructions and is called the Commit Pointer Buffer. These lists translate a logical source or destination of an incoming instruction to their pseudo-register locations. The results produced by the instructions after execution is written into the pseudo-register at which the pseudo-pointer points. The 32-bit data remains in the same location. Only the pseudo-pointers are copied form one list to the other when an instruction completes execution. The pseudo-pointers are 6-bits requiring a 6-bit data transfer as compared to a 32-bit data transfer earlier. Whenever a mispredicted branch is identified, the list having pseudo-pointers of the completed instruction is copied to the list having pseudo-pointers of the uncompleted instruction. This is done to bring the machine state back to where it was before the mispredicted branch or jump instruction.

This process of assigning pseudo-pointers to the incoming instructions and transferring pseudo-pointers instead of the data itself is more efficient.

2.1.3 Handling out of order execution

As each functional unit has different latencies, even though the instructions are fetched in order, they don't complete in order. To force the instruction to complete and change the machine state in order, a special first-in-first-out (FIFO) wrap-around queue is used, called the Reorder Buffer. The Reorder Buffer maintains the ordering of the instructions and makes sure instructions complete or and committed in order.

When instructions are issued in the pipeline, they reserve a slot in the Reorder Buffer. The slots in the reorder buffer are reserved in program order and each instruction is tagged with a reorder buffer slot number. NOP's in the instruction stream do not find a spot in the reorder buffer. Every cycle the reorder buffer tries to commit as much as four instructions if they have completed execution.

Committing and issuing of the slots are achieved by two pointers, called the Commit-Pointer and the Issue-Pointer. The issue-pointer points to the locations from where the reorder buffer slots have to be issued, and the commit-pointer points at the location of the last committed instruction.

Mispredicted branches are detected at the top of the reorder buffer queue or when the commit-pointer points to the branch instructions. When a misprediction is detected, the processor restores itself by canceling all the instructions after the mispredicted branch. A new instruction block is then fetched from the correct target address.

Detection of misprediction is delayed till that particular branch or jump instruction reaches the top of the reorder buffer queue even thought the branch conditions are evaluated in the instruction execution stage of the pipeline. Hence even though the misprediction could have been detected at the end of instruction execution pipeline stage, it is delayed until the branch or jump instruction reaches the top of the reorder buffer queue. The time between the evaluation of the branch condition and detection of misprediction can be several clock cycles as there might be some instructions before the

branch or jump instruction in the reorder buffer that are yet to be committed. This is inefficient, because knowingly wrong instructions are fetched from the time the branch condition was evaluated until misprediction is detected if the branch or jump instruction has mispredicted. This is done to keep the architecture simple.

2.1.4 Handling dynamic scheduling of instructions

An instruction should not be allowed to execute unless both the source operands are ready. This is called a Read after Write (RAW) hazard. This hazard is due to inherent dependencies in the instruction codes, and is more prominent in the super-scalar architecture, since more then one instruction is fetched in a clock cycle and there is a greater possibility of fetching instructions with dependencies in the same instruction block. This problem is avoided by dynamically scheduling the instruction at run time.

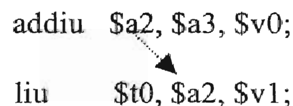


Figure 2.2: Sample code.

It can be seen form the source code above that the first instruction has a destination register of '$a2' and the second instruction has the same register as one of its source operands. Unless the first instruction finishes execution, second instruction should not execute to avoid erroneous execution, since the data of '$a2' would not have been updated by the first instruction.

The Reservation Station is a small buffer before the execution pipeline that holds the instruction until both the source operands are ready. Each functional unit has its own reservation station. For this design the size of the reservation station is kept as four; hence it can hold four instructions simultaneously. The reservation station is capable of issuing out of order instructions into the execution pipeline the both the source operands are ready and there is no structural hazard. For an instruction waiting for source operands, the reservation station snoop's the common data-buses every clock cycle, and if the data is available, the instruction is issued for execution in the next clock-cycle.

2.1.5 Handling Multiply and Divide Instruction

Results produced by the multiply or divide instructions are 64-bit. These results are not directly written into the architectural registers. Instead they are written into a small buffer called the Hi-Lo Buffer as defined by the MIPS instruction set. The Hi-Lo buffers are written during the last stage of the instruction execution. There are special instructions to copy these data to the architectural registers from the Hi-Lo Buffer.

2.1.6 Handling Memory Stores

Memories are traditionally slower then the processor, hence whenever a load or a store instruction completes the processor has to wait until the data memory is accessed. This causes stalls in the processor. These stalls are avoided in this design by using a special purpose buffer called the Store Buffer. The Store Buffer is written instead of the memory once the store instruction completes, and the store buffer interacts with the memory without holding up the processor.

The Store Buffer writes into the memory whenever the memory data bus is available and there are no pending loads from the memory. Load instructions are given preference over the store instruction, as loads from the memory hold up the execution of further instructions in the processor if there is data dependency causing the processor to stall. The Load instruction first scans the store buffer before it looks for the data in the memory, as the latest copy of the data might be in the store buffer. This ensures the latest copy of the data is always picked by the load instruction.

2.1.7 Handling Data Consistency

Inherent data dependencies in the execution codes may lead to some inconsistencies in handling the data in the case of this processor as each incoming instruction's logical destination registers is renamed to a new pseudo-register. There can be a situation when two or more instructions fetched in the same instruction block have identical logical destination registers. This causes aliasing of that logical destination register, as more than one pseudo-pointer points to it. This might cause errors when an incoming instruction refers to the same register for data by reading its pseudo-pointer from the issue pointer buffer.

This inconsistency is removed by doing destination overwrite while updating the issue pointer buffer. The pseudo-pointer of the latest instruction overwrites the pseudo-pointer picked for the other instructions that have the same logical destination in an instruction block. Destination overwrite removes the ambiguity for the subsequent incoming instructions while accessing the pseudo-pointers for data.

Data inconsistency may also occur when logical source registers of an incoming instruction are the same as the logical destination register of another instruction in the same instruction block. This inconsistency occurs because the issue pointer buffer is read before it is written for a particular instruction block. The issue pointer buffer is not updated with the latest pseudo-pointer picked for the logical destinations when it reads the pseudo-pointer from logical source registers. Incorrect pseudo-pointers are read causing an erroneous operation. This can be avoided by overwriting the logical sources with the newly picked pseudo-pointers before reading the data.

The special hardware components discussed above have distributed timing over the seven pipeline stages. These pipeline stages will be discussed in the next part of the chapter.

2.2 Pipeline Stages

2.2.0 Introduction

Pipelining is done for maximum utilization of the processor hardware components. If the processor is un-pipelined, processor hardware such as the Instruction Decoder is used once during an instruction lifetime. Instruction lifetime is the latency of the processor, which is typically more then one cycle. Hence the hardware is idle for the majority of the time, which is inefficient. To reduce this inefficiency, the instruction lifetime is divided into independent stages. This is called pipelining and each stage is called a pipeline stage. Since the pipeline stages are independent of each other, each

hardware component can be used in every stage for different instructions. This makes the utilization of the hardware components much more efficient. Pipelining enables the processor to process more than one independent instruction every clock cycle, and the exact number of independent instructions executing simultaneously at a given time depends on the number of pipeline stages.

This design under verification has seven pipeline stages as named earlier. This section talks about each pipeline stage's function in execution of the instruction.

2.2.1 Instruction Fetch Stage

Instruction fetch is the first stage in the lifetime of an instruction in the processor. This stage fetches instructions from the instruction cache. As the processor fetches four instructions in a clock cycle, instructions are fetched from four consecutive locations starting from the program counter.

The branch prediction mechanism works in this pipeline stage since it is necessary to determine the next program counter, where the next block will be fetched for the next clock cycle. The current program counter is used to index the branch target buffer (BTB) and the branch prediction buffer (BPB). If there is a hit in the branch prediction buffer and the branch target buffer, and the prediction is *taken*, then the next program counter is the branch target address obtained from the branch target buffer. On the other hand if the prediction is *not taken*, the next program counter is the address of the fifth instruction from the program counter. When a branch is encountered for the first time, it is always

predicted to be *not taken* since no data is available on this branch in the branch target buffer or the branch prediction buffer.

2.2.2 Instruction Decode/Issue Stage

The instruction block that is fetched is passed to this stage for decoding. Decoding is done in the first half of this stage. The decoder decodes the opcodes for each instruction, the source and destination register address, and it also produces a group of control signals that aid in the proper execution of the instruction. These control signals identify the function unit the instruction is intended for. It also distinguishes between the immediate mode instruction, having immediate data as one of the operands to the register mode instruction. It also points out if there are any NOP's present in the instruction block, and a few others that will be discussed in later chapters in detail.

Each instruction needs two-source operands for execution and one destination register to write the results after execution. The destinations are to be renamed to the Pseudo-register locations. This is done by looking at the allocate bits of each location of the register file and finding four location which are not allocated, i.e. empty. This is achieved using a special component called the Prioritizer. If the prioritizer fails to find four empty locations it issues a structural stall until four empty locations are found. The worst-case delay of the prioritizer is less the 10 nsec. which is one clock period and is one pipeline stage delay.

The source operands obtained from the decoder are decoded again to find the pseudo-pointer for these registers from the issue pointer buffer in the second half of this stage as shown in the Figure 2.1.

If the instruction block has more the one branch or jump instruction, the instructions after the delay slot of the first branch or jump instruction are cancelled and re-fetched in the next clock.

2.2.3 Instruction Dispatch

Instruction dispatch is the third state of the pipeline. In this stage the pseudo-pointers picked up by the prioritizer in the instruction decode state is written in the issue pointer buffer after destination overwrite, if required, and the valid bits for corresponding locations are reset marking them invalid. The source pseudo-pointers picked in the last stage passes through source overwrite logic before they are used to decode the value buffer locations in the first half of this stage, and the value buffer is read in the second half of this stage if the valid bits for these particular locations are set. If the valid bit for a particular location is not set, the corresponding operand is marked invalid, and the instruction waits in the reservation station for the value in that particular location to be ready. These values are read directly from the common data-bus once they are written back.

Reorder buffer locations are allocated for all the new incoming instructions in program order. This is done to prevent out of order committing of the instruction at the

end of the pipeline. Each instruction is tagged with their reorder buffer location address to facilitate prioritizing the instruction in the reservation station for execution. If the reorder buffer is full, there is a structural stall until the time the locations are available.

Each instruction with the source operands and the reorder buffer location address is written into their corresponding reservation station identified by the functional unit codes produced by the instruction decode. If the reorder buffer is full, there is a dispatch stall until the locations are available in the reservation station. There are also structural stalls if there are more instructions of a particular type than the number of functional units for executing that type of instruction. In this case, the instructions that were successfully dispatched into the reservation station are marked, and the un-dispatched instructions are dispatched again until they are written into the reservation station.

2.2.4 Instruction Execution

The instruction stays in the reservation station until both the source operands are ready. For reading the source operands that were marked invalid, the reservation station snoop's the common data bus every clock cycle. When the data for that particular source register is on the common data bus, it is written in the reservation station, and the location is marked as valid and ready for execution.

The instructions that are marked ready for execution are issued to the functional unit for execution. There are five functional units. There are two ALU's, one MDU, one BJU and one LSU.

Each ALU, CP0 and BJU is a single stage unit and takes one clock cycle to finish execution, while the MDU takes four clock cycles to execute and the LSU takes two clock cycles to execute.

The renamed destination Pseudo-pointers are decoded in this stage to update the data in the value buffer; this is called pre-decoding. As the value buffer is updated in the first half of the next clock cycle, there is no time to decode the destination Pseudo-pointers in the same half cycle when the value buffer is updated.

2.2.5 Instruction Write Back

As the name suggests, the results of an instruction execution is written back in this stage. There are a total of two data buses called the common data-buses. A write back controller assigns the data buses to different functional units depending on the availability. If the data bus is not available, there is a structural stall that stalls that particular functional unit.

The reservation station is updated if any instruction is waiting for these results. The value buffer is written in the first half of the clock cycle with the pre-decoded address lines, and corresponding valid bits are set indicating the data has been written back for future instructions. The Branch and Jump functional unit has a separate bus to write the result of the execution in the reorder buffer, which is written in the second half of this stage. The branch targets are written into the branch target buffer only if the

branch or jump has mispredicted and the branch instruction is picked for committing by the reorder buffer, the branch prediction buffer is updated every time the branch is completed and is picked for committing by the reorder buffer.

2.2.6 Instruction Commit I

This stage decides whether an instruction will be allowed to change the machine state or commit. The reorder buffer is capable of committing up to four instructions in a clock cycle depending on the status of the instructions. The reorder buffer reads four consecutive instructions starting from the commit-pointer and checks the valid bits for each instruction's destination, to distinguish between complete and incomplete instructions. It selects instruction until the first incomplete instruction to be committed.

When a branch instruction is encountered and is mispredicted, the reorder buffer writes the correct prediction and the corresponding branch target address in the branch prediction buffer and the branch target buffer respectively. It issues a restore signal to indicate the misprediction and flushes all the instructions after the mispredicted branch from the reorder buffer. At the restore signal, the processor restores the machine state to the state it was before the mispredicted branch was fetched by copying all the commit pointer buffer to the issue pointer buffer, flushing all the reservation stations, asynchronously resetting all the pipeline registers to nullify all the instruction in the processor pipeline, and setting the program counter to the correct branch target address. If the encountered branch is predicted correctly, the prediction in the branch prediction buffer is updated.

2.2.7 Instruction Commit II

The instruction selected in the last stage for committing, are committed in this stage by writing the destination pseudo-pointer into the commit pointer buffer. The commit pointer buffer is read in the first half of this stage to read the old pseudo-pointers, to allow the de-allocation of these old pseudo-pointers which can be over written in the next half cycle. The commit bits for the corresponding destination pseudo-pointers read from the reorder buffer is set, indicating the particular instruction is no longer speculated and has altered the machine state.

This chapter talked about the overall architecture of the processor outlining the different issues involved in this design. It discussed at length the different pipeline stages in the processor and how each hardware component interacts in these stages. The next few chapters will focus on different hardware components overviewed in this chapter and explain in detail the design issues of each hardware component.

Chapter 3

Design of Branch Prediction Mechanism

3.0 Introduction

Good branch prediction is imperative for good performance of a processor; this is more critical for a super-scalar processor. Since an instruction block is more than one instruction wide in a super-scalar processor, the penalty for misprediction is much higher, as was explained in Chapter 2. This processor implements a two-bit saturating counter for each branch that is predicted. The most significant bit of the counter is the prediction for that branch, a logical '1' represents a *taken* branch and a logical '0' represents a *not taken* branch.

