Handling of Precise Interrupts in Pipelining Systems

By
Amal M. Barakat Al- Dweik

Supervisor
Dr. Sami Serhan

Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in
Computer Science

Faculty of Graduate Studies
University of Jordan

May ٢٠٠٢

This thesis was successfully defended and approved on: 30-May-2002.

Examination Committee **Signature**

Dr. Sami Serhan, Chairman
Assist. Prof. Of Compiler Design ……………………………………

Prof. Oqeili Saleh, Member
Prof. Of Computer Architecture ……………………………………

Dr. Mohammed Mahafzah, Member
Assoc. Prof. Of Parallel Architecture ……………………………………

Dr. Andraws Sweidan, Member
Assist. Prof. Of Complex Systems ……………………………………

Dr. Imad Salah, Member
Assist. Prof. Of Complex Systems ……………………………………

DEDICATION

To all our people in Palestine

To my mother, my father, brothers, and sisters.

ACKNOWLEDGEMENT

I'm grateful to Dr. Sami for his idea of the thesis. I would like to thank him for his continuous encouragement, support, and help. A special thank for all the colleagues in the University of Jordan for their support.

I acknowledge the German Academic Service Exchange (DAAD) for granting me the scholarship along my study in the master.

LIST OF CONTENTS

LIST OF TABLES

List of Figures

Handling of Precise Interrupts in Pipelining Systems

By
Amal M. Barakat Al-Dweik

Supervisor
Dr. Sami Serhan

Abstract

Handling of precise interrupts in pipelining systems is a significant field of research in the computer architecture. Today's high-performance processors employ out-of-order execution, which permits instructions to be overtaken by later instructions. Many schemes were used, in the time of interrupt, to maintain the sequential state, and not modifying process state in an order different from that defined by the sequential architectural model.

The thesis mainly is concerned with two techniques: In Order Completion, and the Out- of Order Completion. Both techniques are implemented and tested. The Out Of Order Completion technique consists of: Reorder Buffer, Reorder Buffer with Bypass Paths, History Buffer, and the Future File schemes. The results obtained and the comparisons between all of these techniques are presented throughout the thesis.

A new approach is proposed in this thesis. It is a compromised scheme between the In Order Completion and the Out Of Order Completion. The control logic, the flow of information, and the hardware components are stated. After the new proposed scheme is analyzed, we found that it gives better results than the other two techniques regarding to the hardware components and the control logic used in this scheme.

1. INTRODUCTION

1.1 Preface:

Most current computer architectures are based on a sequential model of program execution. In contrast, a high performance implementation may be pipelined, permitting several instructions to be executed at the same time. Complex hardware schemes are required to maintain the sequential state since the use of a sequential architecture and a pipelined implementation clash at the time of an interrupt; pipelined instructions may modify the process state in an order different from that defined by the sequential architectural model. At the time an interrupt condition is detected, the hardware may not be in a state that is consistent with any specific program counter value (James *et al.,* 1988). When an interrupt occurs, the interrupted process state must be saved (James *et al*., 1988).

Interrupts are precise if the saved processor states are consistent with the in-order states defined by the program sequence where one instruction completes before the next begins (Sang-Joon et al., 1999).

Therefore, today's high-performance microprocessors employ out-of-order execution to raise performance. In contrast to in-order execution, out-of-order execution permits these instructions, which block the execution stream, to be overtaken by later instructions. This results in better utilization of the function units and in a higher performance compared to the pipelined in-order designs (Daniel *et al*., 2001). The out-of-order issue of instructions has drastically increased the utilization of the precise interrupt mechanism to handle exceptional events (Amerl *et al*., 200).

1.2 Objectives of Handling Precise Interrupts

In computers, there is a problem in writing programs; that is programs are not linear! One or more pipeline stages may have to be flushed in processing an exception. The number of flushed stages depends on where the exception is triggered in the pipeline. Complex hardware schemes are required to maintain the sequential state. The out-of-order issue can cause a serious problem at the time of interrupts, because it makes the process states different from those defined by the program sequence (Sang-Joon *et al*., 1999). Interrupt handlers should not modify any of source registers in the interrupted operation because subsequent operations with respect to the original program order that use the same source register would need to be re-issued and re-executed after resumption. The processor cannot re-execute those operations because they may have already updated the processor state. So, we have to preserve registers in the interrupted processor state.

In pipelined machines, most difficult exceptions have two properties:

a. They occur within the instruction.
b. They must be restartable.

Although precise interrupt is preferable, the implementation of it is difficult because of the following:

1. Interrupts can occur anywhere during the execution of an instruction.
2. Multiple instructions are being executed, and each may have partially updated state information.
3. Multiple instructions are being executed, so multiple interrupts can occur at the same time.

So, with the current trends in the design of processors and operating systems, the cost of handling exceptions precisely is also becoming extremely expensive; this is because of their implementation. Exceptional situations are harder to handle in a pipelined machine because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the machine or not.

1.3 Previous Works

There are many studies, which attempted to study or to develop the handling of precise interrupts in pipelining systems. James E. Smith and Gurindar S. Sohi have some of these researches in their research in the superscalar processing (James *et al*., 1995) focusing on converting an ostensibly sequential program into a more parallel one. They used the phase in which they recommend of committing the Process State in correct order so that precise interrupts can be supported. Pierguido V.C. Caironi, Lorenzo Mezzalira, Mariagiovanna Sami presented, in (pierguido *et al*., 1996), a hardware solution to the problem of precise interrupts and exceptions in superscalar RISC CPU architectures. This solution called Context Reorder Buffer (CRB) which is based both on the reorder buffer architecture presented by Smith and Pleszkun, and on the concept of context. Daniel Kroning proved, in (Daniel, 2001), the data consistency of the pipelined machines and presented a generic approach to speculative execution, where he proposed a data consistency criterion for such a machine. He applied this method in order to implement and prove DLX pipeline with branch prediction and precise interrupts. E .Ozer, S.W.Sathaye, K.N.Menezes, S. Banerjia, M.D.Jennings and T.M.Conte have mentioned in their study, that “interrupt handling in out-of-order execution processors requires complex hardware schemes to maintain the sequential state, especially implementing the precise interrupts”. They apply in their work the reorder buffer with future file and the history buffer methods to Very Long Instruction width (VLIW) processors, and present a new scheme, called the " Current-State Buffer (CSB)". Daniel Kroning in his paper, (Daniel *et al.*, 2001), combines the Tomasulo Scheduler with a reorder buffer, which implements precise interrupts. He gave a mathematical correctness proof for this enhanced scheduling algorithm. James E. Smith, Andrew R. Plezkun in (James *et al.*, 1988) described the precise interrupt problem and discussed five solutions, which are In-Order Instruction completion, Reorder Buffer, Reorder Buffer with Bypass Paths, History Buffer, and Future File. They discussed the performance for each method examined. David Alan Gilbert in (David ,1997) described a solution to the problem of dependency and exception-handling mechanisms in the context of a third generation asynchronous implementation of the Advanced Risk Management (ARM) instruction set architecture. The Reorder Buffer was the basis of the architecture and the novel enhancements as Hwu and Patt proposed. They described an enhanced mechanism, which reduces the need to stall the pipeline.

1.4 Main Points of Contribution

We built our own simulators to implement the schemes examined and to verify the results obtained. There was not any complete algorithm, we wrote these algorithms with their full implementation, and did all the analysis for these schemes. We searched for any ready used simulator, but unfortunately, we found only some, which is related to pipelining without any handling of the interrupts. They mentioned that interrupts problem is another heavy work that is not matter of concern. Even though, these simulators were built with the full corporation between several universities in the United States of America and the IBM Company, they were incomplete.

The following are the main points of contribution presented in the thesis:

1. Two architectural models are used throughout the thesis: The scalar architecture and the VLIW processors.
2. All algorithms used in the thesis are stated, implemented, and analyzed.
3. The In-order Completion, and the Out of Order Completion techniques are used. The Out of Order Schemes used are: Reorder Buffer, the Reorder Buffer with Bypass paths, the History Buffer, and the Future File. The performance obtained from their implementation is discussed.
4. A new technique is suggested to handle the precise interrupts in pipelining systems. It is a compromised technique between the In Order Completion and Out of Order Completion.
5. The proposed scheme is verified to be better than other schemes. It is compared with the other schemes, regarding to the hardware components. Results and analysis are introduced.
6. As a conclusion, a comparative analysis for all schemes used is presented and analyzed.

1.5 Organization of the Thesis

Chapter two presents the background knowledge of the thesis. The literature review of the thesis is the subject of chapter three. The basic approaches for handling the precise interrupts in the pipelining systems are fully discussed, and expanded throughout the thesis.

The implementation part of the thesis is discussed fully in chapter four and chapter five. Chapter four discusses the In Order Completion scheme. The Out of Order schemes, their full analysis and discussion are introduced in chapter five. Chapter six discusses the compromised scheme, which is suggested in the thesis. The final implementation chapter is chapter seven; it introduces the precise interrupts handling in VLIW processors.

Chapter eight contains the final conclusion of the thesis and the extensions suggested as a future work.

2. BACKGROUND KNOWLEDGE

This chapter is a brief review of different fundamental subjects related to the thesis. The basics, the terminologies, and many concepts used throughout the thesis are defined in the following pages.

2.1 Pipelining

A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages. Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

A pipelined processor consists of a sequential, linear list of segments; where each segment performs one computational task or group of tasks. Pipelining increases the CPU instruction throughput; which means that a program runs faster and has lower total execution time. It also increases the number of instructions completed per unit of time. But it does not reduce the time of execution of individual instructions. In fact, it slightly increases the execution time of each instruction due to the overhead in the pipelined control.

Pipelining is used to obtain improvements in processing time that would be unobtainable with existing non-pipelined technology. Similarly, the goal for the IBM 360/91 was an improvement of one to two orders of magnitude over the 7090. Technology advances could only bring about a four-fold improvement. In a more recent example, the 6502 microprocessor had a throughput similar to the 8080 processor running at a clock rate four times faster. This was due to the pipelined architecture of the 6502 versus the non-pipelined 8080.

There are two types of pipelines: Instructional pipeline where different stages of an instruction fetch and instruction execution are handled in a pipeline, and Arithmetic pipeline where different stages of an arithmetic operation are handled along the pipeline.

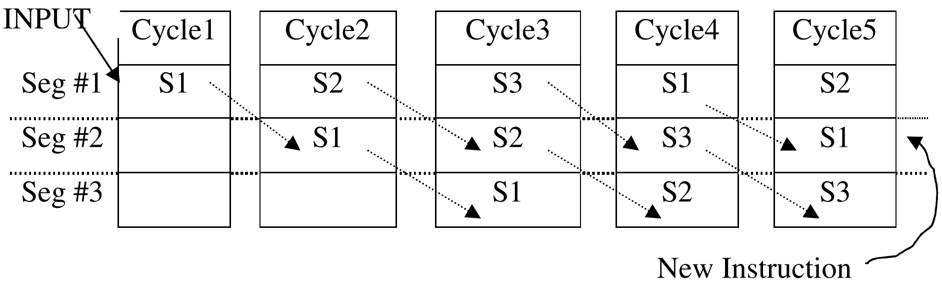


Figure 2.1**.** Notional diagram of a pipelined processor; the segments are arranged vertically, and time moves along the horizontal axis.

A pipelined processor can be represented in two dimensions, as shown in Figure 2.1. Here, the pipelined segments (Seg #1 through Seg #3) are arranged vertically, so the data can flow from the input at the top left downward to the output of the pipeline (after Segment 3) (URL, scism).

There are three things that one must observe about the pipeline.

1. The work (in a computer) is divided up into pieces that fit more or less into the segments allocated for them.
2. This implies that in order for the pipeline to work efficiently and smoothly, the work partitions must each take about the same time to complete. Otherwise, the longest partition requiring time T would hold up the pipeline, and every segment would have to take time T to complete its work. For fast segments, this would mean much idle time.
3. In order for the pipeline to work smoothly, there must be few (if any) exceptions or hazards that cause errors or delays within the pipeline. Otherwise, the

instruction will have to be reloaded and the pipeline restarted with the same instruction that causes the exception. There are additional problems we need to discuss about pipelined processors, which we will consider shortly.

If an N-segment pipeline is empty before an instruction starts, then N + (N-1) cycles or segments of the pipeline are required to execute the instruction, because it takes N cycles to fill the pipe.

Note that we just used the term "cycle" and "segment" synonymously. In the type of pipelines each segment takes one cycle to complete its work. Thus, an N-segmented pipeline takes a minimum time of N cycles to execute one instruction.

Pipeline Hazards

Pipelined processors have several problems associated with controlling smooth, efficient execution of instructions on the pipeline.

There are two disadvantages of pipelined architecture:

1. The complexity.
2. The inability to continuously run the pipeline at full speed, i.e. the pipeline stalls.

There are many reasons that make pipelining not running at full speed. There is a phenomenon called pipeline hazards, which disrupts the smooth execution of the pipeline. The resulting delays in the pipeline flow are called bubbles. These pipeline hazards include

1. *Structural Hazards:* occur when different instructions collide while trying to access the same piece of hardware in the same segment of pipeline. Having redundant hardware for the segments wherein the collision occurs can alleviate this type of hazard. Occasionally, it is possible to insert stalls or reorder instructions to omit this type of hazard.
2. *Data Hazards:* occur when an instruction depends on the result of a previous instruction still in the pipeline, where the result has not yet been computed. The simplest remedy is to insert stalls in the execution sequence, which reduces the pipeline's efficiency. The solution to data dependencies is two fold. First, one can *forward* the result of the Arithmetic and Logic Unit (ALU) to the write back or data fetch stages. Second, in selected instances, it is possible to restructure the code to eliminate some data dependencies.
3. *Control Hazards:* can result from branch instructions. The branch target address might not be ready in the time for the branch to be taken, which results in *stalls* (dead segments) in the pipeline that have to be inserted as local wait events, until processing can resume after the branch target is executed. Control hazards can be alleviated through accurate branch prediction (which is difficult), and by *delayed branch* strategies.

These issues can successfully be dealt with. But detecting and avoiding the hazards lead to a considerable increase in hardware complexity. The control paths controlling the gating between stages can contain more circuit levels than the data paths being controlled. The processor can stall on different events:

1. A *cache misses*. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.

2. A *hazard in pipeline***.** Eliminating a hazard often requires that some instructions in the pipeline allowed proceeding while others be delayed. When an instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never be cleared.

In pipelining, there is a considerable increase in hardware complexity. Other problem arises when a branch instruction comes along; it is impossible to know in advance, which path the program is going to take and, if the machine guesses wrong, all the partially processed instructions in the pipeline must be replaced.

When a data dependency does occur, there are two possible strategies:

1. Detect the data dependencies and hold up the pipeline completely until the dependencies have been resolved (by instruction already in the pipeline being fully executed).
2. Allow all instructions to be fetched into the pipeline, but only allow independent instructions to proceed to their completion, and delay instructions, which are dependent upon other, not yet, executed (Barry, 1996).

Practically there are four types of data dependency, which are:

1. Read-After-Write (RAW): it exists if a read operation occurs before a previous write operation has been completed, and hence the read operation would obtain an incorrect value (a value not yet updated).
2. Write-After-Read (WAR): it exists when a write operation occurs before a previous read operation has time to complete, and again the read operation would obtain an incorrect value (a prematurely updated value).

3. Write-After-Write: it exists if there are two write operations upon a location such that the second write operation in the pipeline completes before the first. Hence the written value will be altered by the first write operation when it completes.
4. Read-After-Read: in which read operations occur out of order, doesn't normally cause incorrect results (Barry, 1996).

Out-Of-Order Execution

Data dependencies and different latencies of the functional units can cause additional delays, which reduce performance. In order to eliminate these delays, the rule of in-order execution of all instruction phases must be dropped. The result is an *out-of-order execution* algorithm. An out-of-order execution algorithm tries to increase performance by distributing the instructions among the available hardware components regardless of their original order. There are two main requirements for such an algorithm:

- The algorithm must maintain data consistency.
- The algorithm is supposed to achieve a high utilization of the functional units to reduce the delays. (Daniel *et al*., 2001)

Pipeline Performance Analysis

When designing instruction sets for pipelining, there is a set of guidelines that can be used, as follows:

1. Avoid variable instruction lengths whenever possible:

a. Variable length instructions complicate hazard detection and precise exception handling.

b. Sometimes it is worth to use variable instruction lengths, because of performance advantages, i.e., caches.

i. If were used frequently, the added complexity is dealt with by freezing the pipeline.

2. Avoid sophisticated addressing modes:

a. Addressing modes that update registers (post-auto increment) complicates exceptions and hazard detection.

b. It also makes it harder to restart instructions.

c. Allowing addressing modes with multiple memory accesses also complicates pipelining.

3. Don't allow self-modifying code:

a. Since it is possible that the instruction being modified is already in the pipeline, the address being written must constantly be checked.

i. If it is found, then the pipeline must be flushed or the instruction is updated!

b. Even if it's not in the pipeline, it could be in the instruction cache.

4. Avoid implicitly setting condition codes (CCs) in instructions:

a. This makes it harder to avoid control hazards since it's impossible to determine if CCs are set on purpose or as a side effect.

b. For implementations that set the CC almost unconditionally:

i. Makes instruction reordering difficult, since it is hard to find instructions that can be scheduled between the condition evaluation and the branch.

Performance analysis helps one to intelligently determine whether or not a given processor is suitable computationally for a specific application as discussed fully in (Schmalz, 1998).

Effect of Exceptions

For purposes of discussion, assume that we have M instructions executing on an N-segment pipeline with no stalls, but that a fraction f_{ex} of the instructions raise an exception in the EX stage. Further assume that each exception requires that

(a) The pipeline segments before the EX stage be flushed,

(b) The exception be handled, requiring an average of H cycles per exception, then that

(c) The instruction causing the exception and its following instructions is reloaded into the pipeline.

Exceptions as Hazards

Hardware and software must work together in any architecture, especially in a pipeline processor. Here, the processor control must be designed so that the following steps occur when an exception is detected:

- Hardware detects an exception (e.g., overflow in the ALU) and stops the offending instruction at the EX stage.
- Pipeline loader and scheduler allow all prior instructions (e.g., those already in the pipeline in MEM and WB) to complete.

- All instructions that are present in the pipeline after the exception is detected are flushed from the pipeline.
- The address of the offending instruction (usually the address in main memory) is saved in the EPC register, and a code describing the exception is saved in the Cause register.
- Hardware control branches to the exception handling routine (part of the operating system).
- The exception handler performs one of three actions:

1. Notify the user of the exception (e.g., divide-by-zero or arithmetic-overflow) then terminate the program;
2. Try to correct or mitigate the exception then restart the offending instruction; or
3. If the exception is a kind interrupt (e.g., an I/O request), then save the program/pipeline state, service the interrupt request, then restart the program at the instruction pointed to by EPC + 4. (M.S Schmalz, 1998).

In any case, the pipeline is flushed as described.

In general, we can say that, if a pipeline has N segments, and the EX stage is at segment 1 < i < N, then two observations are key to the prediction of pipeline performance:

- *Flushing*: negates the processing of the (i-1) instructions following the offending instruction. These must be reloaded into the pipe, at the cost of i cycles (one cycle to flush, i-1 cycles to reload the i-1 instructions *after* the exception is processed).
- *Completing:* the N-i instructions that were loaded into the pipeline prior to the offending instruction takes N-i clock cycles, which are executed:

1. Prior to, or
2. Concurrently with, the reloading of the instructions i-1 that followed the I-th instruction (in the EX stage).

It is readily seen that the total number of wasted cycles equals

$(i-1) + (N-i) = N - 1,$ which is precisely the number of cycles that it takes to set up or reload the pipeline.

The proliferation of unproductive cycles can be mitigated by the following technique:

1. Freeze the pipeline state as soon as an exception is detected.
2. Process the exception via the exception handler, and decide whether or not to halt or restart the pipeline.
3. If the pipeline is restarted, reload the (i-1) instructions following the offending instruction, concurrently with completing execution of the (N-i) instructions that were being processed prior to the offending instruction.

If Step 3 can be performed as stated, then the best-case penalty is only one cycle, plus the time incurred by executing the exception handler. If the entire pipeline needs to be flushed and restarted, then the worst-case penalty is N cycles incurred by flushing the pipe, then reloading the pipeline *after* the instructions preceding the offending instruction have been executed. If the offending instruction must be restarted, then a maximum of i cycles are lost (one cycle for flush, plus (i-1) cycles to restart the instructions in the pipe following the offending instruction).

2.2 Interrupts

Sometimes during the execution of the instructions, something interrupts the regular execution sequence, and control transfers to a piece of code known as the interrupt handler, whose purpose is to process the interrupt. The interrupt handler takes appropriate action, and then, possibly, allows normal execution to resume (Mayan, 1996).

So, interrupts, also known as faults and exceptions, are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is raised at the point where the error is detected; it may be handled by the surrounding code block or by any code block that directly or indirectly invoke the code block where the error occurred (Python, 2001).

When exception occurs, pipelined control takes the following steps to save state safely:

1. Turns off all writes for faulting instruction and its successors, e.g., by turning them into no-ops.
2. Forces trap instruction into pipeline on next Instruction Fetch (IF).
3. Save Program Counter (PC) of faulting instruction so that user program execution can restart with fetch of that instruction after Operating System (OS) has finished handling exception.

If delayed branches are used, instructions in pipeline are not necessarily in sequence, so each PC has to be saved in step 3. Reloading PCs and restarting instruction stream is performed by special instructions (e.g., RFE in DLX).

In general, an exception handler should preserve all registers. However, there are several special cases where you may want to squeeze a register value before returning. Nevertheless, you should not arbitrarily modify registers in an exception handling routine unless you are intended to immediately abort the execution of your program.

To properly process an interrupt, also called as exception, an interrupt handler must identify the interrupting instruction, determine the corrective action, and determine which registers should be used for input and output. While processing the interrupt, the handler must modify the state associated with the program. Finally, after processing the interrupt, it must prompt the processor to resume normal execution if appropriate. The hardware must provide mechanisms that enable the interrupt handler to accomplish all these tasks. Most processors do this by implementing precise interrupts (Mayan, 1996).

When the interrupt handler has completed processing, it must transfer control to the program to resume normal execution with as little disruption as possible; thus, the architecture must define a restart mechanism. Together, the interrupt state specification and the restart mechanism define the interrupt model of the architecture.

Normally, the program counter governs control flow through a program. As a result, the compiler (or programmer) knows each register’s values, the instructions that generated those values, and how the control can reach those instructions. Unlike a normal program, an interrupt handler cannot expect from where and under what conditions it will be invoked. However, to properly process an interrupt, the interrupt handler needs information about the interrupted program. So the architecture must specify the assumptions an interrupt handler can make about when it will be invoked and what the machine state will be generated at that point. This specification is the interrupt state specification (Mayan, 1996).

The constraints imposed by the interrupt handler make it clear that the interrupt state must meet the following requirements:

1. All instructions that were issued before the excepting instructions could be complete before control enters the interrupt handler.
2. The state should appear as it would if no instruction is issued after the excepting instruction.
3. The address of the excepting instruction must be available to the interrupt handler.

If the interrupt state satisfies theses conditions, the restart mechanism is obvious: after processing the interrupt, the handler must branch to either the interrupting instruction (and execute it in the new state, in which it should not cause an interrupt) or the succeeding instruction (Mayan, 1996).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop (Python, 2001). In a pipelined machine an instruction is executed step by step and is completed using several clock cycles. Unfortunately, other instructions in the pipeline can raise exceptions that may force the machine to abort the instructions in the pipeline before they are completed.

Interrupt Classification

Software interrupts includes the following:

1. I/O device request.
2. Invoking an operating system service from a user program (system call).
3. Breakpoint (programmer-requested interrupt).

4. Integer arithmetic overflow or underflow; FP arithmetic anomaly.
5. Page fault.
6. Misaligned memory accesses (if alignment is required).
7. Memory protection violation.
8. Using an undefined instruction.
9. Hardware malfunction.
10. Power failure.

Five types can characterize the requirements on exceptions:

Synchronous versus Asynchronous.

If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is *synchronous*. With the exception of hardware malfunctions, devices external to the processor and memory cause asynchronous events.

Asynchronous events usually can be handled after the completion of the current instruction, which makes them *easier* to handle.

User Requested versus Coerced

If the user task directly asks for it, it is a *user-requested* event. In some sense, user-requested exceptions are not really exceptions, since they are *predictable*. They are treated as exceptions, because the same mechanisms that are used to save and restore the state are used for these user-requested events. Because the only function of an instruction that triggers this exception is to cause the exception, *user-requested exceptions* can always be handled after the instruction has completed.

Coerced exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are *harder* to implement because they are *not predictable*.

User Maskable versus user Nonmaskable

If an event can be *masked or disabled* by a user task, it is user *maskable*. This mask simply controls whether the hardware responds to the exception or not.

Within versus Between Instructions

This classification depends on whether the event prevents instruction completion by occurring *in the middle (within)* of execution or whether it is recognized *between* instructions. Exceptions that occur *within* instructions are *always synchronous*, since the instruction triggers the exception.

It is *harder* to implement exceptions that occur *within* instructions than between instructions, since the instruction must be stopped and restarted.

Resume versus Terminate

If the program's execution always stops after the interrupt, it is a *terminating* event. If the program's execution continues after the interrupt, it is a resuming event.

It is *easier* to implement exceptions that *terminate* execution, since the machine need not be able to restart execution of the same program after handling the exception.

The difficult task is implementing interrupts occurring within instructions, where the instruction must be resumed because it requires another program to be invoked to save the state of the executing program. So the steps required are:

1. Correct the cause of the exception.
2. Restore the state of the program before the instruction that caused the exception.
3. Start the program from the instruction that caused the exception.

If a pipeline provides the ability for the machine to handle the exception, save the state, and restart without affecting the execution of the program, the *pipeline* or machine is said to be *restartable.* Almost all machines today are *restartable*, at least for integer pipelines, because it is needed to implement virtual memory.

2.3 Precise Interrupts

The definition of precise interrupt reflects execution in a sequential architecture. In a sequential architecture, instructions are issued serially. An instruction runs to completion before the next one issue. From the precise interrupt point of view, two types of interrupts:

1. Detect before issue: e.g. illegal opcode, privileged instructions, some external interrupts.
2. Detect during execution: e.g. page fault, arithmetic, and some external interrupts.

