REGISTER RENAMING ALGORITHM FOR FAST BRANCH MISPREDICTION RECOVERY IN SUPERSCALAR PROCESSOR

BY

BALACHANDER GANESAN

Bachelor of Engineering

University of Madras

Chennai, India

2001

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

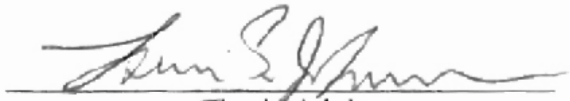
REGISTER RENAMING ALGORITHM FOR

FAST BRANCH MISPREDICTION

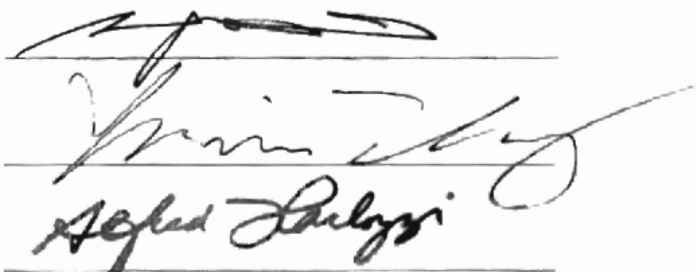
RECOVERY IN SUPERSCALAR

PROCESSOR

Thesis Approved:



Thesis Advisor



Dean of the Graduate College

ACKNOWLEDGEMENTS

My sincere gratitude is due to my advisor Dr. Louis G. Johnson for his inspiration, guidance and continuous encouragement throughout my thesis. The presented work is the result of his support and motivation. I also thank Dr. Yumin Zhang and Dr Weili Zhang for serving as my committee members. Their comments and suggestions are greatly appreciated. I extend my thanks to Anshuman Anand, for his valuable contributions to this thesis. My heartfelt thanks to my family members for their extreme love and continuing support. Finally, I would like to thank my friends for their encouragement and moral support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The ever increasing demand for more computing power has led to research on ways to improve processor performance. Instructions per Clock (IPC) is considered as one of the most significant performance metrics. For a processor executing a program, IPC is the average number of instructions which is executed in one clock cycle. Performance is given by,

$$Performance = \frac{IPC}{IC * T}.$$

Where, IC is the instruction count and T is the clock period.

Thus, improving IPC can result in improved performance. Development of linear pipelined processors, which had higher throughput compared to their un-pipelined counterparts, was the first step towards meeting the demand for more computing power. Although IPC for linear pipelined processor handling one instruction per stage is ideally one, practically it can never reach its ideal limit. This is due to the various pipeline stalls which occur due to data hazards and control hazards [9]. Because of the problem of limited IPC in a linear pipelined processor due to various hazards, alternate architectures handling multiple instructions in a pipeline stage, with potentially higher IPC were proposed and successfully implemented. They are the Very Long Instruction Word (VLIW) architecture and the Superscalar architecture. The fundamental idea behind these

architectures is to take advantage of the inherent Instruction Level Parallelism (ILP) present in the program, by executing multiple instructions simultaneously using parallel functional units.

VLIW architecture relies heavily on compilers to identify and encode instructions in a way suitable for parallel execution by the functional units. Hardware is simplified considerably, because the compiler does the instruction scheduling. If the compiler cannot find an instruction to fill a slot, it inserts NOPs, thereby increasing the code size.

In the superscalar approach, hardware has inbuilt scheduling logic, to identify and execute instructions in parallel. The responsibility of keeping the functional units busy is assigned to the dynamic scheduling unit [9]. To realize the performance improvement from processing multiple instructions, a good dynamic scheduling logic which detects and issues instructions (possibly out-of order) is imperative. The scheduling is normally based on availability of the required source data required to start execution and also the instruction's order in the program.

1.1 Problem Statement

Assuming multiple copies of functional units are available, multiple instructions can be fetched and executed simultaneously, as long as there are no dependencies among the fetched instructions. These dependencies may be caused by the lack of physical registers to store data (name dependency) or due to the producer-consumer relationship (true data

dependency) among instructions. Hence, to increase the number of executable instructions, some of these dependencies must be removed. Register renaming is the process by which such name dependencies are removed. This is accomplished by allocating a different physical register to store data for the logical destination register in an instruction. This mapping is stored in a mapping table (MT), so that, later instructions with that logical destination register as a source register, can read the data from the physical register. Figure 1.1 illustrates the various dependencies and the elimination of name dependency by register renaming. In the figure, the logical source registers are assumed to have been already renamed to some physical register. Also, Figure 1.1 shows the mapping of logical registers to physical registers after the execution of ADD and SUB instructions.

Another glaring problem is the presence of control dependent instructions in the multiple instructions fetched to execute in parallel. To sustain a good fetch rate, branch prediction has been successfully implemented to determine the next fetch address and also predict if the control dependent instructions must be executed. If a branch prediction is later realized to be wrong, the speculatively executed instructions must be cancelled before updating the system state.

a) True Data Dependency

ADD R1 R0, R2
SUB R4 R1, R3

Register Renaming →

	Initial	ADD	SUB
R0	P0	P0	P0
R1	-	P1	P1
R2	P2	P2	P2
R3	P3	P3	P3
R4	-	-	P4

a) True Data Dependency

ADD P1 P0, P2
SUB P4 P1, P3

b) Name Dependency (WAR)

ADD R1 R0, R2
SUB R2 R4, R3

Register Renaming →

	Initial	ADD	SUB
R0	P0	P0	P0
R1	-	P1	P1
R2	P2	P2	P5
R3	P3	P3	P3
R4	P4	P4	P4

b) Dependency Eliminated

ADD P1 P0, P2
SUB P5 P4, P3

c) Name Dependency (WAW)

ADD R1 R0, R2
SUB R1 R4, R3

Register Renaming →

	Initial	ADD	SUB
R0	P0	P0	P0
R1	-	P1	P5
R2	P2	P2	P2
R3	P3	P3	P3
R4	P4	P4	P4

c) Dependency Eliminated

ADD P1 P0, P2
SUB P5 P4, P3

Figure 1.1 Dependency Elimination

When register renaming, dynamic scheduling and branch prediction are present in an implementation, the process of recovery from a branch misprediction or exception is challenging. This is because of the possible out-of order execution of instructions and also the need to restore the mapping table (MT) to a state it would have been in, if the speculated instructions were not executed. This restoration phase can potentially take multiple cycles of operation resulting in an increased penalty for branch miprediction.

1.2 Previous Work

This section summarizes the previous work done in register renaming and means to exploit ILP. The study done in [7] showed the instruction fetch limitation due to control dependent instructions in the fetch group. The paper argues that, regardless of scheduling, the average number of instruction that can be fetched and executed per cycle cannot exceed 1/p, where p is the probability of a taken branch or jump instruction. They proposed dual port BTB (DBTB) and dual port instruction cache (organized as a memory bank) as means to overcome this fetch limitation. The DBTB can provide target address for the next two fetches, which are simultaneously read using the dual port instruction cache [7]. This increases the average number of instructions fetched. There is one main concern about this scheme. Because of the sequential nature of operation of the proposed DBTB, for a large BTB the lookup time can be very large and can possibly affect the clock time.

Work presented in [2] gives an overall picture on register renaming and classifies it based on the type of rename buffer (split or merged) and the kind of register mapping (Mapping Table or mapping in the rename buffer) used. It points out, for a merged implementation of the rename buffer, there is no need for data transfer, and hence, hardware is simplified. It also suggested the following guideline for choosing the size of storage modules (rename buffer, ROB, dispatch window) involved in register renaming.

$$wdw \leq rb \leq rob$$

where, *wdw* is the width of the dispatch window, *rb* is total number of rename buffers and *rob* is size of reorder-buffer.

To reduce the number of read ports in the rename buffer, a scheme called dispatch bound operand read is suggested in [2], in which data is read only for the dispatched instructions. Since the number of dispatched instructions is generally less than the number of issued instructions, one can have less read ports in the register file [2].

All the work summarized below focuses on efficient usage of rename registers as means to reduce the size of the rename buffer. In [3] a hardware scheme is proposed for efficient usage of rename registers through early register release. They claim this can result in reduced number of rename registers and hence a faster clock rate, assuming that the register file is in the critical timing path for clock period. In the proposed scheme, a rename register is scheduled for early release if it is identified that there are no more pending reads from that rename register [3]. It uses a Last Use Table (LUT) and extended ROB fields to keep track and identify registers for early release. The LUT needs to be heavily ported (32 read and 24 writes) for an 8-wide superscalar processor. Although performance improvement of 3% to 6% was reported for numerical programs, cost effective implementation of such a heavily ported LUT is questionable.

In a conventional register renaming scheme, a physical register is allocated during decode and reclaimed when another instruction with the same logical destination commits. In such a scheme, the physical register is allocated for a time which is more than required

[4]. The physical register is wasted (contains no value) from the time it is allocated (during instruction decode) until execution is completed. This increases the need for more physical registers. The proposed scheme in [4] allocates a physical register for a much shorter time than the conventional scheme, thereby, reducing the need a for large physical register file. During decode, virtual tags (which does not address storage locations) are assigned to the logical destination registers. These tags are stored in a General Map Table (GMT) [4]. These virtual tags address another table, Physical Map Table (PMT) which stores the corresponding physical register address. A physical register is allocated only when the instruction completes and at that time, the PMT is updated. Misprediction recovery in such a scheme requires the sequential popping out of instruction form the ROB to set up values in GMT and PMT. Potentially this may take several cycles, increasing the branch misprediction penalty.

Another attempt to achieve efficient register usage, by reusing a physical register for storing the result of two instructions is summarized below. In this scheme, a physical register is reused whenever it is detected that an incoming instruction is likely to produce the same result as one of the previously completed instruction [5]. For example, a 'move' instruction just transfers data from one register to another. The logical destination register's entry in the mapping table for such 'move' instructions is updated with the previously allocated physical register's address, holding the same value. A counter corresponding to that physical register is incremented, indicating that the physical register is shared by two logical destination registers [5]. This scheme enables the removal of

'move' instructions from execution, thereby reducing the dependency latencies of instructions following the 'move' instruction.

1.3 Design Approach

This design is focused on designing a register renaming and scheduling scheme aimed towards fast misprediction and exception recovery. Fast recovery from branch misprediction or exception will reduce the penalty involved with those events and hence, can result in an improved IPC.

The design uses a merged physical register file holding both correct and speculated data. This approach is advantageous in the sense that data transfers between register files can be eliminated, resulting in a simpler hardware. There are two mapping tables (MT) whose content addresses the physical register file. These mapping tables are indexed using the logical register numbers appearing in the instruction. The content of the first mapping table (IP) is updated speculatively at the time of decode. The second mapping table (CP) is updated when the instruction commits. Thus, CP always has the correct mapping for the logical registers to the physical register file, where the data is stored. During a branch misprediction recovery sequence or exception, the contents of CP are copied into IP through a special internal port. IP and CP are designed as an integrated unit with the internal port to simultaneously copy contents of CP to IP during restore. This can be accomplished in one cycle. Figure 1.2 is an illustration of the register renaming scheme.

In Figure 1.2, A, V, C are used to indicated if the physical register file is allocated, if its data is valid (finished execution) and if it has been committed, respectively. The figure shows the state of MT, physical register file and status bits after a fetch group has been decoded and dispatched. As seen from Figure 1.2, the allocated physical register file locations are set to '1', preventing future allocation of that location as a rename register until that location is reclaimed. The Reorder Buffer (ROB) holds the logical register number and rename register number for all instructions, in program order.

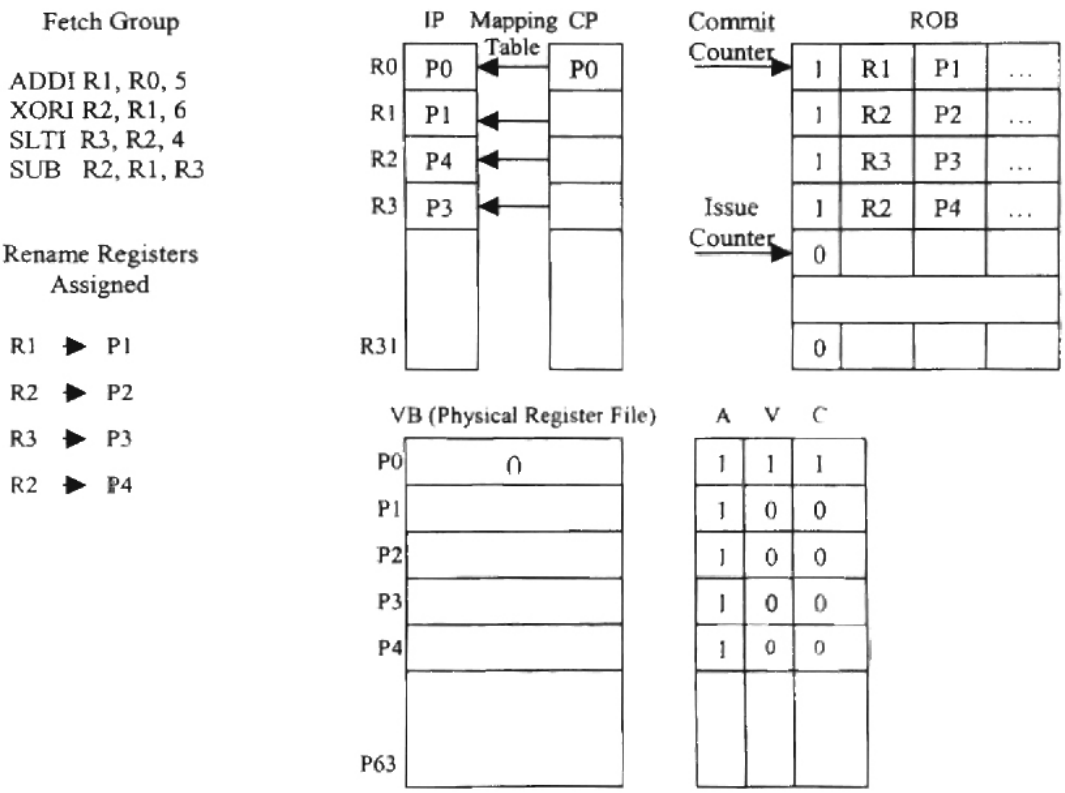


Figure 1.2 Illustration-1of Register Renaming

Figure 1.3 shows the state of modules after ADDI and XORI have committed, SLTI has finished execution but not committed and SUB is still waiting to be executed. For

instructions which have finished execution and written results to VB, V is set to '1'. For instructions which have committed, C is set to '1' and CP is updated with the assigned rename register's address. The ROB in Figure 1.3, has committed the first two instructions (ADDI and XORI) and has updated the corresponding CP mapping table with the rename register assigned for that instruction. The Commit Counter now points to an uncommitted instruction in the ROB. A Detailed explanation on the design of the Status Bits and the ROB is presented in Chapter 3.

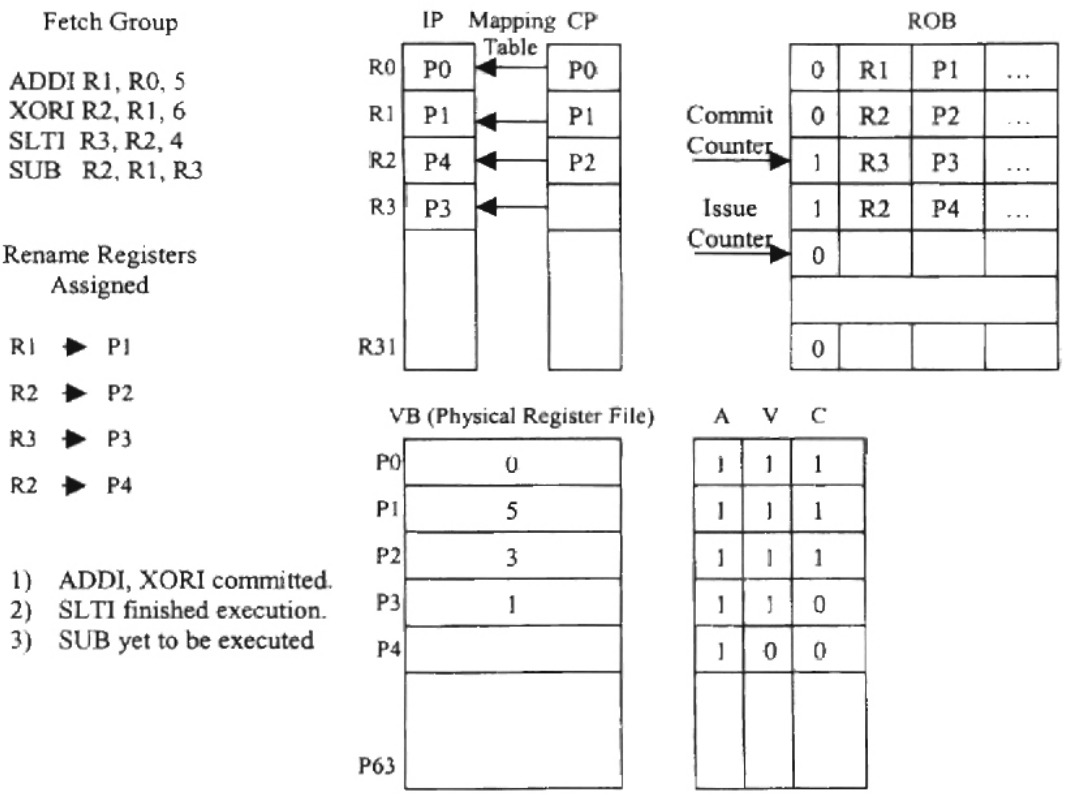


Figure 1.3 Illustration-2 of Register Renaming

1.4 Contributions to Thesis

This design was a joined project with Anshuman. RTL design of Branch Target Buffer (BTB), Branch Prediction Buffer (BPB), Mapping Table (IP/CP), Priority Encoder, Decode, Over write Logic, ALU and Branch Execution Logic was done in [12]. Refer to [12] for detailed implementation details about those modules.

Chapter 2

Architecture Overview and Organization

This chapter introduces the various pipeline stages and their roles in the superscalar processor. The various stages are Fetch, Decode/Issue, Dispatch, Reservation Station, Execution, Write Back, Commit-1 and Commit-2. The process of register renaming is accomplished in the Issue and Dispatch stages. Figure 2.1 shows the microarchitecture of the processor.

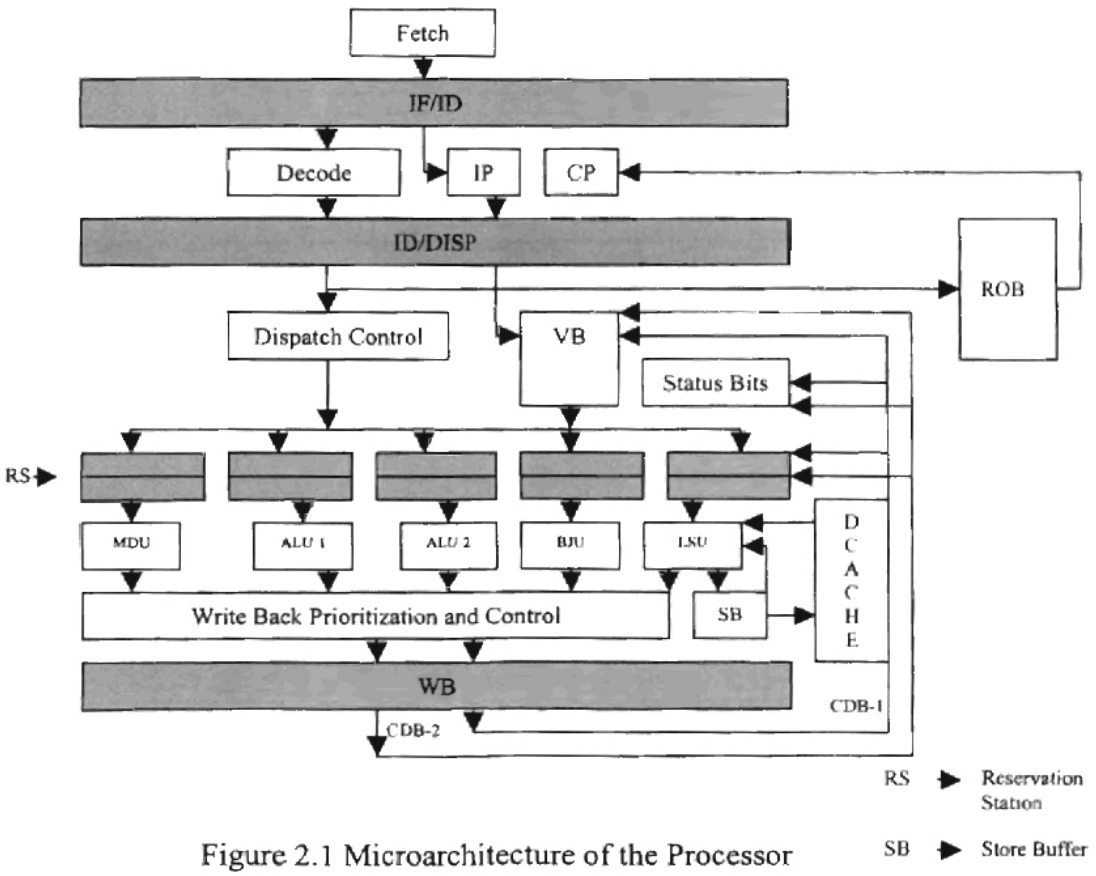


Figure 2.1 Microarchitecture of the Processor

2.1 Fetch

Fetch is the first stage of the architecture. Its primary role is to fetch and provide a predetermined number of instructions (fetch group) for processing. The size of the fetch group is chosen depending on available Instruction Level Parallelism (ILP) and willingness to incur an extra cost of hardware for performance improvement (cost increases approximately as square of the number of ports in the register file). The primary limitation to ILP is due to dependencies between instructions. In the fetch stage, the ILP is limited by control dependent instructions (instructions occurring in the shadow of branch/jump). In a fetch group, instructions following a taken branch/jump should not be executed. Thus, the actual number of executable instructions fetched, is lesser than the size of fetch group [7]. For a larger fetch group, the probability of occurrence of a taken branch or jump is larger. So, the average percentage of executable instructions in such a fetch group will be less.