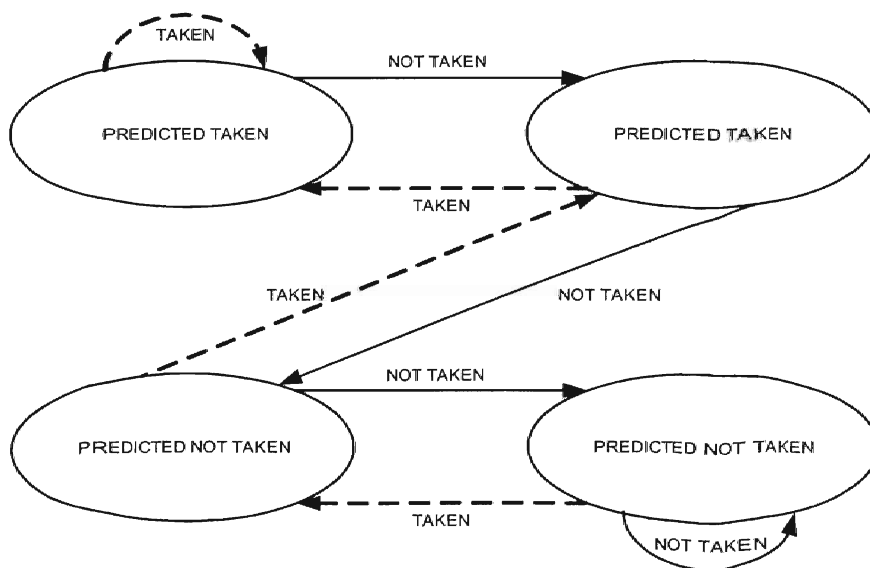


Figure 3.1: States in a two-bit prediction scheme.

Two bits are used to encode four states as shown in the diagram. Two-bit prediction is better then a one-bit prediction scheme because a branch that strongly favors *taken* or *not taken*, as many branches do, will be mispredicted less frequently.

This processor implements two special cache's to implement this scheme called the Branch Target Buffer and the Branch Prediction Buffer. The correct target address of a branch is stored in the branch target buffer, and the corresponding predictions are stored in the branch prediction buffer. These buffers are updated when branch misprediction is detected in the reorder buffer.

Branch and Jump instructions are treated the same way and the target addresses of both types of instructions are stored in the same branch target buffer; there is no separate jump target buffer in this design. The scheme predicts one branch in a clock cycle. If there is more then one branch or jump instructions in an instruction block, the instructions after the delay slot of the first *taken* branch or jump instruction are cancelled.

The following chapter talks about the design of these buffers, talking in detail about the architecture of each of these buffers.

3.1 Branch Target Buffer

The branch prediction mechanism works in the fetch stage of the pipeline. As the instructions are not yet decoded, branch or jump instructions cannot be identified in the fetch group. Since the branch or jump instruction's address is not known to index the branch target buffer. It is indexed using the lower sixteen bits of the current program

counter to fetch data from four consecutive locations in the branch target buffer. As four locations are read simultaneously, the branch target buffer has to be implemented as a four-port memory architecture, which is expensive. Hence, the branch target buffer is implemented as a four-memory bank structure. Each memory back has one read–write port, and all memory banks can be accessed simultaneously independent of each other. The last four bits of the memory address, the program counter in this case, identify the memory bank, and depending on the number of locations in the branch target buffer, the memory address is divided into index and tag fields as shown below.

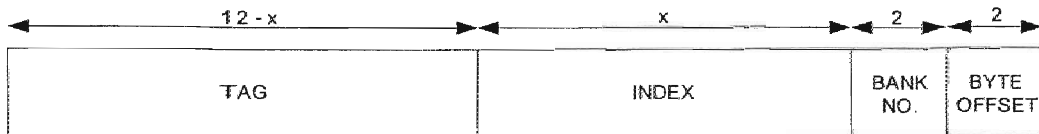


Figure 3.2: Memory Address Division to access Branch Target Buffer.

Each location of the branch target buffer memory bank consists of a tag, a branch target address and a valid bit. Valid bits identify the valid locations in the branch target buffer, and tags are used to determine hits in the branch target buffer. As four consecutive locations are read and their tags compared with the, program counter, there is a possibility of a hit on more then one memory bank simultaneously. In this case the branch target address from the nearest bank to the program counter is read out and the others are ignored.

Since only a *taken* branch or jump instruction is written into the branch target buffer, it is not updated every time the branch is detected in the reorder buffer. Hence separate read and write ports were not necessary. When a *taken* branch is detected in the reorder buffer the target address corresponding to that instruction is written into the

branch target buffer with the address of the branch or jump instruction. A structural stall is issued causing the instruction fetch to stall for that cycle. The architecture is as shown in the diagram below.

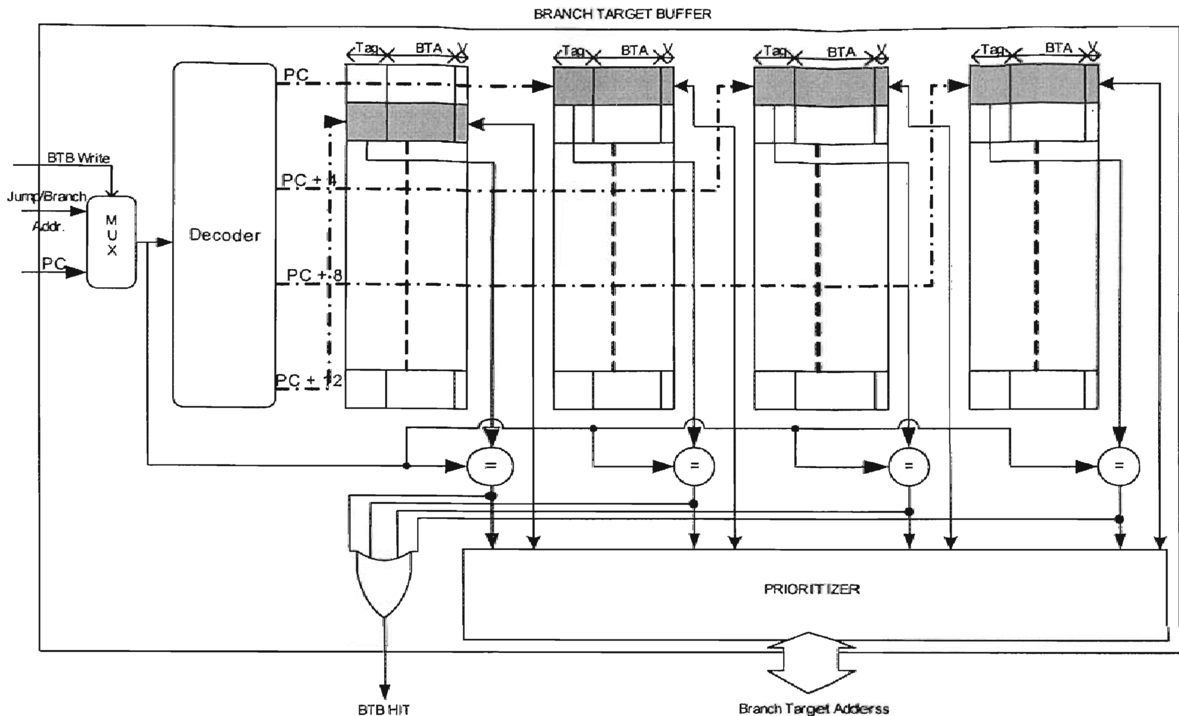


Figure 3.3 Branch target buffer architecture

The branch target buffer hit is the logical OR of hits detected for all the memory banks. The hits from the individual memory bank and corresponding target addresses are prioritized using the prioritizer logic to give the branch target address corresponding to the nearest memory bank to the program counter as the output. Whenever a mispredicted *taken* branch is detected in the reorder buffer, a write signal is set and the multiplexer, as shown in the diagram above, selects the branch or jump instruction address to index the branch target buffer. The tag and the correct branch target address are written and the valid bit set in the memory bank indicated by the two least significant bits of the

instruction word address. Since the exact instruction address of the branch and jump instructions is known at the time of writing the branch target address, only one of the four select signals coming out of the decoder is active, at that time.

3.2 Branch Prediction Buffer

The branch prediction buffer is a small special purpose cache that holds the prediction for every corresponding branch in the branch target buffer. The branch prediction buffer is also implemented as a four memory-bank structure since four consecutive locations corresponding to the branch target buffer have to be read simultaneously in the instruction fetch stage. Initially every branch is predicted *not taken*, as the branch has never executed there is no data available for that particular branch. The branch prediction buffer is updated every time a branch is encountered in the reorder buffer irrespective of its outcome. Hence if there is only one read-write port, there will be a structural stall every time a branch is encountered in the reorder buffer, which is inefficient. To avoid stalling every time a branch is encountered in the reorder buffer, the branch prediction buffer is provided with two separate ports, one for reading and the other for writing. The only time a structural stall can occur is when the branch prediction buffer reads and writes the same location in the same memory-bank simultaneously. The architecture of the branch prediction buffer is as shown in the Figure 3.4. The address decoding for tags, bank number and the index is the same as the branch target buffer discussed above.

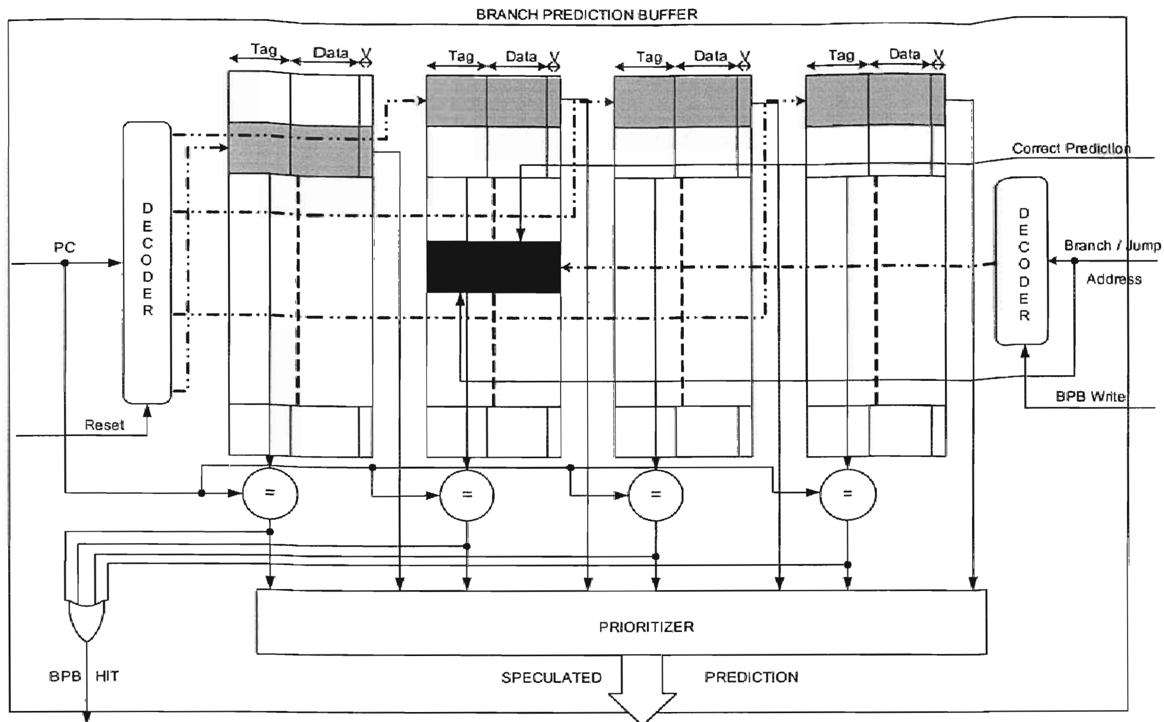


Figure 3.4: Branch prediction buffer architecture

It can be seen from the diagram that there are two separate decoders for the branch prediction buffer, one decodes the program counter for reading from the buffer and the other decodes the branch or jump instruction address to write the updated prediction or the correct prediction into the buffer. Each location of the branch prediction buffer consists of a tag, two-bit predictor and a valid bit. Valid bit indicates the validity of the data in that location and is set every time a correct prediction is written. The tags are used to generate the hit signal from the branch prediction buffer by comparing the tags with a part of the program counter. Whenever a branch is detected in the reorder buffer a 'BPB Write' signal is set enabling the decoder and writing into the branch prediction buffer. Since the exact branch or jump instruction address is known at the time of writing the correct prediction, the decoder has only one location in the branch prediction buffer to

decode, unlike the decoder used for reading. Since the branch or jump instructions cannot be identified in the instruction fetch stage, four consecutive locations are to be fetched starting from the program counter. A logical OR of the hits from each of the four memory- banks give the branch prediction buffer hit signal. As more than one location is read every time from the branch prediction buffer, there can be simultaneous hits on different memory-banks. The hit on a memory-bank nearest to the program counter is selected, and the prediction corresponding to that hit is read out as the speculative prediction of the branch prediction buffer using a prioritizing mechanism as shown in the diagram above, ignoring others.

The branch target buffer and the branch prediction buffer were not implemented at the gate-level, but were implemented behaviorally. We tried keeping them both realistic enough so that they can be implemented at the gate-level if considered necessary. The size of both the buffers is kept as a design variable to study the impact of the different sizes on the overall performance of the processor. The branch target buffer and the branch prediction buffer work in parallel with the instruction cache. Instruction cache is assumed to be infinite in size and cache misses are not simulated in the design. This mechanism predicts only the first branch in the instruction block and ignores other branches, if any, in the instruction block.

Chapter 4

Instruction Set Architecture Decoder & Prioritizer

4.0 Introduction

The design under consideration is based on the Million Instruction Per Second (MIPS) instruction set architecture. The MIPS instruction set has four different version namely MIPS I, MIPS II, MIPS III & MIPS IV. This processor understands and executes the MIPS I instruction set. The decoder is the second stage in the lifetime of the instruction in the processor after the instruction fetch. Instruction decode is done in the first half of the instruction decode/issue pipeline stage. MIPS I instructions are divided into seven functional groups as follows:

1. Load and Store instructions.
2. Arithmetic and Logic instruction.
3. Multiply and divide instructions
4. Branch and jump instruction
5. Co processor 0 instruction.
6. Miscellaneous instructions.
7. Floating-point instructions.