When an instruction interrupts, processor hardware immediately transfers control to the interrupts handler. Interrupt is precise if the machine state at the time of the interrupt is identical to the state that would exist if the implementation were sequential. This state, known as precise state, meets the following conditions:

1. All instructions that issue prior to the interrupting instruction have completed.
2. No instruction has been issued after the interrupting instruction.
3. The program counter points to the interrupting instruction, is the precise program counter.

If all the implementation's interrupts are precise, we say that it follows the precise interrupt model (Mayan, 1996). The conditions, under which precise instructions are either necessary or desirable, are:

1. For I/O and timer instruction, (external), precise process state makes restarting possible.
2. In virtual memory system, (internal), precise instructions allow a process to be correctly restarted after a page fault has been serviced.
3. For software debugging, it is desirable for the saved state to be precise. This information can be helpful in isolating the exact instruction and circumstances that cause the exception condition.
4. For refined recovery from arithmetic exception, software routines may be able to take steps, rescale floating-point numbers, to allow a process to continue. Some end cases of modern floating-point arithmetic systems might best be handled by software, gradual underflow in the proposed IEEE floating point standard.
5. Unimplemented opcodes can be simulated by system software in a way transparent to the programmer if interrupts are precise. So, lower performance models of architecture can maintain compatibility with higher performance models using extended instruction sets.
6. Virtual machine can be implemented if privileged instruction-faults cause precise instructions.

We can implement the precise interrupt model simply on a non-pipelined, sequential architecture implementation. But modern processors use pipelining to improve performance, complicating implementation of precise interrupts (Mayan, 1996).

Unfortunately, pipelining is an important mechanism for improving processor performance, and interferes with the processor's ability to handle precise interrupts. Techniques that implement precise interrupts on pipelined processors use a large amount of extra hardware or reduce performance, or both. To gain some insight into the problem, there is a taxonomy that divides interrupts into four classes. For each class we ask the following questions:

1. Can we interrupt some interrupts precisely yet avoid the performance and /or hardware penalty?
2. Which interrupts are essential for machine operations? Conversely, which interrupts can we implement imprecisely without impairing the machine's ability to run programs correctly?
3. What benefits can we gain from discarding precision for some interrupts? Since we must implement the rest of the interrupts precisely, will the implementation still incur a similar performance and/or hardware cost?

The characteristics listed for the precise-interrupt model are identical to the requirements of the general interrupt handler.

2.4 Imprecise Interrupts

Because of the multiple instructions that can be in various stages of execution at any given moment in time, handling an interrupt is one of the more complex tasks. An imprecise interrupt can result from an instruction exception while the precise address of the instruction causing the exception is not known! The difficulty arising from imprecise interrupts should be viewed as a complexity to be overcome, not as an inherent defect in pipelining.

Some imprecise interrupts are guaranteed to be precise. So, we can implement it as follows: when an instruction interrupts the processor, if any instruction is issued after the interrupting instruction has completed, all instructions between the interrupting instruction and the last completed instruction run to completion. Control transfers to the interrupt handler in this state. Normal execution can resume after the last completed instruction (Mayan, 1996).

Usually, an instruction will cause an interrupt is determined in the nth cycle of its execution. If this n is no longer than the number of cycles necessary to execute the shortest instruction, that interrupt is precise. No instruction is issued after the interrupting instruction can have completed, so only instructions issued before the interrupting instruction will run to completion (Mayan, 1996).

2.5 Tomasulo’s Algorithm

Tomasulo called his mechanism the Common Data Bus, because the mechanism is more expansive than a simple bus, it is usually referred to as *Tomasulo’s algorithm* instead. The underlying principle used is this: *when the data is stale, keep track of where new data will be coming from*. Here is how the principle is used. The register file holds data, for brief windows in time, data words in the register file are stable, and in that they are soon to be overwritten by an instruction that has not yet completed. Take, for example, the following code:

```
lw r1, 16(r2)
addi r1, r1, 1
```

Ignoring dependences between the instructions, the load instruction would be likely to take longer time than the add-immediate, because the load performs both an add-immediate and a memory-access: it requires an add-immediate of register 2 with the value 16 to generate the load address. Only after the address is generated, the memory access begins. Therefore, by the time that the addi is ready to read the value of r1 out of the register file, it is likely that the load is still in mid-execution. If this is the case, then r1 contains stable data that cannot be used for computation by the addi or any other instruction that follows the load. Previous architectures would either stall in this instance or use forwarding paths in the pipeline.

Tomasulo’s algorithm uses a different mechanism: instead of keeping track of the data in r1, it keeps track of the data’s source, i.e. the load instruction which will update r1 in the near future. When the load is decoded, it is en-queued in a numbered *reservation station* awaiting execution; r1 is tagged as invalid; and the register file holds the reservation station Identity (ID) instead of r1’s contents.

Therefore, rather than keeping track of the data value, the register keeps track of where the new data will come from. When the addi instruction is decoded and en-queued, it reads the ID from the register file and is placed in its own reservation station, knowing that one of its operands is invalid, but also knowing the unique ID of the instruction that will produce the operand. That unique ID is the ID of the reservation station holding the load instruction.

This information is used to forward operands from the functional units to the instructions waiting data in reservation stations. Whenever a functional unit produces a result value (either an ALU result or a load-word memory request), the functional unit producing the value broadcasts that value as well as the corresponding instruction's ID on the Common Data Bus. All instructions sitting in reservation stations look to this bus and gate in the data whenever one of its operands is invalid and the corresponding tag matches the ID of the data on the common data bus. As soon as an instruction's operands are all valid, the instruction is ready to execute, whether this is before or after the instructions that come before it in the instruction stream. The register file also monitors this bus, and if the ID on the bus matches the ID in any invalid register, the data is gated into that register, and the register is marked as valid.

The architecture is very simple but extremely powerful and capable of resolving all dependencies through the register file. It also provides a form of register renaming that allows the simultaneous or out-of-order execution of multiple reads and writes to the same register. The algorithm, as well as numerous variations on it, has become a stable in modern high-performance CPU design.

2.6 Register Renaming

In the past, renaming has been implemented in the instruction dispatch stage. This stage is considered to be one of the most critical stages in superscalar processors in determining cycle time. The renaming process can be implemented in the instruction fetch stage. Mapping takes place before set selection; this means that all sets go through the mapping logic, which must be replicated according to the associativity of the cache.

The scheme may require that each source register name be mapped using the mapping table to its associated physical register. If there are 32 arhitected registers, the mapping is implemented in the mapping logic as a 32:1 mux. Using a pass-transistor based multiplexer implementation in CMOS; this will probably involve 4 logic stages. The source register fields must control several multiplexes; this requires they to be initially powered up to obtain a fanout estimated to be about 40.

The critical path is determined either by the tag match logic required for cache operations or by the mapping logic discussed previously. The path through the tag match logic is influenced by four factors, namely, the associativity of the cache, its size, the address size, and the fan-out requirements of the tag compared. The tag match logic starts off by checking, depending on the cache size, possibly the high 18-24 bits of the address and the program counter for equality, which can be implemented in about 4 stages of logic stages. The result bits then serve as control for a number of multiplexes. If the implementation attempts to issue 4 instructions per cycle, this implies a fanout of about 1284. This requires a powering tree that will add 2-3 logic stages, assuming some of the powering is buried in the compare.

Putting all the requirements together, we observe that both paths require about the same number of logic stages: about 6-7. This suggests that, assuming CMOS technology and a set associative cache, the proposed scheme may not increase the machine cycle time.

Notice that only the source registers go through the mapping logic; specifically, the time at which the opcodes become available to the instruction decode stage does not change. This may allow us to overlap some of the mapping with the initial phases of the instruction decode in the decode stage, if necessary.

None of the other mechanisms required is added to the cycle time. Updating the map to reflect any remapped registers is not time critical. It can be done in the next cycle, in parallel with the I-cache access. The mechanism to free physical registers is also not time-critical. All that matters is that there be enough free physical registers, not which ones, i.e. throughput matters, not latency. Expanding the number of physical registers by the number required per cycle can compensate adding an extra cycle to the time to free a physical register.

Generally, there will be several instructions with no output register, e.g., branches, and some with only one input register. Superscalar issue adds a related problem if some source register name is the same as the result register name of some instructions fetched at the same time, but prior in the program order, it must use the new physical register allocated to that register name, rather than the entry in the mapping table. It would seem that we would need to do some decoding in the instruction-fetch stage.

However, we can delay these decisions till the next cycle. If the instruction did not have an output, or multiple instructions had the same output register, the mapping is updated appropriately. Thus, decodes are not added to the critical path. A similar fix up has to be performed in the instruction. The fanout can be of this order, or worse, even if fewer instructions are issued per cycle, depending on the physical organization and read out of the cache arrays, decode stage. Determining the appropriate register to be used can be done in parallel with register access. If the physical register, which is provided by mapping table, is incorrect; the value was read can be discarded. This results in no loss in performance. Either the instruction did not need that value, or the wrong register was accessed because some other instructions fetched in parallel had the same output register name. In the second case, the value needed is the one that will be provided by the other instruction. That instruction, obviously, cannot have been completed, so the instruction with the incorrectly mapped register cannot be issued immediately, anyway. Thus, there is at least a full cycle in which to determine the correct register mapping.

One critical path in register renaming is the path from the instruction fetch to execution. On a vanillaRISC processor, this path is implemented in two stages, Instruction-Fetch and Instruction-Decode/Operand-Fetch. Logically, it follows the following steps:

Fetch ~ Decode ~ ReadValues ---* Execute

Adding renaming adds an extra step:

Fetch ~ Decode ~ Rename ~ ReadValues ~ Execute

If these steps are implemented serially, obviously either an extra stage will be required, as in [2, 9], or the cycle-time of the existing two stages will be increased. The only way to avoid this is to implement some of the steps in parallel. The time to perform an associative lookup is obviously going to be greater than that of a register access. It may turn out that the associative access will end up increasing the cycle time.

The mapping mechanism used can be executed in parallel with elements of the instruction fetch stage without impacting the critical path through that stage. The rest of the path should not change from that in a vanillaRISC processor. In particular, reading values involves a normal register access. Associative lookup requires content-addressable storage to be implemented efficiently. This storage requires complex and expensive hardware.

One major source of complexity is determining when physical registers can be reused. This happens when the register has been written to, there be no outstanding reads and the associated architect register has been remapped. These must all be true if all instructions up to and including the instruction that displaced the physical register has been completed. Clearly, the instruction that allocated the physical register must have been completed, and therefore the register must have been written to. All the instructions that could access this register must precede the displacing instruction, and since they must have completed, all reads of the register must have been accomplished. And, of course, the register must have been remapped. This criterion is fairly simple to implement.

2.7 VLIW Processors

A VLIW processor has many independent functional units but it doesn't try to schedule them dynamically. Each clock cycle the processor fetches a very long instruction formatted with a separate field for each functional unit. Control is very simple: each instruction field is sent to its respective functional unit. The processor usually has no logic to detect hazards and cause stalls so the compiler must schedule the code so no hazards will occur at run time (Kenneth E. Batcher, 2002).

Limitations in Multiple-Issue Processors

The VLIW processor just described issues up to four instructions per clock cycle: will a multiple-issue processor with ten times as many resources, issuing 40 instructions per clock cycle, run ten times as fast. The following factors make it difficult to scale up a multiple-issue processor (Kenneth E. Batcher, 2002):

- Limitations of Instruction Level Parallelism (ILP) in programs*:* Each program has only so much ILP. Most of the ILP is in parallelizable loops and some loops aren't parallelizable.

1. Some fraction of a program is not parallelizable so Amdahl's Law limits the speedup: the more one speeds up the parallelizable part of a program the less time a processor spends in that part.
2. Pipeline depth magnifies the problem. To get good usage of the resources one must find about:

- (Number of functional units) * (average pipeline depth)
- Independent operations in the code. The higher this product the less likely the code has that many independent operations.
- In general, large problems have more ILP than small problems. A processor with a large number of instruction issues per clock cycle needs large problems to solve: small problems will waste its resources. Manufacturers of such machines are lucky that there are always a number of customers with very large problems to solve.

3. Building the hardware: Scaling up the number of functional units (issues per cycle) adds a burden to other hardware resources.

- Doubling the number of functional units doubles the number of operands fetched and written each clock cycle: each register file needs twice as many registers, twice as many read ports, and twice as many write ports. Twice as many instructions must be issued so twice as much code must be fetched each cycle. Memory data traffic is also doubled: twice as many memory operands to read/write each cycle so the functional units aren't starved for data.
- It's much easier to add more functional units than it is to scale up the other hardware resources. Memory and register file technology puts a limit on how far a multiple-issue processor can be scaled up.

4. Superscalar Limitations: Dynamic scheduling imposes a limit to the number of functional units: each clock cycle it compares all result destinations with all instructions waiting for source operands. Doubling the number of functional units quadruples the complexity of dynamic scheduling; tripling the resources multiplies the complexity nine-fold. The cost of dynamic scheduling grows as the square of the number of functional units.

- VLIW Processor Limitations: Static scheduling by the compiler keeps all functional units in lock step. If any functional unit stalls (for a cache miss, page fault, etc.) all other functional units must stall as well: the number of functional units multiplies the cost in performance for any stall.

1. Changing the number of functional units in a VLIW processor changes the binary code: binary code for a scaled-up version is not compatible with code for a low-cost scaled-back version.
2. Compiler Support for Exploiting ILP

Detecting and Eliminating Dependencies

Dependencies must be detected in order to re-schedule code, determine which loops have parallelism, and to eliminate name dependencies (Kenneth E. Batcher, 2002).

A data dependence is relatively easy to find if the operand producing the dependence is a scalar with the same name in all instructions that define and use it. Arrays complicate detection: x [i] and x [j] refer to the same variable when i = j. Pointers in languages like C are even worse: several pointers can point to the same variable.

A loop is a good place to look for ILP: if there is no cycle of dependencies the iterations of the loop can be run in parallel. Consider the following example:

```
For i: = 1 to 100 do
Begin
        A [i]: = B [i] + C [i];
        D [i]: = A [i] * E [i];
End;
```

There are no loop-carried dependencies so the 100 iterations can be run in parallel but the two statements in the loop body can't be interchanged because of the data dependence involving A [i]. Optimum machine code for the second statement in the loop body would not explicitly load *A [i]* from memory but use the result of the first statement (Kenneth E. Batcher, 2002).

Software Pipelining

Software pipelining is a technique for reorganizing loops so that each iteration of the new loop contains instructions from different iterations of the original loop.

Software pipelining and loop unrolling are two techniques for hiding pipeline latencies. Software pipelining uses less code in the loop body and less registers but loop unrolling also reduces loop overhead. For pipelines with very long latencies a compiler might want to use both techniques (Kenneth E. Batcher, 2002).

Trace Scheduling

For processors with many instruction issues per clock cycle a compiler might have to resort to trace scheduling to find enough ILP to keep the processor busy: it extends loop-unrolling by finding parallelism across other conditional branches besides loop branches.

The effectiveness of trace scheduling depends on how well the compiler can predict the most likely outcomes of conditional branches. From these predictions it selects a trace*:* a sequence of basic blocks that the processor will frequently follow. Code in the trace is then re-scheduled to achieve a high issue rate (Kenneth E. Batcher, 2002).

Hardware Support for Extracting More Parallelism

The effectiveness of the compiler optimizations is limited when the behavior of branches is not predictable. Here we show some ways of modifying the hardware to exploit ILP (Kenneth E. Batcher, 2002).

Conditional Instructions

A compiler can use conditional instructions for speculative code as long as the exception behavior of a program is not changed. When the condition is false, a conditional instruction should act like a true no operation (nop) and never cause an exception. When the condition is true, exceptions are allowed. Conditional instructions are helpful for implementing short alternative control flows. Control dependencies are replaced by data dependencies, which makes scheduling easier. The following factors limit the utility of conditional instructions (Kenneth E. Batcher, 2002).

- Time is still consumed when the condition is false. This is important when implementing a long alternative control flow: the same time is always consumed whether control goes through the alternative or not.
- They only test one condition. Additional instructions are required to test the AND or OR of several conditions.
- Only a few instructions can be made conditional without impacting clock rate.

Compiler Speculation with Hardware Support

There are two constraints that should be satisfied when instructions are moved. Whenever a compiler speculates it violates the constraint, where an instruction that's control dependent on a branch can't be moved to a place where it is no longer control dependent on that branch. The constraint can be violated as long as the program still runs correctly: that is, the exception behavior and the data flow of the program are preserved (Kenneth E. Batcher, 2002).

Preserving the exception behavior of a program imposes a severe restriction on the amount of speculation a compiler can perform. In particular, all memory reference instructions and most Floating Point (FP) instructions can cause exceptions so a compiler can't execute any of these instructions speculatively (Kenneth E. Batcher, 2002).

Compiler speculation can be allowed to modify the exception behavior of a program as long as the program still runs correctly. There are three methods to allow this, so the compiler can speculate more ambitiously:

1. Hardware-Software Cooperation for Speculation: Each program-caused exception is either resuming or terminating.
2. A resuming exception is handled in the normal way: the Operating System (OS) trap-handler is called to fix the cause and the program is resumed with the offending instruction. The program will still run correctly.
3. A terminating exception is handled differently: rather than terminating the program, the program resumes with the offending instruction producing an

incorrect result. If a speculative instruction caused the terminating exception the incorrect result is not used and the program still runs correctly.

There are two problems with this method of hardware support for compiler speculation:

4. Speculative coding may suffer a higher frequency of resuming exceptions to slow it down. Instead of improving performance with speculative code the net result could be a loss in performance!
5. Bugs in code may not be visible.

This method of hardware support is only acceptable if it can be turned off so fatal errors always terminate programs. Every programmer should use this hardware mode to thoroughly debug a non-speculative machine version of the source code before compiling a speculative machine version (Kenneth E. Batcher, 2002).

- Speculation with Poison Bits*:* Poison bits make code bugs more visible. A bit is added to each machine instruction to flag it as speculative or non-speculative.

1. A terminating exception caused by a non-speculative instruction is always fatal and the program is terminated with a fatal error message.
2. An exception caused by a speculative instruction is never fatal and the program continues: an exception that's normally terminating is changed to a resuming exception with the offending instruction producing a bad result.
3. To flag a bad result, a poison bit is added to every register. Only speculative instructions are allowed to use poisoned register values: a poisoned source operand produces a poisoned result. If a non-speculative instruction tries to use a poisoned register value, a terminating exception occurs. Store-to-memory

instructions are always non-speculative so memory can neither store poisoned values nor any values at poisoned addresses.

4. One complication with this method of hardware support is when the OS saves and restores registers: poison bits must also be saved and restored.

- Speculative Instructions with Renaming*:* With the two previous methods the compiler has to rename registers whenever scheduling creates WAR and/or WAW hazards: ambitious speculation may use up too many registers. An alternative is to provide buffering and renaming in hardware much as Tomasulo's algorithm does.

1. An instruction that's control dependent on a branch is said to be boosted if the compiler schedules it to execute before the branch. A boosted instruction is flagged to indicate whether the compiler is assuming the branch will be taken or the branch will be untaken. The result of a boosted instruction can be forwarded to other boosted instructions (assuming the same branch outcome) but it is not stored in a register until the actual branch outcome is known: if the branch outcome agrees with the assumption the result of a boosted instruction is stored in the destination register; otherwise the result is ignored (Kenneth E. Batcher, 2002).
2. The method can be extended to allow boosting instructions across multiple branches.

Hardware-Based Speculation

Hardware-based speculation is complex and requires substantial hardware resources but it has some advantages over compiler-based speculation (Kenneth E. Batcher, 2002):

1. Hardware can disambiguate memory references to allow greater speculation.
2. Hardware branch-prediction can be more accurate than compiler branch-prediction.
3. Precise exceptions are easier to generate.
4. No compensating or bookkeeping code is required.

The basic idea is simple: let instructions execute in any order but don't commit their results to registers or memory until it's safe to do so.

Advantages of VLIW (Wen-mei W. Hwu. 1999):

- No runtime dependence checks against previously or simultaneously issued operations
- No runtime scheduling decisions.

The disadvantages can be summarized as:

- No tolerance for different or variable latencies.
- No object code compatibility
- No tolerance for a difference in the set of functional units (Wen-mei W. Hwu. 1999).

A conventional sequential program has a Unit Assumed Latency (UAL). The semantics of the program are understood by assuming each instruction is completed before the next one is issued. There is another scheduling model, which is called Non-UAL (NUAL) scheduling models. It consists of two models (Wen-mei W. Hwu., 1999):

1. Equals (EQ) Model

- Each operation takes exactly its specified latency i.e. the destination register will not be written until latency number of cycles.
- Produce slightly shorter schedules due to register reuse

2. Less-Than-or-Equals (LEQ) Model

- An operation may take than or equal to its specified latency i.e. the destination register can be written any time from issue to latency cycles
- Simplifies the implementation of precise interrupts Provides binary compatibility when latencies are reduced

VLIW as an Architecture

1. Defining attributes

- NUAL: non-unit assumed latencies.
- Multiop: multiple simultaneously issueable operations per instruction:

   a. No flow dependencies between these Operations.
   b. Output and anti-dependencies specified by the assumed latencies

2. Advantages over a sequential architecture (Uniop, UAL)
   a. Explicitly provides independence information in the program
   b. No runtime dependence checks against previously or simultaneously issued operations

c. Economy of register usage

Permits an especially simple mechanism for object code compatibility (Wen-mei W. Hwu, 1999).

2.8 Summary

Four major subjects are discussed in this chapter: the pipelining with, the interrupts as a general concept, then the concept of the precise interrupts and all related difficulties and concepts are fully pointed out. The concept of the imprecise interrupts was stated in brief. Finally, the Tomasulo’s algorithm and register renaming are discussed.

3. SCHEMES FOR HANDLING PRECISE INTERRUPTS IN PIPELINING SYSYEMS

3.1 Preface

This chapter presents different schemes for handling precise interrupts in pipelining systems that is used throughout the thesis.

There are several solutions to the problem of the study. The following are the main solutions in general.

First Solution:

- Ignore the problem (imprecise exceptions):

1. This may be fast and easy, but it's difficult to debug programs without precise exceptions.

- Many modern CPUs, i.e. DEC Alpha 21064, IBM Power-1 and MIPS R800, provide a precise mode that allows only a single outstanding Floating Point (FP) instruction at any time.

1. This mode is much slower than the imprecise mode, but it makes debugging possible.

Second Solution:

- Buffer the results and delay commitment:

1. In this case, the CPU doesn't actually make any state (register or memory) changes until the instruction is guaranteed to finish.
2. This becomes difficult when the difference in running time among operations is large.

- Lots of intermediate results have to be buffered (and forwarded, if necessary).

Variations of the **second** ***solution:***

- History file:

1. This technique saves the original values of the registers that have been changed recently.
2. If an exception occurs, the original values can be retrieved from this cache.
3. Note that the file has to have enough entries for one register modification per cycle for the longest possible instruction.
4. Similar to the solution used for the VAX for auto-increment and auto-decrement addressing.

- Future file:

1. This method stores the newer values for registers.
2. When all earlier instructions have completed, the main register file is updated from the future file.
3. On an exception, the main register file has the precise values for the interrupted state.

Third Solution:

- Keep enough information for the trap handler to create a precise sequence for the exception:

1. The instructions in the pipeline and the corresponding PCs must be saved.
2. After the exception, the software finishes any instructions that precede the latest instruction, which is completed.
3. Technique is used in the SPARC architecture.

Fourth Solution:

- Allow instruction issue only if it is known that all previous instructions will complete without causing an exception.

1. The floating point functional units must determine if an exception is possible early in the EX stage, first couple clocks, in order to prevent the following instructions from completing.
2. Sometimes it requires stalling the pipeline in order to maintain precise interrupts.
3. The R4000 and Pentium solution.

3.2 In-order completion

In-order execution does not fully utilize all functional parts of a CPU. The rule of in-order execution prohibits that subsequent instructions overtake previous instructions. Clearly, if instructions completed in the order they were issued, we could handle an interrupting instruction by allowing it to reach its last pipeline stage and then preventing the completion of all subsequent instructions. This scheme guarantee precise state. The original MIPS implementation used a similar scheme (Mayan, 1996).

Instructions modify the process state only when all previously issued instructions are known to be free of exceptions. This strategy is most easily implemented when pipeline delays in the parallel functional units are fixed. That is, they do not depend on the operands, only on the function. Thus, the result bus can be reserved at the time of issue. However, forcing in-order completion can degrade performance (Mayan, 1996).

3.3 Reorder Buffer Scheme

The reorder buffer is a FIFO (First In First Out) queue that is placed between the output of the functional units and the write-back port of the register file. It keeps the register file in a precise state. The entries of the reorder buffer are en-queued when instructions are issued, each entry contains the following fields: destination register, result value, PC, interrupt status, valid. Instructions are removed from the head of the queue when they have valid result values after they have written back their results. Exceptions are checked for an instruction when the instruction reaches the head of the queue. If an instruction causes an exception all entries behind it in the reorder buffer are discarded and do not write back their results. Bypassing the result values from the reorder buffer is required for maximum performance.

The reorder buffer was developed to solve the problem that, in many pipelined computers even those with in-order instruction issue, execution results are frequently produced out-of-order. For example, this happens when functional units have different latencies. This is what Smith and Pleszkun mean by “pipelined” processors: in-order pipes with functional units that have different latencies. In previous machines, results were typically written to the register file as soon as they were produced, and if the results were produced out-of-order, they could therefore update machine state out-of-order. Such an organization causes problems in the case of handling precise interrupts, during which the machine state is required to reflect that of a sequential machine with in-order instruction completion, otherwise the interrupt is not considered “precise”. Smith and Pleszkun solve the problem by providing a mechanism to allow instructions that generate results out-of-order to be completed in-order. Changes to the state of the machine (register file, memory system) are limited to the time of instruction completion, which is handled in program order, and therefore the state of the machine always reflects that of a sequential implementation (UMD, 2000).

The fundamental idea is coupling of instruction execution and instruction completion. Whereas in previous pipeline organizations execution and completion could be treated as an atomic multi-cycle operation, the reorder buffer separates out the concept of instruction completion as a phase of the instruction life cycle that may or may not happen on the cycle following the generation of results. Thus, the reorder buffer behaves like a holding tank for instructions while they are in the process of being executed (including decode, operand fetch, execution by an ALU, possible memory access, and write-back to the register file). It is easy to imagine many possible structures that would perform such a function (UMD, 2000).