2.1.1 Fetch Limitation

In the MIPS-1 ISA, branch and jump instructions have a delay slot which must be executed irrespective of the action of the branch/jump instruction. The following analysis shows that, the average number of executable instructions of a fetch group, will be higher than an ISA without a branch delay slot. It is assumed that a compiler always fills the delay slot with a useful instruction that is not another branch, the branch and its delay slot instruction are always kept in the same fetch group, fetch stage can always predict if

branch is taken and the fetch group size is greater than two. Let 's' be the size of the fetch group and 'p' be the probability of a taken branch/jump.

Average Executable Instructions (No delay slot):

$$S_E = \frac{1-(1-p)^s}{p}$$

Since p < 1, for very large values of 's', S_E saturates to 1/p. Therefore, the maximum number of executable instructions in this case is 1/p [7].

Average Executable Instructions (Branch Delay Slot):

The average executable instructions is calculated by the weighted sum of all the possible fetch cases [7]. The probability of fetching only one executable instruction in a fetch group is zero. This is because, even if the first instruction is a branch, we can execute it along with its delay slot instruction.

$P_1 = 0$

The probability of fetching only two executable instructions in a fetch group is 'p'. This can occur if the first instruction is a taken branch or jump. Hence it has a probability of 'p'.

$P_2 = p$

The probability of fetching only three executable instructions in fetch group is '(1-p)p'. This can occur if the second instruction is a taken branch/jump and the first instruction is not a taken branch/jump.

$P_3 = (1-p)p$

The probability of fetching only four executable instructions in fetch group is '$(1-p)^2p$'. This can occur if the third instruction is a taken branch or jump and the first two instructions are not taken branches or jumps.

$P_4 = (1-p)^2 p$

This continues until a taken branch/jump does not occur until the third to last instruction of the fetch group. Hence,

$P_{(s-2)} = (1-p)^{(s-4)} p.$

The probability of fetching only (s-1) executable instructions is as shown below. The first term occurs from the case that the first (s-3) instructions are not taken branches or jumps and the next instruction is a taken branch or jump. The second term occurs when a taken branch or jump is in the last position of the fetch group and all other instructions are not a taken branch or jump. This requires the branch to be cancelled and re-fetched in the next cycle to keep the delay slot instruction in the same fetch group as the branch/jump.

$P_{(s-1)} = (1-p)^{(s-3)} p + (1-p)^{(s-1)} p$

The probability of fetching 's' executable instructions is as shown below. The first term occurs when there are no taken branches or jumps in the fetch group and the second term comes if the branch/jump occurs in the second to last position with its delay slot in the last position of the fetch group.

$P_s = (1-p)^s + (1-p)^{(s-2)} p$

All the probabilities developed above are under the assumption that a previous fetch did not have a branch in the last position. If the previous fetch had a branch in the last position then, the number of instructions fetched is two. Therefore, the average number of executable instructions in a fetch group with branch delay slot is given as follows:

$$S_E = (1-(1-p)^{(s-1)}p)(1.P_1 + 2.P_2 + 3.P_3 + ... + s.P_s) + 2(1-p)^{(s-1)}p$$

On simplifying the above equation,

$$S_E = \frac{1-(1-p)^s + p((1-p)^{2s}-1) + p^2((1-p)^s-1) + p^3((1-p)^{2s}+1)}{p(p-1)^2}$$

Typically p = 0.1 for general purpose streams. Table 2.1 shows the improvement in S_E when the branch delay (BD) slot is handled as described above. Using the above scheme can result in improvement of up to 7% in fetching executable instructions.

S	S_E (ND)	S_E (BD)	S	S_E (ND)	S_E (BD)
3	2.710	2.752	10	6.513	6.889
4	3.440	3.517	11	6.861	7.287
5	4.095	4.217	12	7.175	7.647
6	4.685	4.856	13	7.458	7.973
7	5.217	5.439	14	7.712	8.269
8	5.695	5.969	15	7.941	8.536
9	6.125	6.452	16	8.146	8.778

Table 2.1 Instruction Fetch Limitation

2.1.2 Instruction Cache Organization

The Fetch stage is comprised of a Program Counter (PC), which is a loadable register updated every cycle, an instruction cache, Branch Target Buffer (BTB), Branch Prediction Buffer (BPB) and fetch control logic. The program counter is used to index an instruction cache which is organized as a four-bank memory with each memory bank itself being a direct mapped cache. Since fetch has to provide four instructions every cycle to the next stage, the instruction cache must be wide enough to provide them [7]. The instruction cache used in this architecture provides four instructions every cycle without any misses. The Instruction cache must also have the ability to provide instructions when the fetch group spans across two cache blocks. For sustained fetch bandwidth, cache must be able to provide instructions in such cases, without additional clock cycles. This problem is resolved by organizing the instruction cache as a four-bank memory [11] as in Figure [2.1]. For a read operation, PC [3:2] is used to identify the first instruction's bank. The rest of the bits of the PC are used as index and tag, for generating the read/write select lines and tag match, respectively for each bank. If the addresses cross the four word boundary, the input to the decoder, "index", is appropriately incremented to index the next row in the bank, shown as "DI logic". The output of the four banks are reordered using an instruction reordering network to get the original program order, before being passed to Decode/Issue stage.

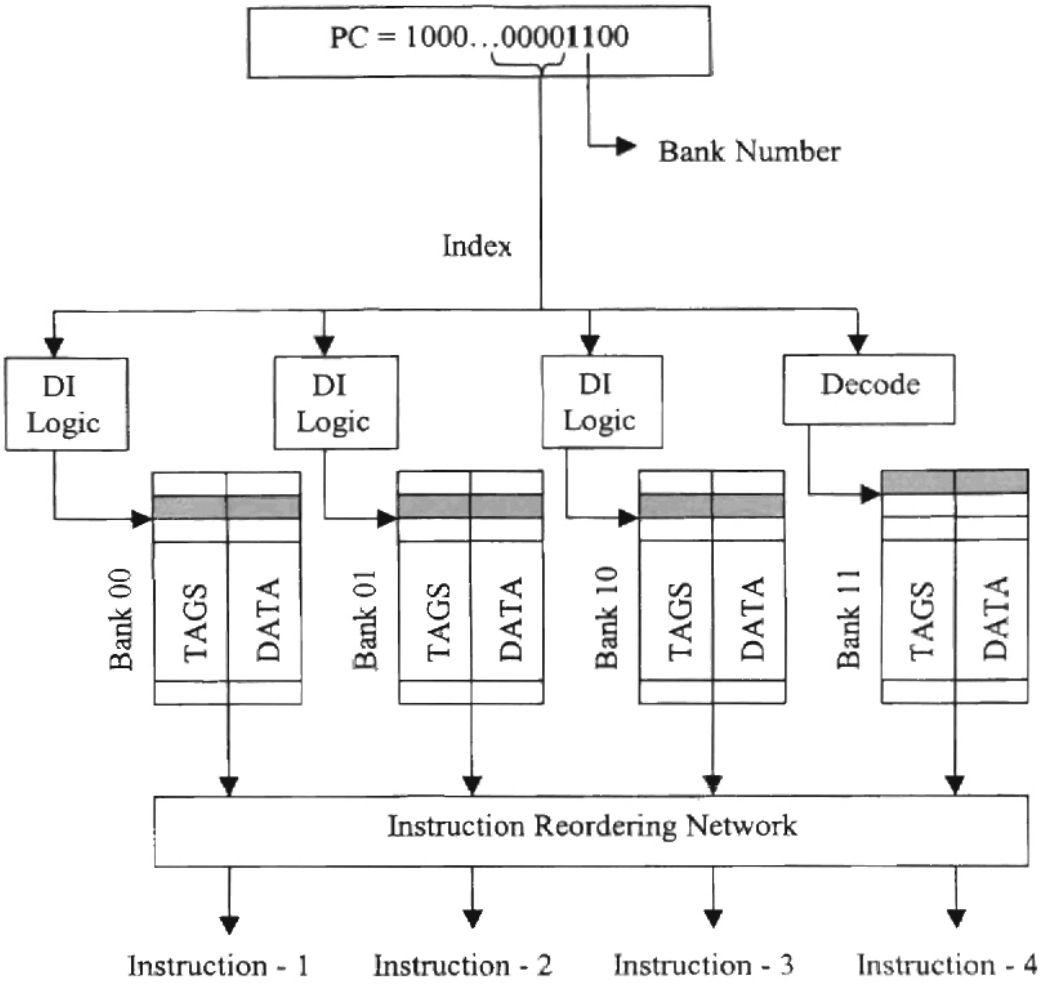


Figure 2.1 Four-bank Direct Mapped Instruction Cache

2.1.3 Branch Prediction

Conditional and unconditional branches (jumps) cause problems for sequential fetching of instructions. Without branch prediction, fetch has to wait till the branch target address is evaluated, which may be several stages down the pipeline, depending on the addressing scheme followed in the Instruction Set Architecture (ISA). This contributes to stalls occurring due to branches and jumps. The impact of these stalls is more severe in superscalar architectures, since the actual branch penalty is the number of stalls

multiplied by the size of fetch group, as compared to branch penalty being equal to number of stalls in scalar pipelines.

To improve performance, control dependent instructions can be allowed to execute as long as they do not change the state of the machine (no unrecoverable update to system register or memory). When the branch is evaluated, if the speculation is correct, useful work is done and the branch penalty is not incurred. Only when the speculation is incorrect, we have to recover and fetch correct instructions and we incur the branch penalty. This prediction is done depending on the history of a branch instruction. The prediction bits are store in Branch Prediction Buffer (BPB) and the target address is stored in Branch Target Buffer (BTB).

2.1.4 Branch Target Buffer (BTB)

The BTB is used to store the target address of the taken branches and jump instructions, so that when those instructions are executed again (for example, a branch instruction evaluating the condition of termination of a loop is executed several times), we have the target address readily available. The PC is used to index the BTB. It is also organized as a four-bank memory similar to instruction cache and it gives the target address of the branch or jump instruction nearest to PC. The target address of a taken branch/jump is written into the BTB at the time the branch/jump is committed. The BTB can either read or write in one cycle. So whenever a target address is written into the BTB, fetch needs to be stalled as we cannot do the look up for the target address for the fetched group.

2.1.5 Branch Prediction Buffer (BPB)

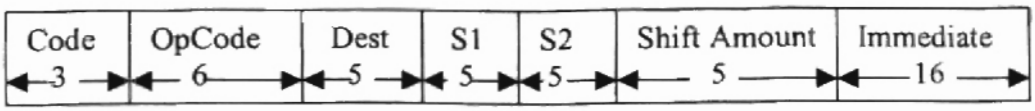
BPB, also organized as a four-bank memory, stores the prediction bits for all jumps and branches. Initially all branches are predicted not taken and updated at commit time using the two bit correlating algorithm. PC is updated to the branch target address only if the branch/jump being fetched has an entry in the BTB and the BPB entry for that branch predicts it to be taken.

2.2 Decode

Decode is the second stage of the pipeline. It decodes the instructions fetched and generates internal codes for instructions depending on the functional unit. The standard decode output format is as shown Figure 2.2.

The "Codes" for various functional units are Co-Processor Zero Unit (CP0) - 001, Multiply Divide Unit (MDU) - 010, Arithmetic Logic Unit (ALU) - 011, Branch Jump Unit (BJU) - 100, Load Store Unit (LSU) – 101. The "OpCode" contains the code for identifying the function of the instruction. Since the execution units are simulated in the highest level of abstraction, the "Opcode" is not decoded. For Link instructions, the "Dest" field contains 11111 so that the return address is written into R31, a requirement in the MIPS architecture.

Register Write Instruction



Branch / Jump Instruction

Code	OpCode	Dest	S1	S2	0000000000	Offset
Code	OpCode	Dest	S1	Instruction Index		

Figure 2.2 Decode Output format

Decode also generates the control signals of a NOP indicator, a branch or jump indicator and a non-register write instruction indicator. Decode has logic built in to cancel branch or jump instructions occurring at the fourth slot of the fetch group. All instructions following the delay slot instruction of a branch/jump predicted as 'taken' and branches in the shadow of a branch/jump predicted as 'not-taken' are also cancelled. This is because the prediction for the second branch in the fetch group is not available since BPB can do a lookup of prediction bits for only one branch per cycle.

2.3 Register Renaming Scheme

Register renaming enables the removal of name dependencies in the fetch group by increasing the number of physical registers internal to the processor. This will result in more Instruction Level Parallelism (ILP). The hardware involved in the register renaming scheme are Issue Pointer Buffer (IP), Commit Pointer Buffer (CP), Prioritizer, Value Buffer (VB), Allocate Bits, Valid Bits, Commit Bits and Reorder Buffer (ROB). IP, CP,

VB are all multiported static RAM type memory elements capable of doing a read and a write in one clock cycle. Brief descriptions of the role of these modules are given below.

Issue Pointer Buffer (IP): IP is a mapping table which maps the logical registers in the program to locations in the Value Buffer, where the data is stored. Read from IP is done in the second half of the "Issue" stage and Write is done in the first half of the "Dispatch" stage.

Value Buffer (VB): It stores the result of the instruction after it is calculated and written back. "Data" for source operands are read in the second half of "Dispatch" and computed "data" is written into "VB" in the first half of "Write Back" stage.

Commit Pointer Buffer (CP): It is also a mapping table, which maps the logical registers in the program to the physical register, Value Buffer, similar to IP. The difference between IP and CP is that the CP is the mapping table for instruction which have completed, whereas, IP is the mapping table for newly (possibly speculatively) fetched instructions. Read is done in the first half of "Commit-2" and Write is done in the second half of "Commit-2".

The logical register, appearing in the program code, addresses the IP, which holds the "pseudo-pointers". These pseudo-pointers in turn address a VB location where data is stored. The logical register number is 5 bits wide (in MIPS ISA), capable of addressing

32 IP locations. The pseudo-pointer stored in IP is 6 bits wide, capable of addressing 64 VB locations. Thus, the actual locations where data can be stored are increased internally.

In the second half of the Issue stage, the pseudo-pointers for the source registers appearing in the program are read from the IP and in first half of Dispatch, source overwrite logic (SOW) processes them and then they are decoded as shown in Figure 2.3. These decoded values index the corresponding locations in VB for data read in second half of Dispatch. SOW replaces the source pseudo-pointers with the corresponding newly picked destination pseudo-pointers, if there are RAW dependencies within a fetch group. The number of comparisons required is given by s^2-s, where 's' is the fetch group size [9]. This square dependence can result in high hardware cost for large fetch groups.

All "register-write" instruction's logical destination registers in the fetch group are allocated new destination registers, pseudo-pointers, in the Issue stage. These pseudo-pointers are written into the IP in first half of Dispatch so that instructions in the next fetch group can read them from IP. Before they are written into IP, destination overwrite logic (DOW) checks for WAW dependencies within a fetch group and if any exist, it replaces the earlier destination pseudo-pointer with a later one. This enables instructions fetched in the subsequent cycles to read the latest source pseudo-pointer from IP. The prioritizer generates the pseudo-pointers by doing a priority encoding of the available 'Allocate Bits'.

Allocate Bits, Valid bits and Commit Bits are the status bits used for book keeping purposes in the register renaming scheme. There are as many status bits of each type as the number of VB locations. Status bits have one to one correspondence with VB locations.

Allocate Bits indicate if the VB location is free for allocation as an internal destination location for an instruction (indicated by '0'), or if it is currently being used by some instruction (indicated by '1'). If the Prioritizer picks a VB location's address as a "pseudo-pointer" during the Issue stage, the corresponding Allocate Bit is set to '1' at the beginning of the Dispatch stage. The Allocate Bit is reset to '0' when a later instruction with the same logical destination register commits. This is the process of de-allocation of internal registers for future instructions.

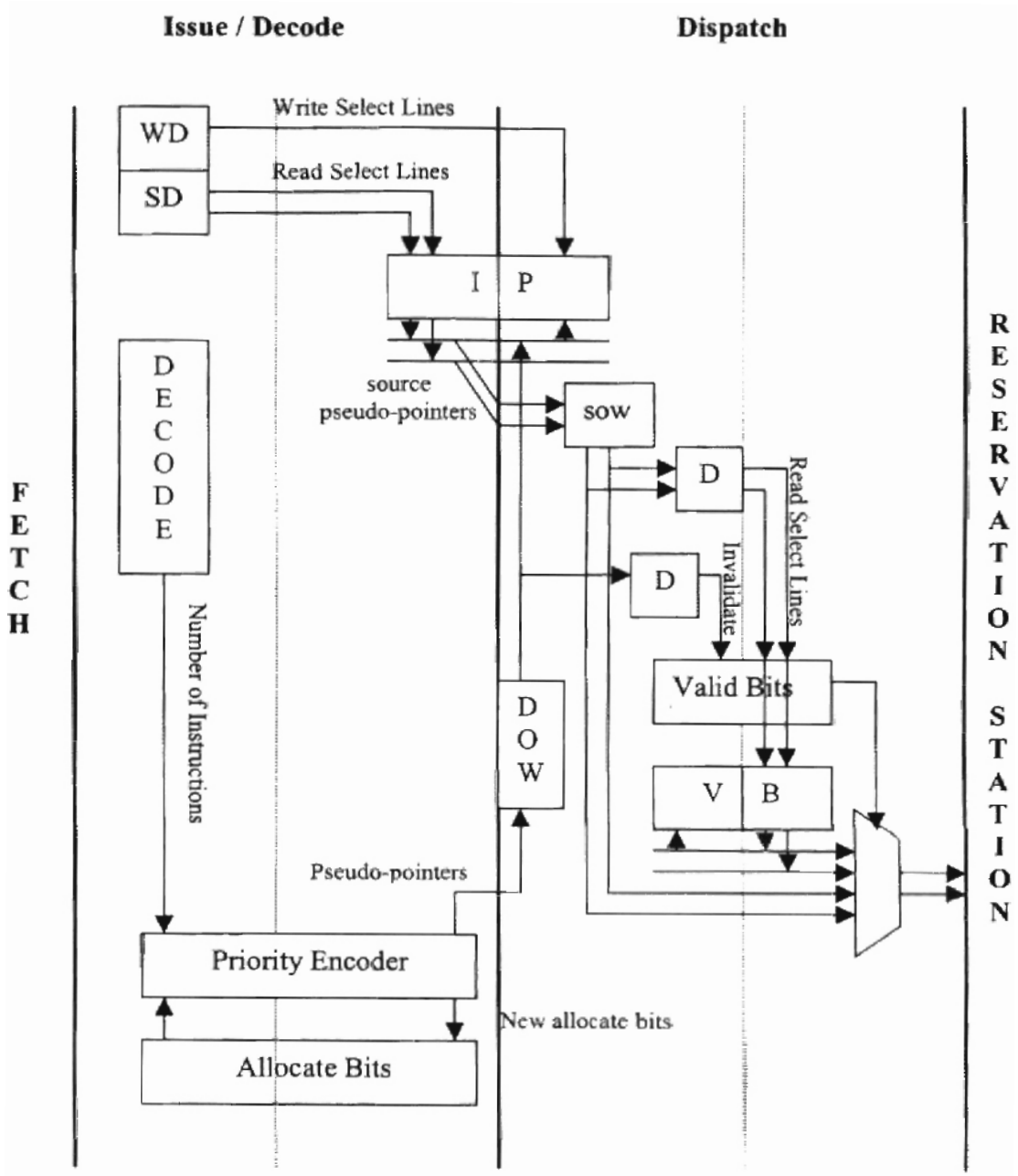


Figure 2.3 Register Renaming in Issue and Dispatch

Valid Bits indicate if the instruction, for which the VB location was assigned, has completed writing back the result (indicated by '1'), or if it is still being evaluated in some pipeline stage (indicated by '0'). In the Dispatch stage, the valid bits of the VB locations, from which source operands are read, are checked. If the Valid Bit is '1', then the operand is ready. If the Valid Bit is '0', the pseudo-pointer is passed to the reservation station in place of the read operand. The instruction waits in the reservation station and snoops common data bus for data. Valid Bit is reset to '0' when that VB location is allocated as a new destination for a logical register.

Commit Bits indicate if the instruction for which that VB location was assigned has been determined to be a correct instruction and if its pseudo-pointer has been written into CP buffer (indicated by '1'). Therefore, during an exception (for example: branch misprediction), only the data with commit bits set to '1', need to be preserved. During a restore (recovery from branch misprediction), the Commit Bits are copied into Allocate bits and Valid Bits, as means of restoring status bits. The Commit Bit of a VB location is reset to '0' when a later instruction with the same logical destination register completes.

2.4 Dynamic Scheduling

Given unlimited machine resources, the "Data flow limit" is a measure of the performance limit of a processor executing a program [11]. RAW hazards (data dependencies) in the code causes the fundamental limit to 'data flow', as those instructions have to wait until the instruction providing its source operand finishes

execution. A Data flow graph (DFG) is used to graphically represent the dependencies among instructions, as in Figure 2.4. The arrows between instructions indicate the data dependency along with the execution latency of the instruction providing the data. The cumulative latency of the longest dependence chain gives the data flow limit in clock cycles [11].

The DFG assumes an execution latency of one clock cycle for ALU instructions and three clock cycles for multiply instruction. The data flow limit for the fragment of code is 6 clock cycles and there are 6 instructions. Hence the maximum IPC possible is 6/6 = 1 instruction per clock.

```
i1: move  $v1, $to
i2: move  $a3, $t1
i3: lw    $a2, 0($a3)
i4: addiu $a1, $v1, 1
i5: sll   $v0, $a1, 0x2
i6: mult  $t1, $a2, $v0
```

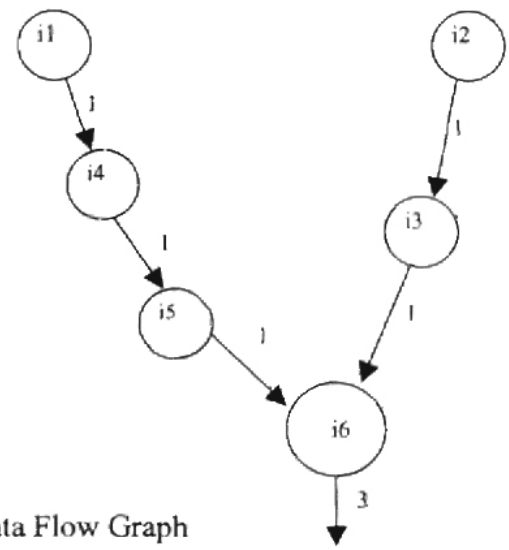


Figure 2.4 Data Flow Graph

Scheduling is one way to leverage ILP to improve overall performance by scheduling instructions with ready operands for execution ahead of data dependent instructions. In dynamic scheduling the hardware decides the order in which the instructions are executed while the program is being run rather than static scheduling by compiler. The primary

objective of the 'Dynamic Schedule' unit is to determine which instructions can proceed to execution at the beginning of the next cycle. In this architecture, the reservation station's issue logic does the scheduling.