The MIPS I instruction set implements delayed loading, the architecture does not allow a particular register 'Rs' to be used as a source register for an instruction following

the load instruction, if this particular register was destination of the load instruction. The instruction following a load instruction is called the load delay slot instruction, since the loaded data can only be used after the load delay slot.

Multiply and divide instructions produce a 64-bit result. These results are written into a Hi-Lo buffer, as defined by the instruction set architecture. Branch and jump instructions are architecturally implemented with a delay slot. Hence, the instruction following the branch or jump instruction has to be executed before the branch or jump instruction takes any action.

A MIPS I instruction is a single 32-bit aligned word. Formats of the different types of instructions are as follows.

In an I-Type (immediate type) instruction, one of the source operands in this type of instruction is immediate data; the format is as shown.

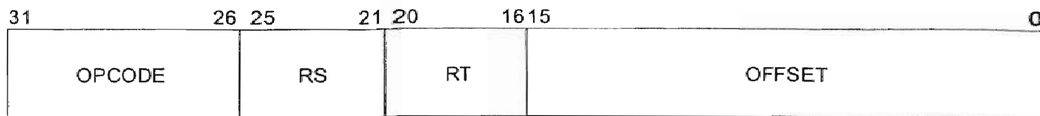


Figure 4.1: I-Type instruction format.

In a J-Type (jump type) instruction, this instruction provides a 26-bit instruction index for calculating the jump target address. The format is as shown.

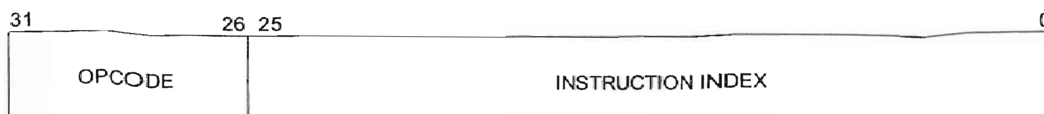


Figure 4.2: J-Type instruction format.

In an R-Type (register type) instruction, this instruction provides both the source operands (RS & RT) and destination (RD) as registers. The Shift amount (SA) is also provided for some specific instruction such as SLL etc.

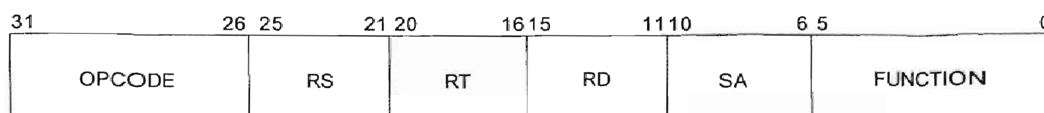


Figure 4.3: R-Type instruction format.

There are 12 load and store instructions, 24 arithmetic and logic instructions, 8 Multiply and divide instructions, 12 branch and jump instructions, 3 co processor instructions and 2 miscellaneous instructions in the MIPS I instruction set. Floating-point instructions are not implemented in the processor design.

The destination registers of each instruction fetched in an instruction block every clock cycle have to be renamed to assign them a new pseudo-register location. This is achieved using a special hardware component called the prioritizer.

This chapter talks about the design of the instruction set decoder and explains the different control signals generated to properly execute every instruction. The chapter also discusses the design of the prioritizer.

4.1 Instruction set architecture decoder design

The decoder operates in the first half of the instruction decode/issue stage of the processor pipeline. The decoder generates a set of control signals that facilitate instruction execution in the processor. Different control signals for the decoder are as shown in the block diagram below. The inputs to the decoder are the instructions fetched in the last stage from the instruction cache in the format described above. Reset and restore signals are also input to the decoder to reset all the outputs of the decoder whenever either of these signals is set.

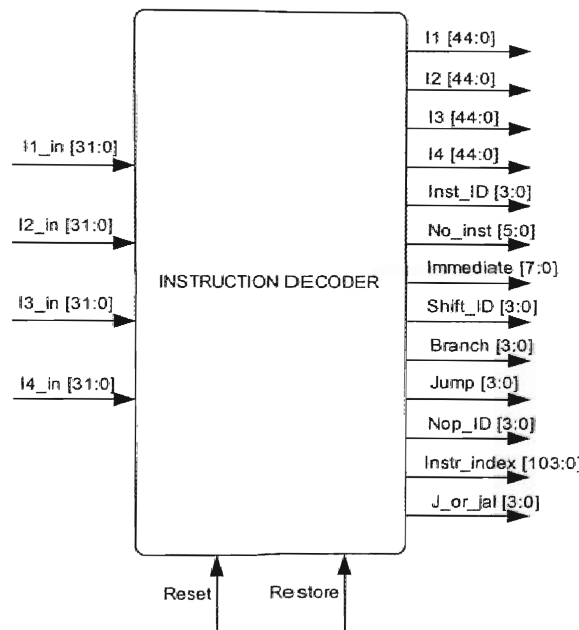


Figure 4.4: Instruction Set Architecture Decoder.

The output format for the instruction I1, I2, I3 and I4 is as shown below

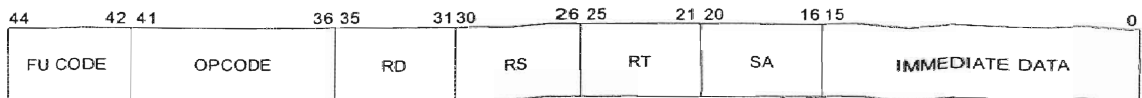


Figure 4.5: Decoder output format for instructions.

The 'fu code' or functional unit codes identity the functional unit for which each instruction is intended. Functional unit codes are different three bit binary numbers assigned to each functional unit. They are as shown in the table below

Functional Units	**Codes**
Null operation (NOP)	000
Co-processor 0	001
Multiply & Divide Unit	010
Arithmetic & Logic Unit	011
Branch & Jump Unit	100
Load & Store Unit	101

Table 4.1: Functional Unit Codes for the Processor

Opcodes are the primary six-bit operation codes that are understood by the functional units for executing the instructions. RD defines the five-bit destination register specifier where the instruction has to write after completion, and RS & RT defines the two five-bit source register specifies for a particular instruction. For instructions that perform shift operations such as SLL etc., the shift amount has to be provided along with other information. This information of the shift amount is conveyed using the SA fields of the instruction coming out of the decoder. For the I-Type instructions one of the source operands is the immediate data that is supplied as part of the instruction. The lower sixteen bits of the decoder's output provides the immediate data if any.

There is a possibility that one or more instruction in the fetched block is a null operation (NOP). Hence, a four-bit control signal named 'inst_ID' gives the position of instruction by setting the bit corresponding to the instruction in the instruction block and resetting all other bits. The prioritizer that finds the pseudo-registers for the incoming instruction block uses this signal.

Identifying the number of instructions in a particular instruction block, given by a six-bit control signal named 'no_instr', is required by the reorder buffer to assign a reorder buffer slot for each of these instructions. This is done to prevent out of order update of the machine state.

For identifying the immediate type instructions in the instruction block, an eight-bit signal named 'immediate' is provided. There are two source operands in each instruction and there are four instructions in an instruction block. Hence, the immediate data can be for any of the eight source operands. Two bits, one for each operand is assigned to each instruction in the instruction block. Whichever operand has immediate data that particular bit is reset and others are set.

Shift instructions need the shift amount as one of the operands. To identify these instructions a four-bit control signal called the 'shift_ID' is used. The bit corresponding to the shift instruction is set and the others are reset.

Branch and jump instructions are identified using two separate two-bit signals called 'branch' and 'jump'. The bit corresponding to the branch or jump instructions are set and the others are reset.

J-Type instructions are identified by a four-bit control signal called 'j_or_jal' and the corresponding instruction index is given on a 104-bit bus called 'inst_index'.

Decoding instructions correctly and identifying each instruction uniquely is a critical step in the correct operation of an instruction. The control signals described in this section help the processor uniquely identify each instruction and execute them in accordance with the rules set by the MIPS instruction set architecture.

4.2 Prioritizer design

To avoid an out of order update of the machine state, the speculated instructions should not be allowed to change the machine state. Hence the logical destination of each instruction fetched in a cycle has to be renamed to a pseudo-register. Prioritizer logic is used to achieve this. This logic looks at the allocate bits of each location of the register file and picks out four locations which have not yet been renamed. If the prioritizer is unable to find four new locations, a structural stall is issued, and the processor stalls until four new locations are found. The overall circuit of the prioritizer is as shown in Figure 4.6.

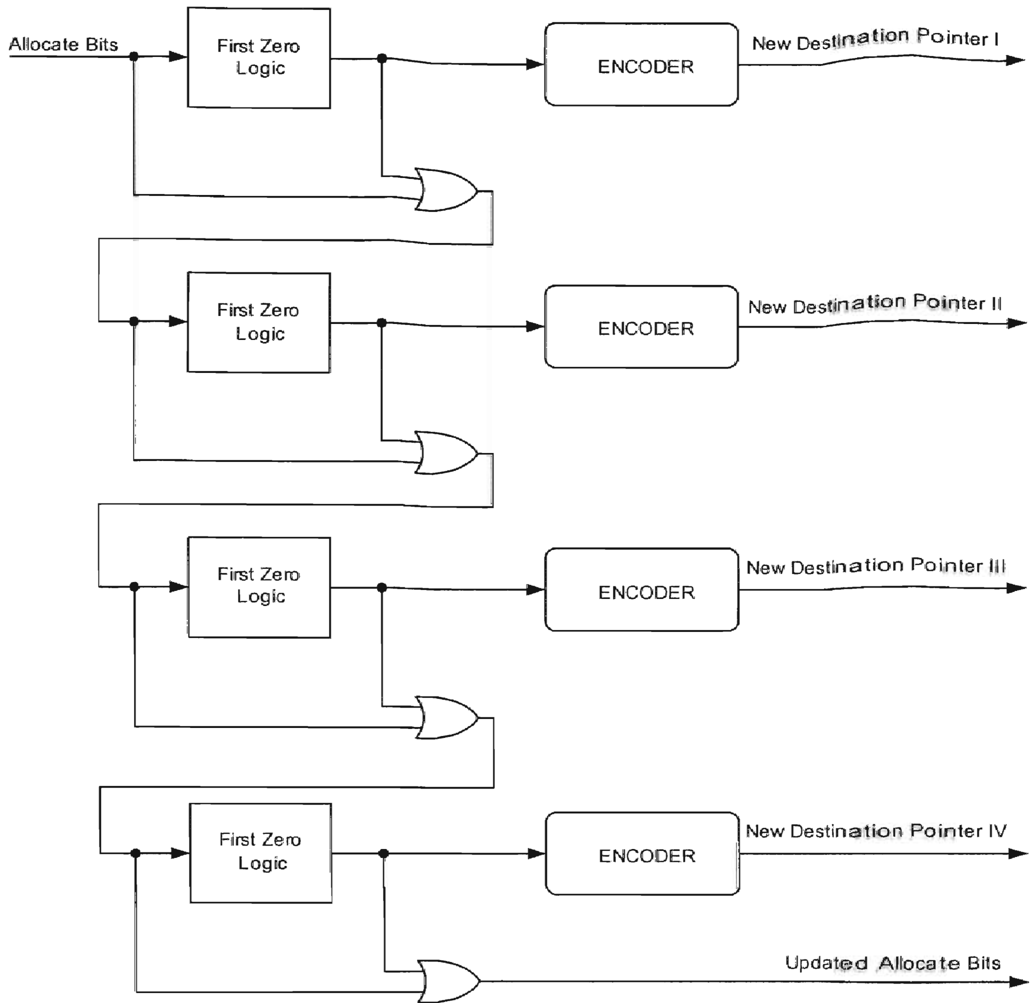


Figure 4.6: Prioritizer.

As can be seen from the diagram the allocate bits are scanned to find the first non-allocated location. That location is encoded using a 64 to 6 bit encoder to give the first new pseudo-pointer location. The first zero logic finds the first non-allocated slot in the allocate bits and sets the corresponding bit and resets all other bits. The output of the first zero logic is ORed with the allocate bits and fed to another first zero logic, and the process repeats until the four new pseudo-registers are found. The prioritizer also

provides the updated allocate bits after the new destinations are found. The first zero logic is as shown below.

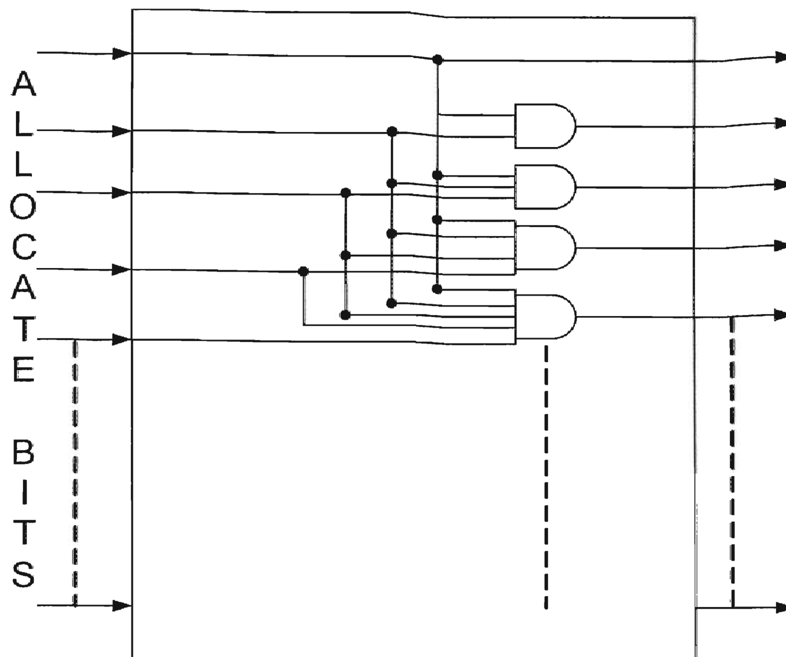


Figure 4.7: First zero logic

Each output bit of the first zero logic is the logical AND of all the allocate bits less than the output bit. This is repeated for all the 64-bits. The output bit corresponding to the first non-allocated location indicated by a logical '0', is set and all the others are reset. As the bit number of the output increases, the input to that particular AND gate increases e.g. for the fifth bit in the output a five input AND gate is required. As the gates with the larger fan-in are slower, AND gates greater than four inputs are implemented as a tree of small AND gates. This reduces the worst-case delay. The worst-case delay of the prioritizer is less than 10 nsec. making it feasible for this processor with a clock rate of 10 nsec.