Thus, the reorder buffer keeps the original program order of the instructions after instruction issue and allows result serialization during the retire stage. State bits are stored if an instruction is on a speculative path, and when the branch is resolved, if the instruction is on a correct path or must be discarded. When an instruction completes, the state is marked in its entry. Exceptions are marked in the reorder buffer entry of the triggering instruction. Reorder buffer entries are allocate in the first issue stage and de-allocated serially when the instruction retires.

When an exception is detected, a flag in the instructions' reorder buffer entry is set indicating the exceptional status. Delaying the handling of the exception ensures that the instruction didn't execute along a speculative path. While the instructions are being committed, the exception flag of the instruction is checked (UMD, 2000).

To implement out-of-order, issue requires a buffer, or instruction window between decoder and functional units, as shown in figure 5.5, which is discussed in. Here, the Reorder Buffer is combined with the Tomasulo's algorithm. The Advantages of Tomasulo's Algorithm:

1. Distribution of hazard detection and control logic (Because of distributed reservation stations and Common Data Bus (CDB): Multiple instructions waiting on a single operand can all start execution as soon as the operand is broadcasted on the common data bus).
2. Elimination of stalls for WAW and WAR hazards (Because of register renaming and storing operands into reservation stations as soon as available).

But how do we implement this instruction window? We have two choices:

1. Centralized Window: Buffer every instruction for every functional unit in a common window.
2. Reservation Stations (RSs): Distribute individual buffers to each of the FUs.

The number of instructions found in the instruction window at any given time is greater than the maximum fetch and decode rate. Potentially, we can issue more instructions in a given cycle than this maximum fetch/decode rate. In general, the maximum number of instructions issued in one cycle can be significantly higher than the average instruction execution rate (e.g. more than a factor of 2).

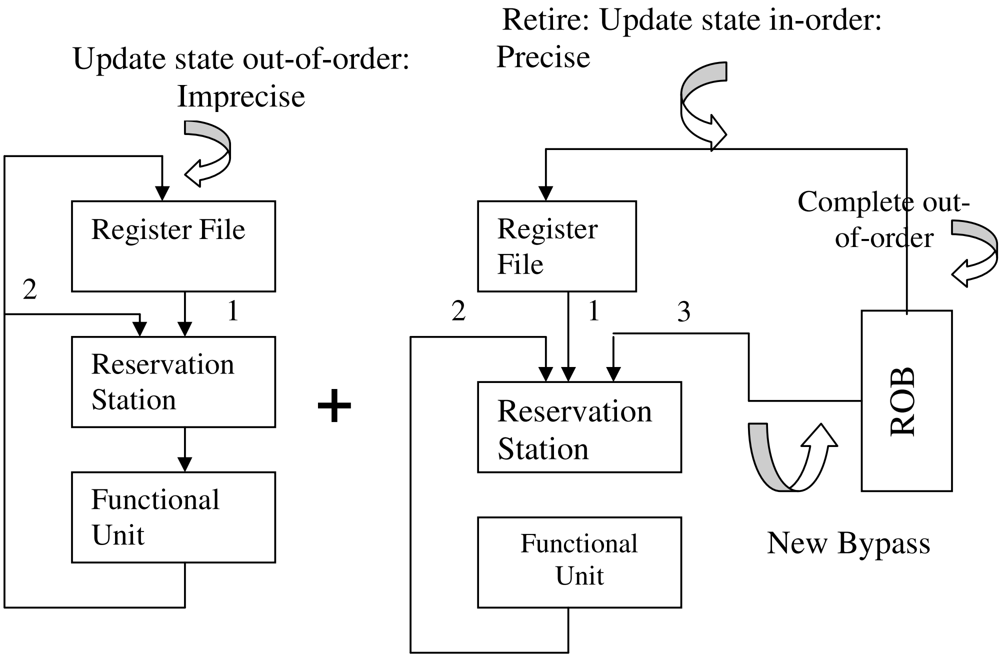


Figure 3.1. Precise Tomasulo Pipeline with ROB

There are some implications of this, some are:

- To maximize performance, we need to support a high instruction issue rate.
- To do this requires simultaneous communication of much operand value to different functional units. Each instruction issued must be accompanied by all of its required operands.
- With a central window, this busing is routed to all functional units.
- With a distributed instruction window (i.e. reservation stations) the window is filled at the average instruction execution rate not the peak rate.

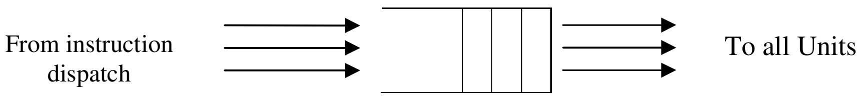
We should note that, in general, if the window is partitioned using RSs, the total size of the instruction window must be larger than if it were centralized in order to support the same amount of look-ahead. Multiple issues and forms of reservation stations are shown in figure 3.2.

Another remarkable point about reservation station; is that reservation stations potentially reduce operand bus routing. Because reservation stations are distributed, they can more easily support the maximum instruction issue rate. In a given cycle, each RS may issue an instruction to an FU since the operands come from local reservation stations. Also, each FU can have a different number of reservation stations assigned to it.

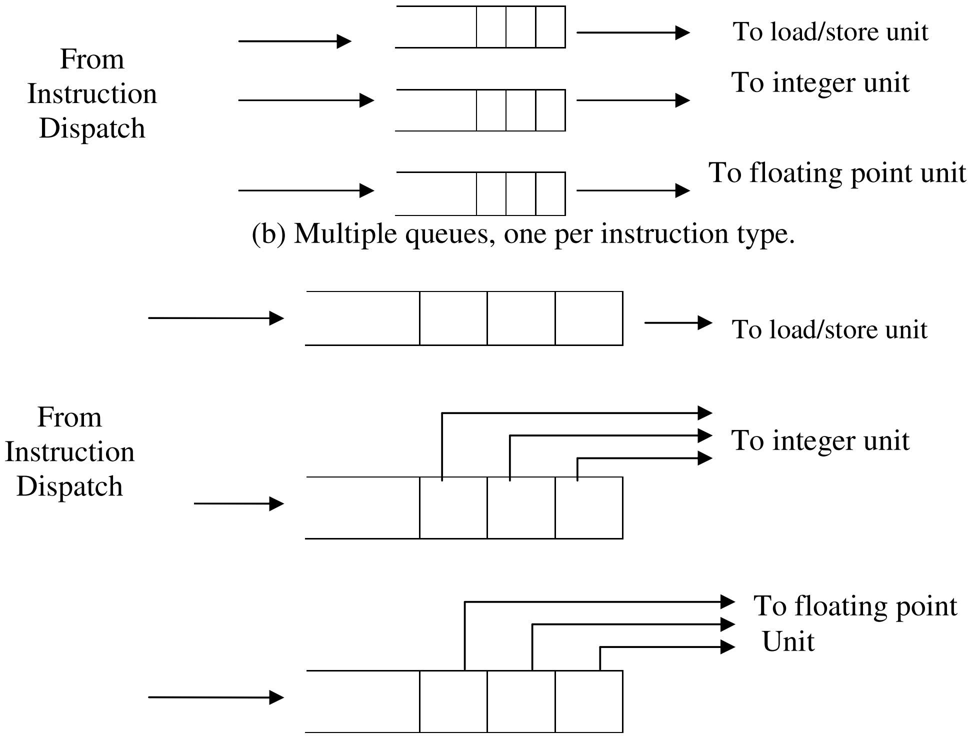
3.4 Reorder Buffer with Forwarding (Bypass) paths

A new mechanism for enforcing RAW dependencies is formed; this is similar to the model presented by Sohi and Vajapeyam (David ,1997).

When an instruction is being issued, the reorder buffer is searched for entries whose destination register field corresponds to a source operand that the instruction needs. If no entry in the reorder buffer matches the register number then the result has already reached the register file and the operand is read from there. If one match is found the instruction issue may be stalled until the reorder buffer entry contains a valid result, whereupon it is used as the operand for the instruction being issued (David ,1997).



a. Single, shared queue



(b) Multiple queues, one per instruction type.

c. Multiple reservation station; one per instruction type.

Figure 3.2. Forms of Reservation Station

In order for results to be used early, bypass path may be provided from the entries in the reorder buffer to the register file output latches as shown in figure 4.8. These paths allow data being held in the reorder buffer to be used in place of register data. The implementation of this method requires comparator for each reorder stage and operand designator. If an operand register designator of an instruction being checked for issue matches a register designator in the reorder buffer, then a multiplexer is set to gate the data from the reorder buffer to the register output latch. In the absence of other issue blockage conditions, the instruction is allowed to be issued, and the data from the reorder is used prior to being written into the register file (James *et al.*, 1988).

There may be bypass paths form some or all of the reorder buffer entries. If multiple bypass paths exist, it is possible for more than one destination entry in the reorder buffer to correspond to a single register. Clearly only the latest reorder buffer entry that corresponds to an operand designator should generate a bypass path to the register output latch. To prevent multiple bypassing of the same register, when an instruction is placed in the reorder buffer, any entries waiting the same destination register designator must be inhibited from matching a bypass check (James *et al*., 1988).

3.5 Future File Scheme

Typically, these and other mechanisms that perform the function of ensuring in-order commitment of machine state are all called by the microprocessor design community “reorder buffers” whether the description is technically accurate or not (UMD, 2000).

Reorder buffer holds only instruction execution states (results are in rename registers). Johnson’s description of a reorder buffer in combination with a so- called future file. The future file is similar to the set of rename registers that are separated from the architectural registers. In contrast, Smith and Pleskun described a reorder buffer in combination with a future file, whereby the reorder buffer and the future file receive and store results at the same time. Other reorder buffer type: The reorder buffer holds the result values of completed instructions instead of rename registers. Moreover the instruction window can be combined with the reorder buffer to a single buffer unit (UMD, 2000).

The future file is a mechanism described by Smith and Pleszkun. There is another modified version of it, which is described by Johnson. It consists of a model similar to the simple reorder buffer with the addition of an extra register file known as the future file. As in the simple reorder buffer, the reorder buffer holds look-ahead state and the register bank hold in-order state. In normal operation the future file holds the architectural state, however upon recovery from exception the architectural state is formed by a combination of the future file and the register bank (David, 1997).

3.6 History Buffer Scheme

The history buffer is one of three mechanisms proposed by Smith and Pleszkun to handle precise interrupts in pipelined processors. Like check pointing mechanisms, the history buffer maintains some state to be restored when an exception is encountered. However, unlike check pointing only the part of the state, which has changed recently, is stored.

3.7 Summary

The main five handling schemes for the precise interrupts in pipelining systems were introduced in this chapter. The first method is the In-Order-Completion of instructions. The second method is the Reorder Buffer, the same structure of this Reorder Buffer but with the forwarding (Bypassing) paths is the third method. The fourth and fifth methods are the Future File and the History Buffer respectively.

Finally, Implementing precise interrupts through in-order completion degrades performance by reducing the amount of pipelining possible. Implementing precise interrupts with out-of-order completion requires significant amount of hardware. Worse, the extra hardware can add to the machine’s cycle time, thus degrading performance. To reduce the costs of interrupt handling with out-of-order completion, we must consider the requirement of the different classes of interrupts (Mayan, 1996).

4. IMPLEMENTATION OF IN ORDER COMPLETION

In this chapter, we introduce the architecture and the structure of the system, which we use. The algorithm of the In-Order Completion scheme is implemented and analyzed in this chapter.

4.1 Architectural Model

For describing the various techniques, an architectural model is chosen as declared in (James *et al*., 1988). It is a register-register architecture where all memory accesses are through registers and all functional operations involve registers. In this respect, it bears some similarity to the CDC and Cray architectures. The process state in this architectural model consists of the program counter, the general-purpose registers, and main memory. The architecture is simple, has a minimal amount of process state, can be easily pipelined, and can be implemented straightforward with parallel functional units like the CDC and Cray architectures (James *et al*., 1988).

In our model as illustrated, we emphasize scalar architectures (as opposed to vector architectures) because of their applicability to a wider range of machines (James *et al*., 1988).

Figure 4.1 shows the parallel pipelined implementation for this simple architecture. It uses an instruction fetch/decode pipeline with processing of instructions in order. The final stage of the fetch/decode pipeline is an issue register where all register interlock conditions are checked. Here the memory access function is implemented as one of the functional units. The registers of the operands are read at the time an instruction is issued. There is a single result bus that returns results to the register file. This bus may be reserved at the time an instruction is issued or when an instruction is approaching completion. A new instruction can be issued every clock period in the absence of register or result bus conflicts.

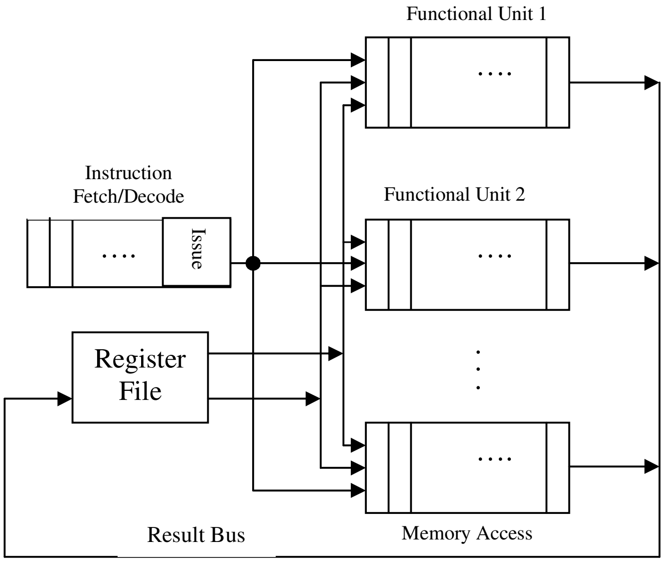


Figure 4.1. Pipelined Implementation of our architectural model (James *et al.*, 1988).

Table 4.1. Sample1 of code used in the work.

State. No.	**Statement**	**Comments**	**Execution time**
0	R1← 0	Init. Loop index	2 clock periods
1	R0 ← 0	Init. Loop Counter	2 clock periods
2	R5 ← 1	Loop Inc. Value	2 clock periods
3	R7 ← 100	Maximum Loop Count	2 clock periods
4	**Loop**: R1← (R2 +A)	Load A(I)	11 clock periods
5	R3 ← (R2 +B)	Load B(I)	11 clock periods
6	R4 ← R1 +*f*R3	Floating Add	6 clock periods
7	R0 ← R0 +R5	Inc. Loop count	2 clock periods
8	(R0 + C) ← R4	Store C (I)	11 clock periods
9	R2 ← R2 +R5	Inc. Loop index	2 clock periods
10	P = Loop: R0 !=R7	Cond. Branch not equal	

Table 4.2. Sample2 of code used in the work.

State. No.	Statement	Execution time
A	LD F6, 34(R2)	1 clock period
B	LD F2, 45(R3)	1 clock period
C	MULTD F0, F2, F4	10 clock periods
D	SUBD F8, F6, F2	1 clock period
E	DIVD F10, F0, F6	40 clock periods
F	ADDD F6, F8, F2	1 clock period

To demonstrate how an imprecise process state may occur in our architectural model, consider the section of code mentioned in table 4.1, which sums the elements of array A and B into array C. Another sample of code used in the work is introduced in table 4.2.

4.2 Interrupts prior to Instruction Issue

Before preceding with the various precise interrupts methods, I would like to first consider interrupts that can be handled the same way by all methods, these are interrupt that may occur prior to instruction issue.

As the architectural model indicates, instructions are queued to be issued in sequence. This simplifies the handling of interrupts since there is no change can be applied to the process state before instructions have been issued.

There are many such interrupts, such as, privileged instruction faults and unimplemented instructions; others may include external interrupts. These exceptions can be checked at the issue stage.

When such an exception occurs, this can be handled as follows:

1. Instruction issue is halted.

2. A wait state is forced until all previously issued instructions have been completed.
3. After they have completed, the process is in a precise state (which includes the precise contents of the registers, the program counter, conditional registers, index registers, and that portion of the main memory being used for data).
4. The precise instruction is the one whose program counter value being held in the issue register. This makes the memory and the program counter in a consistent state.

There are several mechanisms for implementing precise interrupts on pipelining implementations. As we have indicated, the main source of difficulty is the order of the completion, not the order of issue. For simplicity, unless otherwise stated, we assume that instructions are issued in order- that is, in their program order (Mayan, 1996). So, our concentration will be on exceptions occurring after the instructions are issued.

4.3 In-Order Instruction Completion

4.3.1 Scheme Description

With this scheme, as described in section 3.2, instructions modify the process state only when all previously issued instructions are known to be free of exceptions as illustrated in the previous section.

There is logic on the result bus that checks for exception conditions in instructions, as they complete. This control information identifies the functional unit that will supply the result and the destination register of the result. It is also marked "*valid*" with a validity bit. Each clock period, the control information is shifted down one stage toward stage one. When it reaches stage one, it is used during the next clock period to control the result bus so that the functional unit result is placed in the correct result register.

The Result Shift Register is shown in Table 4.3. Here the stages are labeled *1* through *n*, where *n* is the length of the longest functional unit pipeline. An instruction that takes *i* clock period reserves stage *i* of the result shift register at the time it is issued. If the stage already contains valid control information, then the issue is held until the next clock period, and stage *i* is checked once again. An issuing instruction places control information in the result shift register.

Table 4.3. Result Shift Register

Stage	**Functional Unit Source**	**Destination Register**	**Valid**
1			0
2			0
3			0
4			0
5			0
. .	. .	. .	. .
N			0

↑ Shift Upward

Instruction In stage 1 writes the result-bus

4.3.2 Scheme Analysis

The method basically can be used regardless of whether precise interrupts are to be implemented or not. If we still disregard the precise interrupts. It is possible for a short instruction to be placed in the resultant pipeline in stage *i,* when previously issued instructions are in stage *j*, *j>i*. This leads to instruction finishing out of the original program sequence. If the instruction at stage *j* eventually encounters an exception condition, the interrupt will be imprecise because the instruction placed in stage *i* will complete and modify the process state even though the sequential architectural model says *i* does not begin until *j* completes.

To implement precise interrupt with respect to registers using this methodology; an issuing instruction using stage *j* should "reserve" stages *i<=j* that were not previously reserved by other instructions. And they are loaded with null control information, so that they do not affect the process state. This guarantees that instruction-modifying registers finish in order.

Table 4.4. Result Shift Register including the program counter to implement precise interrupts.

(a) Sample 1 (James *et al.*, 1988).

Stage	Functional Unit Source	Destination Register	Valid	Program Counter
1			0	
2			0	
3			0	
4			0	
5	FLPT ADD	4	1	6
. .	. .	. .	. .	. .
N			0	

Shift Upward

Instruction In stage 1 writes the result-bus

(b) Sample 2

Stage	FU/OP	Destination Register	Valid	Program Counter
1				
2	LD (1)		0	A
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				

Shift Upward

Instruction In stage 1 writes the result-bus

To implement precise interrupts with respect to the program counter, the result shift register is widened to include a field for the program counter of each instruction as shown in table 4.4.-a. This field is filled as the instruction is issued. When an instruction with an exception condition appears at the result bus, its program counter is available and becomes part of the saved state.

Experiments:

Many experiments were done to test the scheme in order to handle the precise interrupts. The original functional units (FUs) used, were: Integer FU (ADDD, SUBD), the Floating Point FU, the Load/Store FU, the MultD/DivD FU, and the Branch FU. When we changed these FUs, we used more FUs to increase the throughput. The Integer FU is extended to be two FUs, one for ADDD and the other to SUBD, the same then is done to the MultD/DivD FU, it becomes MultD FU and DivD FU, the Load/Store FU becomes one for Load and the other for Store, the Floating Point FU can be further divided to have one for each floating point operations. The dependency hazards were resolved fully or partially. Here are the results of the experiments done:

A- The first sample of code is applied to the approach. The results were as follows:

- The total number of shifts and tests were the same for all experiments. Fig.4.4 shows the percentage of failed to succeeded tests.
- Fig. 4.2, 4.3 show how is the number of shifts and failed tests are changed in each experiment.

Resolving the dependencies among instructions, but still having sharp peaks at the instructions that has RAW dependency achieves a slight improvement.

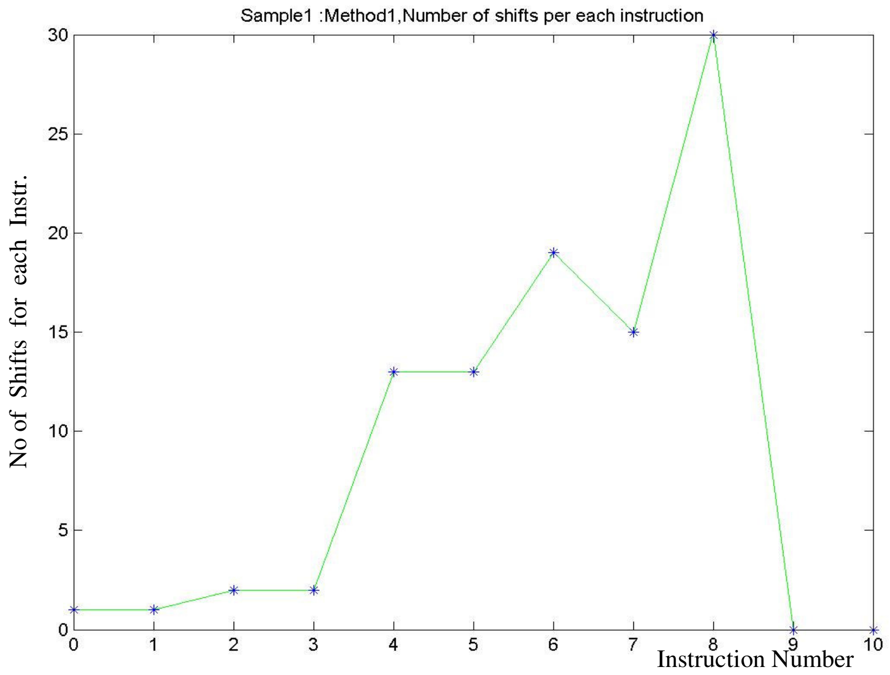
B- The second sample of code is applied. The results were as follows:

- The total number of shifts and tests were not changed in any of the experiments done. Fig.4.6 shows the total number of tests and the percentage of failed to succeeded tests.
- Fig. 4.5, 4.7 show how is the number of shifts and failed tests are changed after each modification.

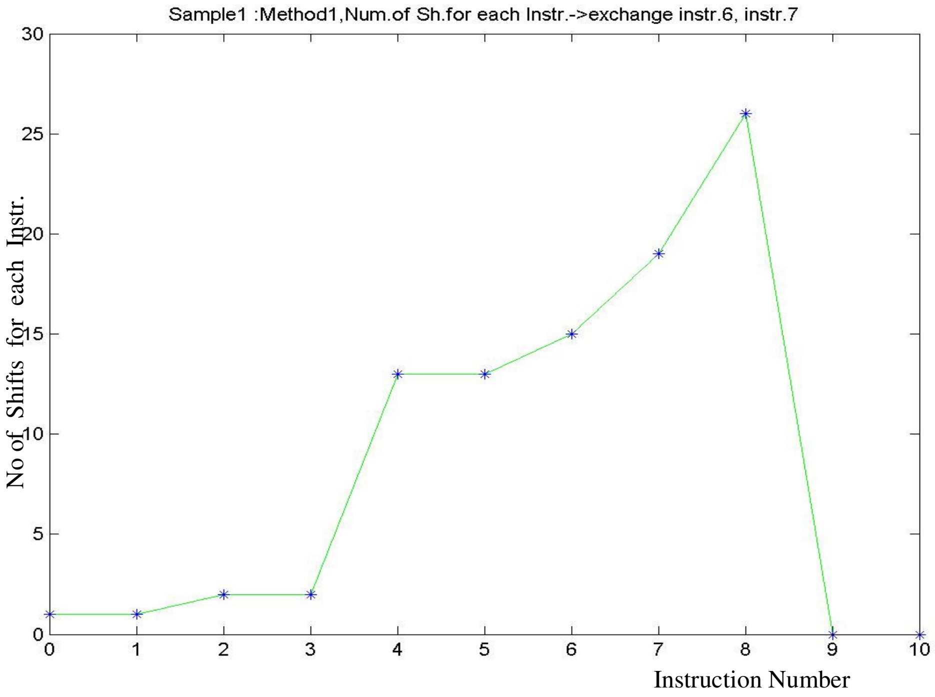
As we resolve dependencies among instructions, we get better performance regarding to the number of failed tests for each Instructions, i.e., tests are succeeded more.

So, when there is no hazard dependency, this will give better performance to the algorithm used. No need to change the FUs, since all instructions are issued and executed in order, and no conjunction of instructions can occur at the FU.

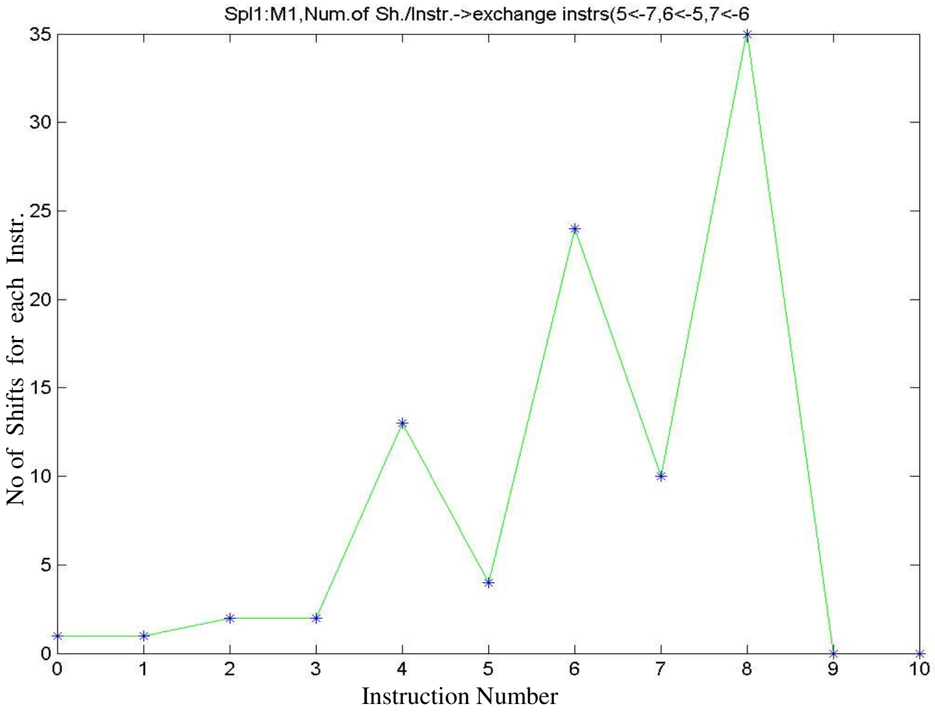
The main advantages of the In Order Completion are that it is simple and easy to be implemented. Also, it is free of any kind of dependency. The nature of instructions and dependencies among them affect the results obtained.