'Dispatch' writes instructions into RS, based on the availability of slots in RS. Dispatch, stalls the pipeline until all instructions in a fetch group have been written into the RS. Instructions wait in the RS until all operands required for execution are available. Instructions with ready source operands immediately proceed to execution [11]. This can occur even if it results in out of order execution. If multiple instructions have operands ready, RS prioritizes and schedules the earlier instruction for execution. RS efficiently handles RAW hazards, by snooping the Common Data Bus (CDB) for their operands. A detailed explanation of implementation of dynamic scheduling using RS is given in the next chapter.

2.5 Execution and Write Back

The choice of the type of execution units is based on the ISA. For the MIPS-1 ISA, with only integer operations allowed the instruction type's fractions are as follows. About 40% of the instructions form the ALU instructions, 22% are Branch/Jump instructions, 15% are Multiply/Divide instructions and the rest 23% are Load/Store instructions.

Multiply/Divide instructions require optimized hardware for good performance and the pipeline latency for these instructions tend to be greater than other instructions (latency of

3 in this architecture). Hence, Multiply/Divide instructions are processed by a separate functional unit (MDU). Load/Store instructions require two pipeline stages (address generation and memory access) for their execution. Latency of ALU and Branch/Jump instructions are reduced by not forcing them to go through the memory access stage required for Load/Store instructions. Hence, Load/Store instructions are processed by a separate function unit (LSU).

For a typical program approximately 40% of the instructions are ALU instructions, 20% are Branch/Jump instructions and 40% are Load/Store instructions [11]. Hence, for a four wide superscalar processor we will have, 1.6 ALU instructions, 0.80 Branch/Jump instruction, and 1.6 Load/Store instructions. A single Load/Store Unit (LSU) is used instead of two LSU because of the complexity of designing multiple data memory interfaces [11]. Hence, we have 2 Arithmetic and Logic Units (ALU), 1 Branch /Jump Unit (BJU), 1 Load /Store Unit (LSU) and 1 Multiply /Divide Unit (MDU).

During the Write Back stage, the functional units write their results in the common data bus (CDB), broadcast it to the snooping RS for a tag match and also update the Value Buffer (VB) with the data. Depending on instruction mix and data dependency variation during the program execution, the contention for the CDB will vary over the duration of program execution. These CDBs are loaded heavily by the snooping RS. Hence, they need to be carefully designed so that the 'Write Back' stage does not become the bottleneck for clock speed. The optimum number of CDBs can be determined by performance simulations and analyzing the average percentage utilization of CDB over

the simulation of the program. In this design, two CDBs are provided to write back the results of two instructions simultaneously.

2.6 Commit

Instructions finish execution out of order and wait in the ROB until all previous instructions are committed. Commit handles instructions in program order. It is accomplished by implementing the Reorder Buffer (ROB) as a first-in first-out (FIFO). All fetched instructions, excluding NOPs, are written into ROB during the second half of the Dispatch stage. The valid bit for the corresponding ROB location is set to '1'. This valid bit indicates if the ROB content is a valid instruction.

In the first half of every cycle, ROB reads the first four instructions in the queue. For register write instructions, the commit logic checks the valid bit of the instruction's pseudo-pointer indexed VB location, as shown in Figure 2.5, to determine if that instruction has completed. For non-register write instructions like store and branch, ROB itself has a completion bit indicating if it has completed. Non-register write instructions update the completion bit in ROB upon execution. Implementation details of the ROB are presented in the next chapter.

An instruction read from the ROB is determined to be complete if the valid bit of that instruction and all instructions preceding it in the commit group are set to '1'. Figure 2.5 shows a commit group size of two for simplicity. The same organization will apply for a

commit group of four by adding extra read and write ports to the modules. Once an instruction is determined to have been completed, the CP is updated with the new pseudo-pointer of the completing instruction, if it is a register write instruction. This is done in 'Commit-2' stage. De-allocation of registers must take place in this stage. The pseudo-pointers which are being replaced, "Old pseudo-pointers", are read from CP and those locations in VB are de-allocated and de-committed (allocate and commit bits set to '0' respectively).

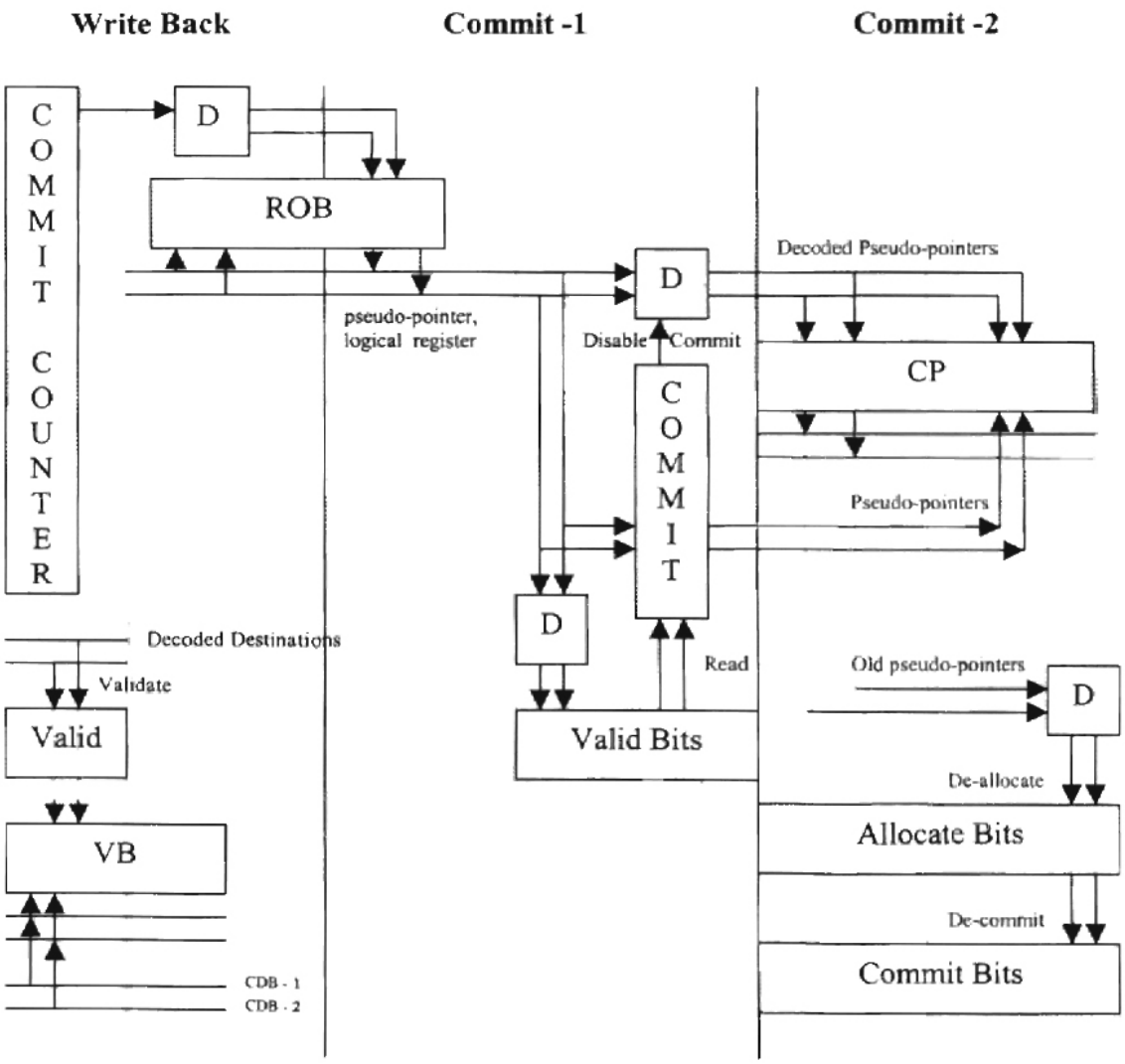


Figure 2.5 Instruction Commit Organization

Commit Logic checks for branch misprediction among the instructions read from ROB. If it encounters a mispredicted branch which has completed, it waits till its delay slot instruction completes and other instructions in the fetch group updates the CP with the new pseudo pointer and then it initiates recovery by signaling a 'Restore'.

During the 'Restore' cycle, the contents of CP are copied into IP through a special internal write port. The valid bits of all ROB locations are reset (set to '0') preventing incorrect instructions from updating CP. Thus, instructions executed due to wrong speculation never update the CP (incorrect instructions not allowed to update the system state). CP may be considered as the mapping table for correctly executed instructions.

Chapter 3

Design and Implementation

This chapter discusses the various implementation details of the hardware modules involved in the register renaming and dynamic scheduling schemes.

3.1 Reorder Buffer (ROB)

The ROB maintains the sequential ordering of the instructions as appearing in the program code, which allows updating the machine state in program order. All fetched instructions are assigned a location in the ROB in the order they appear in the program. As the processor speculatively fetches and executes control dependent code (instructions fetched after branch or jump), speculated instructions must not change the state of the machine (system registers and memory) until it is ensured that the branch or jump instruction was predicted correctly. To accomplish this, we wait until the branch / jump condition is resolved and when the branch / jump commits, a misprediction bit (set in the ROB) is checked. If a misprediction is detected, all instructions following that branch / jump's delay slot instruction are cancelled. If the speculation is correct the subsequent instructions proceed to commit.

The ROB is implemented as a circular buffer implementing a FIFO queue. The Issue Counter adds the fetched instructions to the end of the queue and the Commit Counter

commits instructions from the beginning of the queue. By the process of committing, Commit Logic frees up locations in ROB for new instructions fetched by issue. Figure 3.1 shows the functional operation of the ROB implementing a FIFO queue.

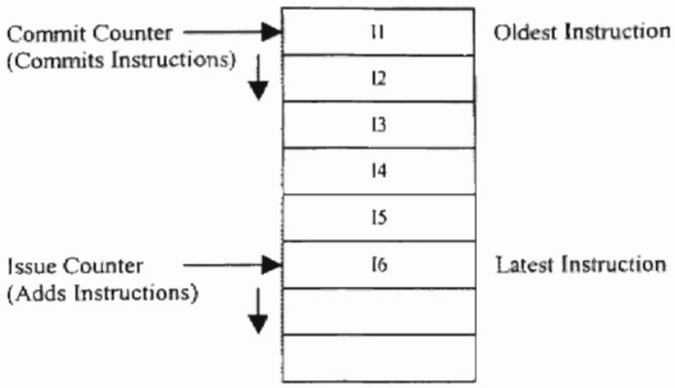


Figure 3.1 FIFO Reorder Buffer

If the Issue Counter or Commit Counter reaches the end of the ROB it wraps around and starts adding or committing instructions from the beginning of the ROB respectively. At any point of time during a program run, the number of occupied ROB locations determines the number of instructions in flight. This number is called the Instruction Window. Ideally, it should be greater than the sum of reservation station entries, latencies of the individual execution pipelines, size of the store buffer and size of the dispatch buffer [2]. The number of reservation station entries is 20. The sum of latency of all execution pipelines is 8. The size of the store buffer is 10. The size of the dispatch buffer is 4. The number ROB entries must be greater than 42. In this design, the ROB size is 64.

Every ROB entry has the fields shown in the Figure 3.2. The valid bit 'V', is set to '1' when an instruction is issued and reset to '0' during restore. 'Code' identifies the type of

instruction as, Register write: 000, Non-register write: 001, Link instruction: 010, Branch/Jump: 100, Store: 111. This is required by the commit logic to determine the condition of completion. 'Pred', holds the 2-bit prediction bits for branch/jump instructions, set at the time of completion of branch/jump instruction. The value of 'Pred' is 00, for non branch/jump instructions, 11, for jump instructions and is whatever the prediction algorithm generates, for branch instructions. The 'Logical Dest', is required to select the 'CP' location to write the destination pseudo-pointer. The 'Rename-Pointer', is the rename register number given to the destination register of the instruction. 'DS' is set, if the branch/jump instruction had a delay slot instruction. 'BA' and 'BTA' holds the branch instruction's address and target address respectively. 'M' is set, if the branch/jump instruction is mispredicted. The 'C' is set at the time of completion of non-register write instructions and link instructions.

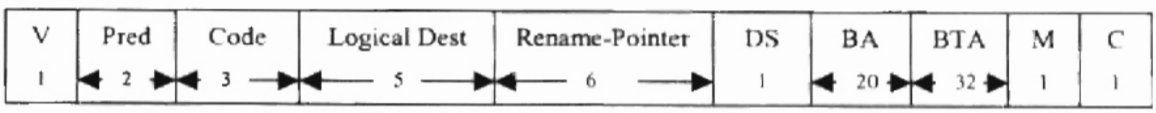


Figure 3.2 ROB Field Format

The ROB is organized as a multiported SRAM with 4 read / write and 3 write ports. ROB 'read' is in the first half and 'write' is in the second half of clock cycle. The number of write ports is determined by the size of the fetch group (to write all fetched instructions into the ROB) and the number of execution units handling non-register write, branch/jump instructions (to set C, Pred in ROB). The number of read ports is determined by the maximum number of instructions designed to be committed per cycle. To avoid

frequent stalls due to lack of space in the ROB, the number of instructions processed for commit must be at least equal to the size of the fetch group.

Figure 3.3 shows one ROB cell modeled as a latch in the RTL description of the design. The buses I1_Bus, I2_Bus, I3_Bus, I4_Bus are time multiplexed Read / Write buses and B_Bus, M_Bus, L_Bus are write buses required to set the complete, misprediction, target address, prediction fields in the ROB, for branch/jump, multiply/divide and load/store instructions respectively.

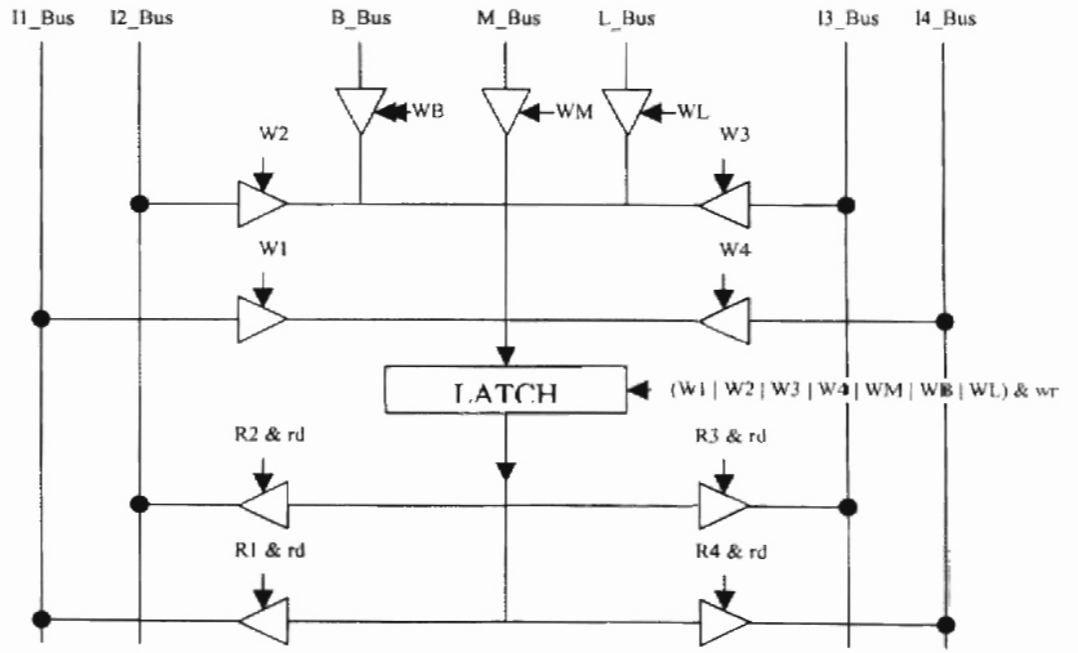


Figure 3.3 RTL Design of a ROB Cell

The functional organization of ROB and Commit logic is described in the following paragraphs. The 'Issue Counter' IC, is used to generate the write decode lines of W1, W2, W3, W4 (four consecutive locations of ROB) in the 'Decode/Issue' stage. These

decoded lines are used to select the ROB locations, in the second half of the 'Dispatch'. The 'Issue Counter' is incremented by the number of instructions (NI) fetched, determined by the Instruction Set Architecture Decode (ISAD). The Figure 3.4 shows the write operation of ROB for two cycles of instruction fetches.

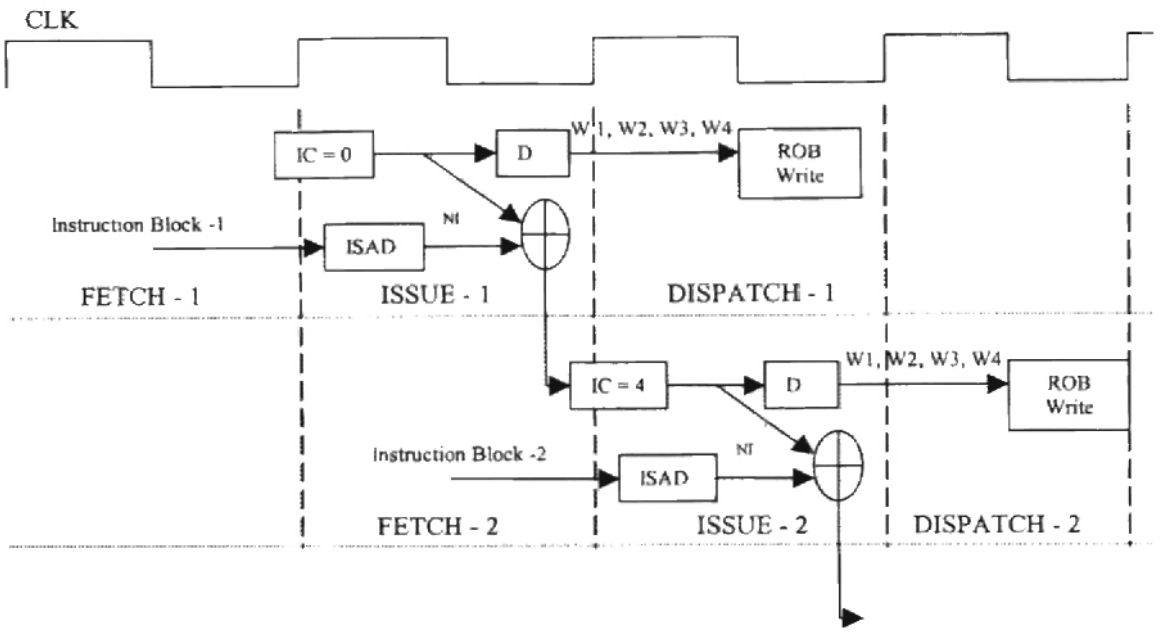


Figure 3.4 ROB Write Operation

The 'Issue Counter' is implemented as a Finite State Machine (FSM) governed by the state transition diagram shown in Figure 3.5. The four states are Reset State (RS), Increment State (IS), Restore State (ReS), Stall State (SS). In the 'RS' state IC is reset to 0. In the 'IS' state, IC is incremented by the number of instructions fetched (NI). In the 'Res' state, IC is set to the Commit Counter (CC). This results in the canceling of uncommitted instructions between IC and CC. A stall occurs, when the number of instructions fetched, is more than the space available in ROB. When stall occurs, ROB

waits for instructions to commit to free up space in the ROB. In this state, IC remains same value until the stall condition is resolved.

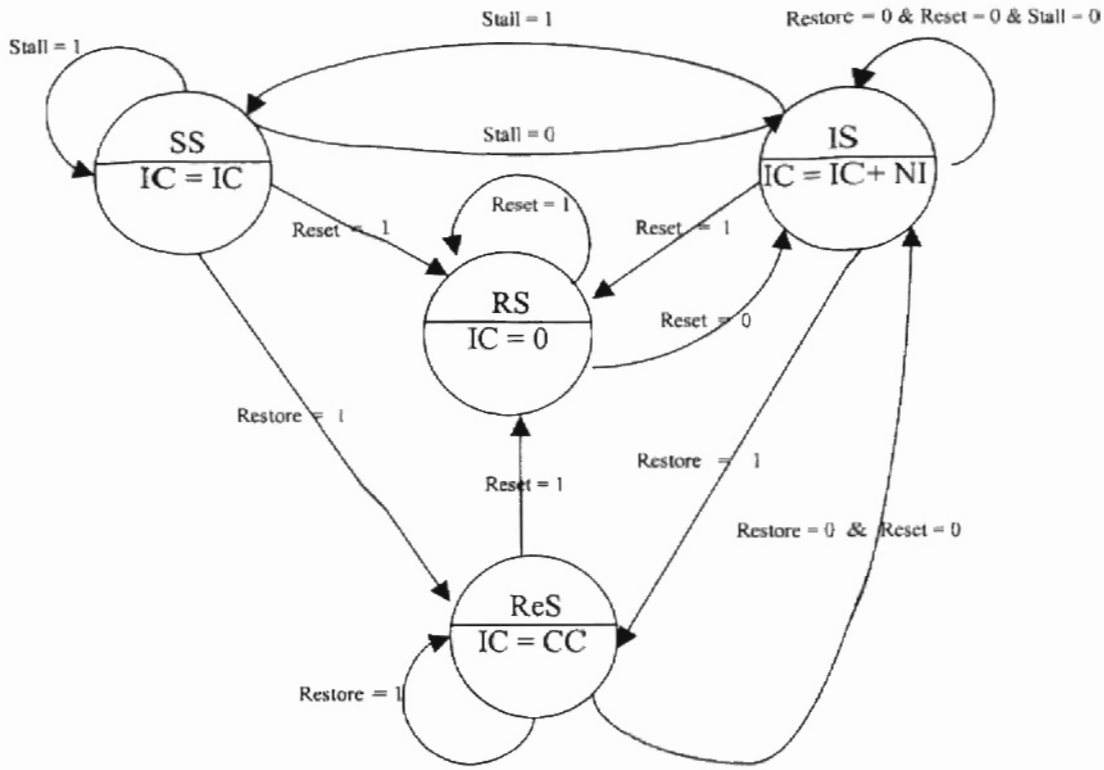


Figure 3.5 Issue Counter State Transition Diagram

The ROB read operation for committing instructions is very similar to the write operation. The Commit Counter (CC) is used to select four consecutive instructions for commit, every cycle. The Commit Logic checks the valid bits for these instructions and determines the number of consecutive instructions that can be committed (NC). CC is incremented by the value of NC. If 'Restore' occurs, CC remains the same value. On 'Reset', CC is reset to '0' and all valid bits are reset to '0'. The functional operation of the process of commit is shown in Figure 3.6. Decode pre-decodes eight select lines for

read. Depending on NC, the 8 to 4 multiplexer selects four consecutive decoded lines for the read operation in the next cycle.