The Prioritizer is an essential part in the intended architecture, as the pseudo-register architecture cannot be realized without renaming the incoming instructions logical destinations. This design is not extendable as the prioritizer is essentially a sequential circuit and the number of stages in the module increases with the increase in the number of new pseudo-pointers required. As the number of stages is increased, the worst-case delay of the circuit also increases making it incompatible with other design with fetch blocks greater than four instructions. The first zero logic slows down as the empty locations are near to the end of the register file. In this design the worst-case delay is less than the clock period making it feasible.

This chapter outlined the basic instruction types in the MIPS I instruction set, and it also pointed out different control signals generated by the instruction set architecture decoder and outlines their use in the correct execution of the instructions. The mechanism of renaming logical destinations of new instructions was discussed in detail, and the limitations on the prioritizer were pointed out. These limitations make this kind of design inextensible for a design with larger instruction block size.

Chapter 5

Register file design

5.0 Introduction

Speculative prediction of branch or jump instructions in a super-scalar processor helps in optimum utilization of hardware and making the processor more efficient, since the processor need not wait for the evaluation of the branch or jump condition before fetching a new instruction block. In the normal processor operation there can be, at any given time, both speculated and un-speculated instructions executing simultaneously. Unless these branch or jump instructions complete execution, the speculated instructions should not be allowed to alter the machine state. To avoid this, results generated by speculated instructions are stored temporarily in registers other then the architectural registers until the branch or jump condition is evaluated. Hence, to complete a speculated instruction, two 32-bit data transfers are required. This is inefficient, as these data transfers require 32-bit data buses. For a super-scalar processor this inefficiency is more prominent as the processor requires multiple 32-bit buses to avoid structural stalls caused by more than one instruction completing simultaneously. The proposed scheme in this study reduces this inefficiency by having additional data storage locations other than the architectural registers called pseudo-registers, and maintaining two translations tables of pointers called the Issue Pointer Buffer and the Commit Pointer Buffer that translate a logical register number to the pseudo-register number. Logical destination of each

incoming instruction is renamed to a pseudo-register. This list of pseudo-pointers is maintained in the Issue Pointer Buffer. When an instruction finishes execution, the results are written in the pseudo-register and the instruction is completed or committed by copying the pseudo-register pointer to the Commit Pointer Buffer. Whenever a misprediction is detected, the Commit pointer buffer overwrites the Issue pointer buffer bringing the processor state back to where it was before the branch was predicted as shown in the block diagram.

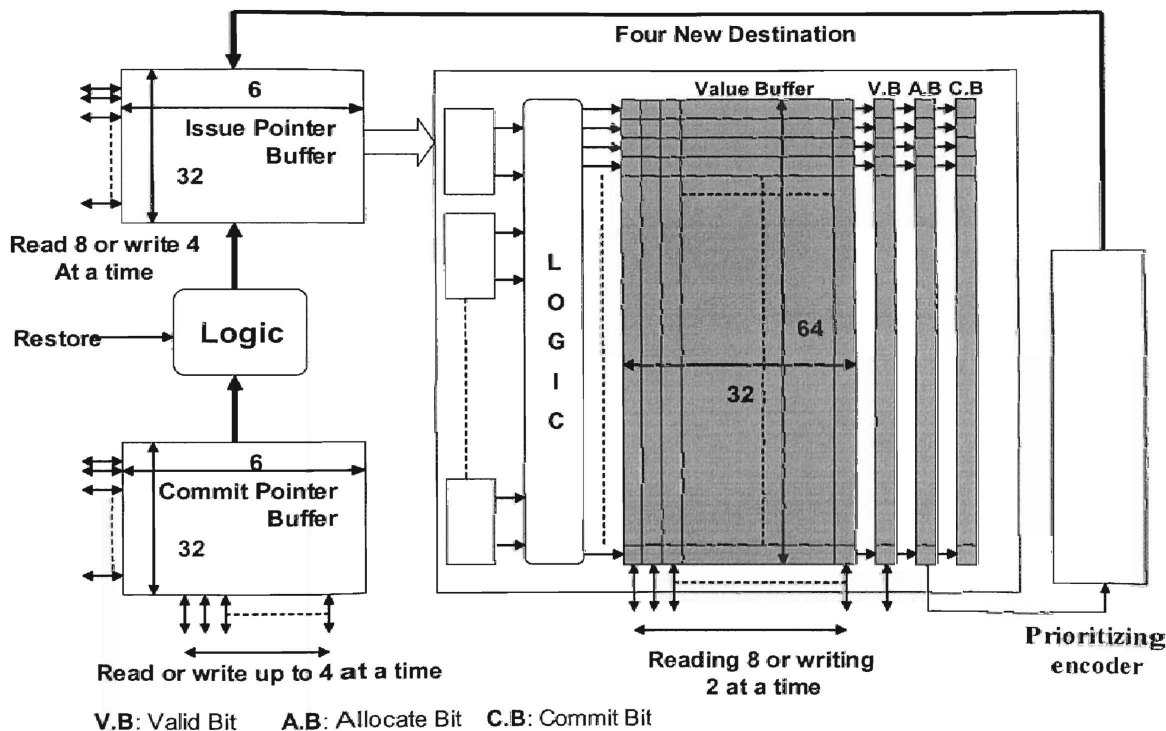


Figure 5.1: Block Diagram of the Register-file.

There are 32 architectural registers or logical registers as defined by the MIPS I instruction set, and there are an additional 32 pseudo-registers. These two sets of registers are implemented as one register file called the Value Buffer. Each value buffer

location is associated with three status bits to identify the locations that has been renamed or allocated to the incoming destination register. They also identify which of the data is committed and which of the location has correct data or valid data. As there are a total of 64 locations in the value buffer, the pseudo-register pointers are 6-bit. Hence, to commit a speculated instruction, a 6-bit data transfer is required as compared to 32-bits in the earlier case, which is more efficient.

This chapter discusses the timing and implementation of the two pointer translation buffers and the value buffer, outlining different design criterions.

5.1 Issue pointer buffer/Commit pointer buffer design

The Issue pointer buffer and the Commit pointer buffer are implemented as a single memory unit having two storage locations at every address as shown in Figure 5.2. Since each instruction block has four instructions and each instruction requires two source operands, a total of eight renamed pseudo-pointers have to be read from the issue pointer buffer concurrently every clock cycle; hence, eight read ports are required on the issue pointer buffer. Moreover, four new pseudo-pointers corresponding to the logical destinations of the four new instructions are to be written into the issue pointer every clock cycle. This requires four write ports on the issue pointer buffer. Since the reading from and writing into the issue pointer buffer are done at different times, we require a total of eight ports, out of which four are read-write as shown in Figure 5.2.

Since the processor tries to commit up to four instructions every clock cycle, the commit pointer buffer requires four write ports. The locations that are being committed are read in the first half cycle for de-committing; hence, four read ports are also required on the commit pointer buffer. As the reading and writing of the commit pointer buffer is done in two different half cycles, a total of four read-write ports are sufficient as shown below.

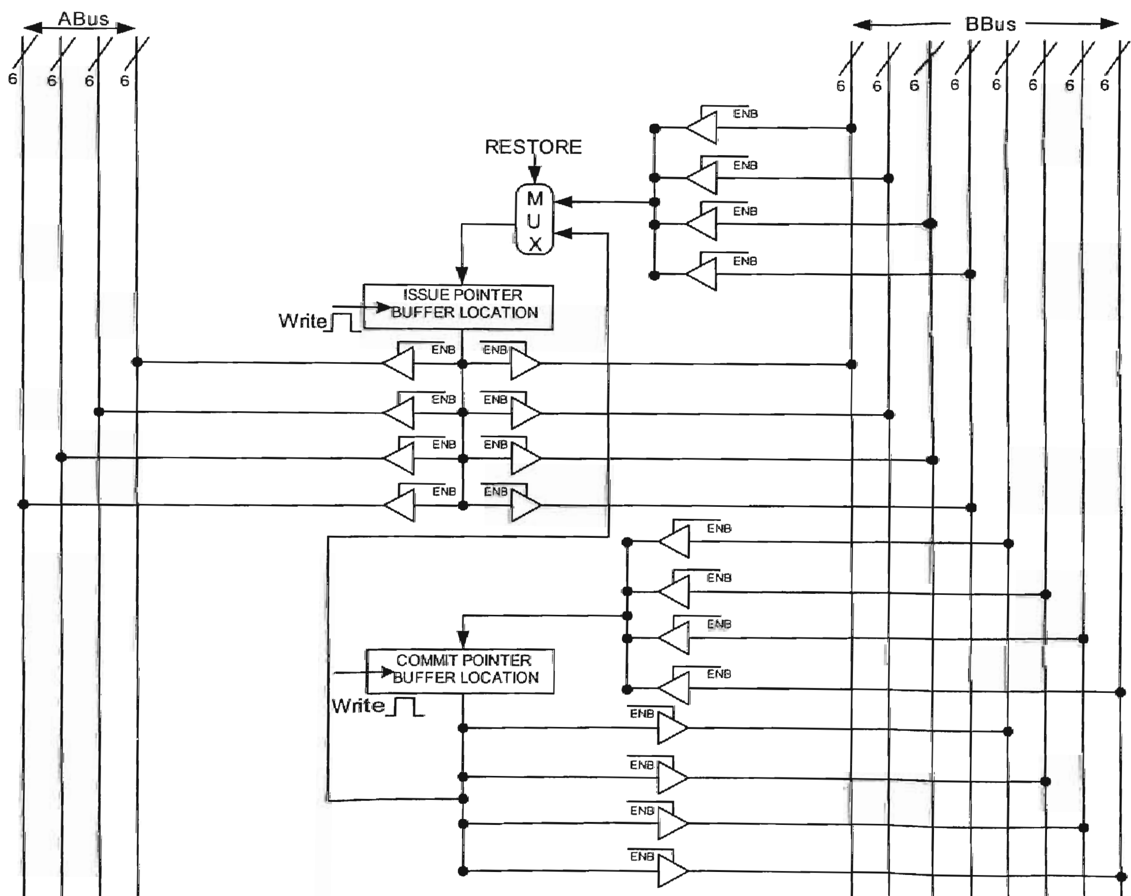


Figure 5.2: Issue Pointer Buffer/Commit Pointer Buffer Implementation.

Figure 5.2 describes the implementation of one memory cell of the issue pointer/commit pointer buffer. The number of columns and rows in these buffers depends

on the width of the pseudo-pointers and logical registers respectively. The MIPS instruction set describes the width of the logical registers as 5-bits, and since there are 64 registers in the processor, the width of the pseudo-pointers are 6-bits. Hence, the number of rows in these buffers are 32 (2^5) and each row is 6-bits wide as described in Figure 5.1. The control signals to the transmission gates are not shown in the figure. These signals are generated using a set of decoders that decode the 5-bit logical register number to a 32-bit control signal. The first row of the issue pointer buffer and the commit pointer buffer is hard wired to logic zero. This is done in accordance to the MIPS instruction set architecture which describes the register '$zero' which always returns logic zero. Two avoid any errors in reading the pseudo-pointers from the logical source registers for the first few initial instructions in the program, all the locations in the issue pointer buffer are forced to logic zero when there is a system reset.

The timing on the output buses of the Issue Pointer Buffer/Commit Pointer Buffer is as shown in Figure 5.3. The issue pointer buffer reads source pseudo-pointers first, before the new destination pseudo-pointers are written. Since the instruction set architecture decoder works in the first half of the instruction decode/issue pipeline stage, the issue pointer buffer is read in the second half of this stage. New destination pointers are written in the first half of the next stage as shown in the timing diagram. This processor tries to commit up to four instructions every clock cycle. The destination pseudo-pointers of the committing instructions overwrite locations in the commit pointer buffer selected by the logical destination register number. The overwritten locations have to be de-committed. Hence, the commit pointer buffer is read in the first half of the

instruction commit II stage and the committed pointers are written in the second half of this stage as show in the timing diagram below.

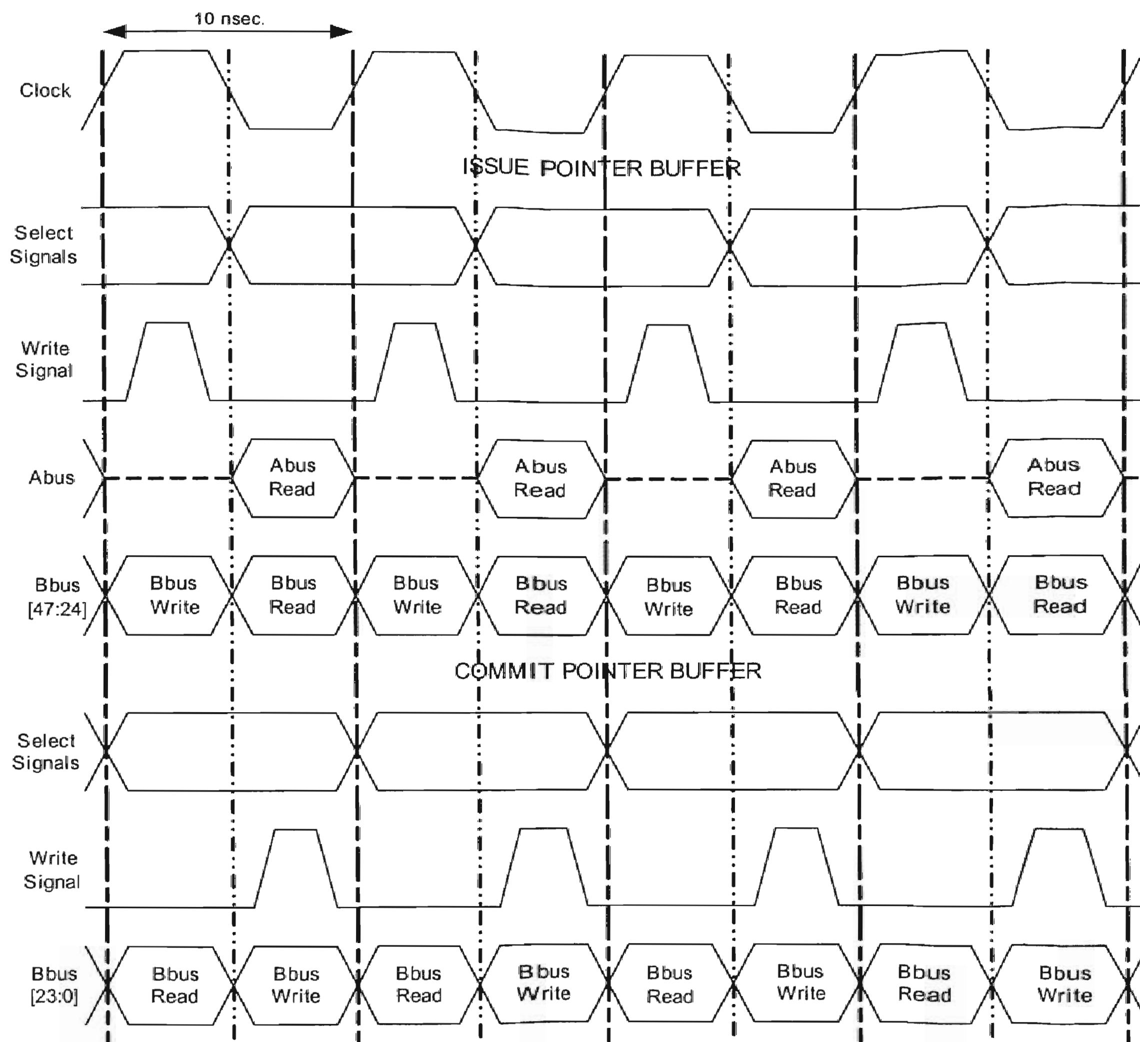


Figure 5.3: Issue Pointer Buffer/Commit Pointer Buffer Timing

5.2 Value Buffer Design

The source pseudo-pointer read from the issue pointer buffer is decoded to read the 32-bit data from the value buffer. The value buffer is a single register file that has both the architectural registers and the additional pseudo-registers. Implementations of the value buffer can be understood from the following figure, which shows the implementation of a single memory cell of the value buffer. There are a total of 64 locations in the value buffer, each 32-bit wide. The first location always returns logic zero representing register '$zero' of the MIPS instruction set architecture.