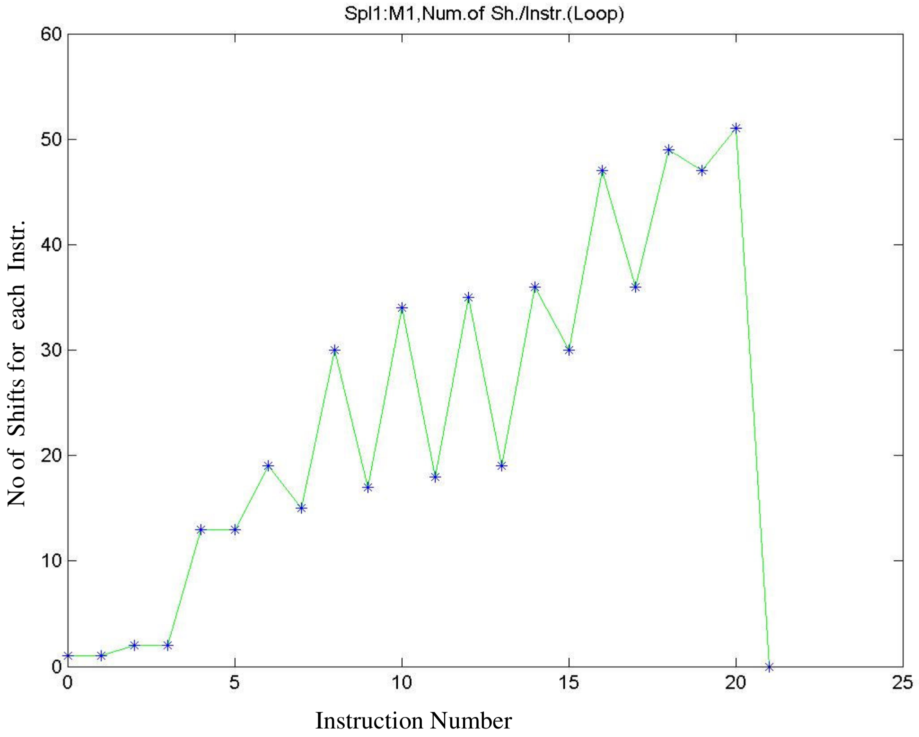
(a) original Code.



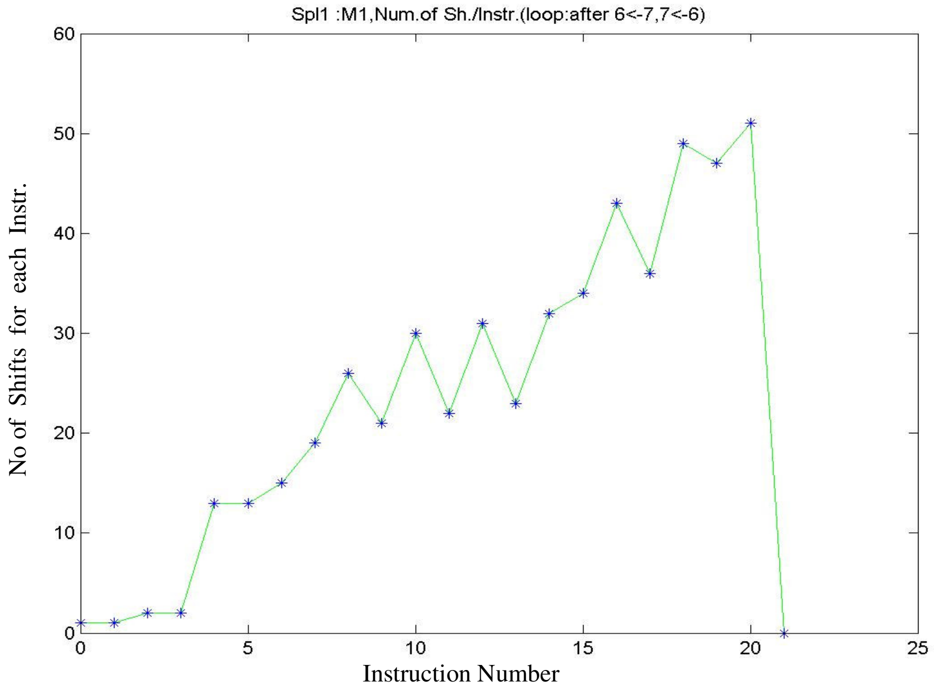
(b) Exchanging Instructions 6,7.



(c) Reordering instructions to resolve all Dependency hazards.

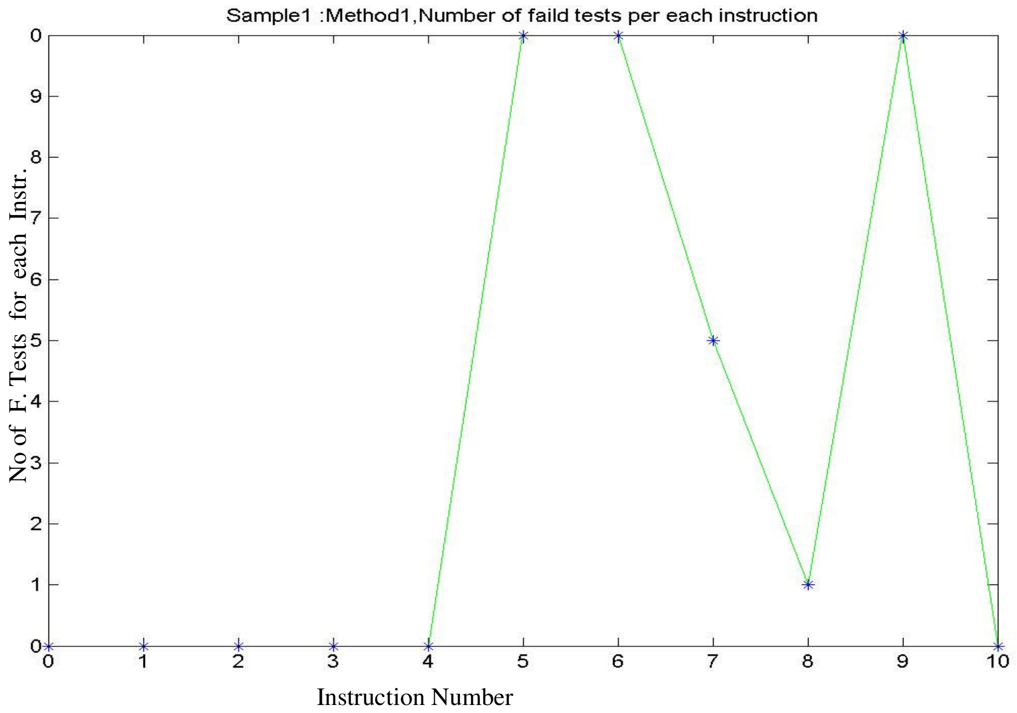


(d) Loop of the original code.

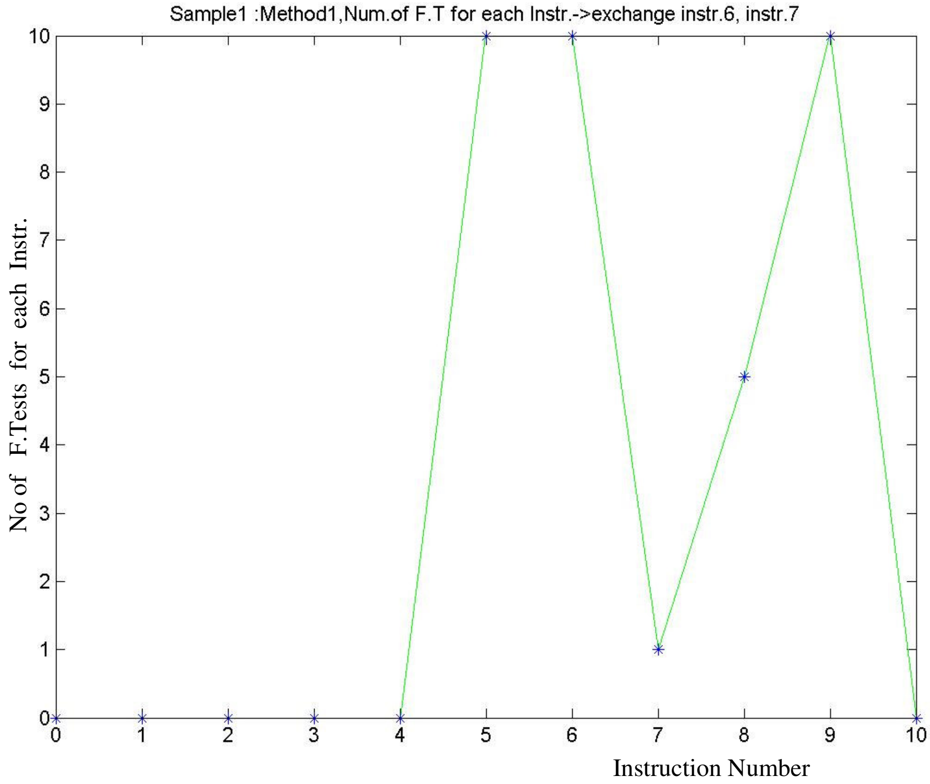


(e) Loop after resolving dependencies.

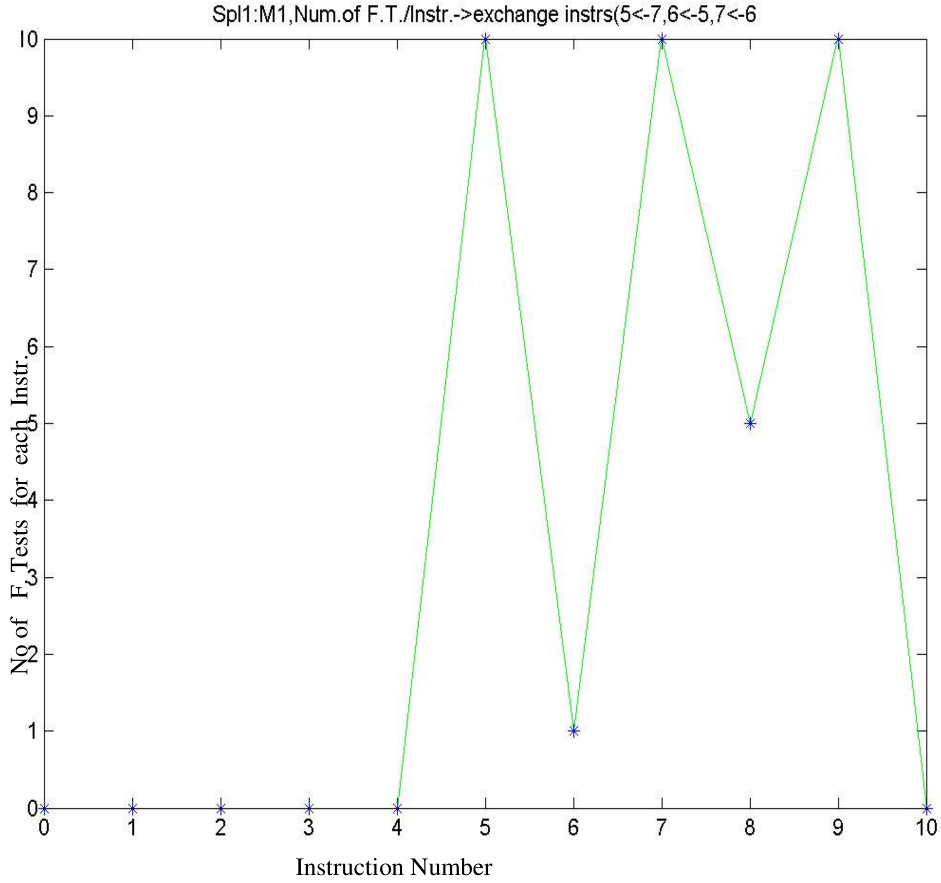
Figure 4.2. In-Order Completion. Sample 1, the number of shifts per each instruction.



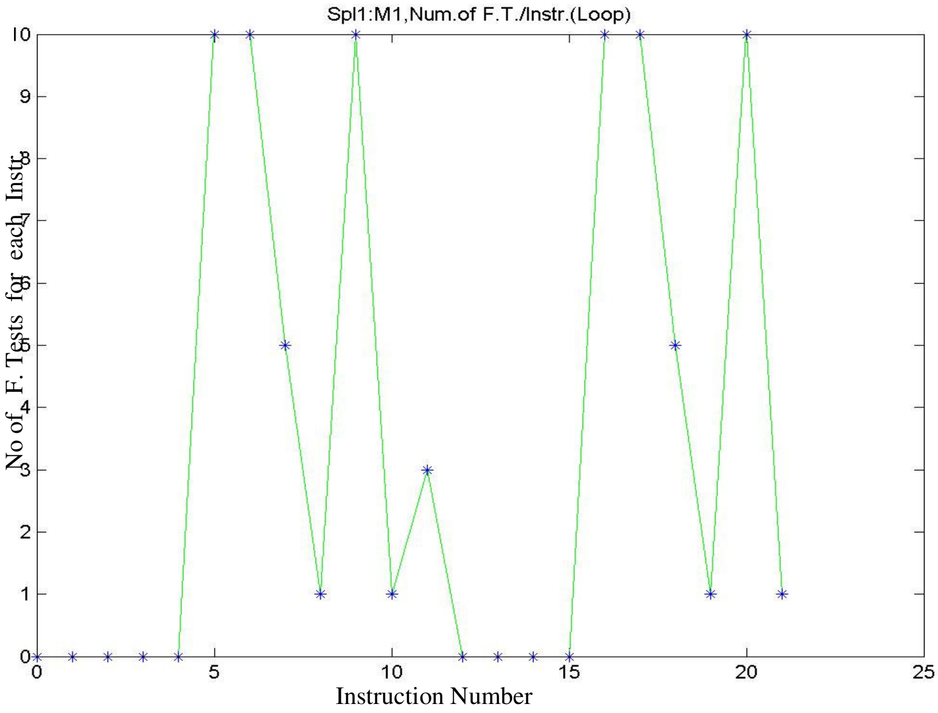
(a) Original Code.



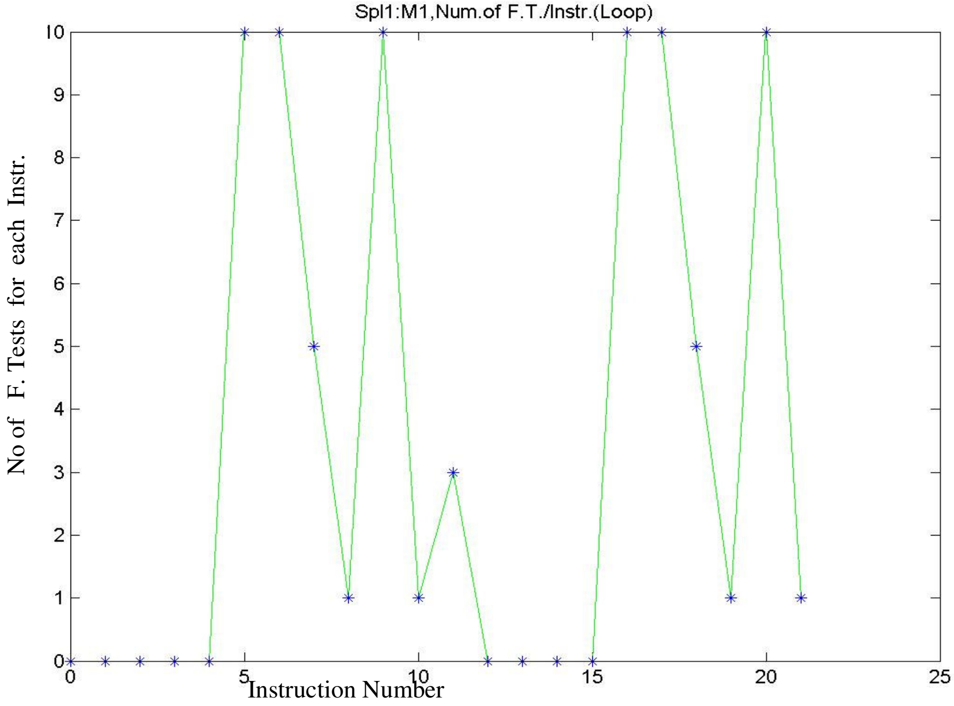
(b) Exchanging Instructions 6,7



(c) Reordering instructions to resolve all Dependency hazards.

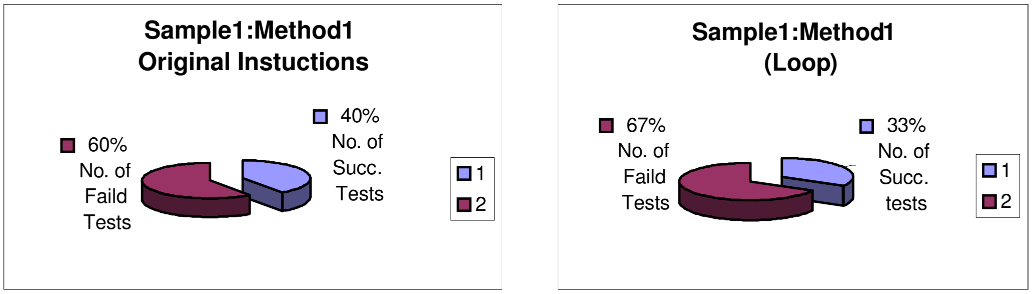


(d) Loop of the original code.



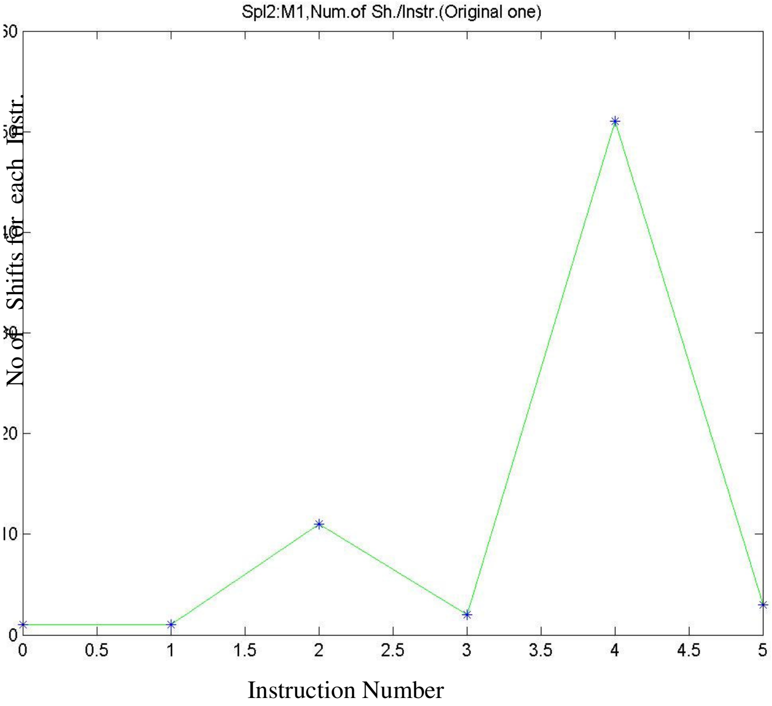
(e) Loop after resolving dependencies.

Figure 4.3. In-Order Completion. Sample 1, the number of failed tests per each instruction

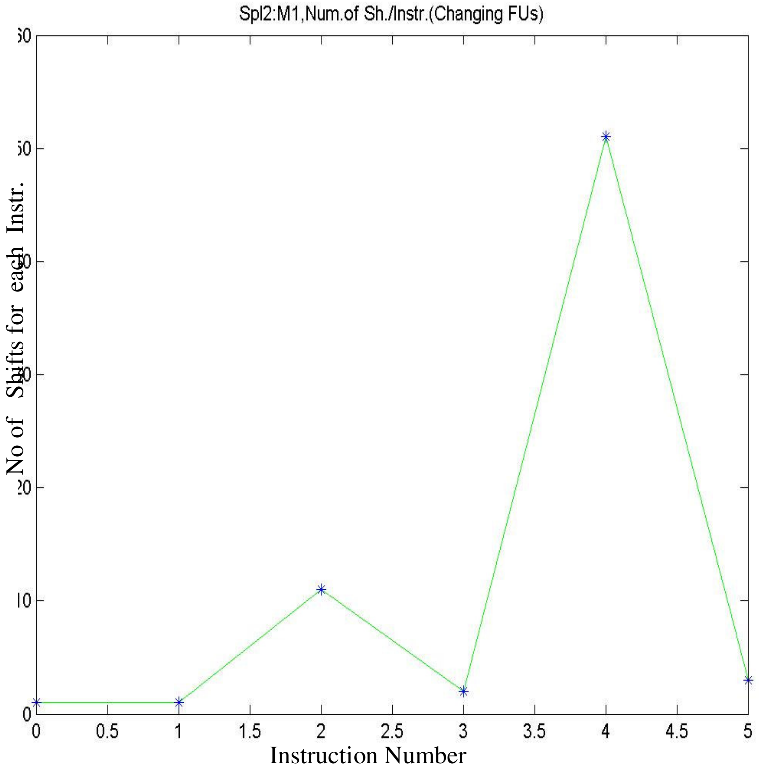


(a) Original Instruction Code (b) Original Instruction Code (Looping)

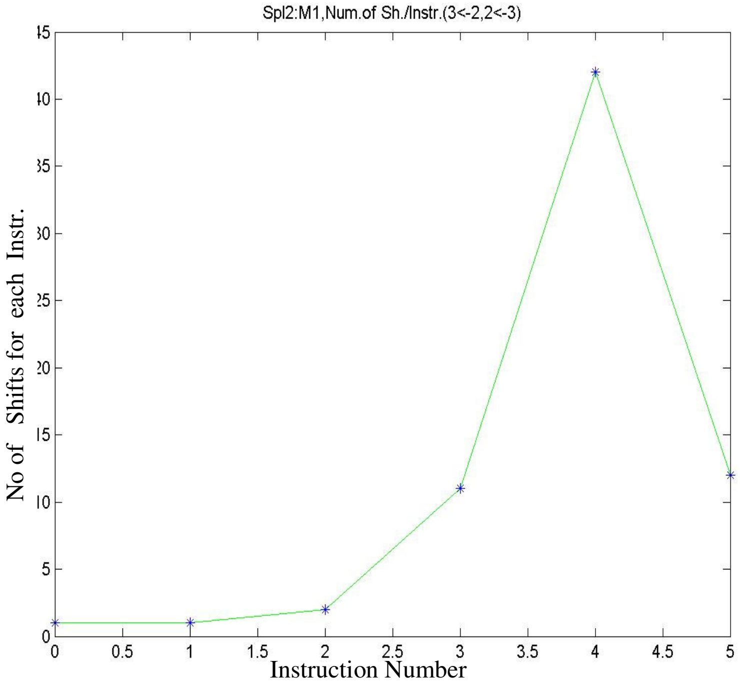
Figure 4.4. In-Order Completion. Sample 1. The number of failed tests and Succeed tests to the Total number of tests.



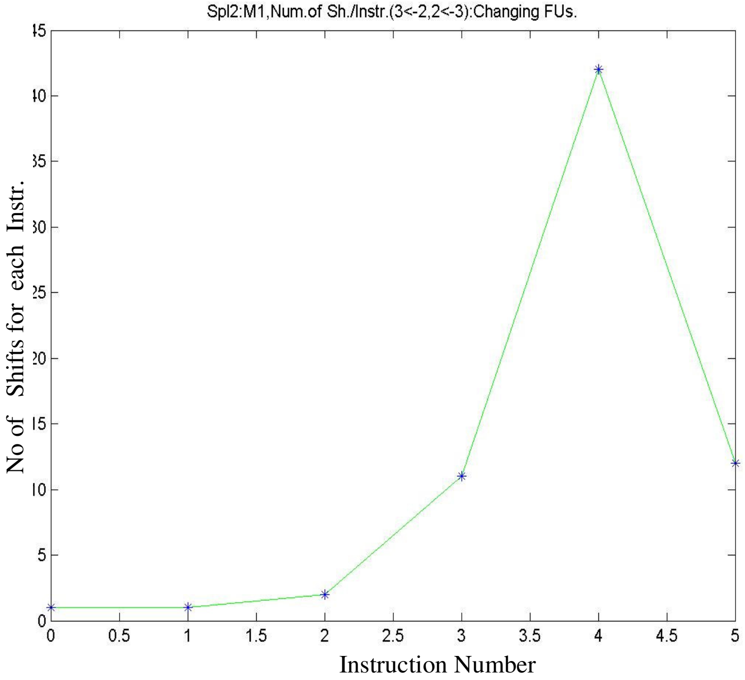
(a) Original Code.



(b) Changing FUs.



(c) Reordering instructions to resolve all Dependency hazards.



(d) Reordering instructions to resolve Dependency and Change FUs.

Figure 4.5. In-Order Completion. Sample 2. The number of shifts per each instruction.