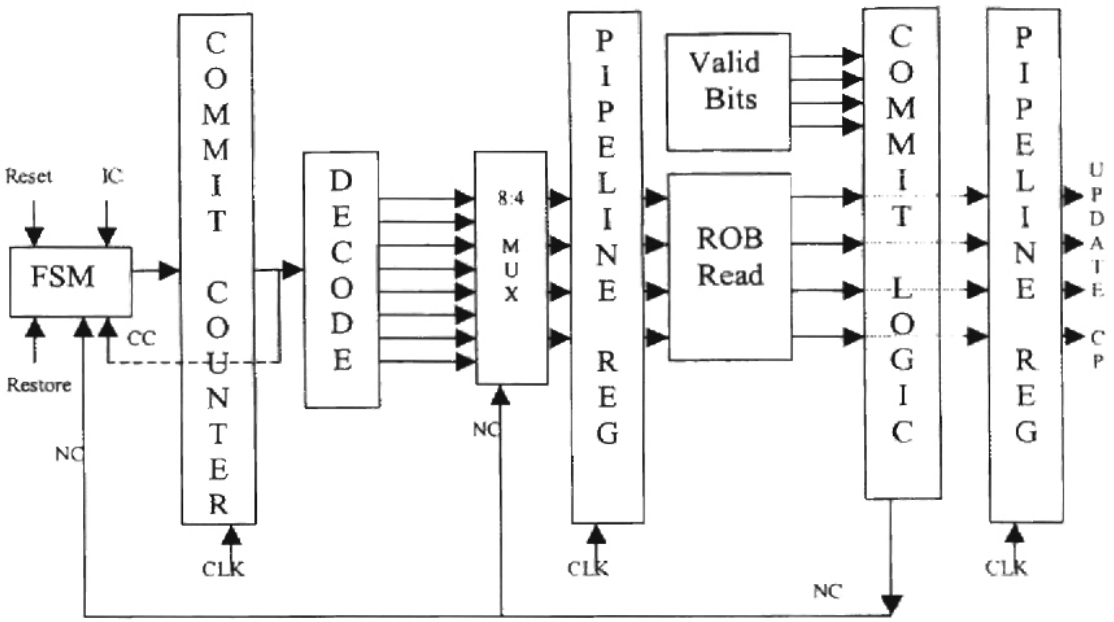


Figure 3.6 ROB Read and Commit Operations

The Commit logic generates NC based on the number of consecutive instructions read from valid ROB locations which have completed execution. If a mispredicted branch is detected in the middle of a commit group, Commit Logic generates control signals to cancel instructions after the mispredicted branch, and keep them from updating CP in the next cycle. Also, Commit has built in logic to allow only one branch to commit in a commit group. This is because the BPB has one write port to update the prediction bit of branch / jump.

The wrap around behavior of the ROB occurs naturally with this scheme, as long as the size of the ROB is some power of 2. In this design, the ROB size is $2^6 = 64$. The size of the Issue Counter and Commit Counter is 6 bits. With finite, 6 bits for representing a value between 0 to 63, once the value crosses 63 the overflow bit is ignored and the values wraps around to 0.

3.2 Status Bits

The 'Status Bits' are comprised of Allocate bits, Valid Bits and Commit Bits. Each of these Status Bits has one to one correspondence to a Value Buffer location and is used for bookkeeping purposes in the register renaming scheme. Allocate Bits indicate, whether a VB location is free to be assigned as a rename-pointer to the destination register of an instruction in the fetch group. Valid Bits indicate if that VB location contains valid data. Commit Bits indicate whether the instruction for which the rename-pointer was assigned, has been committed. Table 3.7 describes the possible states which the VB location can take.

Each Status Bit is modeled as a 1-bit loadable register with decode and write/read operations done in a half clock cycle. This enables simultaneous read / write operations to many bits. In the Allocate Bits, every cycle some bits are allocated (set to '1'), depending on instructions fetched and some are de-allocated (set to '0'), depending on instructions committing. It is guaranteed that allocation and de-allocation will be for different bits since, only already de-allocated bits will be considered for allocation.

negative clock edge. The Logic determines the value to be loaded into the register depending on the various conditions discussed above.

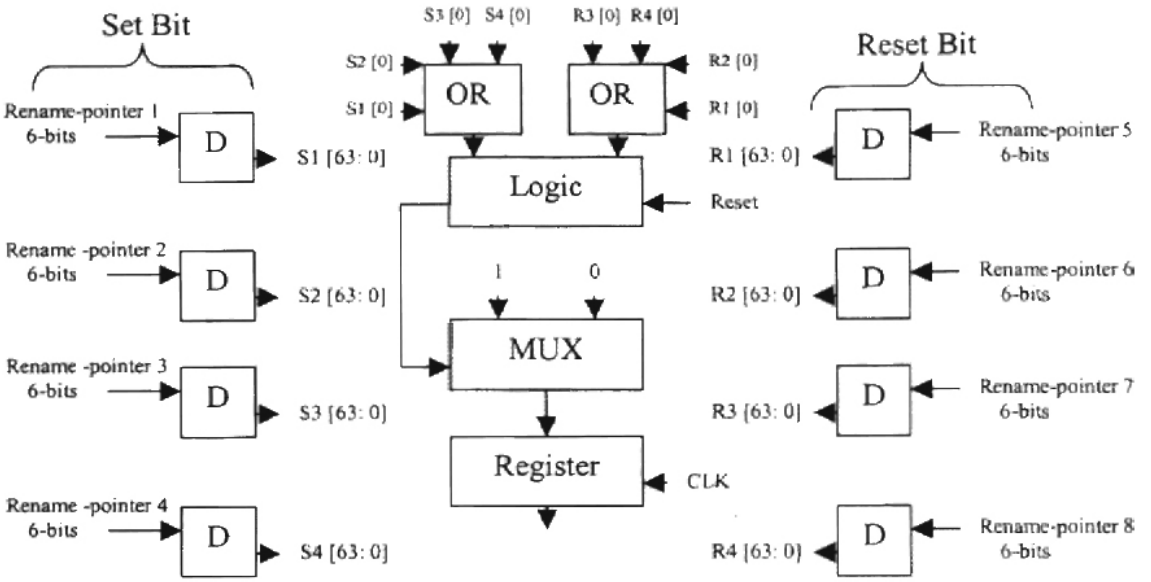


Figure 3.8 Status Bits

The 'Logic' sets or resets the status bit depending on the decoder's output. During Reset, all status bits are set to '0', except the first Allocate bit and Valid Bit which are always set to '1'. During 'Restore', the value of the corresponding Commit Bit is copied to the Allocate and Valid Bits. All Decoders have a disable signal which drives its output to all zeros. There is external logic (not shown) which controls this disable signal, if the number of bits to be set or reset is less than four. For Allocate Bits, the set logic is replaced by pre-computed allocate bits generated by the priority encoder. 'Logic' selects this value to be clocked in every clock.

3.3 Dispatch

Dispatch is the stage in the pipeline where the instructions are assigned a Reservation Station (RS). The read for source operands from the Value Buffer (VB) is done before writing the instructions into the RS. Since a distributed RS scheme is chosen (reason explained in next section), dispatch has to decide the RS to which instructions must be forwarded, depending on the type of instruction and availability of space in the RS. If dispatch cannot assign a RS for all the instruction in that group, it stalls the pipeline until a RS is assigned to all instructions. Since RS has one write port, only one instruction can be dispatched into the RS in a cycle.

The organization of 'Dispatch' is as shown in the Figure 3.9. The register renamed instructions (RR_I1.....RR_I4) are allowed to enter the 'Dispatch' if all instructions in the previous fetch have been assigned a RS. Every register renamed instruction has functional unit code (fu_code), operands and their valid bits (op1, v1, op2, and v2), execution code (exe), pseudo-pointer (dest), reorder-buffer position (reo) and dispatched bit (disp). The 'disp' bit indicates if the instruction has been dispatched. Initially when a new fetch group is processed, 'disp' is set to '0' for all instructions. Every cycle, depending on instructions being written to RS, the 'disp' bit is set to '1' by the 'Update Pending' (UP) logic, as shown in Figure 3.9. Only instructions with 'disp' set to '0' are considered for dispatch into RS by the dispatch logic.

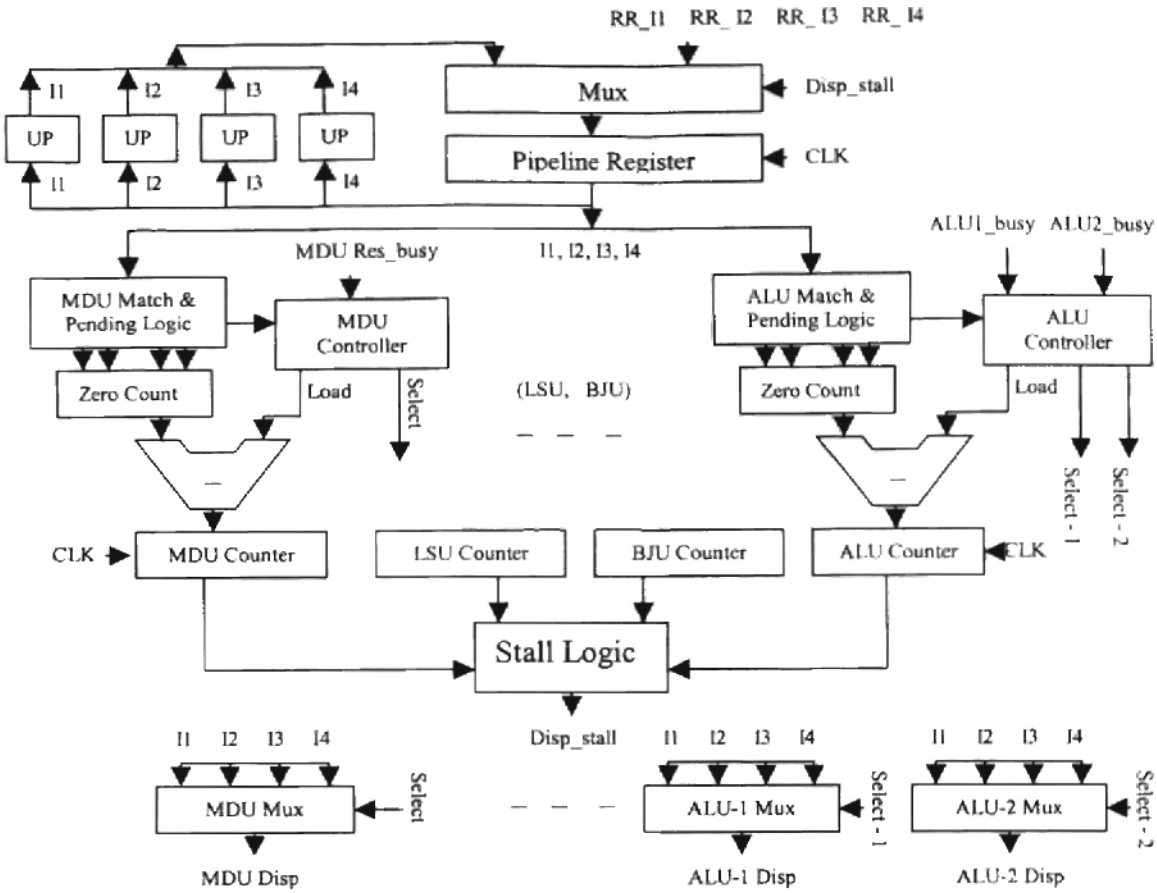


Figure 3.9 Dispatch Organization

Every functional unit has a 'Match and Pending Logic' (MPL), 'Zero Count' (ZC), '3-bit Subtractor' (Sub) and 'Controller' (C). The MPL identifies the instructions which belong to that functional unit which are waiting to be dispatched and indicates those instructions by a '0' in its output. The ZC logic counts the zeros and gives a 3-bit value. Controller determines if an instruction can be assigned a RS by checking the busy bits of the RS. It sets the 'load' signal to '1', if there is an instruction for that functional unit which is waiting to be dispatched and if there is an empty slot in the RS for that functional unit. The 'Sub' subtracts those values and its output value is registered in a negative edge

sensitive register, 'Counter', at the middle of clock. In the second half of clock cycle the 'Counter's' value is the number of instruction to be dispatched in the next cycle for that functional unit.

The 'Stall Logic' issues a dispatch stall if all functional units counter value is not zero. This results in the reprocessing of the same group of instructions next cycle. Since the 'UP logic' sets the 'disp' bits of already dispatched instructions to '1', those are not considered for dispatch when they are processed again due to a dispatch stall.

3.4 Reservation Station

The Reservation Station (RS) is the critical hardware component to incorporate dynamic scheduling in high performance processors. RS may be considered as a temporary storage for instructions, until their source operands become available. As soon as an instruction's operands are available it can be scheduled for execution. By this behavior, RAW hazards are efficiently handled by the RS. The are two major schemes of implementing a RS are distributed and central reservation stations. In the distributed reservation station scheme, every functional unit has its own dedicated RS (smaller number of entries); whereas in the centralized scheme, a common RS (larger number of entries) is used for holding the instructions of all functional units. The distinct advantage of a distributed RS (used in this design) is that, the complexity of the issue logic, which decides the next instruction to be executed from a pool of instructions, is reduced, as the decision is made from a smaller pool of instructions [9]. Also, since the determination of where the instruction needs to be

executed is already done in the dispatch stage, the speed and simplicity of distributed RS is advantageous compared to the centralized scheme. The main advantage of the centralized RS over distributed RS is that the RS space is more efficiently utilized. Since no entry of the centralized RS is permanently reserved for a type of instruction, centralized RS handles the variation in instruction mix better than a distributed RS, and so can reduce dispatch stalls, occurring due to the non-availability of RS entries.

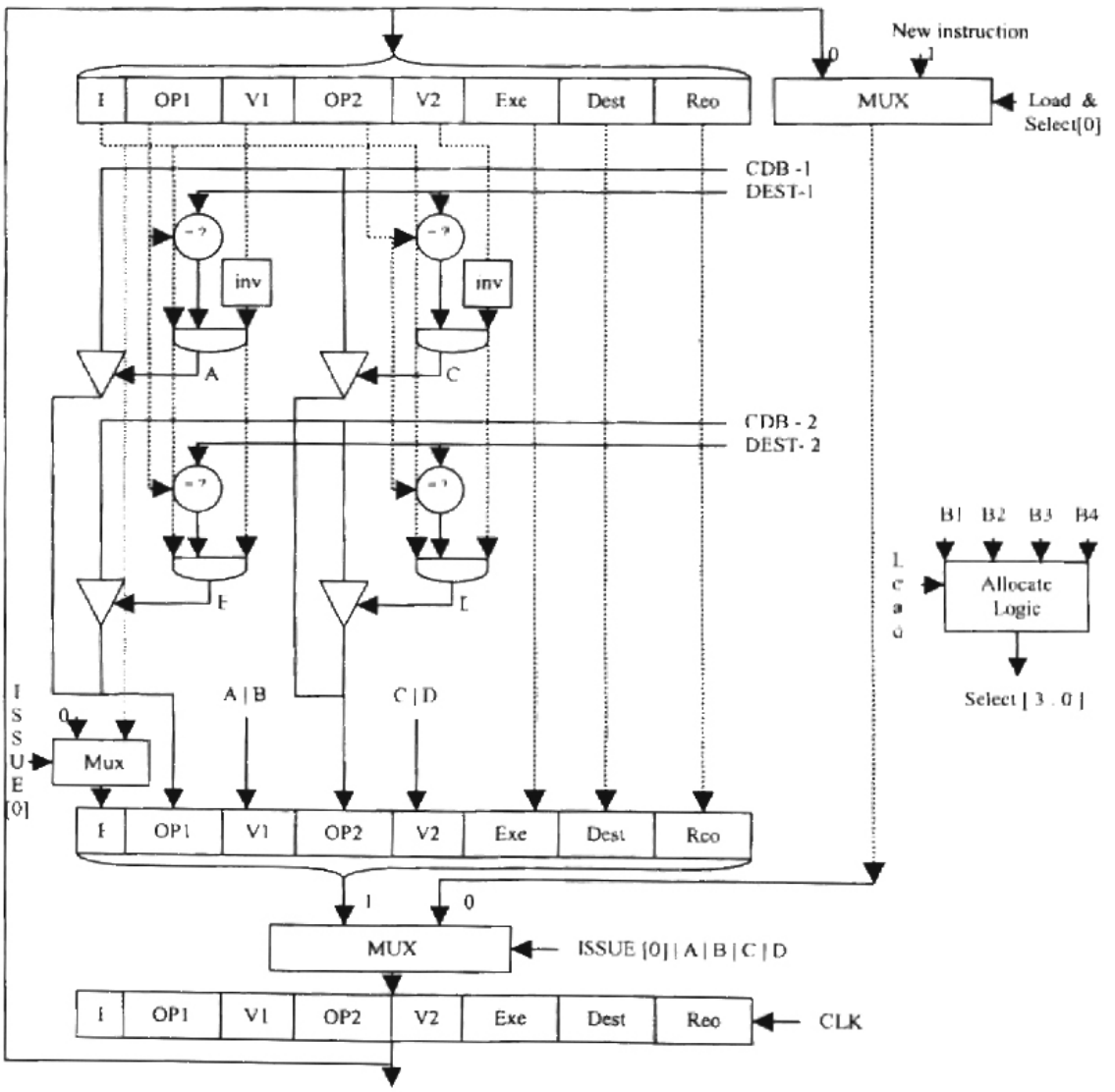


Figure 3.10 Reservation Station Allocate and Wait

RS must allocate an entry for the instruction, wait until its operands are ready, issue an instruction whose operands are ready and free up the entry for future instructions. A RS entry has a busy bit 'B', operands and valid bits (OP1, V1, OP2 and V2), rename-pointer (Dest), execution code (Exe) and the instruction's position in ROB (Reo). Figure 3.10 shows the implementation of the allocate and wait circuitry of the reservation station. 'Allocate Logic' checks the busy bits of all the entries. If 'load' signal is set to '1' by dispatch, it selects the earliest non-busy RS entry for writing the instruction. The busy bit 'B' is set to '1' when an instruction is written into the RS, and it remains the same until it is issued for execution. A RS entry must be retained until a new instruction is written in that location. RS entry must also check all the common data buses (CDB) for operands. This is done my comparing the source rename-pointer (last 6 bits of OP) with the destination rename-pointer broadcasted along with the data in the CDB's. If there is a match of rename-pointers and if RS entry is busy and waiting for operands, data is copied from the CDB into the RS entry and the corresponding valid bit is set to '1'.

The issue logic has the task of selecting one instruction for execution from all instructions in the RS. Only instructions with ready operands can be considered for issue. If multiple instructions have operands ready, then the earliest instructions among them is scheduled for execution. Since the ROB is a wrap around buffer, one cannot readily identify the earliest instruction using the position of an instruction in the ROB. While writing instructions into the ROB, if the Issue Counter (IC) wraps around, then later instructions in that fetch group will be allocated a lower ROB position compared to the earlier ones in the fetch group. To solve this problem, the Commit Counter (CC) position is used as a

reference and then the relative displacement of the instructions with respect to CC is found by subtracting every eligible instruction's 'Reo' from 'CC'. In unsigned binary subtraction with finite bits for output representation, subtracting a larger value (>CC) from an original value (CC) gives a result, which is greater than the result obtained when subtracting a smaller value (<CC) from that original value (CC). Thus, the position of the greatest of the subtracted results will give the position of the instruction to be scheduled for execution at the beginning of next clock cycle, as shown in Figure 3.11. Also, if all entries of RS are empty and an incoming instruction has all operands ready, issue logic directly issues the incoming instruction for execution without allocating an entry in RS. This is done by adding a second level of multiplexers to choose the incoming instruction and canceling the internal 'Load' signal for the 'Allocate Logic'.

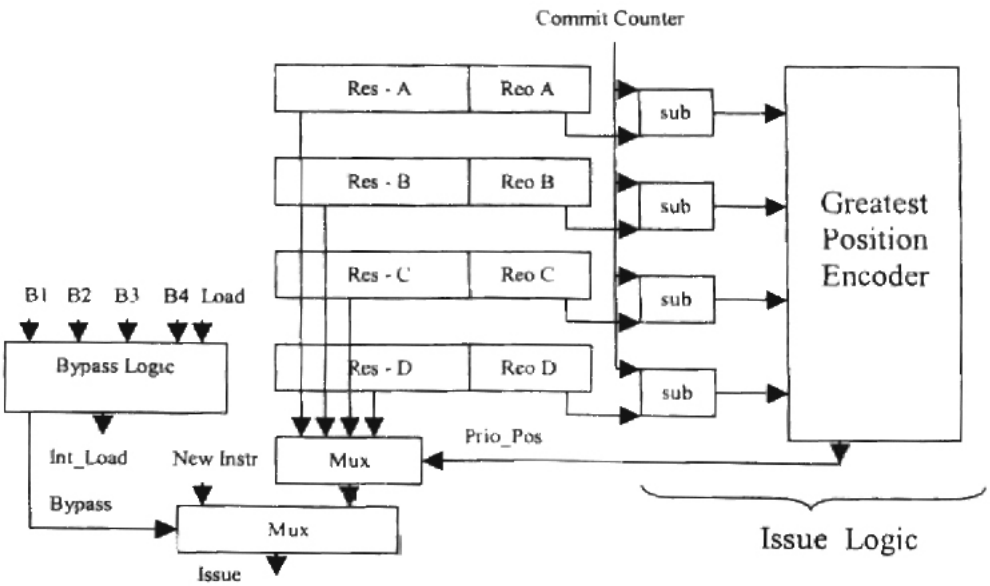


Figure 3.11 Bypass and Issue Logic

Due to the delay in the issue logic instructions are issued into a functional unit delayed by one clock, from the cycle it had both the operands ready. This is shown in Figure 3.12.