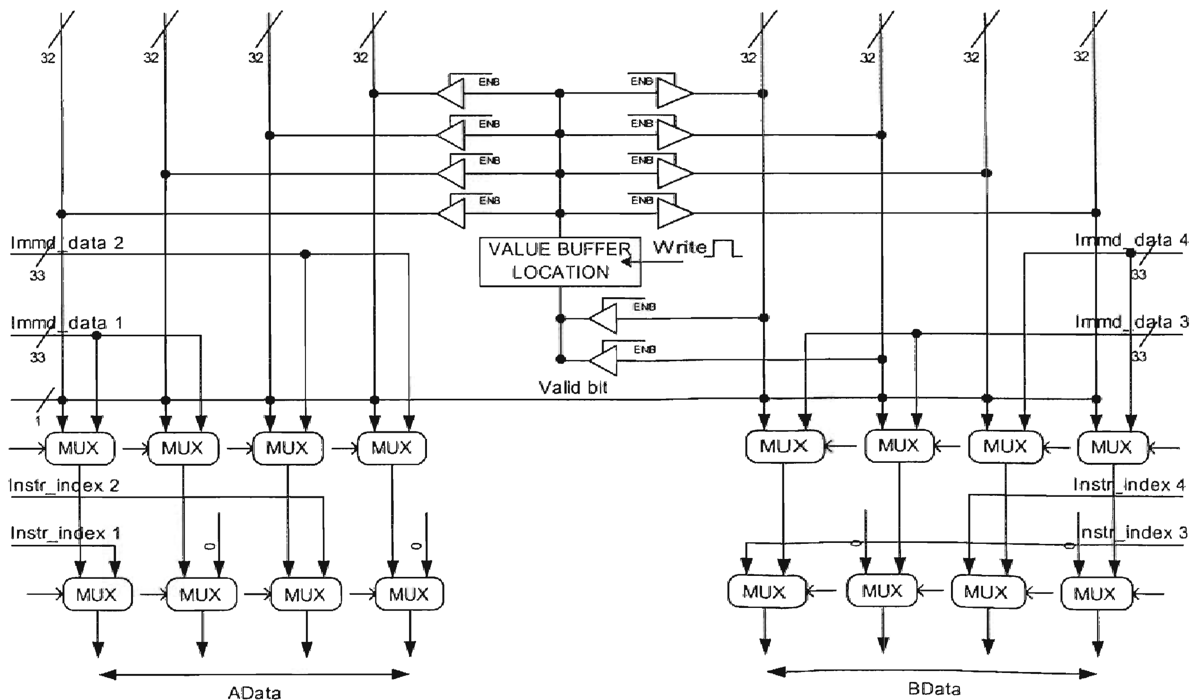


Figure 5.4: Value Buffer Implementation

6-bit pseudo-pointers for each of the sources for the four instructions are decoded in the first half of the instruction dispatch pipeline stage, and the value buffer is read in

the second half of this stage along with the corresponding valid bit as shown. At the end of the instruction execution stage, the results are written into the value buffer, the results are written in the first half of the instruction write-back stage as the destination pseudo-pointers are pre-decoded in the last stage of execution. There are a total of two write back buses called the common data bus I and common data bus II respectively. The common data buses are prioritized using a write-back controller to allow only two concurrent write-backs.

For immediate mode instructions, one of the sources is the immediate data that has to over-ride the data read from the value buffer. Thee first level multiplexers are used for the same. The control signals for these multiplexers are generated by instruction set decoders. As the immediate data supplied in the instruction is 16-bits it is padded with leading logic zeros to make it 32-bits and a 1-bit logic one is attached to it, indicating the data is valid. The second level of multiplexing is used for two special instructions J and JAL. These instructions supply the instruction index which is 26-bits. This instruction index is padded with leading zeros as above and a valid bit is also attached. The control signals for these second level multiplexers are generated by the instruction set architecture decoder.

The timing on the data buses of the value buffer can be seen from the timing diagram shown in Figure 5.5. Whenever an instruction completes execution, the destination pseudo-pointers are pre-decoded as shown in the timing diagram as the 'write

select signal'. The value buffer is read in the second half as the decoder works in the first half as shown.

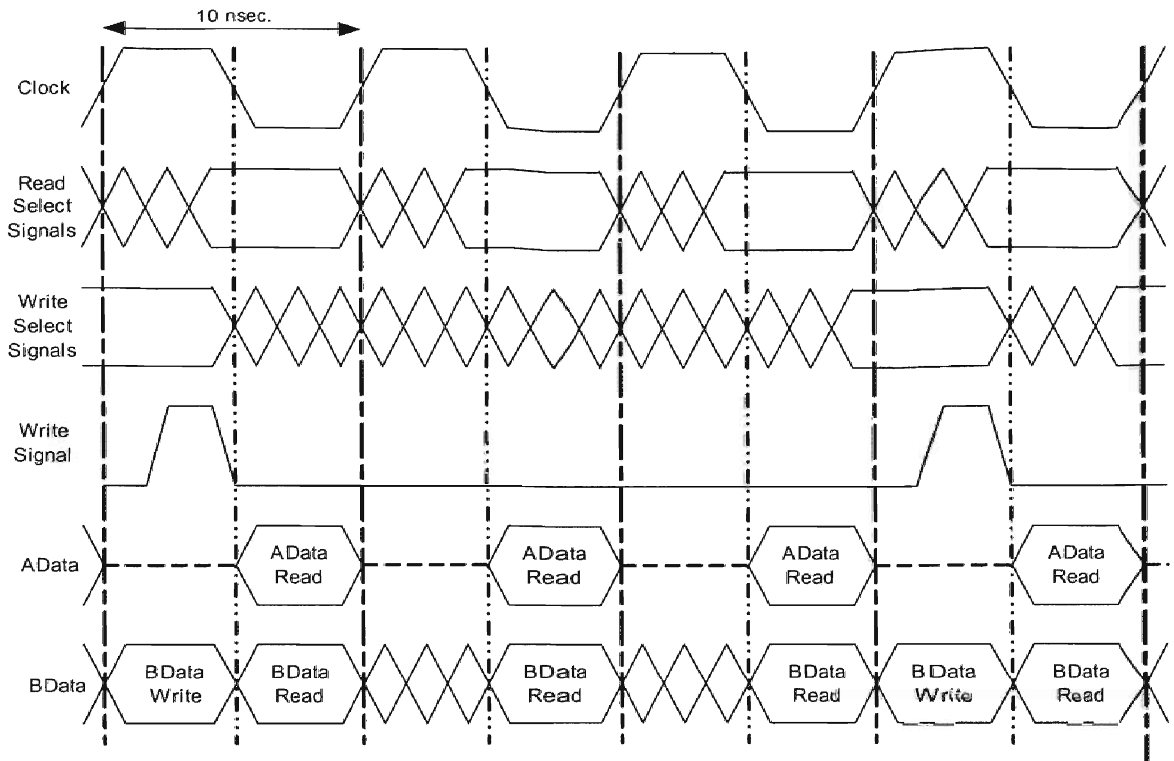


Figure 5.5: Value Buffer timing

Design of the register file that includes the Issue pointer/Commit Pointer buffer and the Value Buffer was discussed in this chapter. The timing on each of the modules was explained. As the issue pointer is read before it is written, destination and source overwrites are required in some cases. These issues will be dealt at length in the next chapter. The next chapter will also talk about the implementation of the execution units.

Chapter 6

Maintaining Data Consistency & Instruction Execution

6.0 Introduction

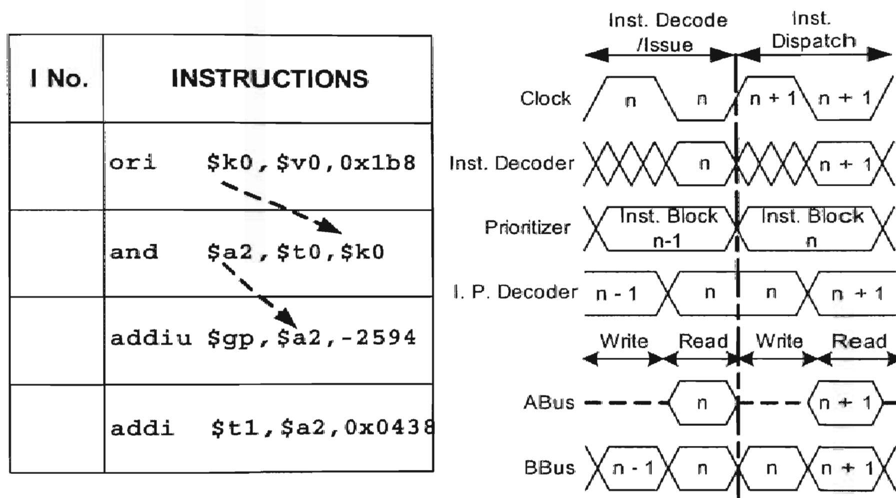
The issue pointer buffer is read before it is written for the same instruction block. This may cause inconsistent reading of the source pseudo-pointer if the logical source register of an instruction is the same as the logical destination register of another instruction in the same instruction fetch block. The new destination pseudo-pointers picked for these instructions are not written into the issue pointer buffer at the time the source pseudo-pointers are read. This inconsistency can cause incorrect data to be fetched from the value buffer causing an erroneous operation. This inconsistency is avoided in this processor by overwriting the sources having the same logical register number as the destination of another instruction by the new destination pseudo-pointer picked by the prioritizer. This kind of overwrite is also required for the new destination pseudo-pointers picked by the prioritizer before they are written in the issue pointer buffer if the logical destination of two or more new instructions are the same. This is done to have the latest pseudo-pointer for the logical destination in the issue pointer buffer.

Instructions sitting in the reservation station are issued to the functional unit for execution when both the source operands are valid and there are no structural stalls. As the basic focus of this processor was to verify the concept of pseudo-registers, these

execution units are implemented behaviorally instead of a detailed implementation at the gate level.

This chapter talks about the implementation of the destination and source overwrite logic. It also talks about two of the five functional units in the processor namely the branch and jump unit and arithmetic and logic unit.

6.1 Source Overwrite Logic



I No.	INSTRUCTIONS
	ori $k0,$v0,0x1b8
	and $a2,$t0,$k0
	addiu $gp,$a2,-2594
	addi $t1,$a2,0x0438

Figure 6.1: Sample Source code and Issue Pointer Buffer timing.

The data inconsistency problem can be understood from the diagram above. As can be seen for the sample code, there are data dependencies between instructions 1 & 2 and between instructions 2 & 3 indicated by the dotted lines. The accompanying timing diagram shows the timing of the prioritizer, the issue pointer buffer bus, the instruction decoder and the issue pointer buffer decoder in the instruction decode/issue stage and

instruction dispatch stage indicated by clock cycles 'n' and 'n + 1' respectively. It can be seen that the new destination pseudo-pointers picked by the prioritizer for the instructions decoded in the nth clock cycle are written into the issue pointer buffer in the (n + 1) clock cycle, and the source pseudo-pointer for these instructions are read in the nth clock cycle. Hence, the issue pointer buffer is read before it is updated causing the inconsistency. This inconsistency is avoided by overwriting the source pseudo-pointers of the instructions having the same logical register number as the logical destination registers of another instruction. The source pseudo-pointers are overwritten with the new destination pseudo-pointer picked for the other instruction before reading the value buffer.

Source overwrite logic is implemented as a group of comparators and multiplexers that compare each logical source register number of each instruction to the logical destination number of all the other instructions. If the comparison identifies that they are equal the new destination pseudo-pointer for that particular logical destination overwrites the source pseudo-pointer as shown in the figure in the next page. RS, RT represents logical source registers and RD represents the logical destination register number. The new destination pseudo-pointers picked by the prioritizer are represented by 'New Dest.'. Sources read from the issue pointer buffers are indicated by RS_IPB and RT_IPB. Overwritten sources are represented by OWRS and OWRT.