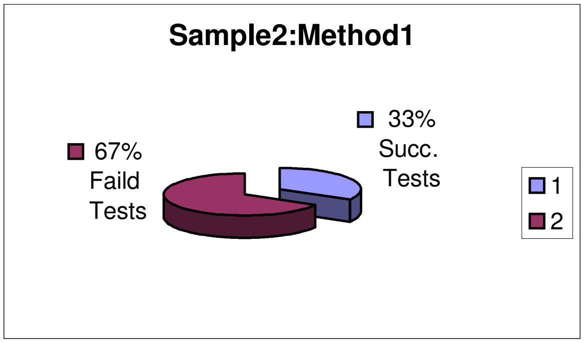
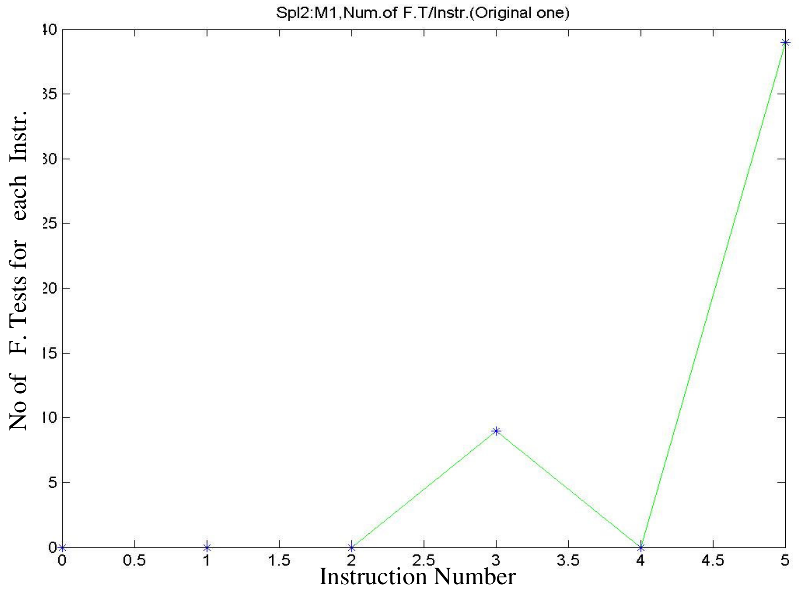
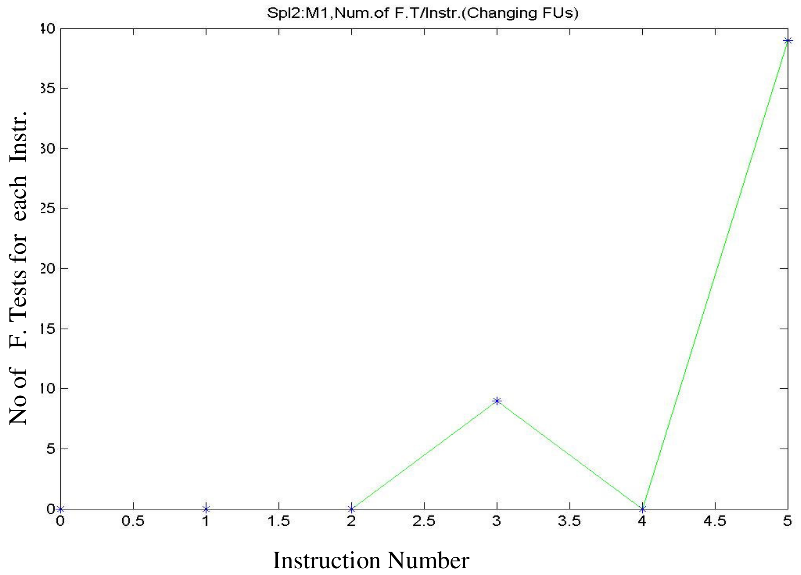


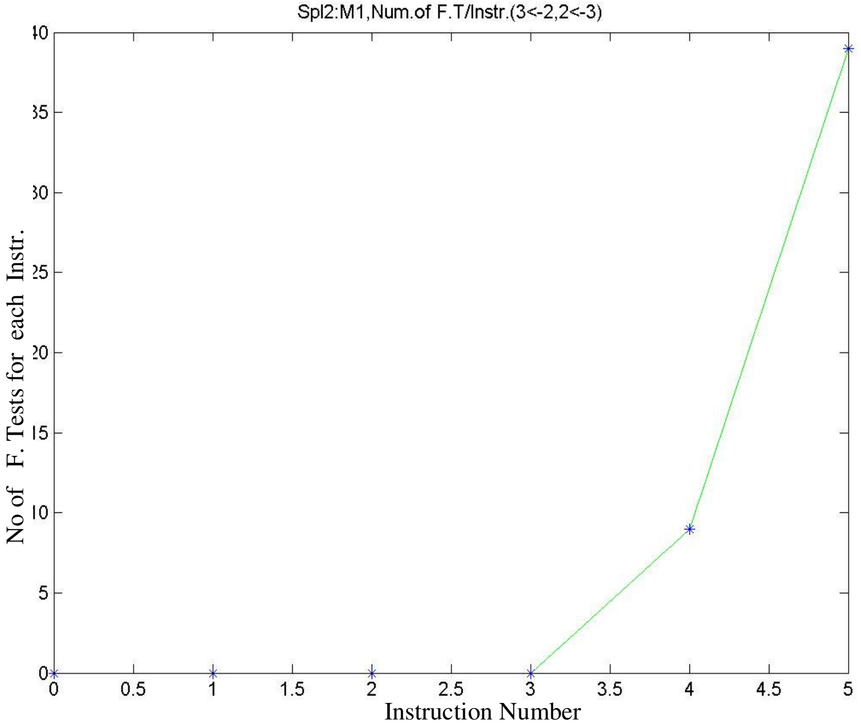
Figure 4.6. In-Order Completion. Sample2. The number of failed tests and Succeed tests to the Total number of tests.



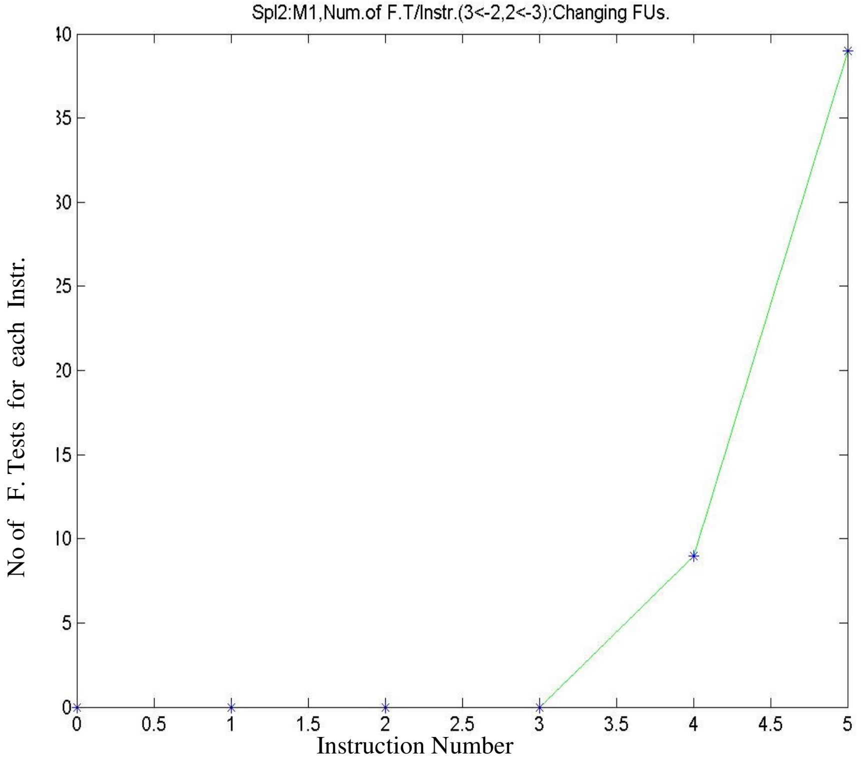
(a) Original Code.



(b) Changing FUs.



(c) Reordering instructions to resolve all Dependency hazards.



(d) Reordering Instructions to resolve dependency and Change FUs.

Figure 4.7. In-Order Completion. Sample 2, the number of Failed Tests per each instruction.

The scheme has the following disadvantages:

1. Fast instructions may sometimes get held up at the issue register even though they have no dependencies and would otherwise issue.
2. Long program execution time because of the held of instructions at the issue stage.

4.4 Summary

The architectural model, and the pipelined performance analysis were discussed in this chapter. Two interrupt schemes were studied which are: the interrupts prior to instruction issue, and the In Order Completion, which is implemented and analyzed. The first scheme acts upon exceptional conditions as soon as they are detected, thereby saving a few cycles per interrupt, but it must also handle situations where an older instruction causes an interrupt after a newer instruction causes its own. In this case, the pipeline would be in the process of handling the newer instruction's exception when the older instruction's exception is detected. The pipeline must abort the interrupt-handler-in-progress and redirect control to handle the exception that was detected next but should be handled first (in program order). Thus, the first scheme requires a bit more logic.

5. Implementation of Out of Order Completion

In this chapter, we introduce the Out- Of -Order schemes; the algorithms and the results are implemented and then fully analyzed.

5.1 Reorder Buffer (Basic Reorder Buffer)

5.1.1 Scheme Description

The reorder buffer is a mechanism suggested by Smith and Pleszkun. The structure of a system with a reorder buffer is shown in figure 5.1. The reorder buffer is a queue holding values returned from the functional units (David, 1997).

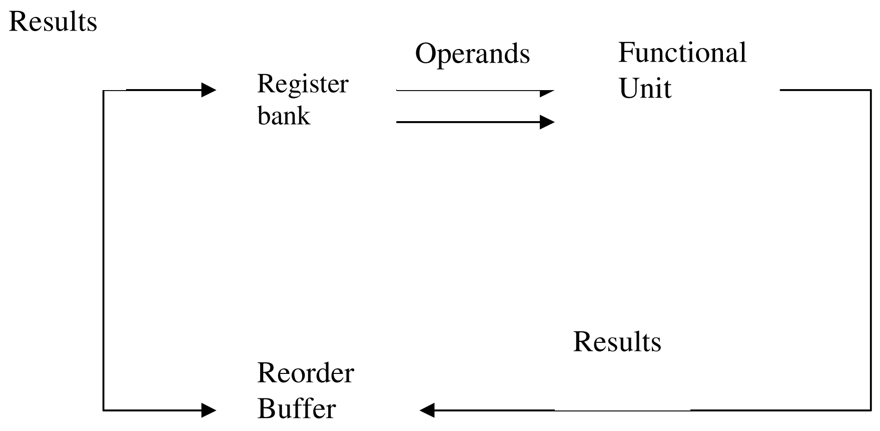


Figure 5.1. Processor organization with a reorder buffer.

The reorder buffer is implemented as a circular FIFO buffer. During instruction issue, a space is reserved in the reorder buffer into which the current *program counter* is written together with the *destination register* identifier. Results returning from the functional units write their results into the allocated spaces in the reorder buffer rather than writing the results directly into the register bank (David, 1997).

Furthermore, each reorder buffer entry has a *valid bit*. The bit indicates that the result of the instruction is in the reorder buffer entry. A reorder buffer entry with active valid bit is called valid reorder buffer entry.

The reorder buffer is accessed using two pointers, the head and tail pointers. We denote the value of the head pointer during cycle *T* by *ROBhead T,* and the value of the tail pointer by *ROBtail T.* Instructions are put in the ROB entry *ROBtail* points to, and removed from the entry *ROBhead* points to. After an instruction is put in the ROB, the *ROBtail* pointer is increased. After an instruction is removed from the ROB, the *ROBhead* pointer is increased. The pointers wrap-around if they reach the end of the ROB. This is illustrated in figure 5.2.

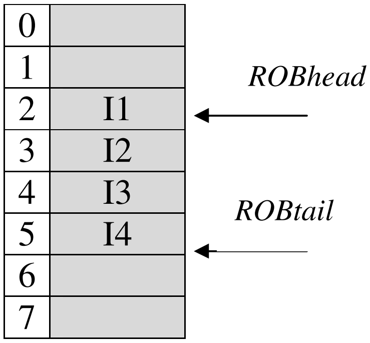


Figure 5.2. Illustration of the reorder buffer pointers

Table 5.1. RSR File used in ROB Scheme

(a) RSR File – sample 1

Stage	Functional Unit Source	Valid	TAG
1		0	
2	Integer ADD	1	5
3		0	
4		0	
5	FLPT ADD	1	4
.	.	.	.
N		0	

(b) RSR File– sample 2

Stage	**FU/OP**	**ROB TAG**	**Valid**	**Program Counter**
1				
2	LD (1)	2		A
3	LD (2)	3		B
4				
5	SUBD	5		D
6				
7				
8				
9	ADDD	7		F
10				
11				
12				
13	MULTD	4		C
14				

The Result Shift Register (RSR) is shown in Table 5.1. The Reorder Buffer (ROB) is shown in figure Table 5.2.

Table 5.2. ROB File used in ROB scheme.

(a) ROB, sample 1

Entry Number	**Des. Register**	**Result**	**Exceptions**	**Valid**	**PC**
3					
4	4			0	6
5	0			0	7
: : :	: : :	: : :	: : :	: : :	: : :

(b) ROB, sample 2

Entry Number	**Des. Register**	**Result**	**Exceptions**	**Valid**	**PC**
3					
4	F8		NO	√	A
5	F2		NO	√	B
6	F0				C
7	F8			√	D
8	F10				E
9	F6			√	F
10					

5.1.2 ROB Scheme Analysis

To Implement the ROB, follow these steps

- *Initialization:*

1. Initialize both ROB and RSR
2. Both tail and head reserved at the first entry of ROB.

- ***Instruction Issue***:

1. The next available ROB (Entry) is pointed by tail.
2. ROB Entry is reserved to the issuing instruction.
3. RSR (TAG) = ROB (Entry), is placed in RSR along with the control information of the instructions.
4. Tail pointer is incremented.

- ***Instruction Completion:***

4. Write value into ROB entry specified by the RSR
5. DO NOT write result into the register file
6. If instruction caused an exception, mark exception bit in the ROB for offending instruction.
7. Check instruction at the head of the ROB

- If the associated instruction is not completed, the slot remains there until it has completed.
- Instructions can continue to be decoded until the reorder buffer is full.
- If completed, check exception bit
- If there is no fault:

a. Commit state (the value is written to the register file).

b. The entry removed from the reorder buffer.

c. Advance head pointer.

8. If there is a fault associated with the value:

1. Issue is stopped in preparation for the interrupt.
2. Squash subsequent instructions in ROB.
3. All further writes into the register file are inhibited.
4. Back-up the tail pointer to the head pointer.
5. The in-order state of the register file is restored.

9. If the instruction causes an exception and software support is needed, the hardware handles the interrupt in the following way:

1. The ROB is flushed; the exceptional PC is saved; the PC is redirected to the appropriate handler.
2. Handler code is executed, typically with privileges enabled.
3. Once a return from interrupt instruction is executed:

a. The exceptional PC is restored

b. The program resumes execution

In this model, there are two primary sources of performance loss: While the exception is being handled, there is no user code in the pipe, and thus no user code executes—the application stalls for the duration of the handler: After the handler returns control to the application, all of the flushed instructions are re-fetched and re-executed, duplicating work that has already been done (Amer *et al*., 2000).

Instructions are en-queued in program order and only when their operands are available. While they are in the reorder buffer program order does not state the sequencing of any particular events. In particular, the instructions might finish executing out of program order.

When an instruction is successfully de-queued from the reorder buffer, its results are *committed* to the machine state. At this point, the instruction is said to be retired. While an instruction is in the process of execution, i.e. before retiring, it may cause an exceptional condition. To ensure that such exceptions are not handled speculatively or out-of-order, exception handling does not occur until instruction commits time. This ensures that no previous instructions have caused as-yet un-handled exceptions. Therefore, all exceptions are handled in program order, and no exception is handled for an instruction that ends up and being discarded (UMD, 2000).

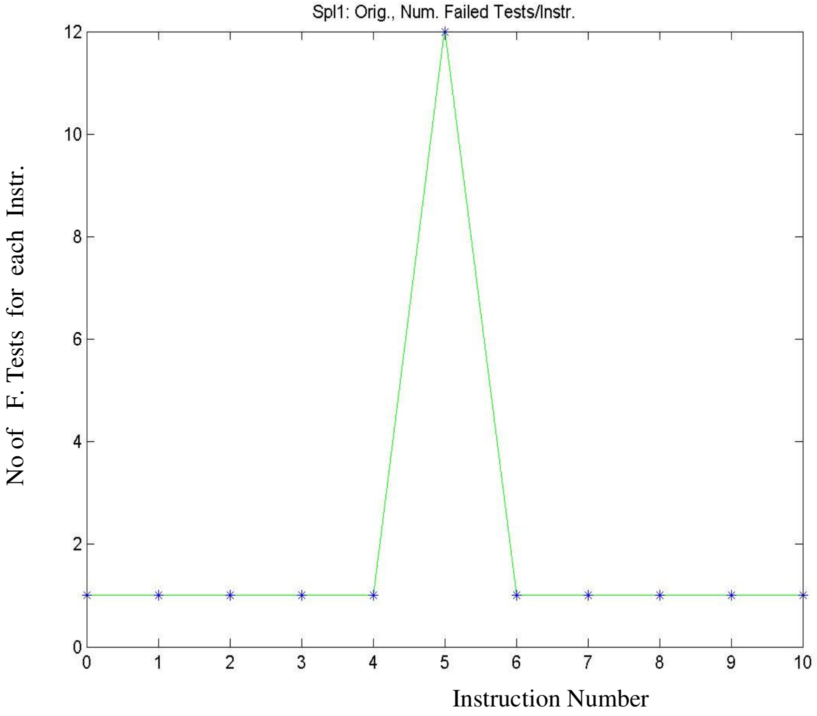
Experiments and Results:

Many experiments were done to test the algorithm. Two samples of code were tested. The instructions in the two samples were repeated several times. Random instructions with different execution times and with variable number of functional units were tested. The same variability of FUs used in the previous chapter (In-Order Completion), were used here. The dependency hazards were resolved fully and partially, to test the results for each variation. Here are the results of the experiments done:

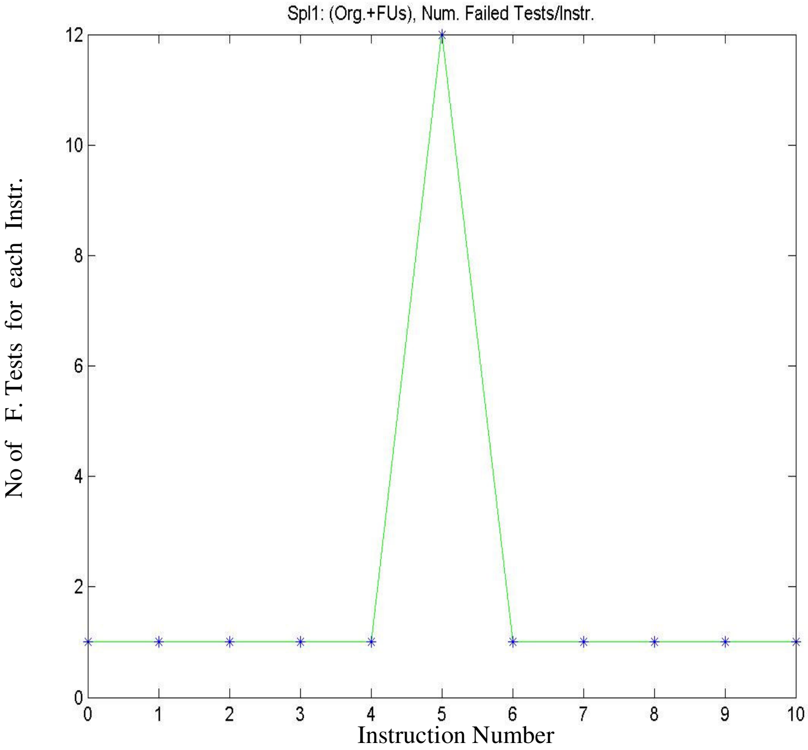
1. The first and the second sample of code, and all their variations were applied to the algorithm. The results were as follows:

- The instructions that have hazard dependencies have sharp peaks in the diagrams graphed for the number of failed tests for each instruction as shown in Fig. 5.3 and 5.4. Other instructions have a constant number for the failed tests.
- The total number of the failed tests and succeeded tests are nearly the same in all experiments. The percentage in all cases is shown in figure Fig.5.5, Fig. 5.6.

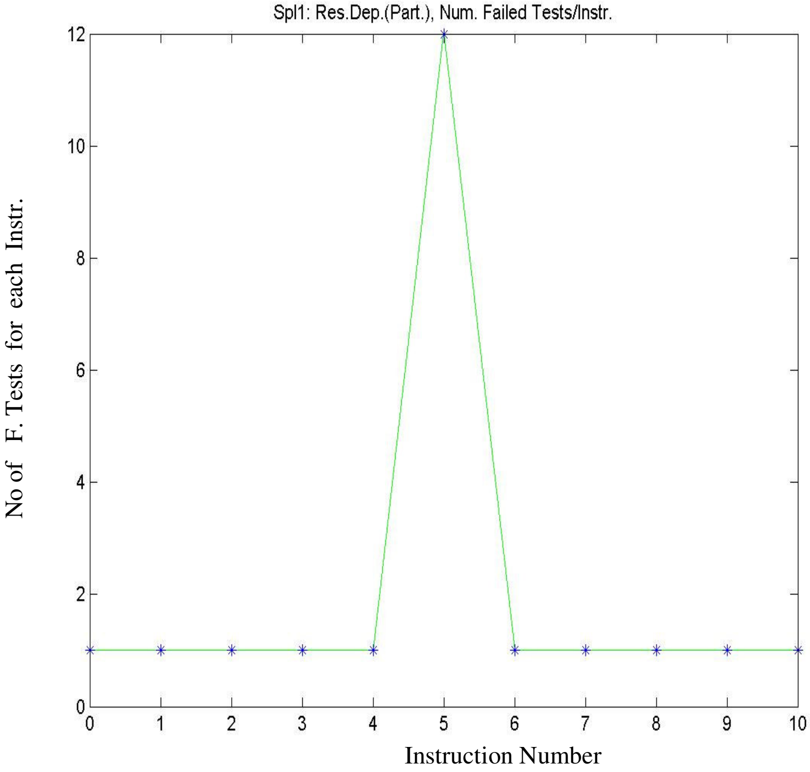
2. The RAW dependencies are solved and this increases the number of shifts necessary for each instruction to be shifted in order to be completely executed, this is shown in Figure 5.5, 5.6 as a peak for resolved instructions.



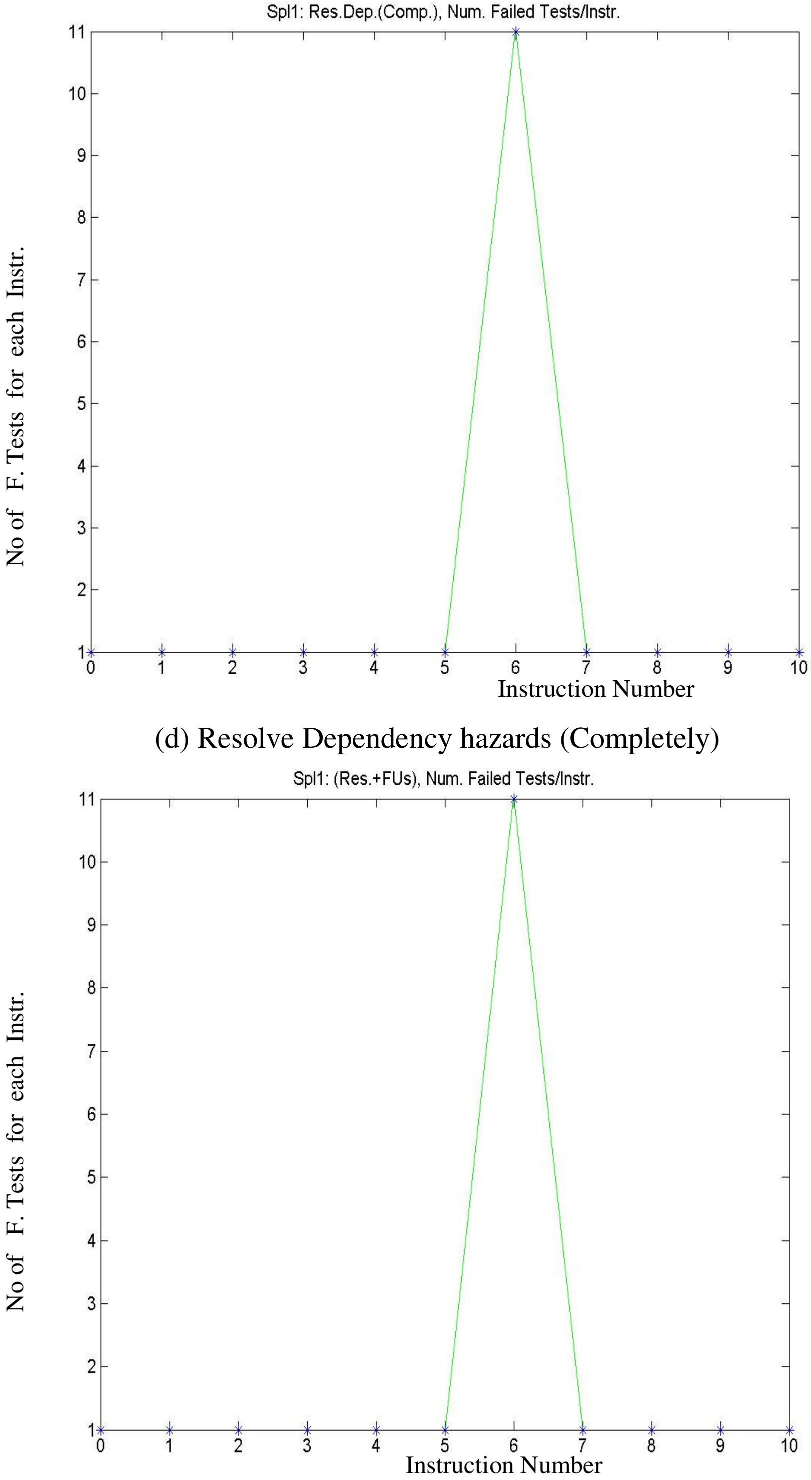
(a) Original Code.



(b) Changing FUs.