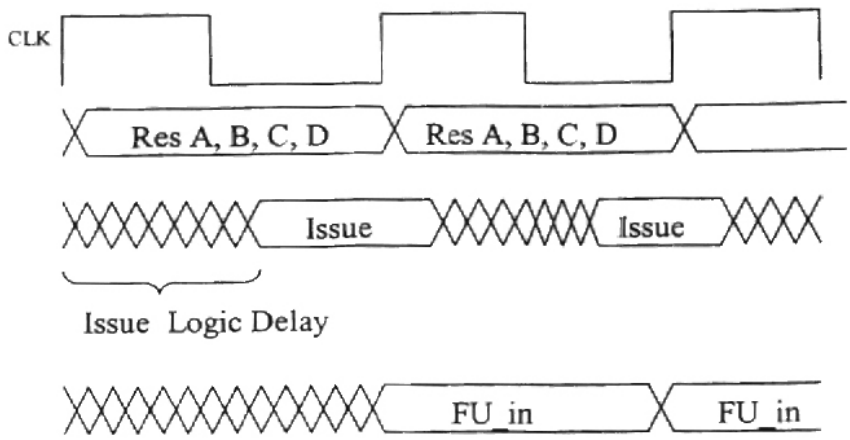


Figure 3.12 Issue Logic Timing

3.5 Execution Units

As mentioned in the previous chapter, this architecture has one MDU, two ALUs, one BJU and one LSU. All the execution units are simulated at a high level of abstraction. The MDU is a three stage pipeline with results in the HILO buffer. The LSU is a two stage pipeline and has a store buffer and a data memory interface. All other execution units are single stage. The instructions are provided to the execution units every cycle by the issue logic of the reservation station. A typical input to an execution unit consists of the two source operands and the execution code 'exe' which indicates the operation to be done with the operands. For some instructions like Jump and Link (JAL), the MIPS ISA computes the target address using an immediate 26-bit instruction index. In those cases, the reservation station has additional storage space for the instruction index and provides it to the execution unit when the instruction is issued.

3.5.1 Multiply and Divide Unit (MDU)

The MDU is simulated as a three stage pipeline and the computed 64-bit result is written into a wrap around buffer called the HILO buffer. As the ROB does not store the result of MDU, this HILO buffer is required to recover from a wrongly speculated multiply or divide instruction. In the MIPS-1 ISA, only MFHI and MFLO instructions write results to the general purpose registers (transfers 32-bit data from HILO to GPR). Other MDU instructions write result to HILO Buffer. When the MDU instruction commits, the corresponding entry in the HILO buffer is set to complete and all other entries are invalidated. If restore occurs due to branch misprediction, all entries except the currently valid and completed entry in the HILO buffer are invalidated.

3.5.2 Load and Store Unit (LSU)

The LSU is a two stage pipeline, the first stage being the address generation stage and the second stage being the memory access stage. The loads and stores to memory are done in program order by issuing the instructions in program order into the LSU execution pipeline. This is done to avoid data hazards for memory which might result from out-of order issue. If Load and Store instructions have the same memory reference address, and if load is supposed to read the data put in memory by store, an out-of order issue of load ahead of store would result in a wrong data being read from memory. Hence the issue logic of the LSU reservation station needs to work differently from the issue logic of other reservation stations. Stores write data into the Store Buffer. The store buffer helps

to cancel the wrongly speculated stores (updates to memory). Since the ROB cannot handle non-register updates, the store buffer is needed to maintain integrity of the program sequence being run. If the store buffer is full and if there is a store pending to be written into it, the LSU pipeline controller stalls the pipeline and waits till an entry is freed in the store buffer. Figure 3.13 shows the organization of the LSU pipeline.

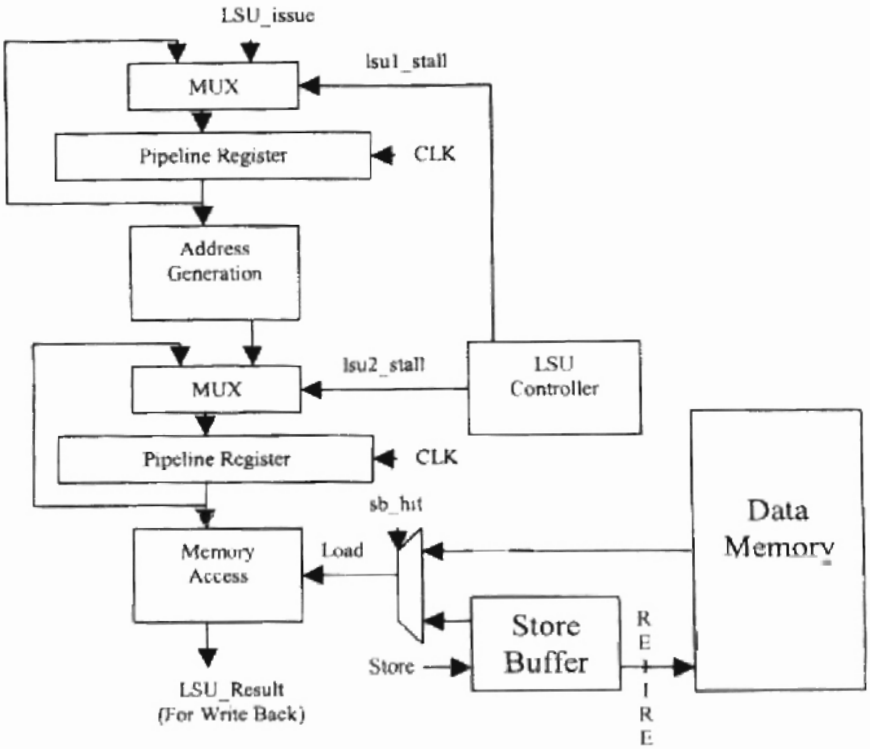


Figure 3.13 Load Store Unit (LSU) Organization

When a store instruction is determined as completed in the ROB, the store buffer controllers set the commit bit in the store buffer for that instruction. Committed stores are written to memory when no load or store instructions are present in the second stage of the LSU. All loads try to read the data from the store buffer. If that address is not present in store buffer it reads it from the memory. Store buffer can be implemented as a small

fully associative cache. In this design the store buffer and memory, along with its interface, are implemented in high level Verilog as arrays.

3.6 Write Back

The responsibility of the write back logic is to allocate the common data bus for functional units, to write results back into the physical register file (Value Buffer). If multiple functional units compete for the CDB the write back logic prioritizes and allocates the CDB to the earliest instruction. The prioritization logic is the same as discussed in 'Dispatch'. If a functional unit was not allotted the CDB in a cycle, the pipeline for that functional unit is stalled, and the instruction is processed for allocation in the next cycle. Other pipelines operate as normal during that stall. Write back logic works in the last stage of the functional unit pipeline. Write back also pre-decodes the write select lines for the destination rename-pointers. All register-write instructions have a write back bit (WB) initially set to '1'. This means the instruction is waiting for CDB allocation. Write back logic considers only those instructions which have WB set to '1' for prioritization. Figure 3.14 shows the organization of Write back with all its components.

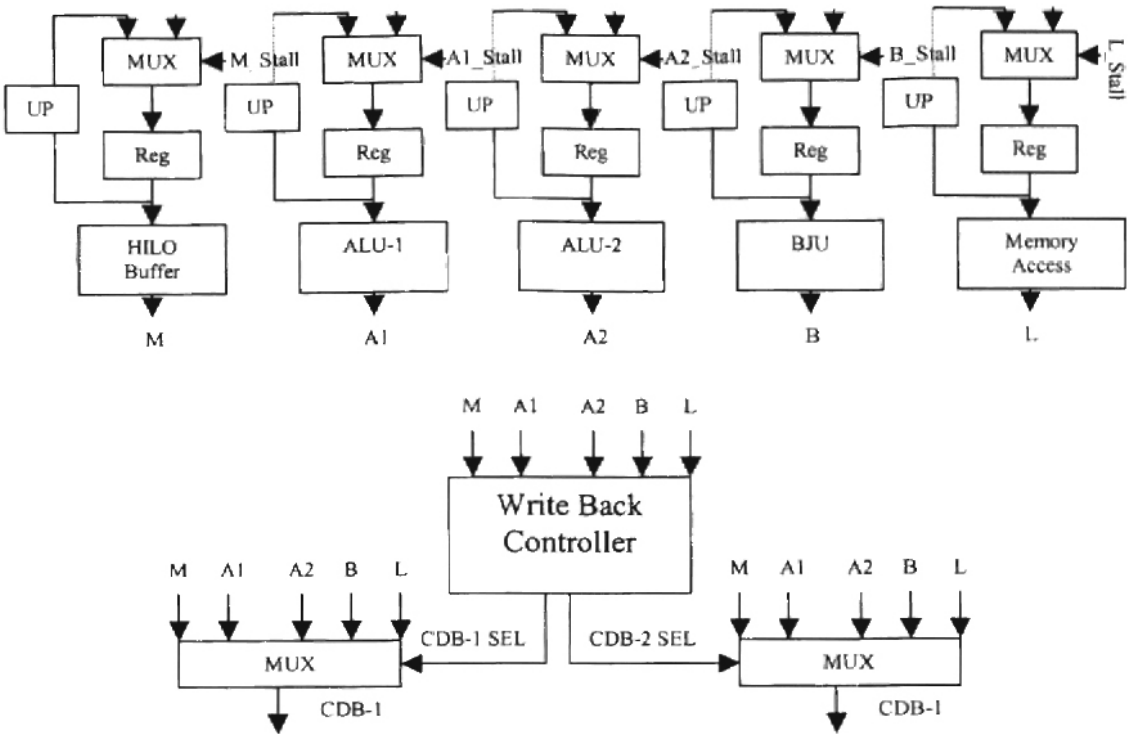


Figure 3.14 Write Back Organization

In Figure 3.14 M, A1, A2, B, L are the results which have the fields 'reo' (reorder buffer position), 'wb-dest' (pre-decoded write select lines), 'data', 'dest' (rename-pointer assigned to the instruction), 'wb' (write back bit). The write back controller prioritizes using the 'reo' of all instructions and allots the CDB to the two earliest instructions waiting for write back. The write back controller generates the stall signal if an instruction waiting for write back is not allotted the common data bus (CDB). UP logic sets the 'wb' bit to '1' if the pipeline is stalled.

Chapter 4

Verification and Results

4.1 Simulation Environment

This section describes the simulation environment and the tools involved in the simulation of the architecture. The tools involved in the simulation are the SDE-MIPS tool kit and Verilog.

4.1.1 SDE-MIPS Tool Kit

The SDE-MIPS tool kit is a GNU C cross-development compiler for MIPS CPUs. It also has a MIPS Simulator with a graphic interface, which can be used for debugging purposes. The SDE-MIPS tool is used for instruction code generation i.e., compiling 'C' programs into assembly level code targeted for the MIPS-1 ISA. The tool kit generates an executable file, compiled into the virtual address space. The command for compiling the program into MIPS-1 ISA is "sde-make SBD= GSIM1B". Compiler flags for code optimizations and turning off floating point instructions can be set in the 'makefile', associated with the 'C' program being compiled. Refer to [13] for more information on compiling in the SDE-MIPS tool kit. The tool kit compiles the executable code in the Executable and Linkable format (ELF) and the executable file has the following sections:

a) .text : holds the executable instructions of the program.

b) .data : contains initialized program data.

c) .rodata : contains read-only data (strings, constants) of program.

d) .sdata : contains initialized data of size less than 'n bytes'. Set by (-G n) flag.

The tool kit also has an 'object dump' utility for ASCII output of executable binaries. The commands for using this utility are "sde-objdump –d exefile >> dump", for disassembly of '.text' and "sde-objdump –s exefile >> datadump", for '.data' section. The '.text' section consists of a 'main' subroutine (compiled program), '__start' (initialization routine) and various other C routines provided by the run-time system.

In the MIPS architecture, sp-relative (stack pointer) addressing is used for accessing local variables in a subroutine's stack. Hence, the initialization routine must initialize the 'sp' register to an upper limit (word aligned) address before running 'main'. The '__start' module is modified to allocate stack space for a program by initializing the sp register to the upper (word aligned) memory address and then making a jump and link (JAL) to 'main'. If the program is self contained for its data (No I/O) and the -G option is turned off, this initialization is enough to start executing the program [10]. Refer to [10] for requirements on starting an application in the MIPS environment. Figure 4.1 shows the memory map layout of a C program compiled using the SDE-MIPS tool kit. The PC is initialized to the beginning of a modified 'start' routine in the .text segment.

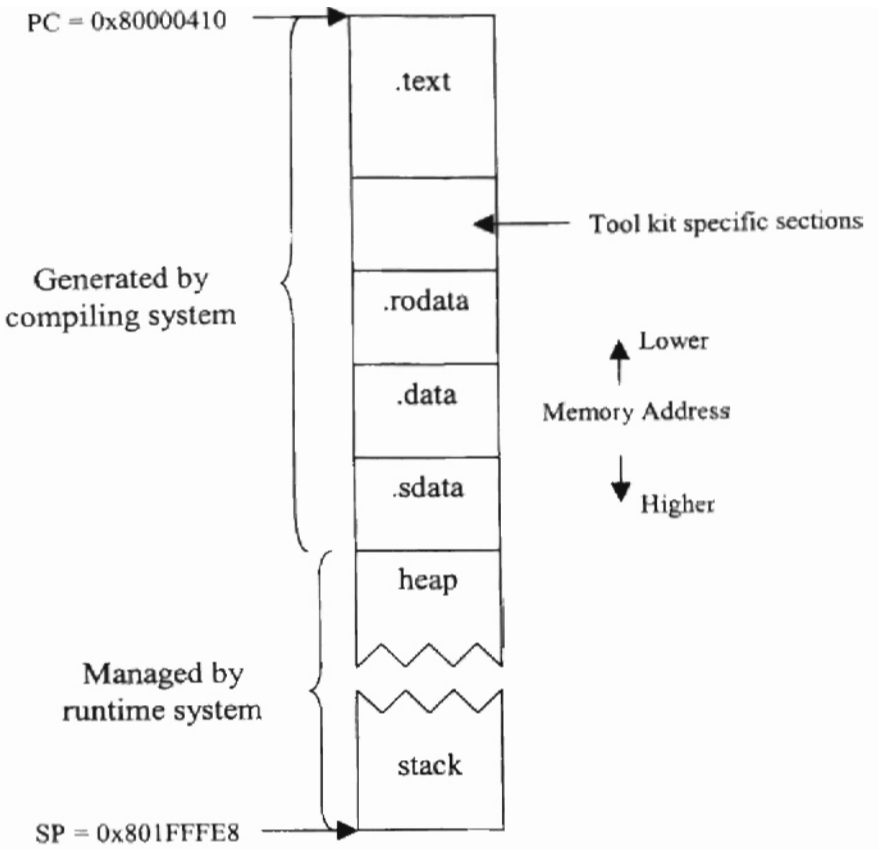


Figure 4.1 SDE-MIPS: Program Memory Map

4.1.2 Verilog Simulation

Verilog is an event driven simulator used for hardware description and verification. It supports structural simulation and behavior simulation. Structural simulation is usually done for timing verification and it is mapped to a technology library. Behavior simulation is done for functionality verification and is independent of technology. Formal

verification of this design is done both structurally and behaviorally. Structural verification is done for individual modules to meet timing requirements and the overall integrated design is simulated behaviorally for functional verification. After obtaining the assembly code from the compiled binary using the 'sde-objdump' utility, it is converted to an ASCII binary file so that it can be loaded into a memory array in verilog. Two separate C programs 'iparser.c' and 'dparser.c' are written to do this conversion. The steps to generate the ASCII binary files are as follows:

1) Compile and run 'iparser.c' with the 'dump' file in the same directory to generate ASCII binary files 'imem' and 'opcode'. These files are to be placed in the 'Binaries' folder in the Fetch directory.
2) If the program does not make any reference to contents in .rodata, .sdata or .data sections during program run, the files 'dmem' and 'data' in the D-MEM folder in the LSU directory can be initialized with data memory addresses till stack pointer and zeros respectively. If not, compile and run 'dparser.c' with the 'datadump' file to generate ASCII binary files 'dmem' and 'data'.

Verilog has an inbuilt function '$readmemb' to load memory arrays from ASCII binary files. Using this function, the arrays are initialized at the beginning of the simulation using an 'initial' block in verilog. The memory arrays constitute the instruction cache (icache) and data cache to which the memory operations take place. The icache and data cache supply the instructions and data to the processor during program execution.

The program end is reached when the control is transferred back to the '__start' routine from the 'main' after stack unwinding. A verilog test file provides the clock and reset signals to the processor. The clock period is 10ns (100 MHz). This clock period is limited by the worst case delay of 8ns in the priority encoder circuit.

4.2 Design Verification

Design verification was carried out at different levels in the top-down hierarchy. Figure 4.2 shows the components in the three levels of hierarchy. The approach followed for verification was to integrate the modules and then verify functional behavior by simulating the design with a handwritten test bench. Although handwritten test files are inefficient, it enables one to check specific cases to expose design or implementation flaws. Verifications done at the top two levels of hierarchy are described below.

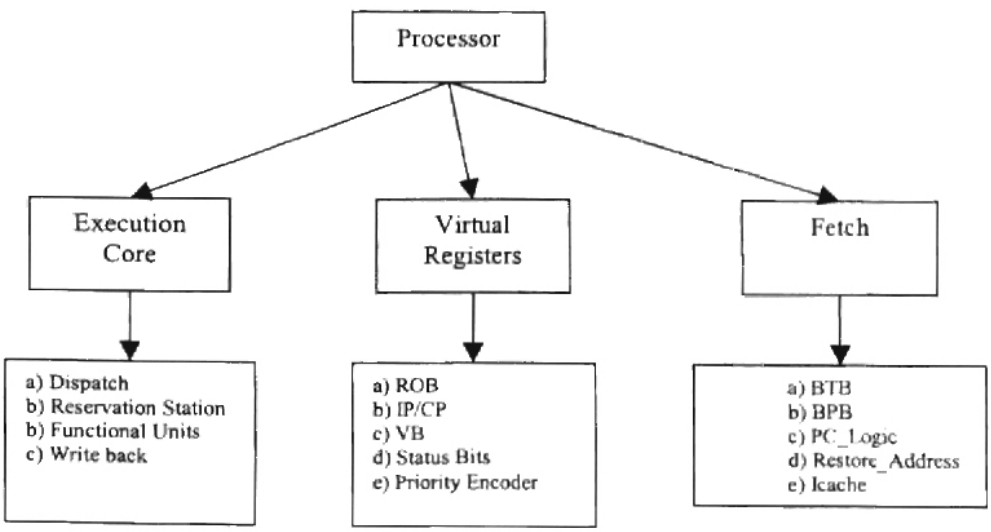


Figure 4.2 Top-Down design hierarchy

Virtual Registers consist of modules involved in register renaming. The objective of this simulation is to verify register renaming in the 'Issue/Decode' and 'Dispatch' stages and register de-allocation and restoring logic in the 'Commit-1' and 'Commit-2' stages. All modules listed under Virtual registers in Figure 4.2 are simulated. The test bench simulates the function of the rest of the processor. Since fetch is not simulated in the hardware for this simulation, the test bench provides Fetch's output to the virtual registers. Refer to Appendix A, for the code which is executed to verify register renaming. In this code fragment, the JAL instruction at 80000424 is mispredicted when it is fetched for the first time since there is no entry in the BTB. Since ten instructions (including the jump delay slot of JAL) have been committed at the time of restoring, Issue Counter and Commit Counter must have the value "001010" immediately after restore. Since there are only six non-identical general purpose registers used as destination registers in the code fragment, in the cycle after restore, status bits must have only those six bits set to '1' apart from the always set bit of first location of VB. Refer to Appendix A, for a cycle by cycle debug trace of the simulation and the snap shots of the ROB outputs leading to 'Restore', due to the misprediction of the JAL instruction.

Design verification of the execution core module is also done. It consists of dispatch logic, reservation stations, functional units and write back logic. The code fragment used to verify Virtual Registers is again used in the execution core's verification. The dynamic scheduling logic must issue the earliest instruction among the ones which have ready data into the execution unit. This can be verified by observing the contents of the reservation station for two successive cycles. The issued instruction's entry in the

reservation station will be invalidated by setting the 'Busy' bit to zero. This must be done for the earliest entry which has all source operands ready. Also, instructions in reservation stations snoop the CDB for data. Proper implementation of this scheme can be verified by setting of the valid bit to '1' for the waiting instructions, when the 'Dest' tag of CDB matches the lower six bits of the operand field of the instruction.

For functional verification of the complete architecture, system states (registers, program stack) are checked for correct data after program execution. It is guaranteed that, if system states have the correct (expected) data after program execution, the program was properly executed.

A 'Bubble Sorting' program was run till completion and system states were checked, as means of functional verification. 'Bubble sorting' is used to sort a list of data in ascending or descending order. Every iteration of the list, results in one sorted value. Typically, the number of checks required to determine the condition for sorting is proportional to n^2, where n is the number of data being sorted. Data to be sorted are loaded into an array. All data loaded into the array are stored in the 'main' subroutine's stack (meant for local variables). Read and update to those data are done using load and store operations. Due to the nature of the bubble sorting algorithm, data dependencies exist in the machine level code, which reduces the possibility of parallel execution of subsequent iterations of loop. Refer to Appendix B for the C code for sorting.

A loop carried dependency is said to exist, if a load from an array depends on a store to that array location from previous iterations of the loop [9]. In the 'main' function of 'bubble.c', the condition of "if (a [i] < a [j+1])" can be resolved only after the execution of the previous iteration of the "for" loop. This is because a[i] set in the previous loop iteration is required to resolve the condition. Thus loop carried dependency exists in the code. Two differently compiled versions of the same 'bubble.c' program are used for verification. The first version is compiled with the (-O3) loop-unrolling option and the other is compiled with (-O2) loop-unrolling turned off. Since loop carried dependency exist, we cannot expect any significant improvement in performance due to loop-unrolling.