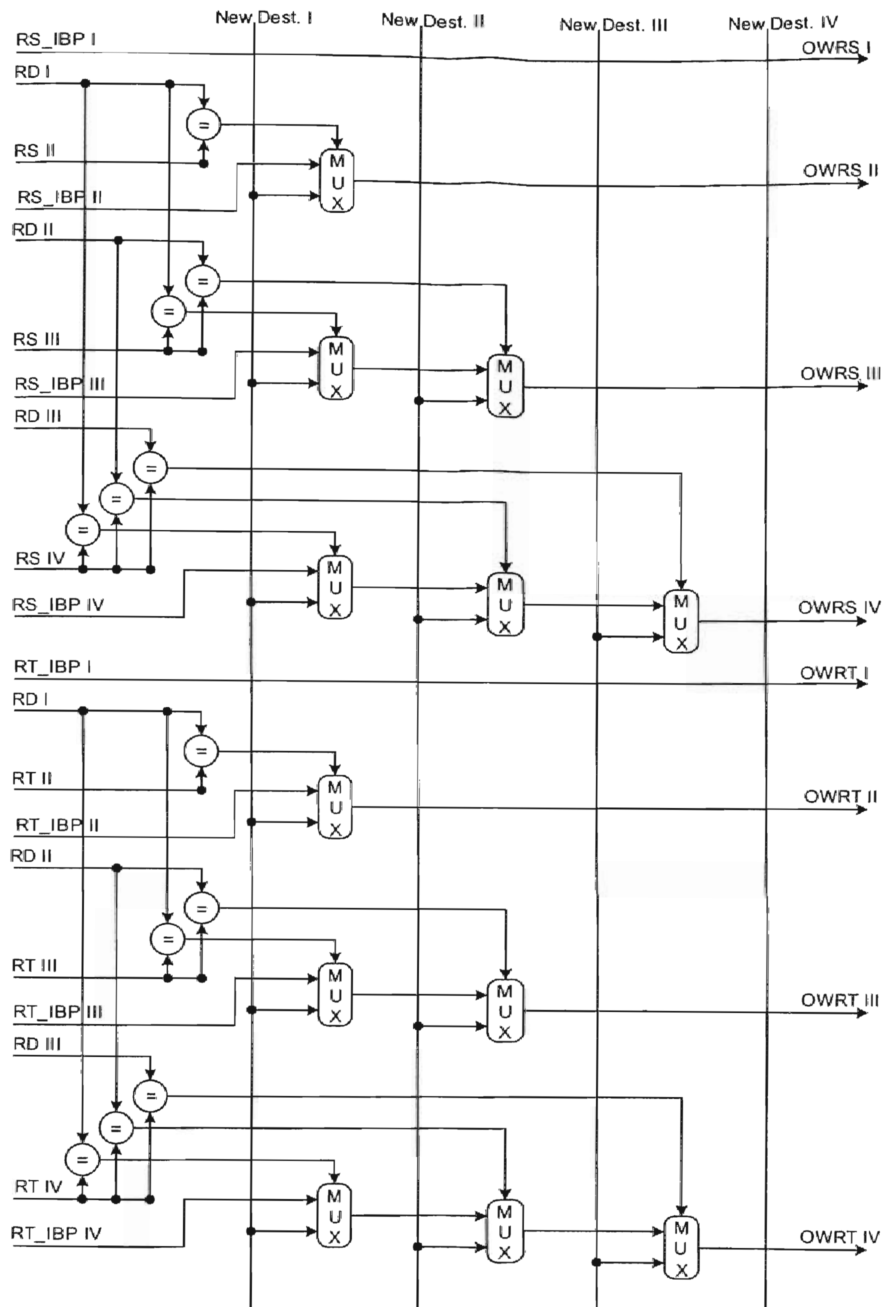
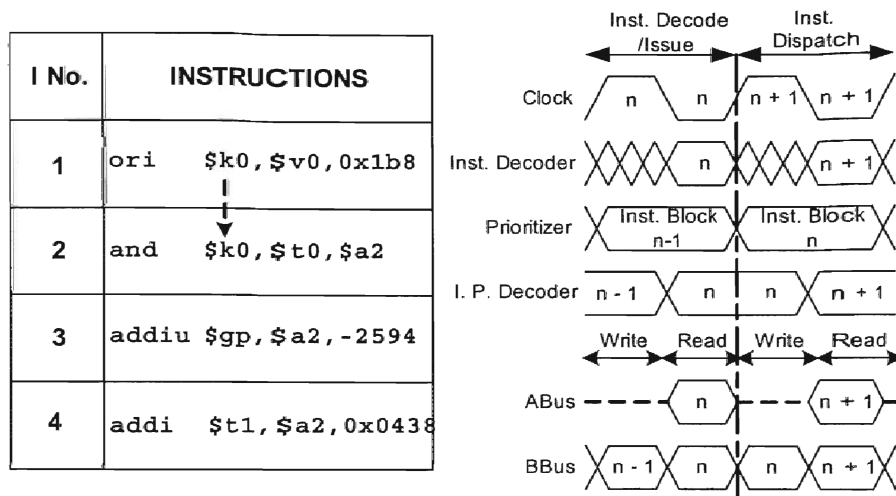


Figure 6.2: Source overwrite logic Implementation.

Source overwrite logic makes sure that the latest pointers are always assigned to the source registers of the instructions, removing the inconsistency arising from the data dependency.

6.2 Destination Overwrite Logic

Another kind of data inconsistency may arise from the situation shown in Figure 6.3 below. This inconsistency arises when, in the same instruction block, there is more than one instruction having the same logical destination registers, as shown.



I No.	INSTRUCTIONS
1	ori $k0,$v0,0x1b8
2	and $k0,$t0,$a2
3	addiu $gp,$a2,-2594
4	addi $t1,$a2,0x0438

Figure 6.3: Sample Source code and Issue Pointer Buffer timing.

It can be seen from the code segment above instruction 1 and 2 have the same logical destination. As the prioritizer picks different destination pseudo-pointers for each instruction in an instruction block, there are two different pseudo-pointers for the same logical register number, '$k0' in this case. This causes ambiguity for the instructions fetched in the future referring to '$k0', as there are more than one pseudo-pointers assigned to it. The correct pseudo-pointer for the register '$k0' should be the pointer picked for instruction 2 or the instruction lower in the block.

This ambiguity is removed in this design by doing a destination overwrite before the new destination pseudo-pointer picked by the prioritizer is written in the issue pointer buffer. The overwrite logic identifies the instructions having the same logical destination and overwriting it with the appropriate new destination pseudo-pointer as shown in the circuit diagram below.

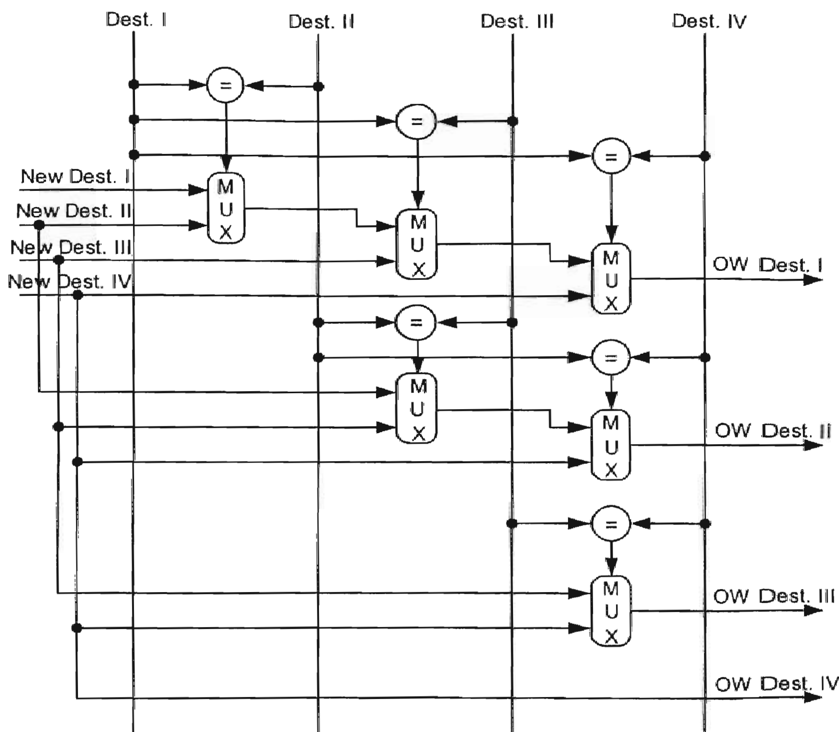


Figure 6.4: Destination overwrite logic Implementation.

As seen from the diagram, the destination overwrite logic compares and identifies the instruction with same logical destination register, and over writes it with the new pseudo-pointer of the instruction lower in the instruction block. Hence the logical destination register of instruction 1 is compared with all the other three instructions, and the logical destination register of instruction 2 is compared with instruction 3 & 4, and so

on for instructions 3 & 4. In the Figure, 'New Dest' represents new destination pseudo-pointers picked by the prioritizer, 'Dest' represent the logical destination registers and 'OW Dest' represents the overwritten destination pseudo-pointers.

The destination overwrite logic work before the new destination pseudo-pointers are written into the issue pointer buffers thereby maintaining an updated list of pseudo-pointers in the issue pointer buffer and removing ambiguity for forthcoming instructions.

6.3 Instruction Execution

Decoded and renamed instructions are executed in this pipeline stage. There are six functional units in this processor for executing different kinds of instructions. As the compiler used in the verification of the design is incapable of handling interrupts and exceptions, the Coprocessor 0 functional unit is implemented as static logic always returning zeros. Each functional unit except the Coprocessor 0 is accompanied with a reservations station to dynamically schedule the instruction. The block diagram of the functional units are shown in Figure 6.5, Coprocessor 0 is not shown in the diagram. The Load Store Unit and the Multiply Unit consist of two and three pipeline stages respectively, as shown in the diagram. There are a total of two arithmetic logic units in the processor since there are more arithmetic and logic instructions as compared to any other types of instructions. The Branch and Jump unit has to write the correct branch target address into the reorder buffer. Hence, apart for the two common data buses, there is an additional bus to do the same called the branch target bus in the block diagram. The

branch target bus is a dedicated bus from the Branch and Jump unit to the reorder buffer. Functional units in this design are implemented at a behavior level since the focus of this study is studying the efficiency of the pseudo-register scheme.

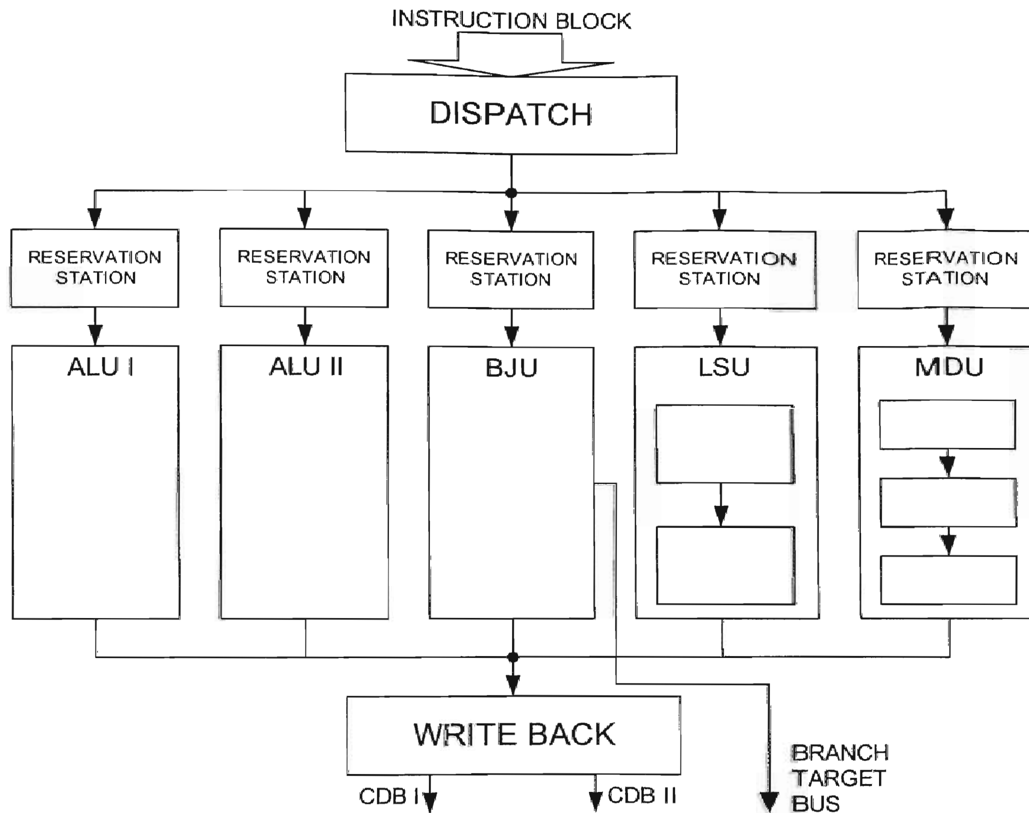


Figure 6.5: Functional units block diagram.

The block diagram of the arithmetic and logic unit is shown in Figure 6.6. It can be seen that the output of the arithmetic and logic unit called the 'alu_result' is 109-bits. Encoding of these results is depicted in Figure 6.7. This result is written on the common data buses in the write back stage.

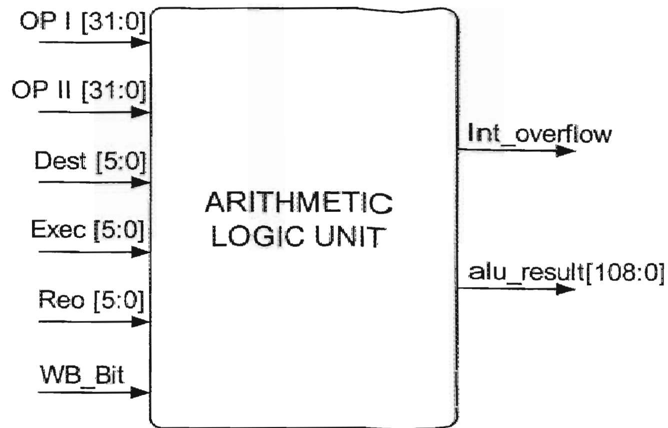


Figure 6.6: Block diagram of Arithmetic and logic unit.

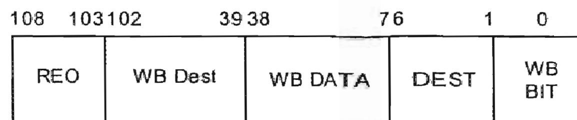


Figure 6.7: ALU result encoding

As seen from Figure 6.7, the first 6-bits of the encoded results represent the reorder buffer location address where the result has to be written to achieve in-order completion. The next 64-bits represent the pre-decoded value buffer location where the results are written after the instruction commits. The data produced by the executions of the instruction is in the next 32-bits containing the arithmetic and logic unit results. The destination pseudo-pointer and a control bit for assigning the common data buses are in the lower 7-bits of the result.