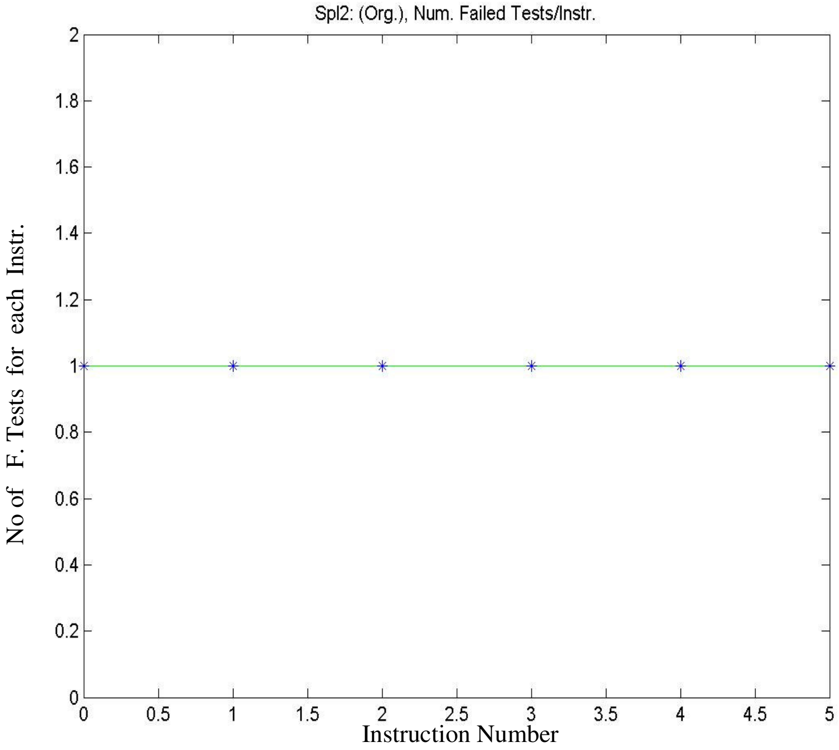
(c) Resolve Dependency hazards (partially)



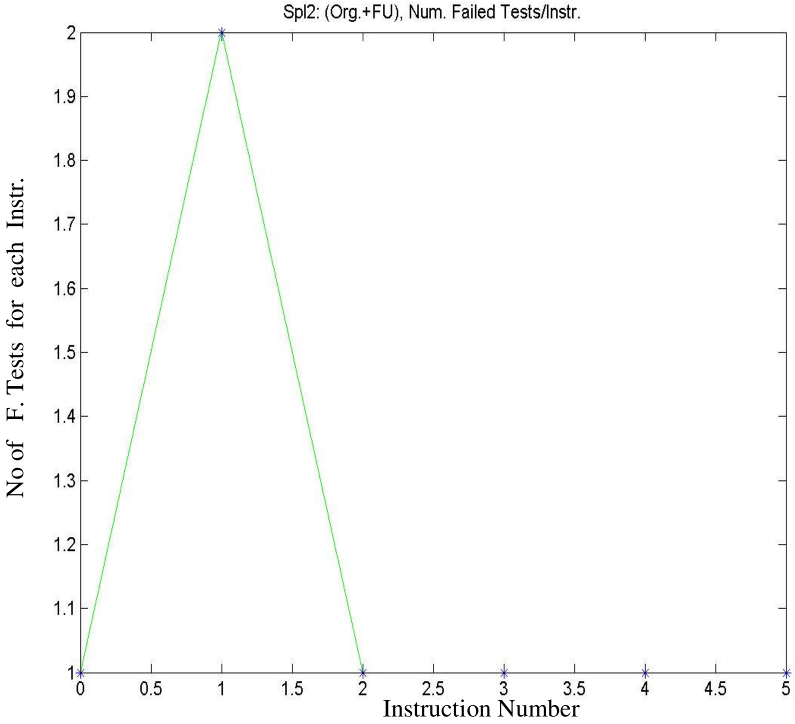
(d) Resolve Dependency hazards (Completely)

(e) Resolving Dependencies & FUs.

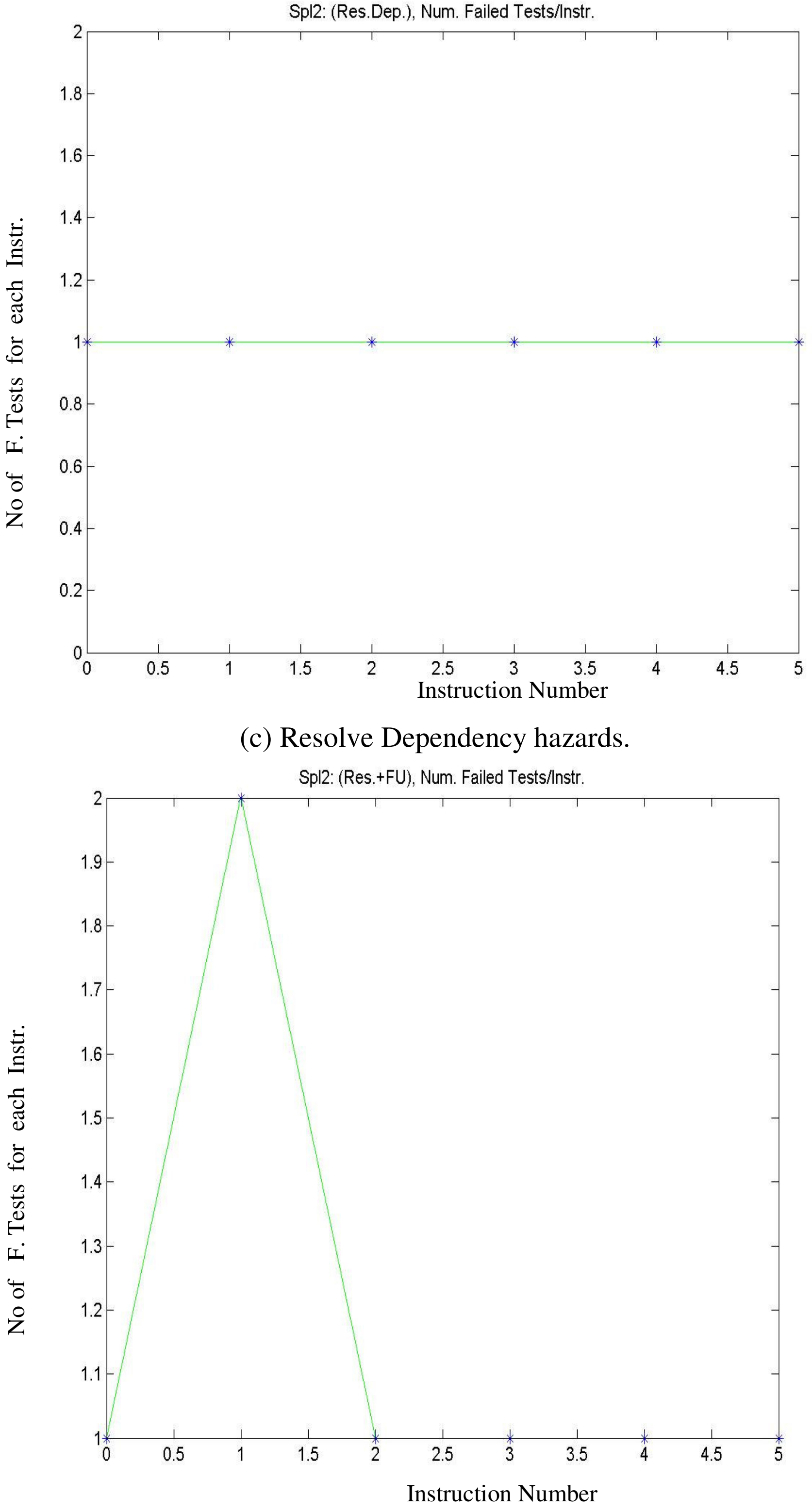
Figure 5.3. ROB. Sample 1: the number of Failed Tests per each instruction.



(a) Original Code.



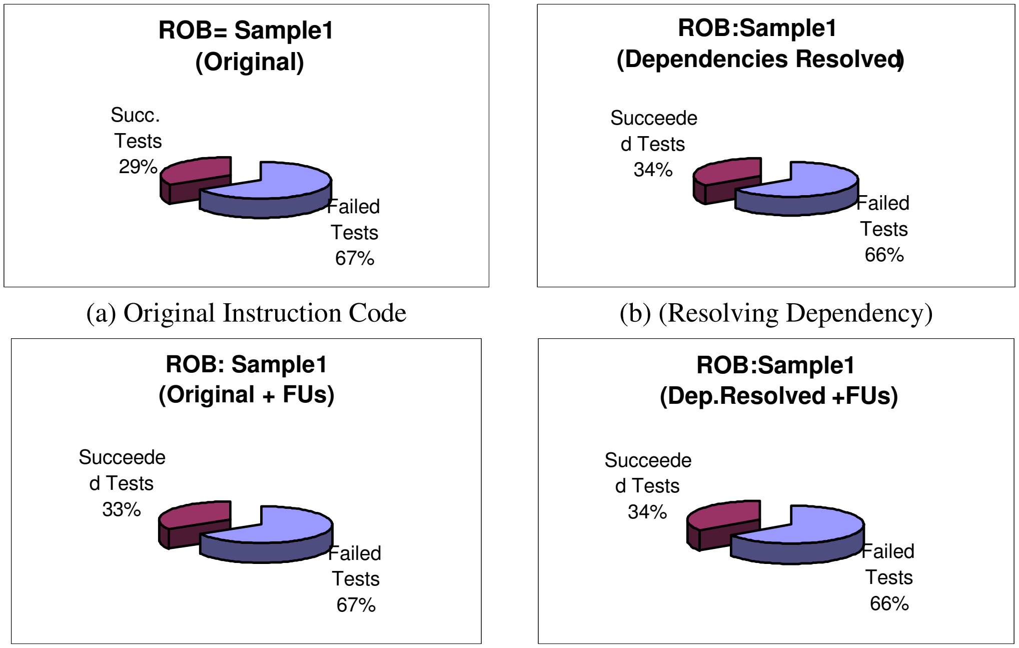
(b) Changing FUs.



(c) Resolve Dependency hazards.

(d) Resolving Dependencies and FUs.

Figure 5.4. ROB. Sample 2: the number of Failed Tests per each instruction.



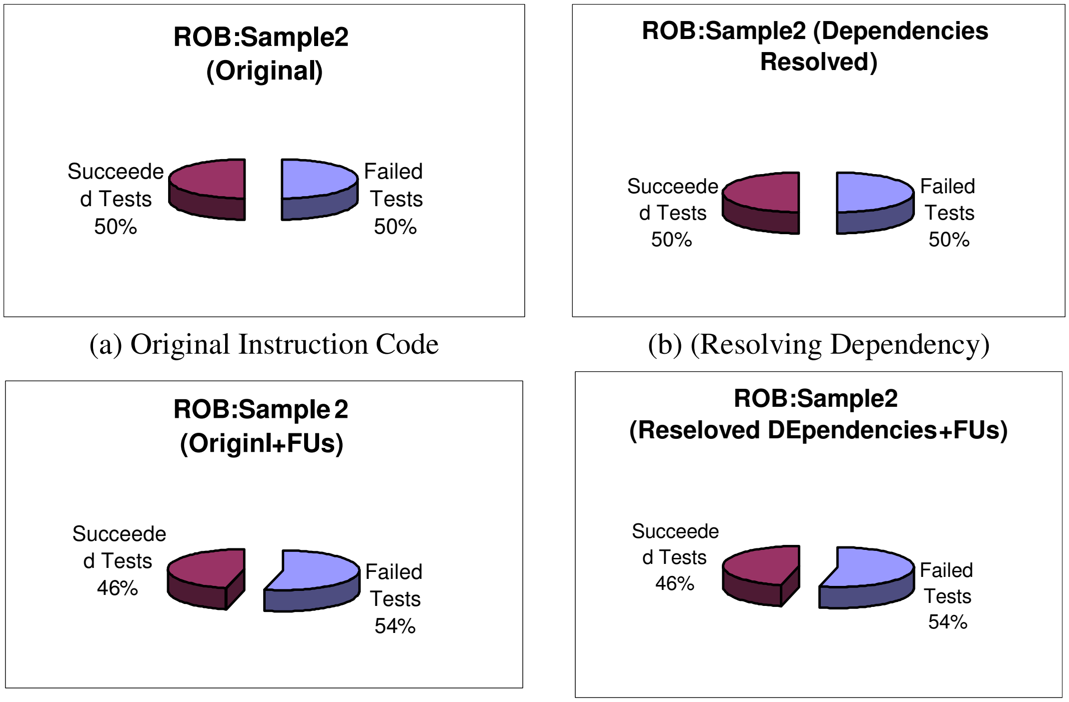
(a) Original Instruction Code

(b) (Resolving Dependency)

(c) Original Instructions (changing FUs.)

(d) Changing FUs and Resolved Dep.

Figure 5.5. ROB. Sample 1. The number of failed tests and Succeed tests to the Total number of tests.



(a) Original Instruction Code

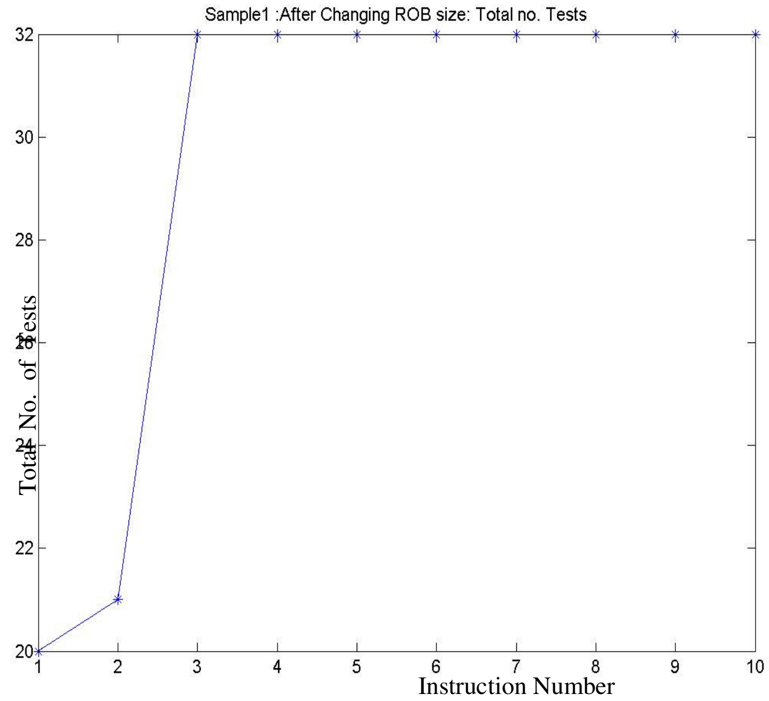
(b) (Resolving Dependency)

(c) Original Instructions (changing FUs.)

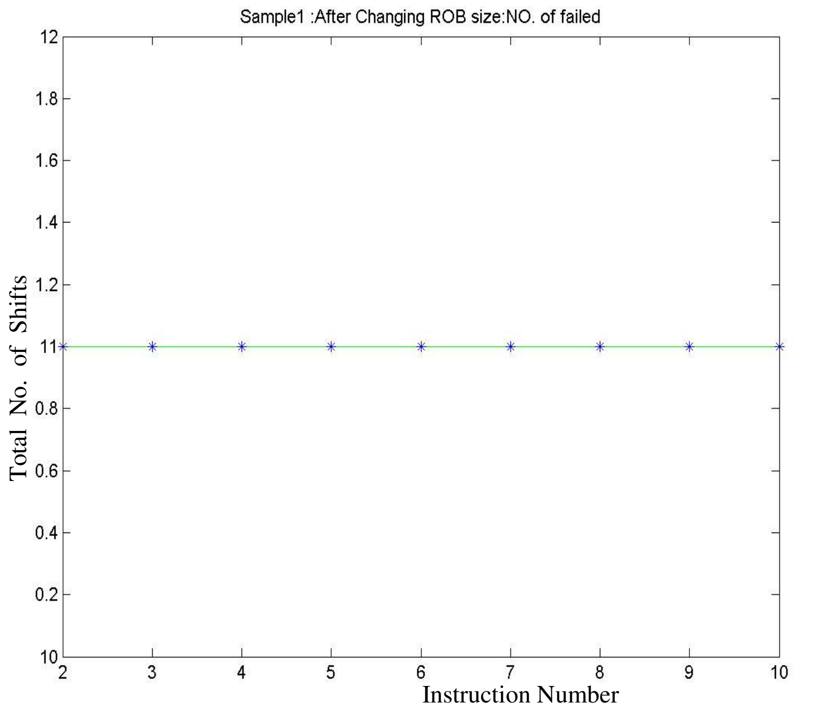
(d) Changing FUs & Resolving Dependencies

Figure 5.6. ROB. Sample 2. The number of failed tests and Succeed tests to the Total number of tests.

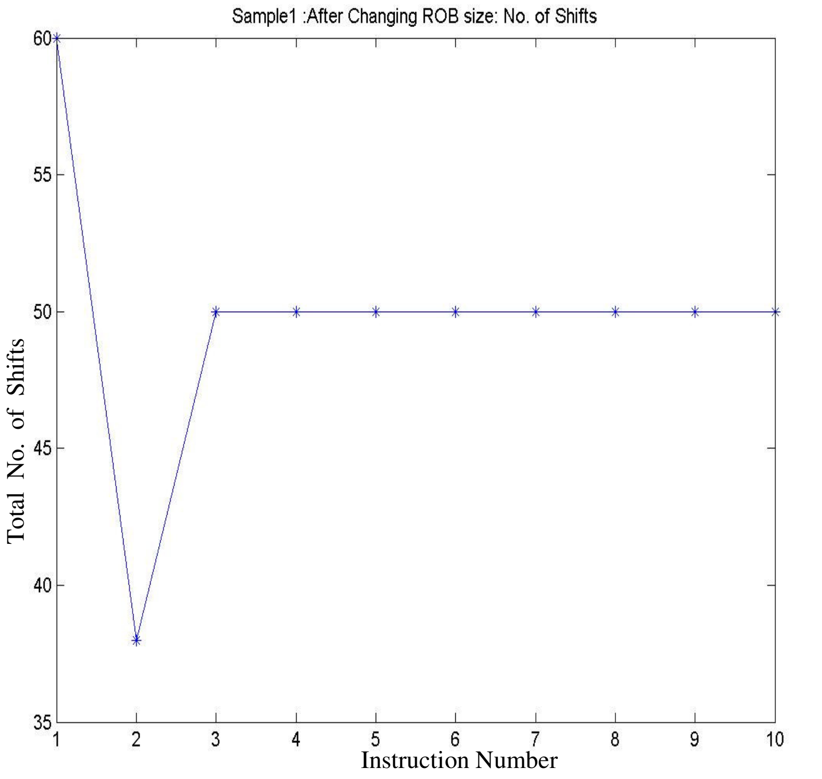
3. As the size of the ROB is changed, different results can be achieved. The total number of tests, the number of failed tests, and the number of shifts at each instruction all increased as the size of the ROB is decreased until we reach to a stable and uniform results regardless of any further expansion. We started with the size of 1 to the size, which is of the RSR. The results can be shown in fig. 5.7. We know the size of both RSR and ROB depends on many factors regarding to the complexity, logic control circuits, and the hardware components in the processor. If the reorder buffer is too small the issue stage will stall waiting for a space, leading to performance loss. But, if we increase it supposing better performance by not having instructions stopped from issuing because of not having empty entries in the ROB, this seems not to have better performance.



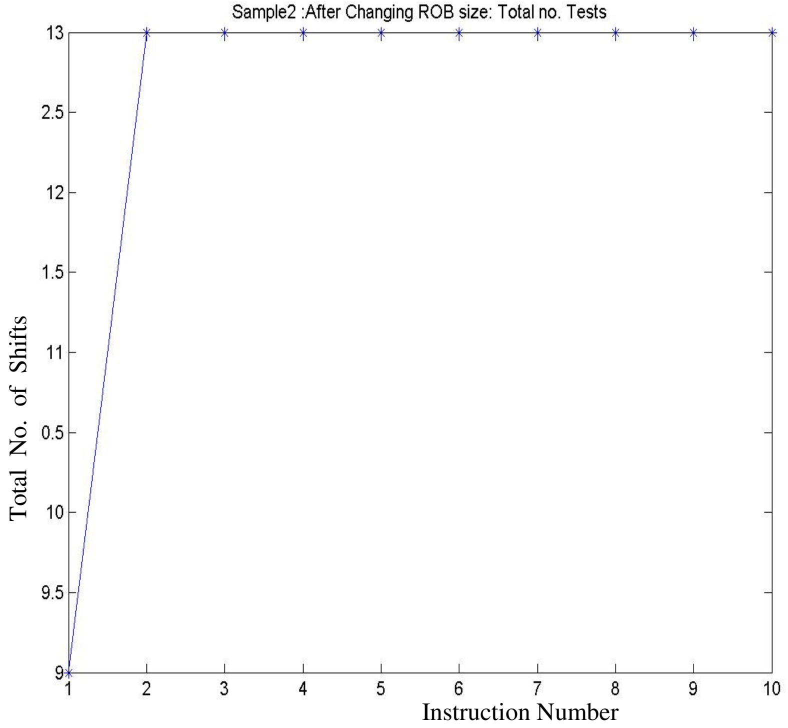
(a) The Total number of tests as the size of ROB changes: Sample 1



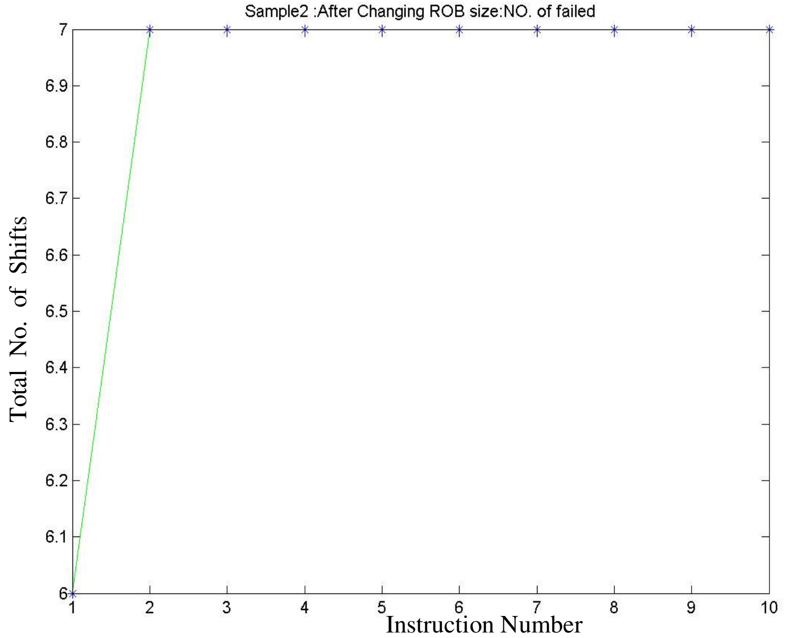
(b) The number of failed tests as the size of ROB changes: Sample 1



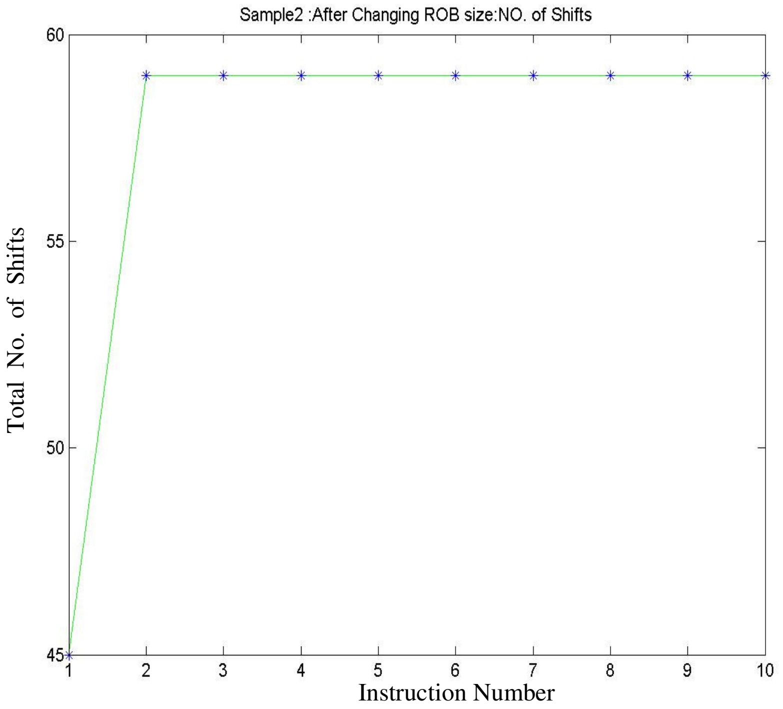
(c) The number of shifts as the size of ROB changes: Sample 1



(d) The Total number of tests as the size of ROB changes: Sample 2



(e) The number of failed tests as the size of ROB changes: Sample 2



(f) The number of shifts as the size of ROB changes: Sample 1

Figure 5.7. ROB. After increasing the size of the ROB.

4. As the time to detect an exception increases, so does the number of instructions that will be re-fetched and re-executed. Clearly, the overhead of taking an interrupt in a modern processor core scales with the following, where each of these is on a growing trend:

- The size of the reorder buffer.
- Pipeline depth.
- Issue-width (Amer *et al*., 2000).

5. As operands are read from the register file the performance degradation due to RAW dependencies is increased since it is necessary to wait for the in-order state to be resolved as results drain into the register bank (David, 1997).
6. We can decrease the time that instructions waited for committing the results by taking the most recent entry for a register from either the register file or the

reorder buffer. So, when using the Reorder Buffer, we may have two variations: either to have:

- Operands must be obtained though the register file. Or
- Operands can be read directly out of the reorder buffer itself if the latest copy of a register value is present in the *result* portion of a reorder buffer entry holding an instruction that has finished executing but has not yet written its result to the register file.

Now we can conclude the following points to be the Reorder Buffer main disadvantages:

A. It suffers a performance penalty.

B. The computed result that is generated out of order is held in the reorder buffer until previous instructions, finishing later, have updated the register file.

C. An instruction dependent on a result being held in the reorder buffer cannot be issued until the result has been written into the register file.