The program has a routine which checks the correctness of sorting and returns '1' upon correct sorting. This value is loaded in the last position of the sorted value. The snap shot of the contents of the program stack before and after the sorting is shown below:

a) Main subroutine 'stack' data before sorting

Address: 801FFFB0 Data: 00000000000000000000000000000000
Address: 801FFFB4 Data: 00000000000000000000000000000000
Address: 801FFFB8 Data: 00000000000000000000000000000000
Address: 801FFFBC Data: 00000000000000000000000000000000
Address: 801FFFC0 Data: 00000000000000000000000000000000
Address: 801FFFC4 Data: 00000000000000000000000000000000
Address: 801FFFC8 Data: 00000000000000000000000000000011
Address: 801FFFCC Data: 00000000000000000000000000010000
Address: 801FFFD0 Data: 00000000000000000000000000000100
Address: 801FFFD4 Data: 00000000000000000000001010011110
Address: 801FFFD8 Data: 00000000000000000000000000111011
Address: 801FFFDC Data: 00000000000000000000000000000000
Address: 801FFFE0 Data: 10000000000000000000010000010000
Address: 801FFFE4 Data: 00000000000000000000000000000000

Address: 801FFFE8 Data: 00000000000000000000000000000000

b) Main subroutine 'stack' data after sorting

Address: 801FFFB0 Data: 00000000000000000000000000000000
Address: 801FFFB4 Data: 00000000000000000000000000000000
Address: 801FFFB8 Data: 00000000000000000000000000000000
Address: 801FFFBC Data: 00000000000000000000000000000000
Address: 801FFFC0 Data: 00000000000000000000000000000011
Address: 801FFFC4 Data: 00000000000000000000000000000000
Address: 801FFFC8 Data: 00000000000000000000010100111110
Address: 801FFFCC Data: 00000000000000000000000000111011
Address: 801FFFD0 Data: 00000000000000000000000000010000
Address: 801FFFD4 Data: 00000000000000000000000000000100
Address: 801FFFD8 Data: 00000000000000000000000000000001
Address: 801FFFDC Data: 00000000000000000000000000000000
Address: 801FFFE0 Data: 10000000000000000000010000010000
Address: 801FFFE4 Data: 00000000000000000000000000000000
Address: 801FFFE8 Data: 00000000000000000000000000000000

As seen from the snap shot of the program stack, the values are sorted to descending order correctly, and the result module returned '1' which is loaded in the address 801FFFD8, the last location in the array, proving the functional correctness of the architecture and its implementation. Table 4.1 presents the performance data collected from the simulation.

The second performance test done is to highlight the potential improvement in performance that can be attained in the presence of parallelism in an executing code fragment. The program computes the sum of the square of numbers in a continuous series. Refer to 'Appendix C' for the C code. Because there is no loop carried dependency in the subsequent iterations of the loop, parallelism exists among instructions

of different loop iterations. This program was executed with an IPC >1. This performance improvement cannot be achieved without register renaming and dynamic scheduling. In a simple linear pipeline, the presence of WAR and WAW hazards due to register name dependency will slow down the processor. Table 4.2 presents the performance data collected from the simulation.

4.3 Performance Data Collection

The performance metrics used for evaluation are Instructions Per Clock (IPC), Misprediction Rate (MR) and Average Percentage Utilization (APU). A Verilog module "data_collector.v" is written to collect and determine these metrics during the program run. In a superscalar processor, because of the parallel pipelines, it is difficult to represent individual stalls in the IPC equation. Moreover, because of speculation, we cannot be sure if those stalls are caused by real or speculated instructions. Hence, IPC can only be determined by the average number of instructions completing from the ROB over the duration of program execution. Since the instructions completing from ROB are all real instructions, we get an accurate estimate of IPC. Therefore IPC is given as follows:

$$IPC = \frac{\sum_{i=1}^{tc} NC_i}{tc}$$

Where, NC_i -- number of instruction commits in the cycle

tc -- total number of clock cycles.

The Range of NC_i is from 0 to commit group size. Typically, commit group size is made the same as the fetch group size. IPC cannot possibly exceed the average number of instructions fetched. Hence, the maximum IPC which one can achieve is the size of fetch group.

The Misprediction Rate (MR) gives an estimate of how often branch or jump instructions are wrongly speculated. Since no useful processing is done with the speculated instructions after a mispredicted branch or jump, IPC is reduced. This is because the average number of correct instructions completing decreases, due to the need for a restore. Restore flushes all the instructions in the pipeline, and hence NC_i for many cycles after restore is '0', until correct instructions fill up the pipeline and start to complete. The Misprediction rate (MR) is estimated as follows:

$$MR = \frac{\sum_{i=1}^{tc} RS_i}{\sum_{i=1}^{tc} BJ_i}$$

Where, RS_i -- Indicates Restore. RS_i is '1' if Restore = 1 else '0'.

BJ_i -- Indicates Branch or Jump instruction complete.

BJ_i is determined from completing branches and jumps, so that, speculated branch and jumps do not account for MR. Lower MR is required for high IPC. This MR is affected by the efficiency of the branch prediction algorithm.

The Average Percentage Utilization (APU) is a metric which can be used for optimization purpose. It indicates the utilization of a hardware module during a program

run. Here, APU for ALU reservation station (ALU_RS) and Common Data Buses (CDB) are calculated. There are 2 ALU_RS with four entries each and 2 CDB. The APU of ALU_RS, CDB are calculated as follows:

$$APU_{ALU\_RS} = \frac{\sum_{i=1}^{tc}(NAB_i / 8)}{tc}$$

Where, NAB_i -- Number of busy ALU RS entries.

tc -- total number of clock cycles.

$$APU_{CDB} = \frac{\sum_{i=1}^{tc}(X_i)}{tc}; X_i = \begin{cases} 1, pwb_i \geq 2 \\ nwb_i / 2, pwb_i < 2 \end{cases}$$

Where, X_i -- Percentage of CDB utilized.

pwb_i -- Number of Pending write-backs.

nwb_i -- Number of write-backs.

tc -- total number of clock cycles.

Table 4.1 summarizes the above discussed performance metrics of the implemented architecture for the programs run to gather performance data. All simulations were run until the completion of the program (i.e., until the PC jumps back to the '_start' routine). The information required to compute the performance metrics are probed at the middle of the second half of the clock, every cycle. The lower IPC in the 'Bubble' program is due to the inherent loop level dependency in the code. Higher than normal MR are registered for 'Bubble' program because, many branch decisions are based on the data being sorted and hence tough to predict. Considerable improvement in IPC is registered for 'SSS'

compared to what would have been achieved if run on a linear pipelined processor. This is due to the register renaming and dynamic scheduling implemented in this architecture.

Metric	Bubble (-O3)	Bubble (-O2)
Total clock cycles (*tc*)	311	360
No of Dispatch Stalls	134	152
Average No. Commits (IPC)	0.5980	0.5401
Misprediction Rate (MR)	42.42%	52.77%
Average RS occupancy (ALU)	35.04%	23.23%
Average RS occupancy (LSU)	50.88%	38.50%
CDB contention Stalls	4	4
Average CDB Utilization	36.97%	34.62%

Table 4.1 Performance on Bubble Sorting

Metric	SSS (-O3)	SSS (-O2)
Total clock cycles (*tc*)	126	148
No of Dispatch Stalls	38	35
Average No. Commits (IPC)	1.103	1.114
Misprediction Rate (MR)	26.66%	14.28%
Average RS occupancy (ALU)	33.63%	45.52%
Average RS occupancy (LSU)	2.57%	2.91%
CDB contention Stalls	2	24
Average CDB Utilization	44.44%	41.21%

Table 4.2 Performance on Squared Series Sum.

Chapter 5

Conclusion

A register renaming algorithm and scheduling unit for a four wide superscalar processor, for fast misprediction recovery and efficient hardware level implementation was proposed and successfully implemented. Performance results show that, this scheme can yield higher IPC compared to a linear pipelined processor, for programs with inherent parallelism.

Future improvements to this work can focus on extending the design and simulation environment to run longer benchmark programs. This would require, figuring out ways of handling I/O operations in the design and during simulation. A cost-performance tradeoff study for additional Common Data Buses (CDB), write ports in Reservation Station (RS) will also be interesting. For reducing the branch penalty still further, modifications to the current register renaming scheme, for an early update of PC with the correct target address can also be explored.

REFERENCES

1. James E. Smith and Gurindar S. Sohi, "*The Microarchitecture of Superscalar Processors*" in Proceedings of the IEEE. Vol 83, No. 12, December 1995.

2. Dezso Sima, Budapest Polytechnic, "*The Design Space of Register Renaming Techniques*" in IEEE Micro, 2000.

3. Teresa Monreal et al., "*Hardware Schemes for Early Register Release*" in Proceedings of the International Conference on parallel Processing (ICPP'02)

4. Antonio Gonzalez et al., "*Virtual Physical Registers*" in High-Performance Computer Architecture (HPCA), February 1998.

5. Stephan Jourdan et al., "*A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification*", in IEEE 1998.

6. Mayan Moudgill et al., "*Register Renaming and Dynamic Seculation: an Alternate Approach*", in IEEE 1993.

7. Steven Wallace and Nader Bagherzadeh, "*Instruction Fetching Mechanism for Superscalar Microprocessors*", University of California, Irvine, USA.

8. Mansur H. Samadzadeh and Loai E. Garalnabi, "*Hardware/ Software Cost Analysis of interrupt Processing Strategies*", in IEEE Micro, June 2001.

9. David A. Patterson, John L. Hennessy, David Goldberg, "*Computer Architecture: A Quantitative Approach*" 2nd Edition, Morgan Kaufman Publishers, January 1996.

10. Dominic Sweetman, "*See MIPS Run*", Morgan Kaufmann; 1st edition, April 1999.

11. John Paul Shen and Mikko H. Lipasti, "*Modern Processor Design: Fundamentals of Superscalar Processors*", Beta Edition, McGraw Hill, 2003.

12. Anshuman Anand, ECEN, Oklahoma State University, "*Architecture Verification of four-wide superscalar Processor*", Masters Thesis, December 2003.

13. Algorithmics SDE-MIPS/Free GNU toolkit 4.0 c, "*Programmer's Guide*", Revision: 1.42, 2001 Algorithmics Ltd.

Appendix A
Design Verification of Virtual Registers

CODE FRAGMENT

This code fragment is used to verify the register renaming scheme. When JAL instruction at 80000424 is fetched for the first time, there are no entries in the BTB and BPB. Hence it will be predicted as not taken. But when the JAL is evaluated in the execution unit, it will be realized that it was mispredicted. Moreover, since it is a link type unconditional jump with a delay slot, the restore must occur only after the delay slot instruction and all instructions above it have finished execution.

Pseudo-pointer assigned for instructions (3),(5),(7),(9) will be de-allocated when instructions (4),(6),(8),(11) are committed respectively. Hence after restore, only six pseudo-pointers will remain allocated and will have valid data. This can be verified by the debug trace printed after simulation of the virtual registers. Data of correct instructions which have completed must be preserved after restore for read by future instructions. The source operand read of instructions (36 to 39) must read the latest values written to those registers by instructions before JAL. This can be verified by their source values in the dispatch stage in the last cycle of simulation.

				Result	Pseudo-pointer
1) 80000400:	0000d025	move	$k0,$zero	$k0 = 00000000;	1
2) 80000404:	00000000	nop			
3) 80000408:	3c1c8001	lui	$gp,0x8001	$gp = 80010000;	2
4) 8000040c:	279cf5de	addiu	$gp,$gp,-2594	$gp = 8000F5DE;	3
5) 80000410:	3c088001	lui	$t0,0x8001	$t0 = 80010000;	4
6) 80000414:	2508a4c8	addiu	$t0,$t0,-23352	$t0 = 8000A4C8;	5
7) 80000418:	241dfff8	li	$sp,-8	$sp = FFFFFFF8;	6
8) 8000041c:	011de824	and	$sp,$t0,$sp	$sp = 8000A4C8;	7
9) 80000420:	3c048000	lui	$a0,0x8000	$a0 = 80000000;	8
10) 80000424:	0c000123	jal	8000048c	$ra = 8000042C;	9
11) 80000428:	24846d14	addiu	$a0,$a0,27924	$a0 = 80006D14;	a
12) 8000042c:	039d282a	slt	$a1,$gp,$sp	$a1 = 00000000;	
13) 80000430	10050004	beq	$zero,$a1,x4		
14) 80000434	00000000	nop			
15) 80000438	3c058000	lui	$a1,0x8000	$a1 = 80000000;	
16) 8000043c	00000000	nop			
17) 80000440	00000000	nop			
18) 80000444	24040001	li	$a0,1	$a0 = 00000001;	
19) 80000448:	3c058000	lui	$a1,0x8000	$a1 = 80000000;	
20) 8000044c:	24a56d08	addiu	$a1,$a1, 27912	$a1 = 80006d08;	
21) 80000450:	3c068000	lui	$a2,0x8000	$a2 = 80000000;	
22) 80000454:	24c66d10	addiu	$a2,$a2, 27920	$a2 = 80006d10;	
23) 80000458:		nop			
24) 8000045c:		nop			
25) 80000460:		nop			
26) 80000464:		nop			
27) 80000468:		nop			
28) 8000046c:		nop			
29) 80000470:		nop			
30) 80000474:		nop			
31) 80000478:		nop			
32) 8000047c:		nop			
33) 80000480:		nop			
34) 80000484:		nop			

```
35) 80000488:                   nop
36) 8000048c:   001a2825        move    $a1,$k0         $a1 = 00000000;
37) 80000490:   001c3025        move    $a2,$gp         $a2 = 8000F5DE;
38) 80000494:   001d3825        move    $a3,$sp         $a3 = 8000A4C8;
39) 80000498:   00084025        move    $a4,$t0         $a4 = 8000A4C8;
40) 8000049c:                    nop
```