Branch and jump unit's block diagram is shown in Figure 6.8. Since the branch and jump unit writes the correct branch target address into the reorder buffer, there is an additional dedicated bus to achieve the same. Hence the output of the branch and jump unit is 'wsb'. This is the pre-decoded address of the reorder buffer where the correct

branch target address is written. The 'brbus' output is the dedicated bus to write the same, 'bju_result' ouptu is used to write the result to the common data buses for the link type branch and jump instructions. The encoding of the 'brbus' is shown in Figure 6.9. The branch and jump unit result encoding is similar to the arithmetic and logic unit's result encoding.

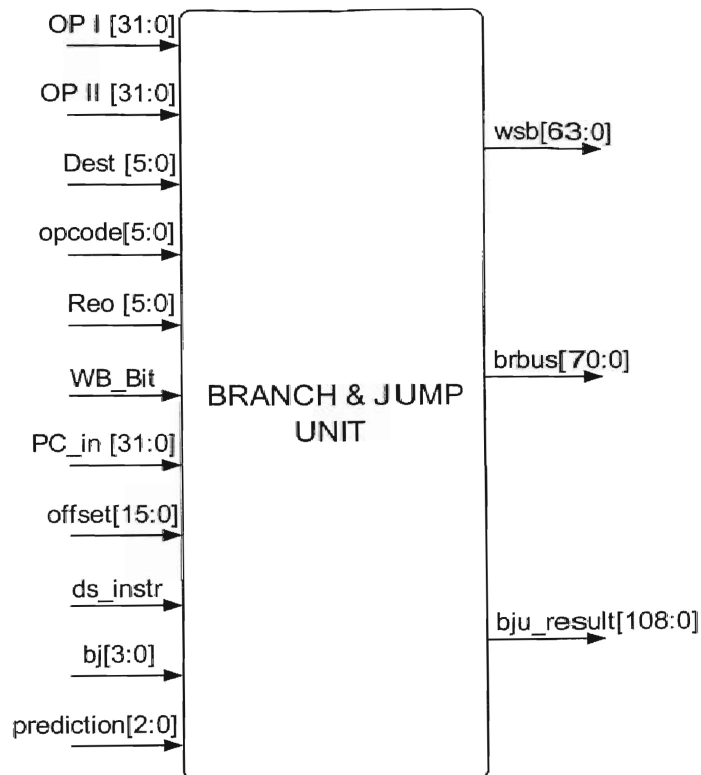


Figure 6.8: Block diagram of Branch and jump unit

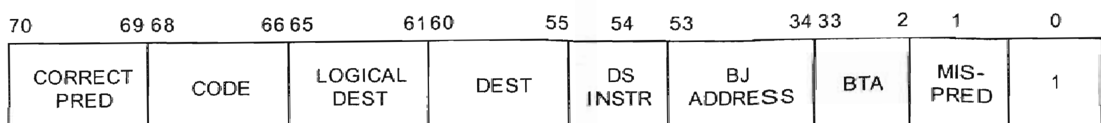


Figure 6.9: Branch bus encoding

The fields of the branch bus can be described as follows:

1. CORRECT PRED: This describes the correct branch action i.e. *taken* or *not taken* after the branch has finished execution.
2. CODE: Code identifies the link type branch or jump instruction. This code is used in the commit stage to commit the instruction.
3. LOGICAL DEST: If the instruction is a link type instruction, these 5-bits contain the logical link address for that instruction, which is R31 as described in the MIPS I instruction set.
4. DEST: This is the pseudo-pointer picked by the prioritizer for the above logical link address.
5. DS INSTR: This field identifies a branch or jump instruction having a delay slot. If the branch instruction has a delay slot, the program counter is not changed unless the delay slot instruction is executed and committed.
6. BJ ADDRESS: These are the lower twenty bits of the branch or jump address. The branch target buffer and the branch prediction buffer are written after decoding the lower bits of the branch or jump instruction address.
7. BTA: This is branch target address computed by the branch and jump functional unit.
8. MIS-PRED: This field identifies if a particular branch or jump instruction is mispredicted or not. When a misprediction is encountered at the top of the reorder buffer queue, system restore is initiated.
9. Last bit is the static bit tied to logic '1' representing the validity of the location written into.

This chapter described some typical data inconsistencies that accompany this type of processor design and also explained how these data inconsistencies are avoided using destination and source overwrite logic. The chapter also gave an over view of the design of two functional units namely the arithmetic and logic units and the branch and jump units. The following chapter will discuss the verification methodology and will discuss results obtained from the simulation.

Chapter 7

Design Verification Methodology & Simulation

7.0 Introduction

The processor design discussed thus far is implemented using Verilog as the hardware description language (HDL). A modular approach was followed to increase the ease of programming and to make the code easier to understand. Modules except functional units and Fetch were done at logic gate-level using the Cadence™ ambit synthesis tool. The design under consideration was tested using the Verilog simulator with instruction and data memory maps of sample 'C' programs having known results. These sample programs were compiled using the SDE-Gcc C compiler since the operating system was not simulated on the processor. The SDE-Gcc C complier is a part of the SDE-MIPS, a cross-development toolkit for MIPS CPU's maintained by Algorithmics, which is freely redistributed. Sample 'C' programs instead of standard testbenches such as the SPEC 95, 99 or 2000 were used for testing due the incapability of this version of the SDE-Gcc C compiler in handling interrupts and exceptions. Doing *objectdump* of the compiler output generated the instruction and data memory map of the sample programs. Since Verilog HDL is capable of reading tab delimited binary files only, a translation program was utilized to translate hexadecimal outputs to binary outputs, which could be understood by Verilog HDL. The instruction and data cache was programmed as a Verilog array that could hold consecutive data and was assumed of

infinite size eliminating cache misses. Finally the intended output was compared with the output generated after the program ran on the processor, validating correct operation.

The testing methodology employed to verify this design is discussed in detail in this chapter. This includes a detailed discussion of the memory model used in the processor and finally the simulation results are presented verifying the processor design.

7.1 Memory Model

Data and instructions cache in this processor are simulated using Verilog HDL arrays. A pair of arrays represents memory in which one if the arrays contain memory addresses and the other contains the corresponding data as shown in the Figure below. Both the arrays are in one is to one correspondence. The memory mode assumes infinite memory size for both the data and instruction cache.

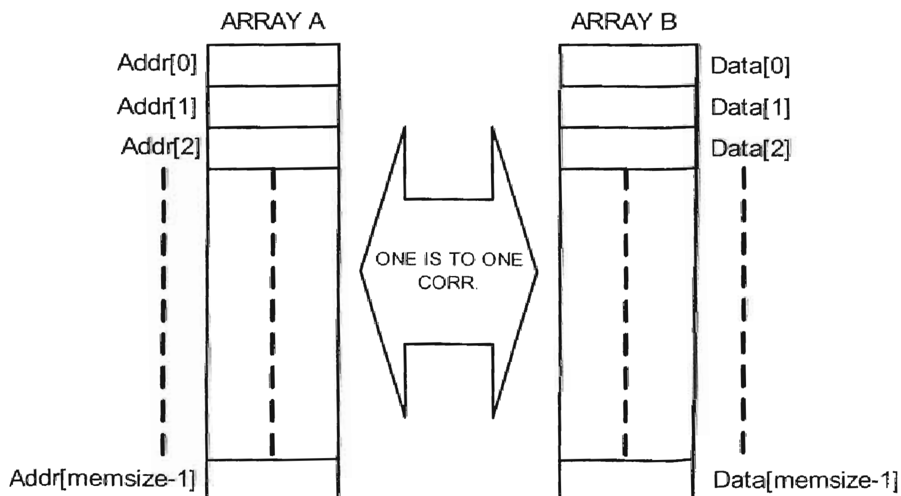


Figure 7.1: Address-Data memory array pair

7.2 Design Verification Methodology

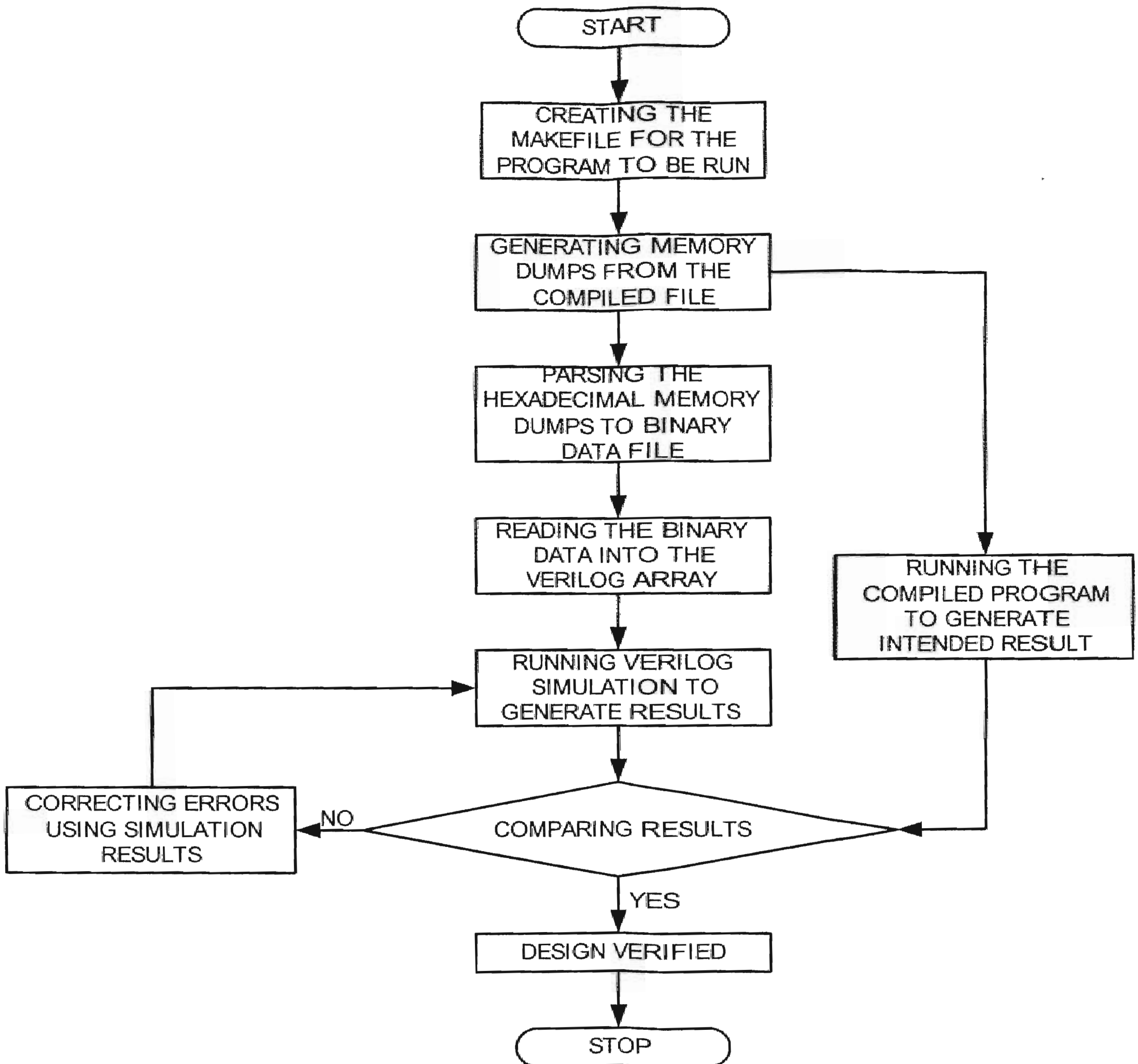


Figure 7.2: Flowchart of design verification methodology

As described above the sample 'C' programs used to test the design were compiled using the SDE-Gcc C Compiler tuned for the MISP I Instruction set. Before compiling a program, a Makefile for the program has to be written that describes

parameters for the compiler to follow like the optimization level and the type of instructions to be included. An example Makefile is shown below.

PROG = prog1

OBJS = prog1.o

CFLAGS = -O3

FLOAT = no

include ../make.mk

The first line of the Makefile, '*PROG = prog1*', declares the name of the 'C' program to be compiled. The second line, '*OBJS = prog1.o*', lists the names the object files that make up the program. Third line, '*CFLAGS = -O3*', declares optimization options for the compiler. Different optimization options for SDE-Gcc C compiler are as follows:

1. O, - O1: This is the lowest level of optimization. A program should be compiled with this level of optimization at least unless debugging the code.
2. - O2: This is the most efficient level of optimization offered by the compiler.
3. - O3: This level optimizes loops by unrolling them. Loop unrolling can cause the code size to increase which may slow the program down.
4. - O4: This is the highest level of optimization possible. This level is seldom used as it blows up the code size significantly.

The forth line '*FLOAT = no*' of the Makefile above identifies whether to include floating point instruction or not. To include the floating-point instruction in the compilers output, this line is modified as '*FLOAT = yes*'. The last line of the Makefile is used to include the information regarding the header libraries.

Once the Makefile is written, the program can be compiled, using the command '*sde-make SBD=GSIM1B*' to generate a ram-file. This file contains the entire compiled program. As the processor under test is compatible with the MIPS I instruction set, the compiler output is restricted to MIPS I instruction set using the command '*SBD=GSIM1B*'. Using the following commands at the command line, '*objdump –D <ram_file name>*' and '*objdump –S <ram_file name>*', can generate instruction dump and data dump respectively. The instruction dump is in a standard format called the Executable and Linkable (ELF) format. Details about the format can be found in the ELF documentation [17].

The Verilog HDL can populate the arrays used as the memory model only with binary data read from a file. Hence, the instruction and data dumps are parsed to generate a tab delimited binary equivalent file for each. The parser takes these dump files as input, and generates two binary data files, one having the memory address and the other having the corresponding data, in accordance with the memory model discussed above.

Once the binary data files are generated, the processor design is ready to be simulated. A Verilog simulation is run on the processor and the results are compared with

the intended result of the program, verifying correct operation. Verilog simulations can be run at two levels of abstraction, at the behavioral level and thee structural level. Structural simulations were run to remove timing violations in the design. As Behavioral simulation is more efficient in simulation time, results are generated using behavioral simulation.