D. The effect on RAW dependencies.

So, the key features of the reorder buffer can be summarized as follows:

- Instructions complete out of order (overlap execution of instructions in buffer).
- Commit in order.
- Consider interrupts at any instruction at commit point, if committing instruction interrupts, squash all later instructions.

Finally, as the electronic fabrication of the components affect the speed and the bandwidth of the processing; at UC Irvine, researchers have been designing a superscalar architecture called SDSP (Superscalar Digital Signal Processors). One of the main components of the design is the scheduling unit, which consists of an instruction window, a reorder buffer, and a register file. It outperforms the previously published reorder buffer in several features. First, the new design decodes four instructions instead of eight. Secondly, the current design runs at 100 MHz as opposed to 20 MHz, using a smaller, three metal technologies. Thirdly, instead of a current bit cell, a look up array is used to dynamically determine the most current entry. This has performance benefits in handling mispredicted branches, since selected entries may be invalidated. Finally, the size of all cells was reduced dramatically by reducing the number of transistors per cell and by combining the shift and storage portions of each cell. But, we should note that the associative lookup could be expensive, especially when we must access the most recent entry for a particular register.

5.2 Reorder Buffer with Bypassing

5.2.1 Scheme Description

One of the primary disadvantages of the reorder buffer described above is its effect on RAW dependencies. This effect can be reduced (or removed) by adding forwarding paths from the reorder buffer around the register bank as shown in figure 5.8 (David, 1997).

When an instruction is being issued, the reorder buffer is searched for entries whose destination register field corresponds to a source operand that the instruction needs. If no entry in the reorder buffer matches the register number then the result has already reached the register file and the operand is read from there.

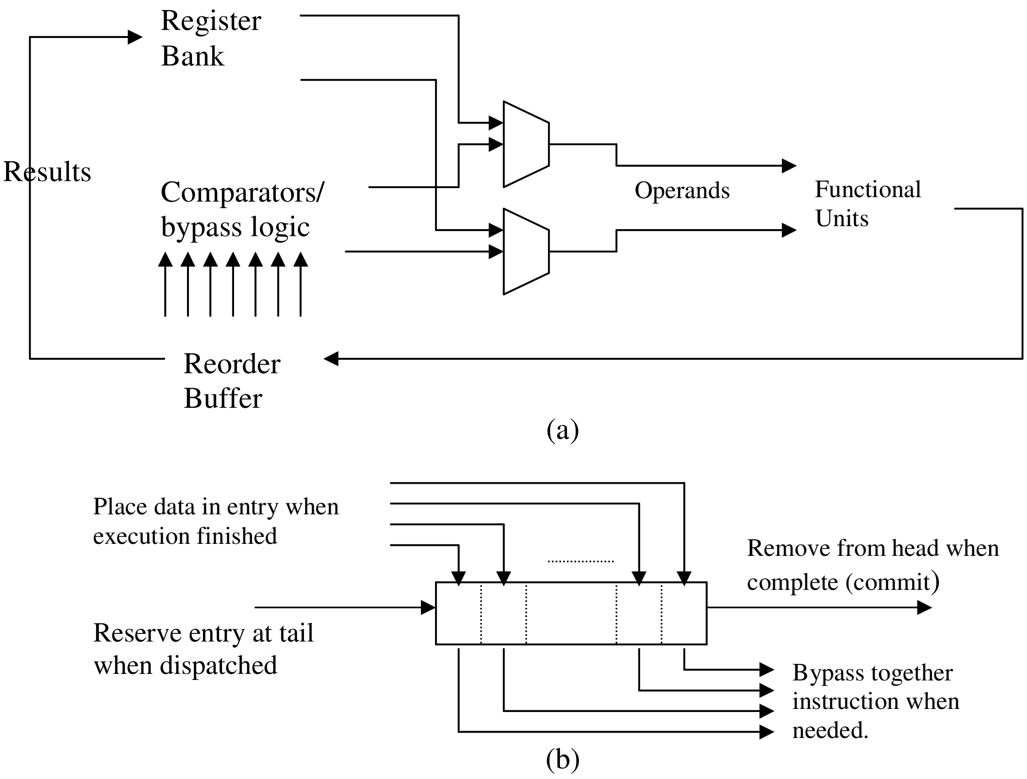


Figure 5.8. Processor organization with a reorder buffer with forwarding

In order for results to be used early, bypass paths may be provided from the entries in the reorder buffer to the register file output latches as shown in figure 5.9. These paths allow data being held in the reorder buffer to be used in place of register data. If one match is found the instruction issue may be stalled until the reorder buffer entry contains a valid result, whereupon it is used as the operand for the instruction being issued (David, 1997).

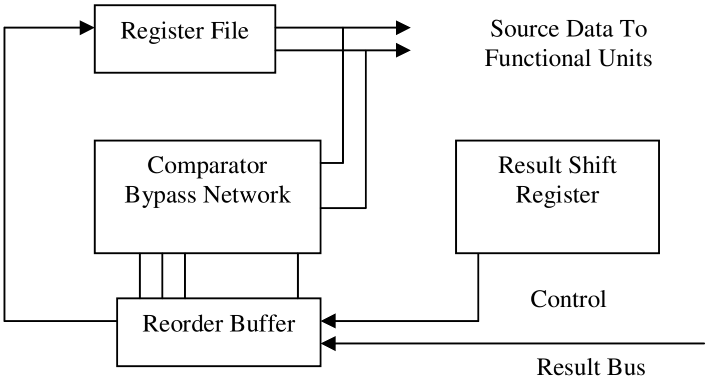


Figure 5.9. Reorder Buffer method with Bypasses.

The implementation of this method requires comparator for each reorder stage and operand designator. If an operand register designator of an instruction being checked for issue matches a register designator in the reorder buffer, then a multiplexer is set to gate the data from the reorder buffer to the register output latch.

5.2.2 Scheme Analysis

The same ROB algorithm is applied with some modification, which is added to improve the pit faults of the basic method.

To Implement the ROB with Forwarding, we can follow these steps

- If instruction is completed, then the value is in ROB
- If instruction is not retired, then the value *not* in register file
- Dependent instruction is dispatched into window, in which the value is either:

a. Not read from register file.
b. Can’t be grabbed from result bus, i.e., Instruction is already completed!
c. Only is placed is in ROB,

- Need to read value from ROB (ROB bypass) (David, 1997).

When bypass paths are added, preciseness with respect to the memory and the program counter is not changed from the basic method. So, the greatest disadvantages with this method are:

- There are number of bypass comparators needed in this scheme.
- The amount of circuitry required for the multiple bypass check, while this circuitry is conceptually simple, there is a great deal of it (James *et al*., 1988).

As we stated above, the primary disadvantage of this mechanism is the complexity of the logic required to test for the presence of registers in the reorder buffer. This consists of a Content Addressable Memory (CAM) whose size increases with the number of entries in the buffer and the number of operands that are forwarded. The comparison is complicated by the need to select the latest version of a register if multiple matches are found.

The main advantage of the reorder buffer with forwarding is that in addition to providing a mechanism for exception handling it also resolves RAW and WAW dependencies (David, 1997). When using ROB with bypassing, if there is a large set of physical registers, ROB allows detection of when to free a physical register. Its handling of WAW dependencies can be seen to be a form of register renaming. Effectively each entry in the buffer is another register, and multiple versions of each register may be presented in the buffer at any time.

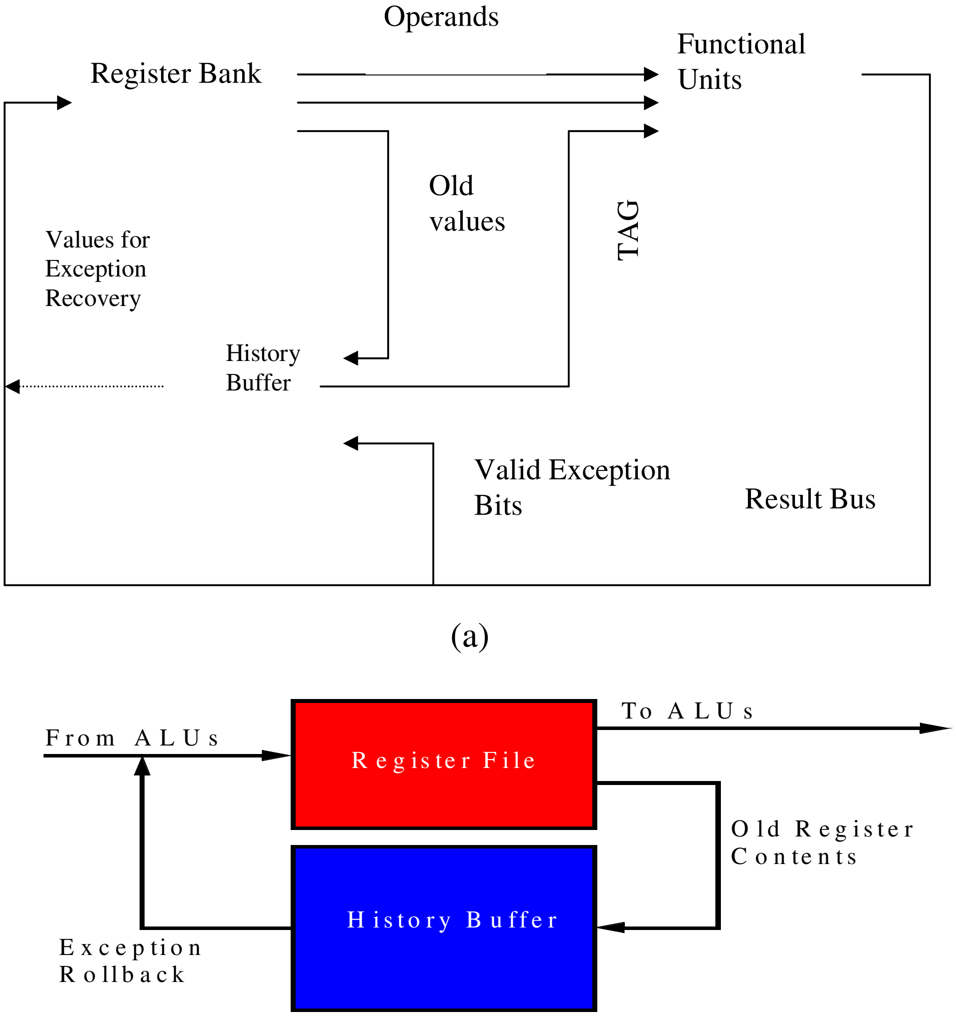
The buffer reorders values so that where there are WAW dependencies the values are written back to the register bank in the correct order, and the search mechanism ensures that instructions which are issued read the most recently allocated version of registers rather than the most recently completed version.

5.3 The History Buffer (HB)

5.3.1 Scheme Description:

The history buffer maintains some state to be restored when an exception is encountered, only the part of the state, which has changed recently, is stored.

A history buffer is a FIFO, to which every instruction is added when it is fetched. When the instruction writes to the register file, the value it overwrites is saved in with it. Instructions are removed from the top of the history buffer when they are completed. Any exception caused by an instruction is reported only when it comes to the top of the stack. At that point, the register file values are restored by "'roll back" basically, starting from the bottom of the buffer, all the saved register values are written back to the register file. The destination register number and program counter are stored in the slot and the exception and valid flags in the slot are cleared (David, 1997).



(a)

(b)

Figure 5.10. The basic structure of a system with a history buffer

Figure 5.10 shows the basic structure of a system with a history buffer while figure 5.11 shows entries to keep original values that can be restored after an exception (David, 1997).

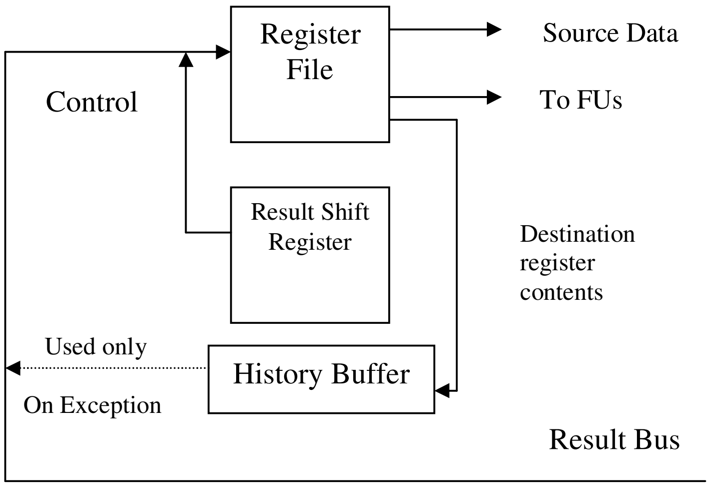


Figure 5.11. Entries to keep original values after an exception

The straightforward adaptation of a history buffer is that it will save with each instruction the *old* physical register mapped by its output register. On an interrupt, both the map and the register file are restored to their precise state by rolling back the history buffer. Results go into register file out-of-order, but keep old state in a history buffer until all previous operations are done. So, it requires hardware to "roll back" history buffer on exception.

Table 5.3. RSR, HB used in the History Buffer Scheme.

(a) The RSR File

Stage	**Functional Unit Source**	**Destination register**	**Valid**	**TAG**
1			0	
2	Integer ADD	0	1	5
3			0	
4			0	
5	FLPT ADD	4	1	4
. .	. .		. .	. .
N			0	

(b) HB File

	Entry Number	Des. Register	Old Value	Exceptions	Valid	PC
	3					
Head →	4	4	40800000	0	0	6
Tail →	5	0	42	0	0	7
	: : :	: : :	: : :	: : :	: : :	: : :

There is an in-order map, which satisfies the invariant that it is the same as the current map used by the instruction at the top of the history buffer. The RSR and HB are shown in Table 5.3.

So, the history buffer has an entry for every instruction. The information contained in this entry includes the instruction, the physical register displaced from the current map by the instruction when it was issued, and a bit indicating whether the instruction has been completed or not.

5.3.2 Scheme Analysis

To Implement the HB scheme, we can follow these steps

- ***On issue***

1. The instruction is added to bottom of the history buffer.
2. The physical register to which the instruction's output register is remapped is also added (assuming, of course, that the instruction has an output register).
3. Copy current value of destination register to HB.
4. Increment head.

- ***On Completion:***

1. Mark HB entry valid.
2. Put old value into history buffer
3. Mark exception bits.
4. When head to HB contains an entry marked valid, the entry checked for exception:

▪ When an entry has exception:

1. Stall issue, waits for pipeline to empty (Flush pipeline).
2. Reload register file from the values in the history buffer (from tail to head).
3. PC at head is precise PC.

▪ If head of ROB has no exception, remove it from buffer.

Interrupt recovery can be facilitated by maintaining an *in-order map,* i.e., the map used by the instruction at the top of the history. This can be implemented by saving, for each instruction in the history buffer, the physical register, for which the instruction's output register was mapped to before being remapped by the instruction. When an instruction is retired from the top of the history buffer, the in-order map is updated appropriately. If exceptions are reported only when the excepting instruction reaches the top of the history buffer, the map can be restored in a single cycle by copying the in-order map instead of rolling back the history buffer.

Experiments:

The same experiments were done as in the previous experiments to test the HB algorithm. The same two samples of code were tested with all their variations. The results were as follows:

1. Both samples are applied to the HB algorithm. The same variations as the previous schemes were used.

- Sharp peaks at the instructions, which have hazard dependencies, obtained as shown clearly in Fig. 5.12 and 5.13. Other instructions have a constant number for the failed tests for the two samples.
- Better results are achieved for the number of failed tests to the succeeded tests as we change the FUs and having no dependency. This is shown in Fig. 5.14, Fig. 5.15.

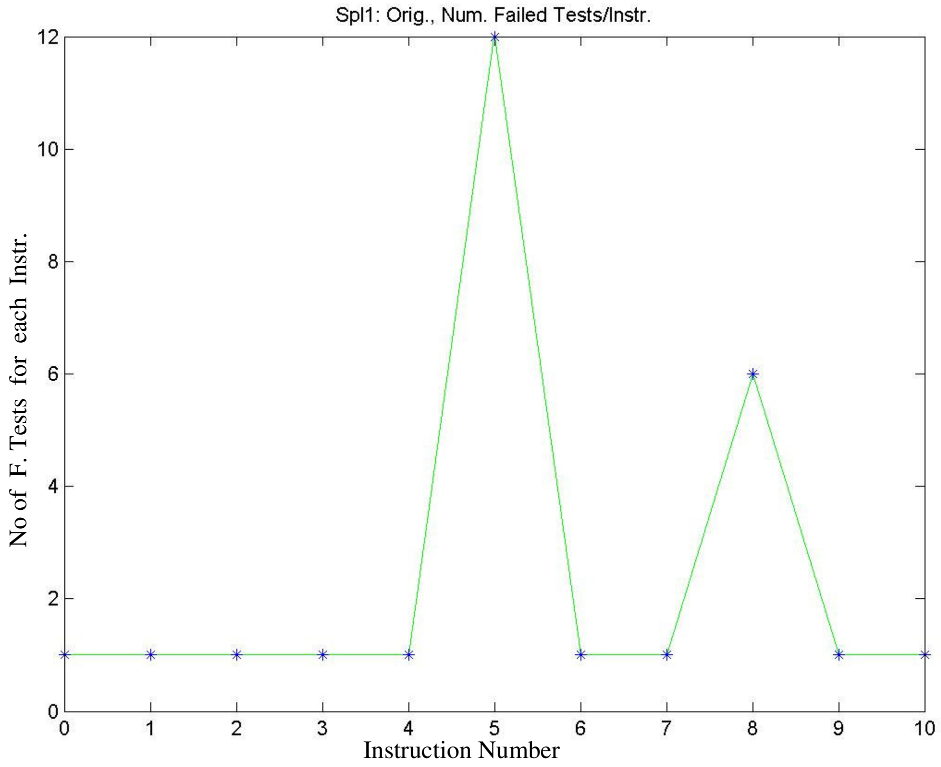
2. The RAW dependencies are completely solved using both the old values of the destination registers and the updated values. This clearly increases the number of shifts necessary for each instruction in order to complete execution. The same considerations mentioned in the previous sections, and in section 4.6 are used here. The history buffer does not help in the resolving of dependencies and it imposes a dependency itself. It may be necessary to wait for the old value of the destination register at issue before it is written into the history buffer. Thus an additional dependency resolving mechanism is needed (David, 1997).
3. We do some experiments for increasing/decreasing the size of the HB, Figure 5.16 shows the results obtained. Increasing the size of HB makes the total number of tests and shifts to be uniform at a certain point. When the size of the HB is the smallest, it gives the less performance. This is because if the history buffer does not contain enough entries, the decode and issue stages will be stalled waiting for a free space in the buffer

before issuing the instruction. But increasing it seems not to have better performance. So, the history buffer can cause a performance degradation beyond that caused by the added complexity of the control logic.

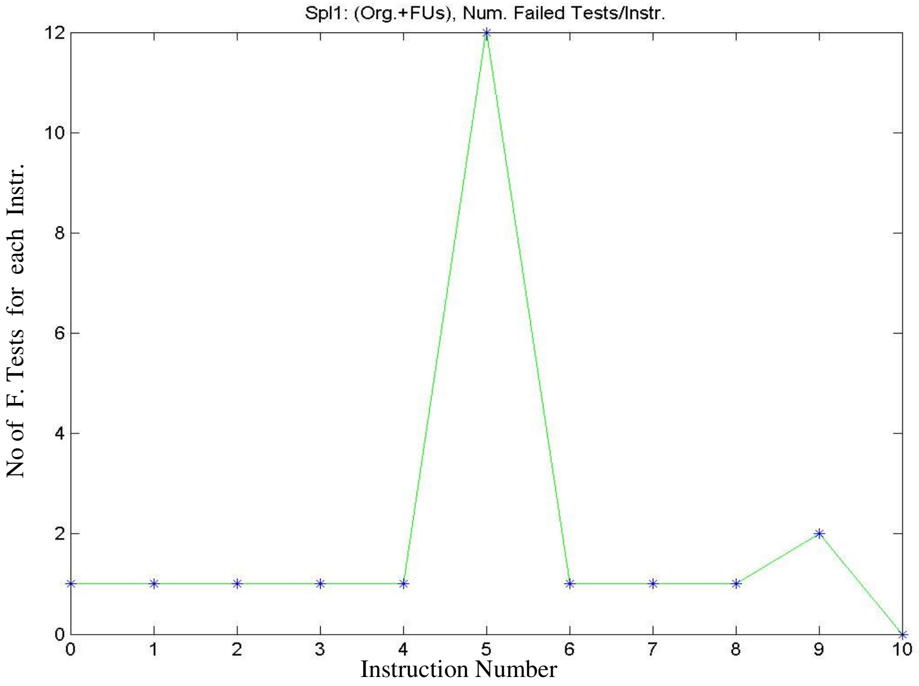
4. In a system using a history buffer the register bank holds the architectural state, as can be seen by the fact that the functional units access values directly from the register bank. The history buffer is used to restore the in-order state when an exception occurs.

So, the history buffer has the following advantages:

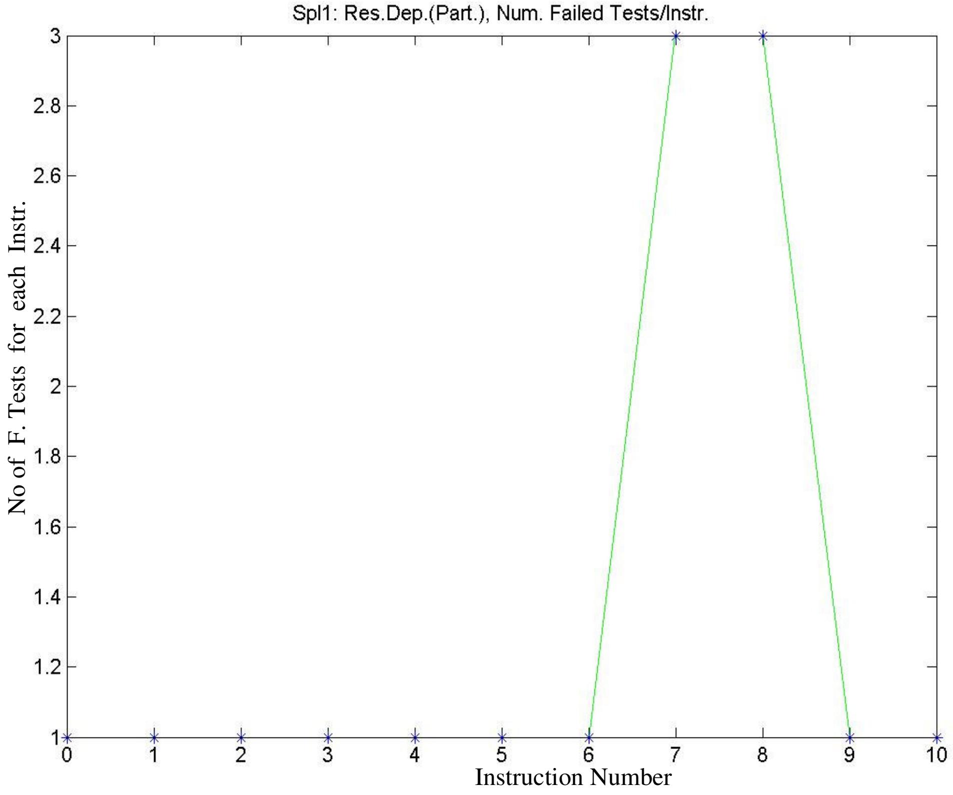
- Only a small part of the state needs to be restored, reducing the amount of storage needed.
- The cost of periodically copying the whole state is removed (David, 1997).



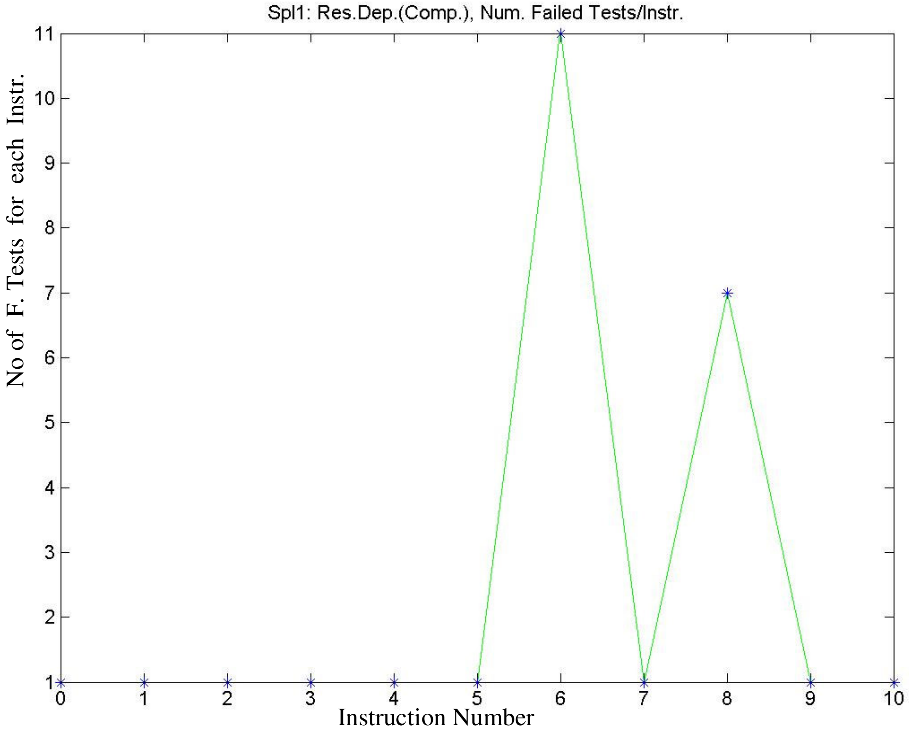
(a) Original Code.