TEST BENCH

The test bench provides the control signals for the design under test. Fetch and rest of the processor's behavior is simulated by the test bench. The test vectors listed under 'Fetch Outputs' is given to the input of 'Issue/Decode' stage. These vectors are chosen to simulate the behavior of Fetch Logic. The test vectors listed under 'ExeCore Outputs' are chosen to simulate the behavior of dispatch, reservation station, execution units and write back stage.

```
`timescale 1ns/10ps
module VR_testbench( );

        reg  clk;
        reg  reset;

        // Fetch  Outputs
        reg[1:0]  pred, stm_pred[0:19];
        reg[31:0]  pred_addr, stm_pred_addr[0:19];
        reg[31:0]  pc, stm_pc[0:19];            //  Address from which instruction block is fetched.
        reg[127:0] instr_block, stm_instr_block[0:19];  //Instruction block fetched for execution

        // ExeCore Outputs
        reg         disp_stall, stm_disp_stall[0:19];   // Dispatch stall
        reg[3:0]    dispatched, stm_dispatched[0:19]; // Indicates instruction yet to be dispatched by '0'
        reg[101:0]  cdb1, stm_cdb1[0:19];               // Common data bus
        reg[101:0]  cdb2, stm_cdb2[0:19];
        reg[63:0]   brdest, stm_brdest[0:19];           // Decoded write address in ROB
        reg[63:0]   wsm, stm_wsm[0:19];
        reg[63:0]   wsl, stm_wsl[0:19];
        reg[70:0]   brdata, stm_brdata[0:19];           // Branch units output to ROB.
        reg[70:0]   mbus, stm_mbus[0:19];
        reg[70:0]   lbus, stm_lbus[0:19];

        // VR's output
        wire        restore;
        wire        vr_stall;
        wire        rob_stall;
        wire        bj4;
        wire        restore_nc;
        wire[5:0]   no_commit;
        wire[3:0]   rob_valid;
        wire[3:0]   rob_complete;
        wire[5:0]   issuecounter;
        wire[5:0]   commitcounter;
        wire[31:0]  bj4_pc;
        wire[44:0]  I1;             // Decode output for instruction 1
        wire[44:0]  I2;             // Decode output for instruction 2
```

```
wire[44:0]  I3;              // Decode output for instruction 3
wire[44:0]  I4;              // Decode output for instruction 4
wire[63:0]  allocatebits;
wire[63:0]  commitbits;
wire[63:0]  validbits;
wire[70:0]  I1Bus;           // ROB read/write Bus1
wire[70:0]  I2Bus;           // ROB read/write Bus2
wire[70:0]  I3Bus;           // ROB read/write Bus3
wire[70:0]  I4Bus;           // ROB read/write Bus4
wire[85:0]  brinfo_disp;
wire[103:0] rr_i1;
wire[103:0] rr_i2;
wire[103:0] rr_i3;
wire[103:0] rr_i4;

integer k, dummy;
```

```
VR dut (clk, reset, instr_block,  pc, pred, pred_addr, disp_stall, dispatched, cdb1, cdb2,  brdest, brdata,
vr_stall, restore, restore_nc,  issuecounter, commitcounter, rr_i1, rr_i2, rr_i3, rr_i4, I1Bus, I2Bus, I3Bus,
I4Bus, no_commit,  rob_valid, rob_complete, mbus,  lbus, wsm, wsl, rob_stall, bj4, bj4_pc, brinfo_disp,
allocatebits, commitbits, validbits, I1, I2, I3, I4);

initial begin
 $timeformat(-9,1,"ns",12);

 // disp_stall = 1 implies pipeline content in dispatch stage must be retained next cycle
 // contents in issue stage is also retained if number of instructions in issue stage is != 0.

 // Cycle 20
 stm_pred[0]          = 2'b00;
 stm_pred_addr[0]     = 32'h00000000;
 stm_pc[0]            = 32'h00000000;
 stm_instr_block[0]   = 128'h00000000000000000000000000000000;
 stm_disp_stall[0]    = 1'b0;
 stm_dispatched[0]    = 4'b0000;
 stm_cdb1[0]          = 102'h00000000000000000000000000;
 stm_cdb2[0]          = 102'h00000000000000000000000000;
 stm_brdest[0]        = 64'h0000000000000000;
 stm_wsm[0]           = 64'h0000000000000000;
 stm_wsl[0]           = 64'h0000000000000000;
 stm_brdata[0]        = 71'h000000000000000000;
 stm_mbus[0]          = 71'h000000000000000000;
 stm_lbus[0]          = 71'h000000000000000000;

 // Cycle 30
 // First fetch group enters 'Issue/Decode' stage.
 // 3 ALU instructions and 1 NOP.

 stm_pred[1]          = 2'b00;
 stm_pred_addr[1]     = 32'h00000000;
 stm_pc[1]            = 32'h80000400;
 stm_instr_block[1]   = 128'h0000d025000000003c1c8001279cf5de;
 stm_disp_stall[1]    = 1'b0;
 stm_dispatched[1]    = 4'b0000;
 stm_cdb1[1]          = 102'h00000000000000000000000000;
 stm_cdb2[1]          = 102'h00000000000000000000000000;
```

```
stm_brdest[1]       = 64'h0000000000000000;
stm_wsm[1]          = 64'h0000000000000000;
stm_wsl[1]          = 64'h0000000000000000;
stm_brdata[1]       = 71'h000000000000000000;
stm_mbus[1]         = 71'h000000000000000000;
stm_lbus[1]         = 71'h000000000000000000;

// Cycle 40
// 'Dispatch Stall = 1' due to 3-ALU instructions in dispatch stage.
// Second fetch group enters 'Issue/Decode' stage.
// 4 alu instructions.

stm_pred[2]         = 2'b00;
stm_pred_addr[2]    = 32'h00000000;
stm_pc[2]           = 32'h80000410;
stm_instr_block[2]  = 128'h3c0880012508a4c8241dfff8011de824;
stm_disp_stall[2]   = 1'b1;
stm_dispatched[2]   = 4'b1010;
stm_cdb1[2]         = 102'h000000000000000000000000000;
stm_cdb2[2]         = 102'h000000000000000000000000000;
stm_brdest[2]       = 64'h0000000000000000;
stm_wsm[2]          = 64'h0000000000000000;
stm_wsl[2]          = 64'h0000000000000000;
stm_brdata[2]       = 71'h000000000000000000;
stm_mbus[2]         = 71'h000000000000000000;
stm_lbus[2]         = 71'h000000000000000000;

// Cycle 50
// Second fetch group processed again in 'Issue/Decode' stage.
// All first fetch group instructions dispatched

stm_pred[3]         = 2'b00;
stm_pred_addr[3]    = 32'h00000000;
stm_pc[3]           = 32'h80000410;
stm_instr_block[3]  = 128'h3c0880012508a4c8241dfff8011de824;
stm_disp_stall[3]   = 1'b0;
stm_dispatched[3]   = 4'b0000;
stm_cdb1[3]         = 102'h000000000000000000000000000;
stm_cdb2[3]         = 102'h000000000000000000000000000;
stm_brdest[3]       = 64'h0000000000000000;
stm_wsm[3]          = 64'h0000000000000000;
stm_wsl[3]          = 64'h0000000000000000;
stm_brdata[3]       = 71'h000000000000000000;
stm_mbus[3]         = 71'h000000000000000000;
stm_lbus[3]         = 71'h000000000000000000;

// Cycle 60
// 'Dispatch Stall = 1' due to 4-alu instructions in dispatch stage.
// Third fetch group enters 'Issue/Decode' stage.
// 3 ALU instructions and 1 JAL.
// Instructions (1) and (3) write back result in CDB.

stm_pred[4]         = 2'b01;
stm_pred_addr[4]    = 32'h00000000;
stm_pc[4]           = 32'h80000420;
stm_instr_block[4]  = 128'h3c0480000c00012324846d14039d282a;
```

```
stm_disp_stall[4]   = 1'b1;
stm_dispatched[4]  = 4'b1100;
stm_cdb1[4]         = {64'h0000000000000002,32'h00000000,6'h01};
stm_cdb2[4]         = {64'h0000000000000004,32'h80010000,6'h02};
stm_brdest[4]       = 64'h0000000000000000;
stm_wsm[4]          = 64'h0000000000000000;
stm_wsl[4]          = 64'h0000000000000000;
stm_brdata[4]       = 71'h00000000000000000;
stm_mbus[4]         = 71'h00000000000000000;
stm_lbus[4]         = 71'h00000000000000000;

// Cycle 70
// Third fetch group processed again in 'Issue/Decode' stage
// All second fetch group instructions dispatched

stm_pred[5]          = 2'b01;
stm_pred_addr[5]   = 32'h00000000;
stm_pc[5]            = 32'h80000420;
stm_instr_block[5]  = 128'h3c0480000c00012324846d14039d282a;
stm_disp_stall[5]   = 1'b0;
stm_dispatched[5]  = 4'b0000;
stm_cdb1[5]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[5]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[5]       = 64'h0000000000000000;
stm_wsm[5]          = 64'h0000000000000000;
stm_wsl[5]          = 64'h0000000000000000;
stm_brdata[5]       = 71'h00000000000000000;
stm_mbus[5]         = 71'h00000000000000000;
stm_lbus[5]         = 71'h00000000000000000;

// Cycle 80
// 'Dispatch Stall = 1' due to 3-alu instructions in dispatch stage.
//  Fourth fetch group enters 'Issue/Decode' stage.
//  1 BJU and 1 ALU instruction.

stm_pred[6]          = 2'b01;
stm_pred_addr[6]   = 32'h00000000;
stm_pc[6]            = 32'h80000430;
stm_instr_block[6]  = 128'h10050004000000003c05800000000000;
stm_disp_stall[6]   = 1'b1;
stm_dispatched[6]  = 4'b1110;
stm_cdb1[6]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[6]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[6]       = 64'h0000000000000000;
stm_wsm[6]          = 64'h0000000000000000;
stm_wsl[6]          = 64'h0000000000000000;
stm_brdata[6]       = 71'h00000000000000000;
stm_mbus[6]         = 71'h00000000000000000;
stm_lbus[6]         = 71'h00000000000000000;


// Cycle 90
// Fourth fetch group processed again in 'Issue/Decode' stage
// All Third fetch group instructions dispatched
// Instruction (4) completes and writes data in CDB.
```

```
stm_pred[7]          = 2'b01;
stm_pred_addr[7]     = 32'h00000000;
stm_pc[7]            = 32'h80000430;
stm_instr_block[7]   = 128'h10050004000000003c05800000000000;
stm_disp_stall[7]    = 1'b0;
stm_dispatched[7]    = 4'b0000;
stm_cdb1[7]          = {64'h0000000000000008,32'h8000f5de,6'h03};
stm_cdb2[7]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[7]        = 64'h0000000000000000;
stm_wsm[7]           = 64'h0000000000000000;
stm_wsl[7]           = 64'h0000000000000000;
stm_brdata[7]        = 71'h00000000000000000000;
stm_mbus[7]          = 71'h00000000000000000000;
stm_lbus[7]          = 71'h00000000000000000000;

// Cycle 100
//  Instruction (5) and (10) complete and write result in CDB.
//  JAL instruction writes branch target address and other information
//  in the location allocated  for JAL in ROB

stm_pred[8]          = 2'b00;
stm_pred_addr[8]     = 32'h00000000;
stm_pc[8]            = 32'h80000440;
stm_instr_block[8]   = 128'h00000000240400013c05800024a56d08;
stm_disp_stall[8]    = 1'b0;
stm_dispatched[8]    = 4'b0000;
stm_cdb1[8]          = {64'h0000000000000010,32'h80010000,6'h04};
stm_cdb2[8]          = {64'h0000000000000200,32'h8000042c,6'h09};
stm_brdest[8]        = 64'h0000000000000100;
stm_wsm[8]           = 64'h0000000000000000;
stm_wsl[8]           = 64'h0000000000000000;
stm_brdata[8]        = {2'b11,3'b010,5'b11111,6'h09,1'b1,20'h00424,32'h8000048c,2'b11};
stm_mbus[8]          = 71'h00000000000000000000;
stm_lbus[8]          = 71'h00000000000000000000;

// Cycle 110
//  Instruction (7) completes and writes result to CDB.
stm_pred[9]          = 2'b00;
stm_pred_addr[9]     = 32'h00000000;
stm_pc[9]            = 32'h80000450;
stm_instr_block[9]   = 128'h3c06800024c66d100000000000000000;
stm_disp_stall[9]    = 1'b1;
stm_dispatched[9]    = 4'b0110;
stm_cdb1[9]          = {64'h0000000000000040,32'hffffffff8,6'h06};
stm_cdb2[9]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[9]        = 64'h0000000000000000;
stm_wsm[9]           = 64'h0000000000000000;
stm_wsl[9]           = 64'h0000000000000000;
stm_brdata[9]        = 71'h00000000000000000000;
stm_mbus[9]          = 71'h00000000000000000000;
stm_lbus[9]          = 71'h00000000000000000000;

// Cycle 120
stm_pred[10]         = 2'b00;
stm_pred_addr[10]    = 32'h00000000;
stm_pc[10]           = 32'h80000450;
```

```
stm_instr_block[10] = 128'h3c06800024c66d100000000000000000;
stm_disp_stall[10]  = 1'b0;
stm_dispatched[10]  = 4'b0000;
stm_cdb1[10]        = {64'h0000000000000100,32'h80000000,6'h08};
stm_cdb2[10]        = 102'h00000000000000000000000000;
stm_brdest[10]      = 64'h0000000000000000;
stm_wsm[10]         = 64'h0000000000000000;
stm_wsl[10]         = 64'h0000000000000000;
stm_brdata[10]      = 71'h000000000000000000;
stm_mbus[10]        = 71'h000000000000000000;
stm_lbus[10]        = 71'h000000000000000000;

// Cycle 130
stm_pred[11]        = 2'b00;
stm_pred_addr[11]   = 32'h00000000;
stm_pc[11]          = 32'h80000460;
stm_instr_block[11] = 128'h00000000000000000000000000000000;
stm_disp_stall[11]  = 1'b0;
stm_dispatched[11]  = 4'b0000;
stm_cdb1[11]        = {64'h0000000000000020,32'h8000a4c8,6'h05};
stm_cdb2[11]        = {64'h0000000000001000,32'h80000000,6'h0c};
stm_brdest[11]      = 64'h0000000000000000;
stm_wsm[11]         = 64'h0000000000000000;
stm_wsl[11]         = 64'h0000000000000000;
stm_brdata[11]      = 71'h000000000000000000;
stm_mbus[11]        = 71'h000000000000000000;
stm_lbus[11]        = 71'h000000000000000000;

// Cycle 140
stm_pred[12]        = 2'b00;
stm_pred_addr[12]   = 32'h00000000;
stm_pc[12]          = 32'h80000470;
stm_instr_block[12] = 128'h00000000000000000000000000000000;
stm_disp_stall[12]  = 1'b0;
stm_dispatched[12]  = 4'b0000;
stm_cdb1[12]        = {64'h0000000000002000,32'h00000001,6'h0b};
stm_cdb2[12]        = {64'h0000000000004000,32'h80000000,6'h0e};
stm_brdest[12]      = 64'h0000000000000000;
stm_wsm[12]         = 64'h0000000000000000;
stm_wsl[12]         = 64'h0000000000000000;
stm_brdata[12]      = 71'h000000000000000000;
stm_mbus[12]        = 71'h000000000000000000;
stm_lbus[12]        = 71'h000000000000000000;

// Cycle 150
// Branch Delay-Slot instruction completes and write data to CDB.

stm_pred[13]        = 2'b00;
stm_pred_addr[13]   = 32'h00000000;
stm_pc[13]          = 32'h80000480;
stm_instr_block[13] = 128'h0000000000000000000000000001a2825;
stm_disp_stall[13]  = 1'b0;
stm_dispatched[13]  = 4'b0000;
stm_cdb1[13]        = {64'h0000000000000400,32'h80006d14,6'h0a};
stm_cdb2[13]        = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[13]      = 64'h0000000000000000;
```

```
stm_wsm[13]          = 64'h0000000000000000;
stm_wsl[13]          = 64'h0000000000000000;
stm_brdata[13]       = 71'h000000000000000000;
stm_mbus[13]         = 71'h000000000000000000;
stm_lbus[13]         = 71'h000000000000000000;

// Cycle 160
// Instruction (8) writes result to CDB.

stm_pred[14]         = 2'b00;
stm_pred_addr[14]    = 32'h00000000;
stm_pc[14]           = 32'h80000490;
stm_instr_block[14] = 128'h001c3025001d38250008402500000000;
stm_disp_stall[14]   = 1'b0;
stm_dispatched[14]   = 4'b0000;
stm_cdb1[14]         = {64'h0000000000000080,32'h8000a4c8,6'h07};
stm_cdb2[14]         = {64'h0000000000000004,32'h80000000,6'h02};
stm_brdest[14]       = 64'h0000000000000000;
stm_wsm[14]          = 64'h0000000000000000;
stm_wsl[14]          = 64'h0000000000000000;
stm_brdata[14]       = 71'h000000000000000000;
stm_mbus[14]         = 71'h000000000000000000;
stm_lbus[14]         = 71'h000000000000000000;

// Cycle 170
//  Misprediction is detected

stm_pred[15]         = 2'b00;
stm_pred_addr[15]    = 32'h00000000;
stm_pc[15]           = 32'h800004a0;
stm_instr_block[15] = 128'h00000000000000000000000000000000;
stm_disp_stall[15]   = 1'b1;
stm_dispatched[15]   = 4'b0000;
stm_cdb1[15]         = {64'h0000000000008000,32'h80006d08,6'h0f};
stm_cdb2[15]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[15]       = 64'h0000000000000000;
stm_wsm[15]          = 64'h0000000000000000;
stm_wsl[15]          = 64'h0000000000000000;
stm_brdata[15]       = 71'h000000000000000000;
stm_mbus[15]         = 71'h000000000000000000;
stm_lbus[15]         = 71'h000000000000000000;

// Cycle 180
// 'Restore' occurs in this cycle.
//  Since all pipelines registers are asynchronously reset during 'Restore', all vectors are zero.

stm_pred[16]         = 2'b00;
stm_pred_addr[16]    = 32'h00000000;
stm_pc[16]           = 32'h00000000;
stm_instr_block[16] = 128'h00000000000000000000000000000000;
stm_disp_stall[16]   = 1'b0;
stm_dispatched[16]   = 4'b0000;
stm_cdb1[16]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[16]         = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[16]       = 64'h0000000000000000;
stm_wsm[16]          = 64'h0000000000000000;
```

```
stm_wsl[16]          = 64'h0000000000000000;
stm_brdata[16]       = 71'h000000000000000000;
stm_mbus[16]         = 71'h000000000000000000;
stm_lbus[16]         = 71'h000000000000000000;

// Cycle 190
// Fetch starts fetching from correct Fetch stream (800048C).

stm_pred[17]          = 2'b00;
stm_pred_addr[17]     = 32'h00000000;
stm_pc[17]            = 32'h00000000;
stm_instr_block[17]   = 128'h00000000000000000000000000000000;
stm_disp_stall[17]    = 1'b0;
stm_dispatched[17]    = 4'b0000;
stm_cdb1[17]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[17]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[17]        = 64'h0000000000000000;
stm_wsm[17]           = 64'h0000000000000000;
stm_wsl[17]           = 64'h0000000000000000;
stm_brdata[17]        = 71'h000000000000000000;
stm_mbus[17]          = 71'h000000000000000000;
stm_lbus[17]          = 71'h000000000000000000;

// Cycle 200
// New fetch stream enters 'Issue/Decode' stage.

stm_pred[18]          = 2'b00;
stm_pred_addr[18]     = 32'h00000000;
stm_pc[18]            = 32'h8000048c;
stm_instr_block[18]   = 128'h001a2825001c3025001d382500084025;
stm_disp_stall[18]    = 1'b0;
stm_dispatched[18]    = 4'b0000;
stm_cdb1[18]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[18]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[18]        = 64'h0000000000000000;
stm_wsm[18]           = 64'h0000000000000000;
stm_wsl[18]           = 64'h0000000000000000;
stm_brdata[18]        = 71'h000000000000000000;
stm_mbus[18]          = 71'h000000000000000000;
stm_lbus[18]          = 71'h000000000000000000;

// Cycle 210
// New Fetch stream enters Dispatch stage and reads source data from VB

stm_pred[19]          = 2'b00;
stm_pred_addr[19]     = 32'h00000000;
stm_pc[19]            = 32'h8000049c;
stm_instr_block[19]   = 128'h00000000000000000000000000000000;
stm_disp_stall[19]    = 1'b1;
stm_dispatched[19]    = 4'b0000;
stm_cdb1[19]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_cdb2[19]          = {64'h0000000000000000,32'h00000000,6'h00};
stm_brdest[19]        = 64'h0000000000000000;
stm_wsm[19]           = 64'h0000000000000000;
stm_wsl[19]           = 64'h0000000000000000;
stm_brdata[19]        = 71'h000000000000000000;
```

```
  stm_mbus[19]        = 71'h000000000000000000;
  stm_lbus[19]        = 71'h000000000000000000;

end

always #5 clk = ~clk;  // clock toggles every 5 ns. Clock period is 10ns.

initial begin

    clk     = 1;
    reset   = 1;
    disp_stall    = 0;
    instr_block  = 128'h00000000000000000000000000000000;

    #20
    reset = 0;  // reset is held high for 20 ns. (2 cycles)

    for (k = 0; k <= 19; k = k+1) begin
          // test vectors are assigned to the input
          // ports of the design under test.

          pred          = stm_pred[k];
          pred_addr     = stm_pred_addr[k];
          pc            = stm_pc[k];
          instr_block   = stm_instr_block[k];
          cdb1          = stm_cdb1[k];
          cdb2          = stm_cdb2[k];
          brdest        = stm_brdest[k];
          wsm           = stm_wsm[k];
          wsl           = stm_wsl[k];
          brdata        = stm_brdata[k];
          mbus          = stm_mbus[k];
          lbus          = stm_lbus[k];
          #5
          dispatched  = stm_dispatched[k];
          disp_stall    = stm_disp_stall[k];
          #5;
    end

    instr_block     = 128'h00000000000000000000000000000000;
    pc              = 32'h00000000;
    pred            = 2'b00;
    #10;
    $finish; // ends simulation.
end


always begin
   #6
   $display("*****************Time = %t*****************", $realtime);
   $display("");
   $display("       Allocate    = %b", allocatebits);
   $display("       Valid       = %b", validbits);
   $display("       Commit      = %b", commitbits);

   $display("");
```

```
    $display("      IssueCounter      = %b", issuecounter);
    $display("      CommitCounter  = %b", commitcounter);
    $display("");
    $display("      Disp Stall          = %b", disp_stall);
    $display("      Restore  = %b", restore);
    $display("");
    $display("Stage - 2 : Issue/Decode");
    $display("      Decode Out - 1    = %b", I1);
    $display("      Decode Out - 2    = %b", I2);
    $display("      Decode Out - 3    = %b", I3);
    $display("      Decode Out - 4    = %b", I4);
    $display("");
    $display("Stage - 3: Dispatch");

$display(" I1:  Fu_Code=%b Op1=%h V1=%b Op2=%h V2=%b Exe=%h Dest=%h Reo=%h Disp=%b",
rr_i1[87:85], rr_i1[84:53], rr_i1[52], rr_i1[51:20], rr_i1[19], rr_i1[18:13], rr_i1[12:7], rr_i1[6:1], rr_i1[0]);

$display(" I2:  Fu_Code=%b Op1=%h V1=%b Op2=%h V2=%b Exe=%h Dest=%h  Reo=%h Disp=%b",
rr_i2[87:85], rr_i2[84:53], rr_i2[52], rr_i2[51:20], rr_i2[19], rr_i2[18:13], rr_i2[12:7], rr_i2[6:1], rr_i2[0]);

$display(" I3:  Fu_Code=%b Op1=%h V1=%b Op2=%h V2=%b Exe=%h Dest=%h Reo=%h Disp=%b",
rr_i3[87:85], rr_i3[84:53], rr_i3[52], rr_i3[51:20],  rr_i3[19], rr_i3[18:13], rr_i3[12:7], rr_i3[6:1], rr_i3[0]);

$display(" I4:  Fu_Code=%b Op1=%h V1=%b Op2=%h V2=%b Exe=%h Dest=%h Reo=%h Disp=%b",
rr_i4[87:85], rr_i4[84:53], rr_i4[52], rr_i4[51:20], rr_i4[19], rr_i4[18:13], rr_i4[12:7], rr_i4[6:1], rr_i4[0]);

$display("");
#4;
end

endmodule
```