7.3 Simulation

To verify the processor design, a sample 'C' program to calculate the factorial of an integer was run following the methodology discusses above. The main program calls a subroutine called 'calculate' to calculate the factorial of an integer. The returned value is stored as an array element. Optimization level '-O3' was used to take advantage of loop unrolling. Compiler output was used to generate instruction and data memory dumps for the program as explained in the section above. These dumps were loaded in the processor instruction and data caches for execution. The stack part of the data memory before and after the program execution is shown below.

```
Addr :801fffbc    Data: 00000000000000000000000000000000
Addr :801fffc0    Data: 00000000000000000000000000000100
Addr :801fffc4    Data: 00000000000000000000000000000000
Addr :801fffc8    Data: 10000000000000000000010000010000
Addr :801fffcc    Data: 00000000000000000000000000000000
```

Figure 7.3: Stack Memory dump before simulation

```
Addr :801fffbc    Data: 00000000000000000000000000000000
Addr :801fffc0    Data: 00000000000000000000000000000100
Addr :801fffc4    Data: 00000000000000000000000000011000
Addr :801fffc8    Data: 10000000000000000000010000010000
Addr :801fffcc    Data: 00000000000000000000000000000000
```

Figure 7.4: Stack Memory dump after simulation

The program was executed to calculate factorial of 4, which was loaded in the stack memory initially as shown in Figure 7.3. The result was loaded in the consecutive location in the stack after calculation as shown in the Figure 7.4.

7.4 Results

Processor performance was evaluated by measuring the following parameters:

1. Instructions per cycle (IPC).
2. Misprediction Rate.
3. Dispatch Stalls.
4. Percentage utilization of Branch prediction buffer.
5. Percentage utilization of Branch target buffer.

These performance parameters are discussed in detail, in this section.

7.4.0 Instruction per cycle (IPC)

Instruction per cycle (IPC) is the measure of the number of instructions processed on an average for execution every cycle. This can be calculated as follows:

$$IPC = \frac{\sum no\ of\ instruction\ comitted/clock\ cycle}{Total\ number\ of\ clock\ cycle} \tag{7.1}$$

IPC is highly program dependent. For a linear pipeline, it is ideally one, but because of data dependencies it is usually much less. In a multi-linear pipeline, apart from the inherent data dependencies, IPC also depends on the compiler's capability to find four independent instructions that could be executed in parallel. As this processor

fetches instructions speculatively, fetched instructions are not necessarily correct instructions. Hence the data is collected at the output of the reorder buffer.

7.4.1 Misprediction Rate

Misprediction rate is a measure of the efficiency of the prediction mechanism. This can be calculated as follows:

$$Misprediction\ Rate = \frac{Number\ of\ branches\ mispredicted}{Total\ number\ of\ branches} \tag{7.2}$$

In this processor branch or jump instruction misprediction is detected at the top of the reorder buffer queue. System restore is initiated when a mispredicted branch is detected. Hence, calculating the number of restores is same as the number of mispredicted branches.

7.4.2 Dispatch Stalls

Dispatch stalls are the pipeline stalls when the processor is incapable of dispatching instructions because of structural hazards. Dispatch stalls in this design are higher as the new instruction block is not fetched until all the instructions in the previous block are dispatched.

7.4.3 Percentage utilization of BPB and BTB.

This is the measure of the number of locations utilized in the branch prediction buffer and branch target buffer at the end of program execution. This is calculated as follows:

$$\%\ utilization\ of\ BPB\ or\ BTB = \frac{Number\ of\ valid\ locations\ in\ BPB\ or\ BTB}{Total\ number\ of\ locations\ in\ BPB\ or\ BTB} \quad (7.3)$$

Only *taken* branch or jump instruction's target addresses are written into the branch target buffer, whereas the branch prediction buffer is updated for every branch whether *taken* or *not taken*. Hence, the percentage utilization of the branch prediction buffer is higher than the branch target buffer.

The following table summarizes the data collected for the simulation run describe above.

Total number of clock cycles	98
Instruction Per Cycle (IPC)	0.418
Misprediction Rate	41.6 %
Dispatch Stall cycles	33
Percentage utilization of Branch Prediction Buffer	14.06 %
Percentage utilization of Branch Target Buffer	7.81 %

Table 7.1: Simulation Results

Dispatch logic stalls the pipeline until all the instructions fetched in a particular instruction block are dispatched causing more stalls as indicated by the results. This causes the IPC to be reduced as, for the stalled cycle, no new instruction block is fetched. IPC of the processor can be increased and dispatch stalls can be reduced by having more efficient dispatch logic. It might pick instructions incrementally as individual instructions are dispatched instead of waiting for all the instruction in a particular instruction block to

be dispatched before fetching the new instruction block. Secondly, optimal utilization of instruction level parallelism is imperative for a good IPC. Hence, an efficient compiler can significantly improve IPC. The SDE-Gcc C compiler used here causes a reduction in IPC as it is not tuned for a multi-liner processor. Hence IPC could be increased by employing a compiler specifically for multi-linear processors. Since the compiler is incapable of handling interrupts and exceptions, standard testbenches are not used for testing the design. Moreover the sample 'C' program used for testing is kept small to reduce the simulation time. This causes the misprediction rate to be high because the prediction mechanism is still in its initial phase indicated by the percent utilization of branch target buffer and branch prediction buffer. Hence a high percentage of branch and jump instruction are encountered for the first time causing higher misprediction. Better utilization of the Branch target buffer and the Branch prediction buffer can be achieved by running larger programs. This will also cause the misprediction rate to reduce as more and more data about different branches is stored in the branch target buffer and branch prediction buffer.

This chapter discussed in detail the methodology employed for verifying the design and also discussed performance measuring parameters. It also discussed results obtained by simulating a sample 'C' program stating the cause of inefficiency reflected in the data and methods to reduce them.

Chapter 8

Conclusion & Future Work

8.0 Conclusion

The processor architecture presented in this study was aimed at an implementation of pseudo-registers for efficiently handling speculated instructions assuming an infinite instruction and data cache. The machine state should not be altered unless the outcome of the branch instruction preceding the speculated instructions is known. To avoid changing the machine state, a pointer based pseudo-register scheme was employed. Each incoming instruction with a destination was assigned a pseudo-register identified by a pseudo-pointer. These pseudo-pointers were maintained in a translation buffer called the issue pointer buffer. Instead of moving the data from a pseudo-register to an architectural register, the corresponding pseudo-pointer was written into another translation buffer called the commit pointer buffer, whenever an instruction commits. This requires a 6-bit data transfer as compared to a 32-bit data transfer making the design more efficient. There are a total of 64 registers in the implementation of which 32 are architectural registers and the rest are pseudo-registers. As each incoming instruction is renamed to a pseudo-register, data consistency is maintained by doing a destination and source overwrite whenever necessary.

A two-bit saturating counter is used for predicting incoming branch or jump instructions. This mechanism is capable of predicting one branch every cycle. Hence, if there is more than one branch or jump instruction in a fetch group, instructions after the delay slot instruction of the first branch are re-fetched in the next cycle. There are a total of six functional units for parallel execution. Coprocessor 0 is implemented as a static element always returning zeros. Each functional unit except Coprocessor 0 is accompanied by a reservation station for dynamic scheduling of instructions at run time. Hence, instructions can be issued for execution out of program order.

Up to four instructions are dispatched every clock cycle to the reservation station. A new fetch group is dispatched only if all the instructions in the previous fetch group are dispatched successfully. Completed instructions are written back using two write back buses called the common data bus I & II respectively. All instructions are restored to program order in the reorder buffer before altering the machine state. The reorder buffer is a first in first out queue that strictly maintains the program order. Correct branch target addresses are also written into the reorder buffer along with the misprediction bit, as the branch misprediction is detected at the top of the reorder buffer queue. When a misprediction is detected, a system restore is initiated to bring the processor state back to where it was before the mispredicted branch. This includes flushing the reorder buffer, reservation stations and copying all the commit pointer buffer entries to the issue pointer buffer.

This study also discussed a design verification methodology. The above implementation was tested with a sample 'C' program whose final output was known. This 'C' program was compiled using the SDE-Gcc C compiler, which is freely redistributed by Algorithmics. The intended output was compared with the output generated by the processor, verifying the design. Data was collected for calculating the instructions per cycle (IPC), misprediction rate, percentage utilization of branch prediction buffer and branch target buffer and dispatch stalls.

8.1 Future Work

Future work for this design may be considered as a realistic implementation of the instruction and data cache, so that the effect of cache miss could be simulated. To reduce the dispatch stalls, the dispatch logic could be improved by not waiting until all the instructions from a particular block are dispatched before fetching the next block. Instead a incremental fetch should be done as each instruction from a particular fetch block is dispatched. This design is hard to extend for a larger fetch size, since the branch prediction mechanism is capable of predicting one branch or jump instruction every clock cycle. Studies have indicated that there is at lest one branch or jump instruction every five instructions [1]. Hence, there is a high probability of having more than one branch or jump instruction in a larger fetch group. This requires a prediction mechanism which is capable of predicting more than one branch per clock cycle to maintain a high IPC.

The system restore following a mispredicted branch or jump instruction is delayed until it is identified at the top of the reorder buffer queue. There is a finite delay between the time when these instructions finish execution and the system is restored. Hence, instructions are fetched from a wrong program path until the misprediction is identified at the top of the reorder buffer queue, which is inefficient. This inefficiency can be removed by doing an early recovery. In this case correct branch target addresses are written into the program counter as soon as it is calculated and misprediction is detected. Hence, early recovery can be done at the end of the execution stage of the pipeline.

The compiler used for verifying the design is incapable of handling interrupts and exceptions. Hence, real benchmarks can't be run on this processor. To test the design with these benchmarks, a better compiler capable of handling interrupts and exception could be used. And, finally mechanism for reducing the simulation time has to be determined so that more though testing could be done on this implementation for its verification.

References

[1] Chang, M.-C.; Chou, Y.-W.; "*Branch prediction using both global and local branch history information Computers and Digital Techniques*", IEEE Proceedings-, Volume: 149 Issue: 2, March 2002 Page(s): 33 -38

[2] Evers, M.; Tse-Yu Yeh; "*Understanding branches and designing branch predictors for high-performance microprocessors*", Proceedings of the IEEE, Volume: 89 Issue: 11, Nov. 2001 Page(s): 1610 -1620

[3] Wallace, S.; Bagherzadeh, N.; "*Multiple branch and block prediction*" High-Performance Computer Architecture, 1997, Third International Symposium on, 1-5 Feb. 1997 Page(s): 94 -103

[4] Egan, C.; Steven, F.; Steven, G.; "*Delayed branches versus dynamic branch prediction in a high-performance superscalar architecture*" EUROMICRO 97. 'New Frontiers of Information Technology'. Short Contributions, Proceedings of the 23rd Euromicro Conference, 1-4 Sept. 1997 Page(s): 266 -271

[5] Jimenez, D.A.; Keckler, S.W.; Lin, C.; "*The impact of delay on the design of branch predictors*", Microarchitecture, 2000. MICRO-33. Proceedings, 33rd Annual IEEE/ACM International Symposium on, 10-13 Dec. 2000 Page(s): 67 -76

[6] Skadron, K.; Ahuja, P.S.; Martonosi, M.; Clark, D.W.; "*Branch prediction, instruction-window size, and cache size: performance trade-offs and simulation techniques*" Computers, IEEE Transactions on, Volume: 48 Issue: 11, Nov. 1999 Page(s): 1260 -1281

[7] John L. Hennessy and David A. Patterson *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers Inc (1990, 1996).

[8] Dutta, S.; Franklin, M.; "*Block-level prediction for wide-issue superscalar processors*" Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP. IEEE First International Conferencen, Volume: 1, 19-21 Apr 1995 Page(s): 143 -152 vol.1.

[9] Lenell, J.; Bagherzadeh, N.; "*A performance comparison of several superscalar processor models with a VLIW processor*", Parallel Processing Symposium, 1993, Proceedings of Seventh International, 13-16 April 1993, Page(s): 44-48

[10] Smith, J.E.; Sohi, G.S.; "*The microarchitecture of superscalar processors*" Proceedings of the IEEE, Volume: 83 Issue: 12, Dec. 1995 Page(s): 1609 -1624

[11] Manjikian, N.; "*Design issues for prototype implementation of a pipelined superscalar processor in programmable logic*" Communications, Computers and

signal Processing, 2003. PACRIM. 2003 IEEE Pacific Rim Conference on, Volume: 1, Aug 28-30, 2003 Page(s): 155 -158

[12] Balasubramonian, R.; Dwarkadas, S.; Albonesi, D.H.; "*Reducing the complexity of the register file in dynamic superscalar processors*" Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on, Dec. 1-5, 2001 Page(s): 237 -248

[13] Tse-Yu Yeh; Patt, Y.N.; "*Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors*" Microarchitecture, 1993. Proceedings of the 26th Annual International Symposium on, 1-3 Dec. 1993Page(s): 164 -175

[14] Dominic Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers Inc (2002).

[15] Riordan, T.; Grewal, G.P.; Hsu, S.; Kinsel, J.; Libby, J.; March, R.; Mills, M.; Ries, P.; Scofield, R.; "*The MIPS M2000 system*" Computer Design: VLSI in Computers and Processors, 1988. ICCD '88, Proceedings of the 1988 IEEE International Conference on, 3-5 Oct. 1988 Page(s): 366 -369

[16] Martin, A.J.; Lines, A.; Manohar, R.; Nystrom, M.; Penzes, P.; Southworth, R.; Cummings, U.; Tak Kwan Lee; "*The design of an asynchronous MIPS R3000 microprocessor*" Advanced Research in VLSI, 1997. Proceedings, Seventeenth Conference on, 15-16 Sept. 1997 Page(s): 164 -181

[17] *Executable and Linkable Format (ELF)*

http://www.skyfree.org/linux/references/ELF_Format.pdf

VITA

Anshuman Anand

Candidate for the degree of

Master of Science

Thesis: ARCHITECTURAL VERIFICATION OF FOUR-INSTRUCTION SUPERSCALAR PROCESSOR FOR MIPS I INSTRUCTION SET

Major Field: Electrical & Computer Engineering

Biographical:

Personal Data: Born in Hoshirpur, Punjab India, on February 16, 1977, the son of Sushma and Arun Kumar Anand.

Education: Graduated from St. Francis De Sale's College, Nagpur India in March 1995; received Bachelor of Engineering degree in Electronics & Power (Electrical) Engineering from Nagpur University, Nagpur India in March 1999; completed the requirements for the Master of Science degree with a major in Electrical & Computer Engineering at Oklahoma State University in December 2003.

Experience: Worked as a software programmer at Subhashini Software Solutions March 2001 to July 2001; Employed by The Daily O'Collegian as webmaster and graduate assistant, The Daily O'Collegian, Oklahoma State University, August 2001 to present.