SIMULATION OUTPUT

The simulation was run till the restoration process initiated by the misprediction of JAL completes. The successful simulation of this code fragment verifies the correctness of the proposed register renaming scheme. It also shows that, the restore cycle is only one cycle and hence, system recovery can be achieved without much performance penalty. In the final Dispatch stage, the source operand values are the ones written by the last correct instruction with that register as the logical destination register. This proves that, the updates to registers done by correct instructions are preserved after 'Restore'. Comments are inserted before each cycle's trace with the hope that the reader can understand the simulation output better.

```
Highest level modules:
VR_testbench
// Reset cycle
*****************Time =       6.0ns*****************

        Allocate     = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
        Valid        = 00000000000000000000000000000000000000000000000000000000000000001
        Commit       = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

        IssueCounter   = xxxxxx
        CommitCounter  = xxxxxx
```

```
        Disp Stall      = 0
        Restore         = x

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Reset cycle
******************Time =      16.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000000000000000000001
        Valid       = 00000000000000000000000000000000000000000000000000000000000000000001
        Commit      = 00000000000000000000000000000000000000000000000000000000000000000000

        IssueCounter    = 000000
        CommitCounter   = 000000

        Disp Stall      = 0
        Restore         = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Instructions of the first fetch group being fetched in 'Fetch' stage.
******************Time =      26.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000000000000000000001
        Valid       = 00000000000000000000000000000000000000000000000000000000000000000001
        Commit      = 00000000000000000000000000000000000000000000000000000000000000000000

        IssueCounter    = 000000
        CommitCounter   = 000000

        Disp Stall      = 0
        Restore         = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
```

```
        Decode Out - 3   = 00000000000000000000000000000000000000000000000000
        Decode Out - 4   = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// First fetch group in Decode stage.
*****************Time =      36.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000000000000000000000001
        Valid        = 00000000000000000000000000000000000000000000000000000000000000000001
        Commit       = 00000000000000000000000000000000000000000000000000000000000000000000

        IssueCounter  = 000000
        CommitCounter = 000000

        Disp Stall    = 0
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 01110010111010000000000000000000000000000000
        Decode Out - 2   = 00000000000000000000000000000000000000000000
        Decode Out - 3   = 01100111111100000000000000001000000000000001
        Decode Out - 4   = 01100100111100111000000000001111010111011110

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// First Fetch group in Dispatch stage.
// Allocate bits updated for the assigned physical registers.
// Issue Counter incremented as instructions are written into ROB.
// Dispatch stall = 1. 3 ALU instructions. But only 2 ALU instructions can be dispatched.

*****************Time =      46.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000000000000000000001111
        Valid        = 00000000000000000000000000000000000000000000000000000000000000000001
        Commit       = 00000000000000000000000000000000000000000000000000000000000000000000

        IssueCounter  = 000011
        CommitCounter = 000000

        Disp Stall    = 1
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 01100111101000000000000000001000000000000001
        Decode Out - 2   = 01100100101000010000000000001010010011001000
        Decode Out - 3   = 01100100111101000000000000001111111111111000
        Decode Out - 4   = 01110010011101010001110100000000000000000000
```

```
Stage - 3: Dispatch
 RR I1: Fu_Code=011 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=25 Dest=01 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=011 Op1=00000000 V1=1 Op2=00008001 V2=1 Exe=0f Dest=02 Reo=01 Disp=0
 RR I4: Fu_Code=011 Op1=00000002 V1=0 Op2=0000f5de V2=1 Exe=09 Dest=03 Reo=02 Disp=0

// non-dispatched instructions dispatched this cycle.
// issue counter does not increment. Due to dispatch stall.
*****************Time =      56.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000000000000001111
        Valid       = 00000000000000000000000000000000000000000000000000000000000000001
        Commit      = 00000000000000000000000000000000000000000000000000000000000000000

        IssueCounter   = 000011
        CommitCounter  = 000000

        Disp Stall     = 0
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 01100111101000000000000000001000000000000001
        Decode Out - 2  = 01100100101000010000000000000101001001100100O
        Decode Out - 3  = 01100100111101000000000000001111111111111000
        Decode Out - 4  = 01110010011101010001110100000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=011 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=25 Dest=01 Reo=00 Disp=1
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=011 Op1=00000000 V1=1 Op2=00008001 V2=1 Exe=0f Dest=02 Reo=01 Disp=1
 RR I4: Fu_Code=011 Op1=00000002 V1=0 Op2=0000f5de V2=1 Exe=09 Dest=03 Reo=02 Disp=0

// Second fetch group enters dispatch.
// 4 ALU instructions. Hence it has to stall.
// Issue counter incremented by four from previous cycle.
// Instructions (1) and (3) completes and sets corresponding valid bits to '1'.

*****************Time =      66.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000000011111111
        Valid       = 00000000000000000000000000000000000000000000000000000000000000111
        Commit      = 00000000000000000000000000000000000000000000000000000000000000000

        IssueCounter   = 000111
        CommitCounter  = 000000

        Disp Stall     = 1
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 01100111100100000000000000001000000000000000
        Decode Out - 2  = 10000001111111000000000000000000000000000000
        Decode Out - 3  = 01100100100100001000000000000110110100010100
        Decode Out - 4  = 01110101000101111001110100000000000000000000
```

```
Stage - 3: Dispatch
 RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008001 V2=1 Exe=0f Dest=04 Reo=03 Disp=0
 RR I2:  Fu_Code=011 Op1=00000004 V1=0 Op2=0000a4c8 V2=1 Exe=09 Dest=05 Reo=04 Disp=0
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=0000fff8 V2=1 Exe=09 Dest=06 Reo=05 Disp=0
 RR I4:  Fu_Code=011 Op1=00000005 V1=0 Op2=00000006 V2=0 Exe=24 Dest=07 Reo=06 Disp=0

// (1) and (3) committed in this cycle.
// Commit Counter incremented.

*****************Time =       76.0ns*****************

        Allocate      = 00000000000000000000000000000000000000000000000000000011111111
        Valid         = 00000000000000000000000000000000000000000000000000000000000111
        Commit        = 00000000000000000000000000000000000000000000000000000000000000

        IssueCounter  = 000111
        CommitCounter = 000010

        Disp Stall    = 0
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 01100111100100000000000000001000000000000000
        Decode Out - 2  = 10000001111111000000000000000000000000000000
        Decode Out - 3  = 01100100100100001000000000000110110100010100
        Decode Out - 4  = 01110101000101111001110100000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008001 V2=1 Exe=0f Dest=04 Reo=03 Disp=1
 RR I2:  Fu_Code=011 Op1=00000004 V1=0 Op2=0000a4c8 V2=1 Exe=09 Dest=05 Reo=04 Disp=1
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=0000fff8 V2=1 Exe=09 Dest=06  Reo=05 Disp=0
 RR I4:  Fu_Code=011 Op1=00000005 V1=0 Op2=00000006 V2=0 Exe=24 Dest=07 Reo=06 Disp=0

// (1) and (3) updates CP and hence Commit Bit is updated for the corresponding locations.
// Third fetch group enters Dispatch.
// Issue Counter incremented.

*****************Time =       86.0ns*****************

        Allocate      = 00000000000000000000000000000000000000000000000000001111111111
        Valid         = 00000000000000000000000000000000000000000000000000000000000111
        Commit        = 00000000000000000000000000000000000000000000000000000000000110

        IssueCounter  = 001011
        CommitCounter = 000010

        Disp Stall    = 1
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 10000010000000000000010100000000000000000100
        Decode Out - 2  = 00000000000000000000000000000000000000000000
        Decode Out - 3  = 01100111100101000000000000001000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
```

```
 RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=08 Reo=07 Disp=0
 RR I2:  Fu_Code=100 Op1=00000123 V1=1 Op2=00000000 V2=1 Exe=03 Dest=09 Reo=08 Disp=0
 RR I3:  Fu_Code=011 Op1=00000008 V1=0 Op2=00006d14 V2=1 Exe=09 Dest=0a Reo=09 Disp=0
 RR I4:  Fu_Code=011 Op1=00000003 V1=0 Op2=00000007 V2=0 Exe=2a Dest=0b Reo=0a Disp=0

// Instruction (4) writes back and hence, valid bit is set.
*****************Time =      96.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000001111111111111
        Valid       = 00000000000000000000000000000000000000000000000000000000000001111
        Commit      = 00000000000000000000000000000000000000000000000000000000000000110

        IssueCounter   = 001011
        CommitCounter  = 000010

        Disp Stall     = 0
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 10000010000000000000010100000000000000000100
        Decode Out - 2  = 00000000000000000000000000000000000000000000
        Decode Out - 3  = 01100111100101000000000000001000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=08 Reo=07 Disp=1
 RR I2:  Fu_Code=100 Op1=00000123 V1=1 Op2=00000000 V2=1 Exe=03 Dest=09 Reo=08 Disp=1
 RR I3:  Fu_Code=011 Op1=00000008 V1=0 Op2=00006d14 V2=1 Exe=09 Dest=0a Reo=09 Disp=1
 RR I4:  Fu_Code=011 Op1=8000f5de V1=1 Op2=00000007 V2=0 Exe=2a Dest=0b Reo=0a Disp=0

// Instructions (5) and (10) finish execution and set valid bits. JAL writes correct target address into ROB.
// Commit counter incremented due to commit of instruction (4).
*****************Time =      106.0ns*****************

        Allocate    = 00000000000000000000000000000000000000000000000000001111111111111
        Valid       = 00000000000000000000000000000000000000000000000000001000011111
        Commit      = 00000000000000000000000000000000000000000000000000000000000000110

        IssueCounter   = 001101
        CommitCounter  = 000011

        Disp Stall     = 0
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000
        Decode Out - 2  = 01100100100100000000000000000000000000000001
        Decode Out - 3  = 01100111100101000000000000001000000000000000
        Decode Out - 4  = 01100100100101001010000000000110110100001000

Stage - 3: Dispatch
 RR I1:  Fu_Code=100 Op1=00000000 V1=1 Op2=0000000b V2=0 Exe=04 Dest=00 Reo=0b Disp=0
 RR I2:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=0c  Reo=0c Disp=0
 RR I4:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
```

```
// Instruction (4) updates CP and hence, commit bit for that VB location is set to '1'.
// Allocate bit and Valid bit of the VB location, indexed by the previous content of CP (2) is set to '0'.
// physical register number 2 is reclaimed for future allocation.
*****************Time =      116.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000001111111111111011
        Valid        = 00000000000000000000000000000000000000000000000000001001011111
        Commit       = 00000000000000000000000000000000000000000000000000000000001010

        IssueCounter  = 010000
        CommitCounter = 000100

        Disp Stall       = 1
        Restore          = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 01100111100110000000000000001000000000000000
        Decode Out - 2   = 01100100100110001100000000000110110100010000
        Decode Out - 3   = 00000000000000000000000000000000000000000000
        Decode Out - 4   = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2:  Fu_Code=011 Op1=00000000 V1=1 Op2=00000001 V2=1 Exe=09 Dest=0d Reo=0d Disp=0
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=0e Reo=0e Disp=0
 RR I4:  Fu_Code=011 Op1=0000000e V1=0 Op2=00006d08 V2=1 Exe=09 Dest=0f Reo=0f Disp=0

*****************Time =      126.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000011111111111111011
        Valid        = 00000000000000000000000000000000000000000000000000001101011111
        Commit       = 00000000000000000000000000000000000000000000000000000000011010

        IssueCounter  = 010000
        CommitCounter = 000100

        Disp Stall       = 0
        Restore          = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 01100111100110000000000000001000000000000000
        Decode Out - 2   = 01100100100110001100000000000110110100010000
        Decode Out - 3   = 00000000000000000000000000000000000000000000
        Decode Out - 4   = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2:  Fu_Code=011 Op1=00000000 V1=1 Op2=00000001 V2=1 Exe=09 Dest=0d Reo=0d Disp=1
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=0e Reo=0e Disp=1
 RR I4:  Fu_Code=011 Op1=0000000e V1=0 Op2=00006d08 V2=1 Exe=09 Dest=0f Reo=0f Disp=0

*****************Time =      136.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000011111111111111111
        Valid        = 00000000000000000000000000000000000000000000000000001001101111011
        Commit       = 00000000000000000000000000000000000000000000000000000000011010
```

```
        IssueCounter   = 010010
        CommitCounter  = 000100

        Disp Stall     = 0
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=011 Op1=00000000 V1=1 Op2=00008000 V2=1 Exe=0f Dest=02 Reo=10 Disp=0
 RR I2: Fu_Code=011 Op1=00000002 V1=0 Op2=00006d10 V2=1 Exe=09 Dest=10 Reo=11 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

**********Time =     146.0ns**********

        Allocate    = 00000000000000000000000000000000000000000000000011111111111111111
        Valid       = 00000000000000000000000000000000000000000000000011100110111111011
        Commit      = 00000000000000000000000000000000000000000000000000000000000011010

        IssueCounter   = 010010
        CommitCounter  = 000110

        Disp Stall     = 0
        Restore        = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Branch delay slot instruction (11) writes back and sets valid bit to '1'.
**********Time =     156.0ns**********

        Allocate    = 00000000000000000000000000000000000000000000000011111111111101111
        Valid       = 00000000000000000000000000000000000000000000000011101110111111011
        Commit      = 00000000000000000000000000000000000000000000000000000000000001101010

        IssueCounter   = 010010
        CommitCounter  = 000110

        Disp Stall     = 0
        Restore        = 0
```

```
Stage - 2 : Issue/Decode
        Decode Out - 1   = 00000000000000000000000000000000000000000000000000
        Decode Out - 2   = 00000000000000000000000000000000000000000000000000
        Decode Out - 3   = 00000000000000000000000000000000000000000000000000
        Decode Out - 4   = 01110010100101000001101000000090000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Branch Mispredition is detected by commit logic.
*****************Time =      166.0ns*****************

        Allocate      = 00000000000000000000000000000000000000000000001111111111111111
        Valid         = 00000000000000000000000000000000000000000000001110111111101111
        Commit        = 00000000000000000000000000000000000000000000000000001101010

        IssueCounter  = 010011
        CommitCounter = 000110

        Disp Stall    = 0
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 01110010100110000011100000000000000000000000
        Decode Out - 2   = 01110010100111000001110100000000000000000000
        Decode Out - 3   = 01110010101000000001000000000000000000000000
        Decode Out - 4   = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4:  Fu_Code=011 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=25 Dest=04 Reo=12 Disp=0

// Four instructions (8, 9, 10, 11) commit. Commit counter incremented by four from previous cycle.
*****************Time =      176.0ns*****************

        Allocate      = 00000000000000000000000000000000000000000000001111111111111111
        Valid         = 00000000000000000000000000000000000000000000001110111111101111
        Commit        = 00000000000000000000000000000000000000000000000000001101010

        IssueCounter  = 010110
        CommitCounter = 001010

        Disp Stall    = 1
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1   = 00000000000000000000000000000000000000000000000000
        Decode Out - 2   = 00000000000000000000000000000000000000000000000000
        Decode Out - 3   = 00000000000000000000000000000000000000000000000000
        Decode Out - 4   = 00000000000000000000000000000000000000000000000000
```

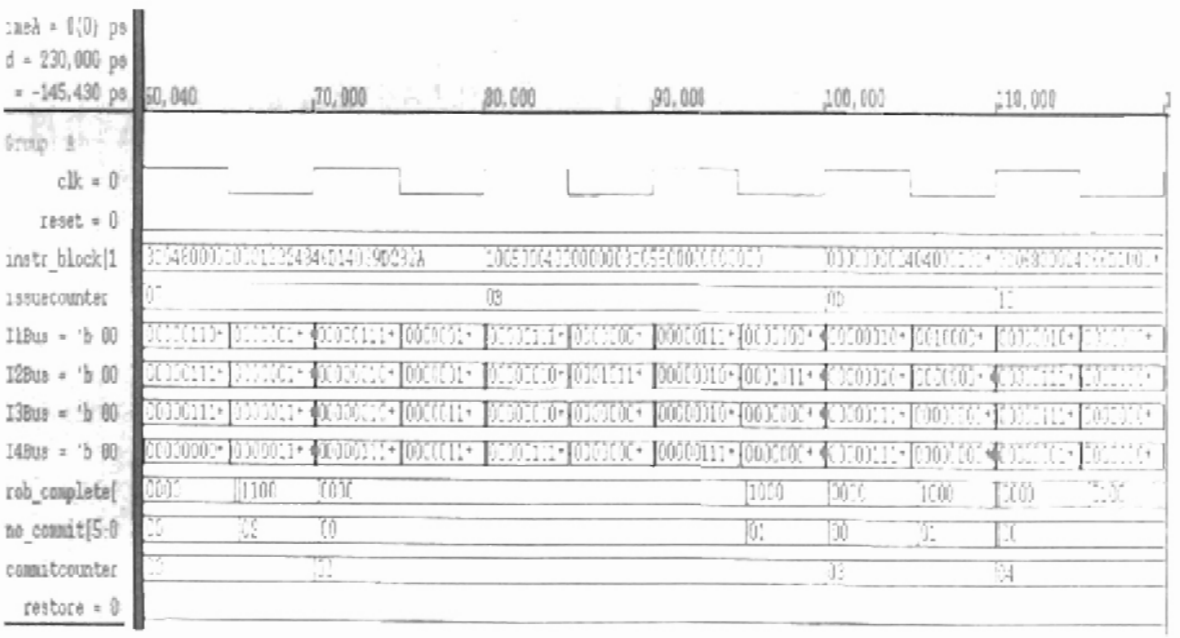
```
Stage - 3: Dispatch
 RR I1: Fu_Code=011 Op1=00000000 V1=1 Op2=8000f5de V2=1 Exe=25 Dest=11 Reo=13 Disp=0
 RR I2: Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=12 Reo=14 Disp=0
 RR I3: Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=13 Reo=15 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Commit Bits of VB locations whose address is stored in CP is set to '1'.
// Restore is set to '1'.
// CP contents are copied to IP internally.
**********Time =      186.0ns**********

        Allocate     = 00000000000000000000000000000000000000000000111111111110101111111
        Valid        = 00000000000000000000000000000000000000000000000000000011010101011
        Commit       = 00000000000000000000000000000000000000000000000000000011010101010

        IssueCounter  = 010110
        CommitCounter = 001010

        Disp Stall    = 0
        Restore       = 1

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Commit Bit's values are copied to Allocate and Valid bits. To clear the changes done to allocate and
// valid bits by speculated instructions.
**********Time =      196.0ns**********

        Allocate     = 00000000000000000000000000000000000000000000000000000011010101011
        Valid        = 00000000000000000000000000000000000000000000000000000011010101011
        Commit       = 00000000000000000000000000000000000000000000000000000011010101010

        IssueCounter  = 001010
        CommitCounter = 001010

        Disp Stall    = 0
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2: Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
```

```
 RR I3:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

*****************Time =      206.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000000000011010101011
        Valid        = 00000000000000000000000000000000000000000000000000000011010101011
        Commit       = 00000000000000000000000000000000000000000000000000000011010101010

        IssueCounter  = 001010
        CommitCounter = 001010

        Disp Stall    = 0
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 01110010100101000001101000000000000000000000
        Decode Out - 2  = 01110010100110000001110000000000000000000000
        Decode Out - 3  = 01110010100111000001110100000000000000000000
        Decode Out - 4  = 01110010101000000000100000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I2:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I3:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0
 RR I4:  Fu_Code=000 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=00 Dest=00 Reo=00 Disp=0

// Instructions fetched from the jumped location enters dispatch. OP2 has the values  computed by
// instructions before the mispredicted JAL. Thus values in register computed by correct instructions
// are preserved during restore, proving correct operation.
*****************Time =      216.0ns*****************

        Allocate     = 00000000000000000000000000000000000000000000000000001111111111
        Valid        = 00000000000000000000000000000000000000000000000000000011010101011
        Commit       = 00000000000000000000000000000000000000000000000000000011010101010

        IssueCounter  = 001110
        CommitCounter = 001010

        Disp Stall    = 1
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000

Stage - 3: Dispatch
 RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=25 Dest=02 Reo=0a Disp=0
 RR I2:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000f5de V2=1 Exe=25 Dest=04 Reo=0b Disp=0
 RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=06 Reo=0c Disp=0
 RR I4:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=08 Reo=0d Disp=0
```

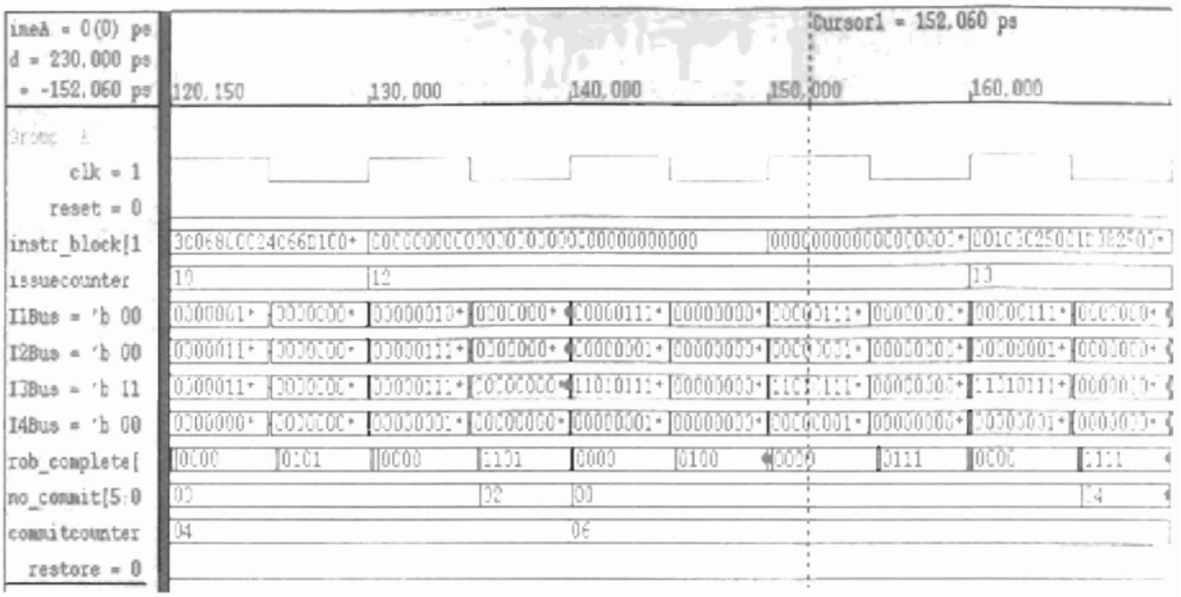
```
******************Time =      226.0ns******************

        Allocate      = 00000000000000000000000000000000000000000000000000011111111111
        Valid         = 00000000000000000000000000000000000000000000000000011010101011
        Commit        = 00000000000000000000000000000000000000000000000000011010101010

        IssueCounter  = 001110
        CommitCounter = 001010

        Disp Stall    = 1
        Restore       = 0

Stage - 2 : Issue/Decode
        Decode Out - 1  = 00000000000000000000000000000000000000000000000000
        Decode Out - 2  = 00000000000000000000000000000000000000000000000000
        Decode Out - 3  = 00000000000000000000000000000000000000000000000000
        Decode Out - 4  = 00000000000000000000000000000000000000000000000000

Stage - 3: Dispatch
  RR I1:  Fu_Code=011 Op1=00000000 V1=1 Op2=00000000 V2=1 Exe=25 Dest=02 Reo=0a Disp=0
  RR I2:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000f5de V2=1 Exe=25 Dest=04 Reo=0b Disp=0
  RR I3:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=06 Reo=0c Disp=0
  RR I4:  Fu_Code=011 Op1=00000000 V1=1 Op2=8000a4c8 V2=1 Exe=25 Dest=08 Reo=0d Disp=0

L422 "VR.test.v": $finish at simulation time 230.0ns
0 simulation events (use +profile or +listcounts option to count) + 469470 accelerated events + 338686
timing check events
CPU time: 1.4 secs to compile + 51.0 secs to link + 78.2 secs in simulation
End of VERILOG-XL 2.8.s008   Oct 20, 2003  21:02:02
```

1) Verification of Virtual Registers Cycles (0-50)

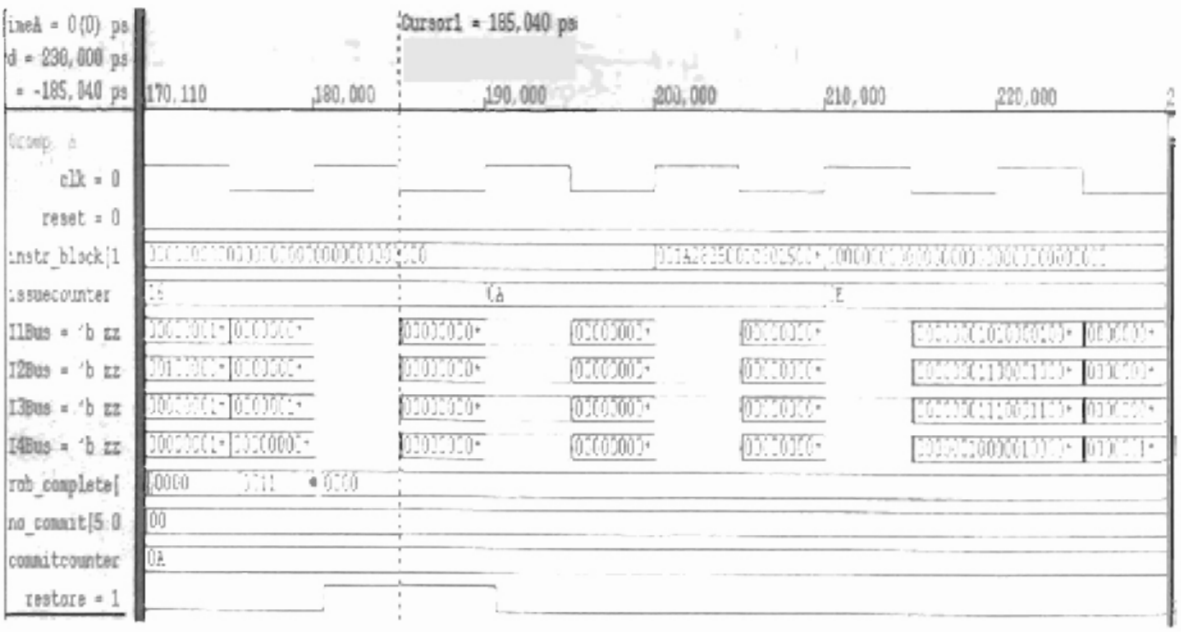
2) Verification of Virtual Registers Cycles (60-110)



3) Verification of Virtual Registers Cycles (120-170)



4) Verification of Virtual Registers Cycles (180-230)



Appendix B
Bubble Sorting Execution

The following C program is a 'Bubble Sorting' program, which sorts a five element array in descending order of its contents. Due to the inherent loop carried dependency in this algorithm, instruction level parallelism is limited.

```
main()
{
  int i, j, k;
  int correct;
  int a[5];
  int result(int, int, int, int, int); // function declaration.

  i = 0;
  j = 0;
  correct = 0;

  // Load Values in array
  a[0] = 3;
  a[1] = 16;
  a[2] = 4;
  a[3] = 670;
  a[4] = 59;

  // Bubble sort
  while(i <= 4){
    for(j = i; j < 4; j= j+1){
     // check if the present element is  greater than
     // the next element. If so, swap the elements.
      if (a[i] < a[j+1]){
        k = a[j+1];
        a[j+1] = a[i];
        a[i] = k;
      }
    }
   i = i+1;
  }

  correct = result(a[0], a[1], a[2], a[3], a[4]);

  // returned value is loaded into last array position
  a[4] = correct;
}

// Subroutine checks for the correct sorting and
// returns  1 if correctly sorted.
int result(int a, int b, int c, int d, int e)
{
 if(a > b && b > c && c > d && d > e)
    return(1);
 else
    return(0);
}
```

Appendix C
Squared Series Sum Execution

The following C program is a 'squared series sum' program, which squares and sums a series using a loop. Notice that, successive iterations of the loop can execute independent of the previous iteration's result. Hence, instruction level parallelism exist.

```
main(){

  int i,sum;
  int a[2];

  i = sum = 0;

  // square and sum
  while (i <= 25){
    sum = sum + (i*i);
    i = i+1;
  }

  //  store the final value
  //  in an array.
  a[0] = sum;
}
```

VITA

Balachander Ganesan

Candidate for the Degree of

Master of Science

Thesis: REGISTER RENAMING ALGORITHM FOR FAST BRANCH MISPREDICTION RECOVERY IN SUPERSCALAR PROCESSOR

Major Field: Electrical and Computer Engineering

Biographical:

Personal Data: Born in Trichy, Tamil Nadu, India on May 21, 1980 the son of Mr. T.R .Ganesan and Mrs. S. Lalitha.

Education: Received Bachelor of Engineering in Electrical and Electronics Engineering from the University of Madras, Chennai, India in May, 2001. Completed the requirements for the Master of Science degree with a major in Electrical and Computer Engineering at Oklahoma State University in December, 2003.

Professional Experience: January 2002 – March 2003: VB Programmer, College Of Business Administration, Oklahoma State University, Stillwater. August 2002 – May 2003: Teaching Assistant, Department of Electrical Engineering, Oklahoma State University, Stillwater.