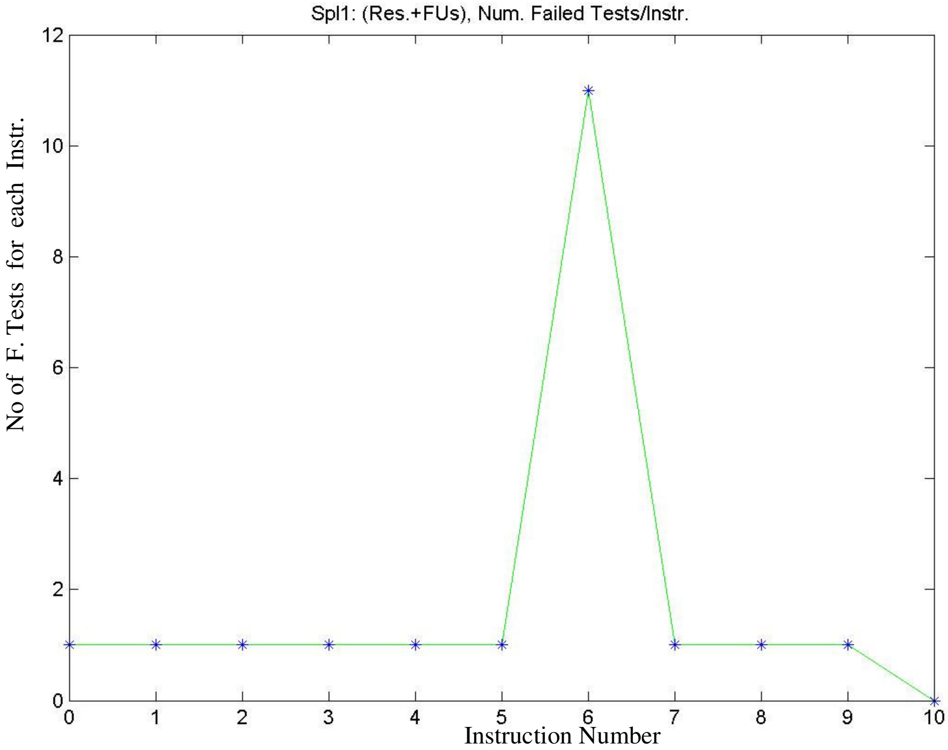
(b) Changing FUs.



(c) Resolve Dependency hazards (partially)

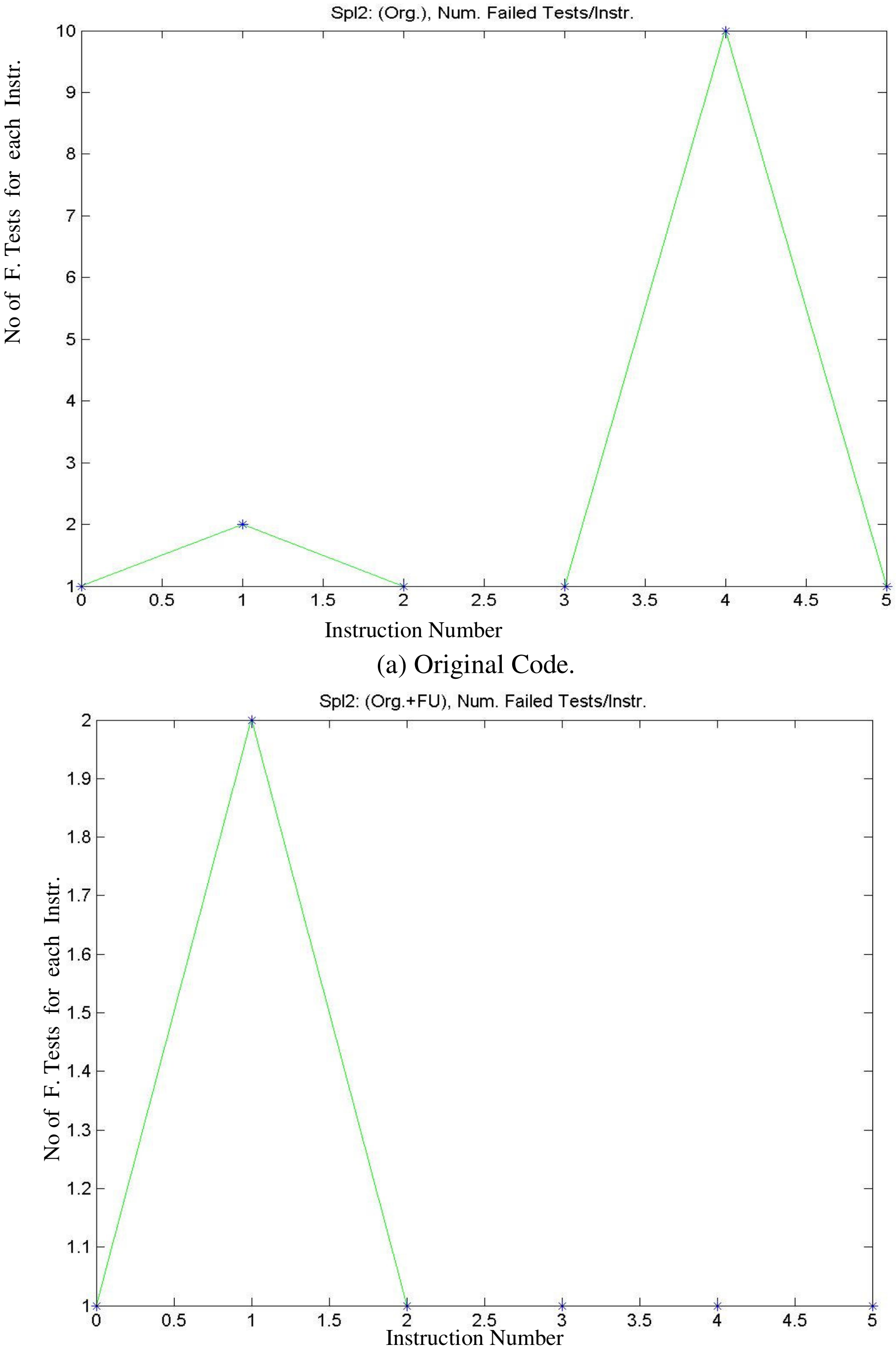


(d) Resolve Dependency hazards (Completely)



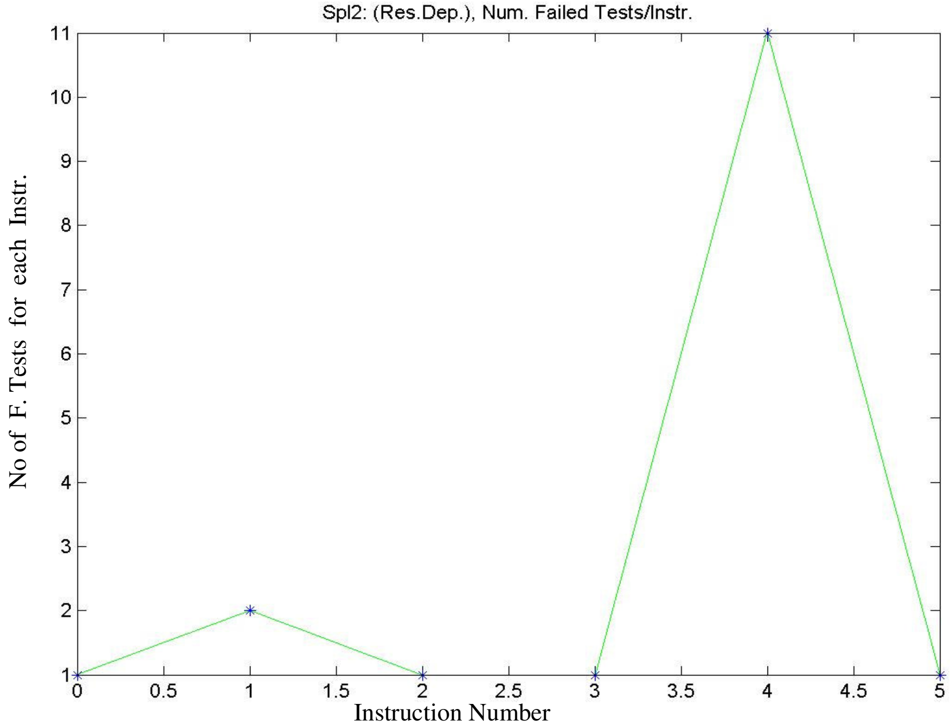
(e) Resolving Dependencies and FUs.

Figure 5.12. HB. Sample 1: the number of Failed Tests per each instruction.

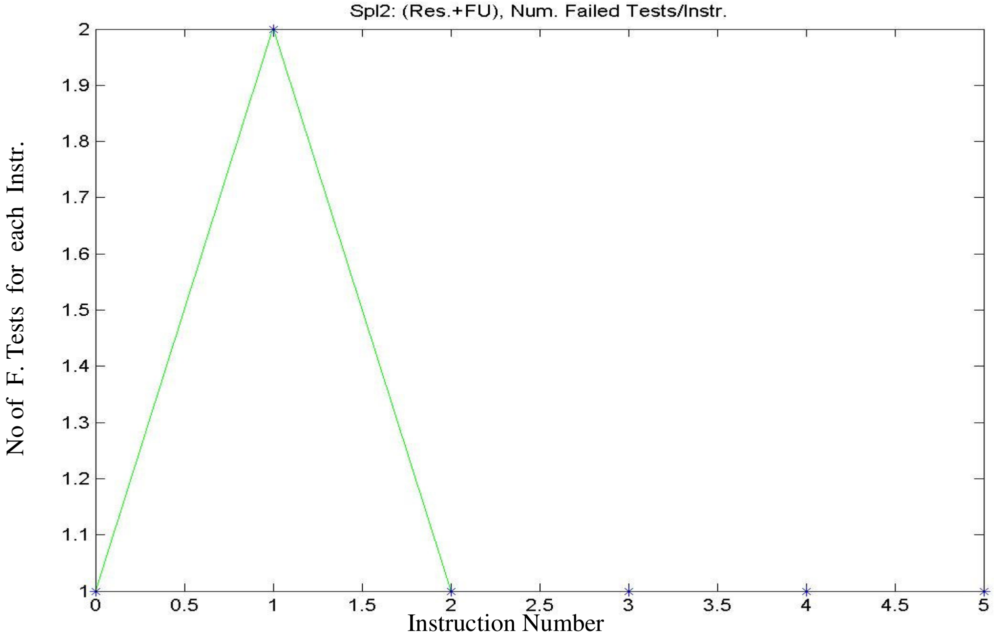


(a) Original Code.

(b) Changing FUs.

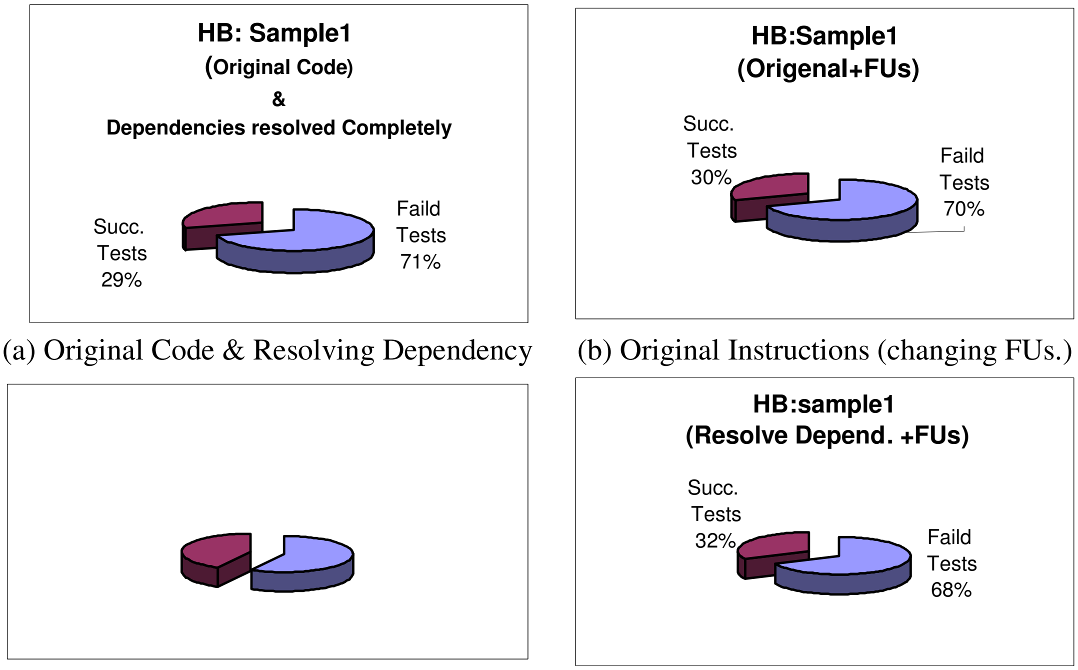


(c) Resolve Dependency hazards.



(d) Resolving Dependencies and FUs.

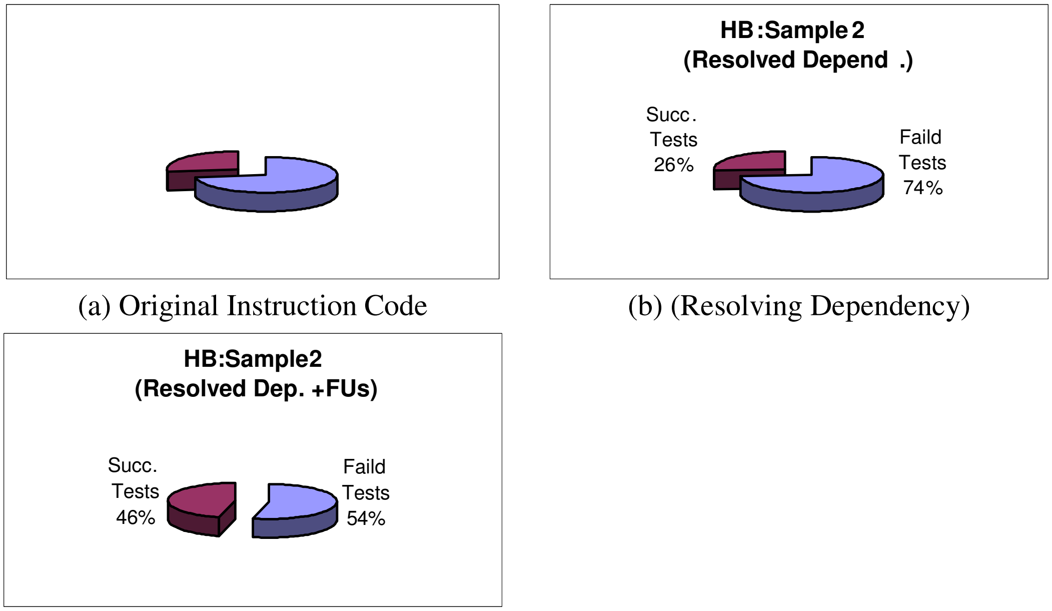
Figure 5.13. HB. Sample 2: the number of Failed Tests per each instruction.



(a) Original Code & Resolving Dependency (b) Original Instructions (changing FUs.)

(c) Resolved dependencies (partially) (d) Changing FUs & Resolving Dependencies

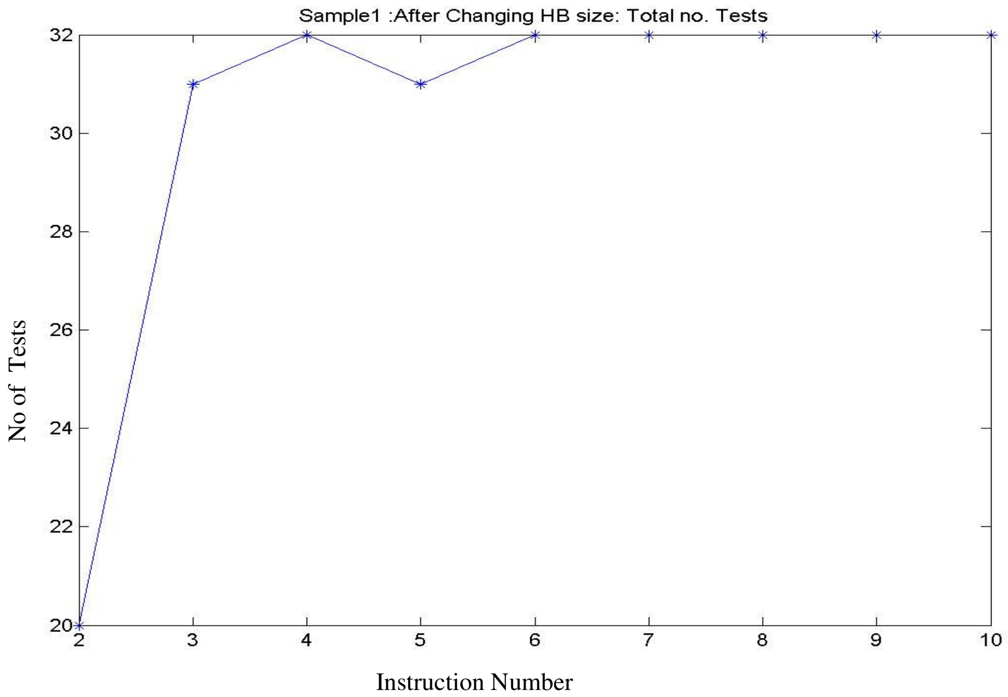
Figure 5.14. HB. Sample 1. The number of failed tests and Succeed tests to the Total number of tests.



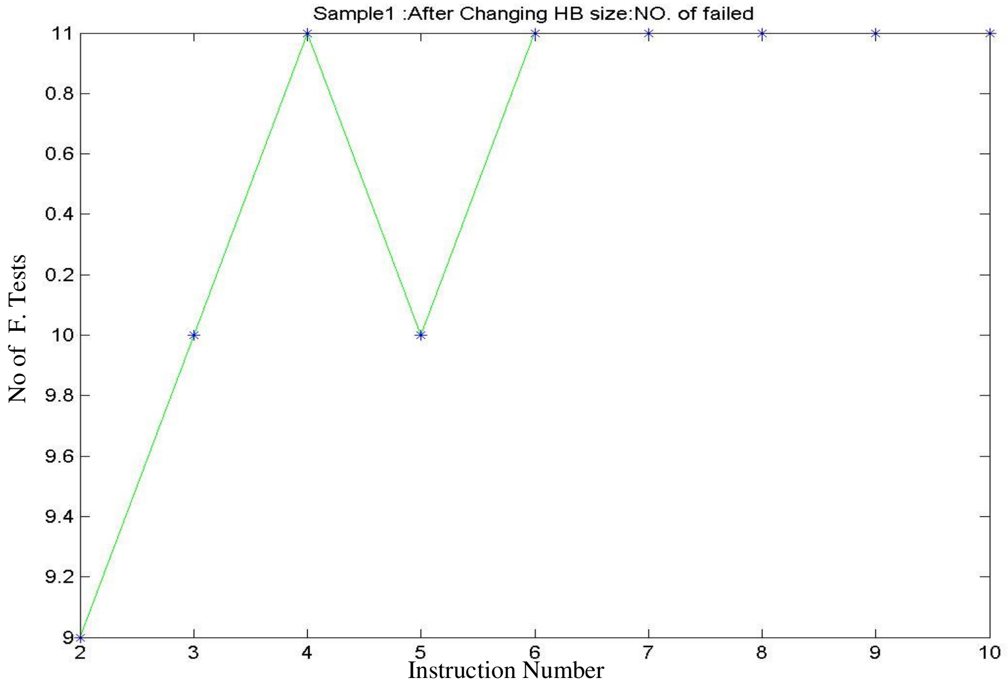
(a) Original Instruction Code (b) (Resolving Dependency)

(c) Changing FUs and Resolving Dependencies

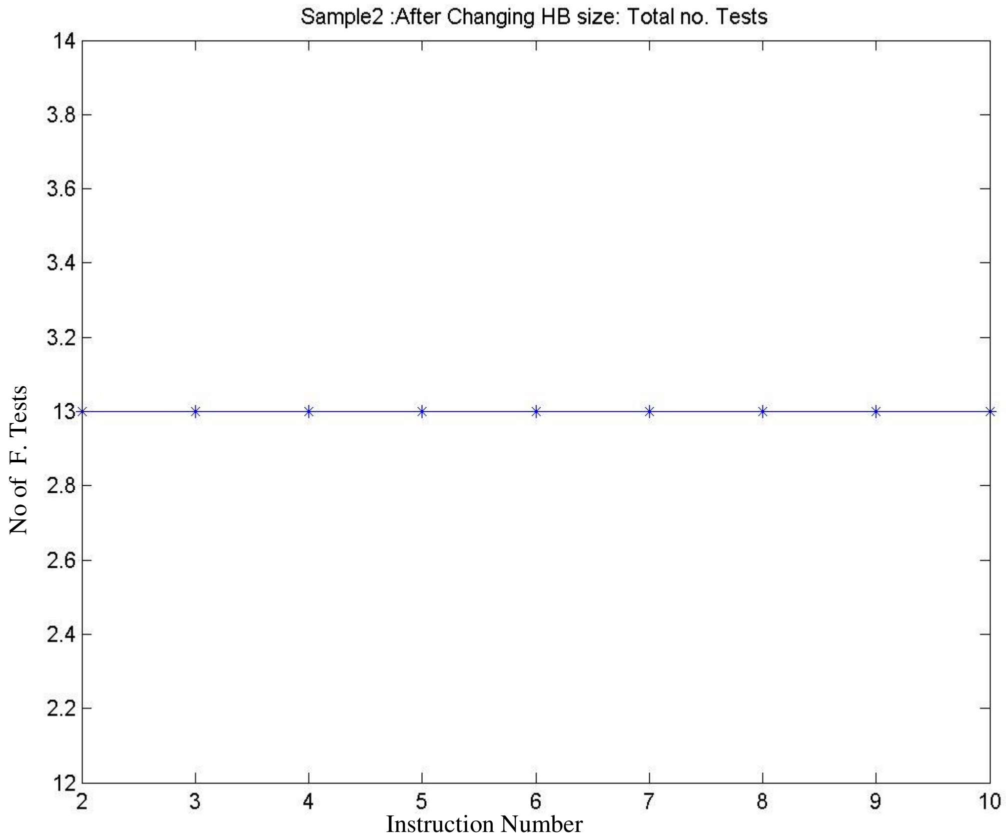
Figure 5.15. HB. Sample 2. The number of failed tests and Succeed tests to the Total number of tests.



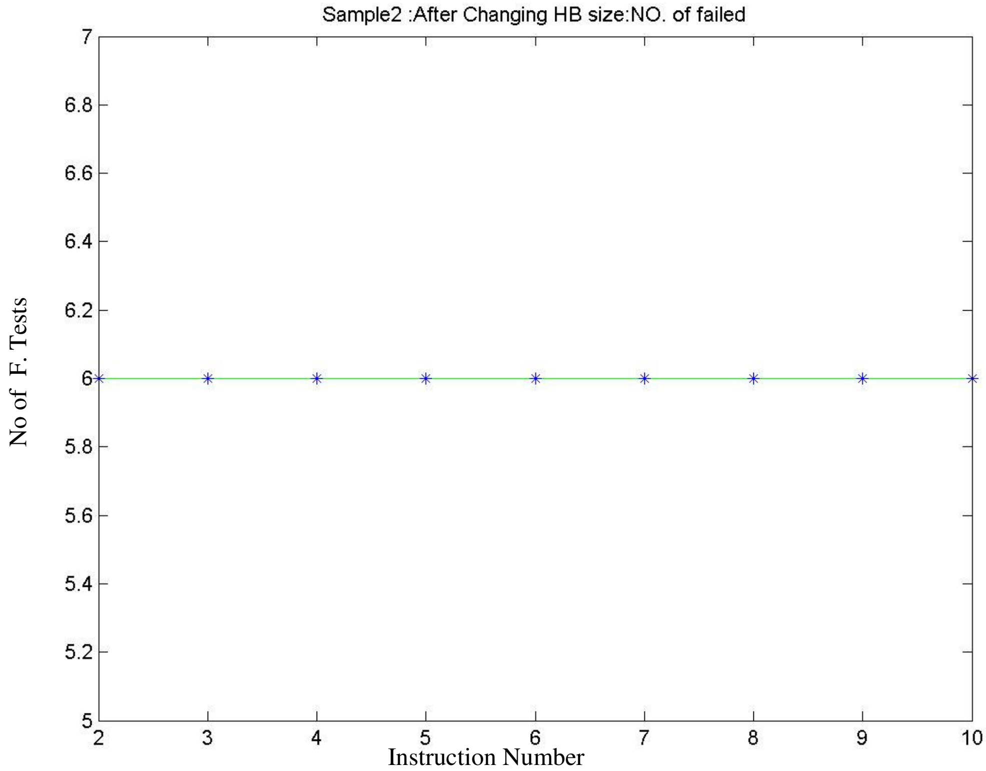
(a) The Total number of tests as the size of ROB changes: Sample 1



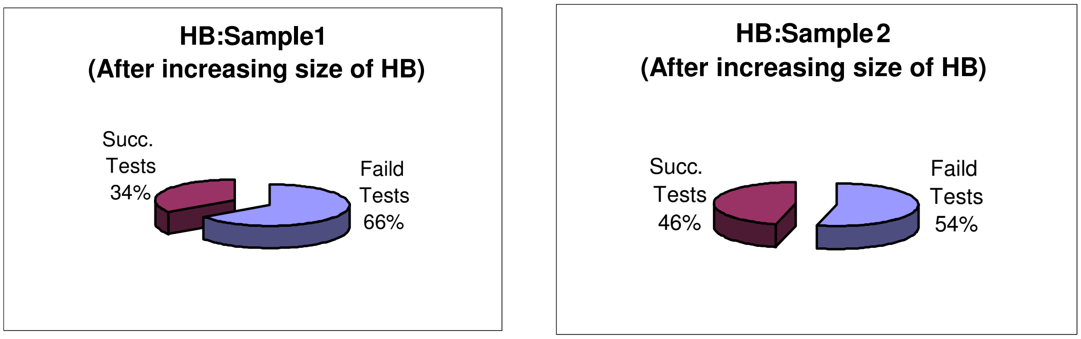
(b) The number of failed tests as the size of ROB changes: Sample 1



(c) The Total number of tests as the size of ROB changes: Sample 2



(d) The number of failed tests as the size of ROB changes: Sample 2



(e) Sample 1: Percentage

(f) Sample 2: Percentage

Figure 5.16. After increasing the size of the ROB.

The main disadvantage is that there only one result bus, so it takes one cycle per entry to restore from the history buffer. In addition to the storage and control logic for the history buffer itself; it requires an extra read port on the register bank to supply the old values of the destination register to get a case in which there is a need for having 3 read ports.

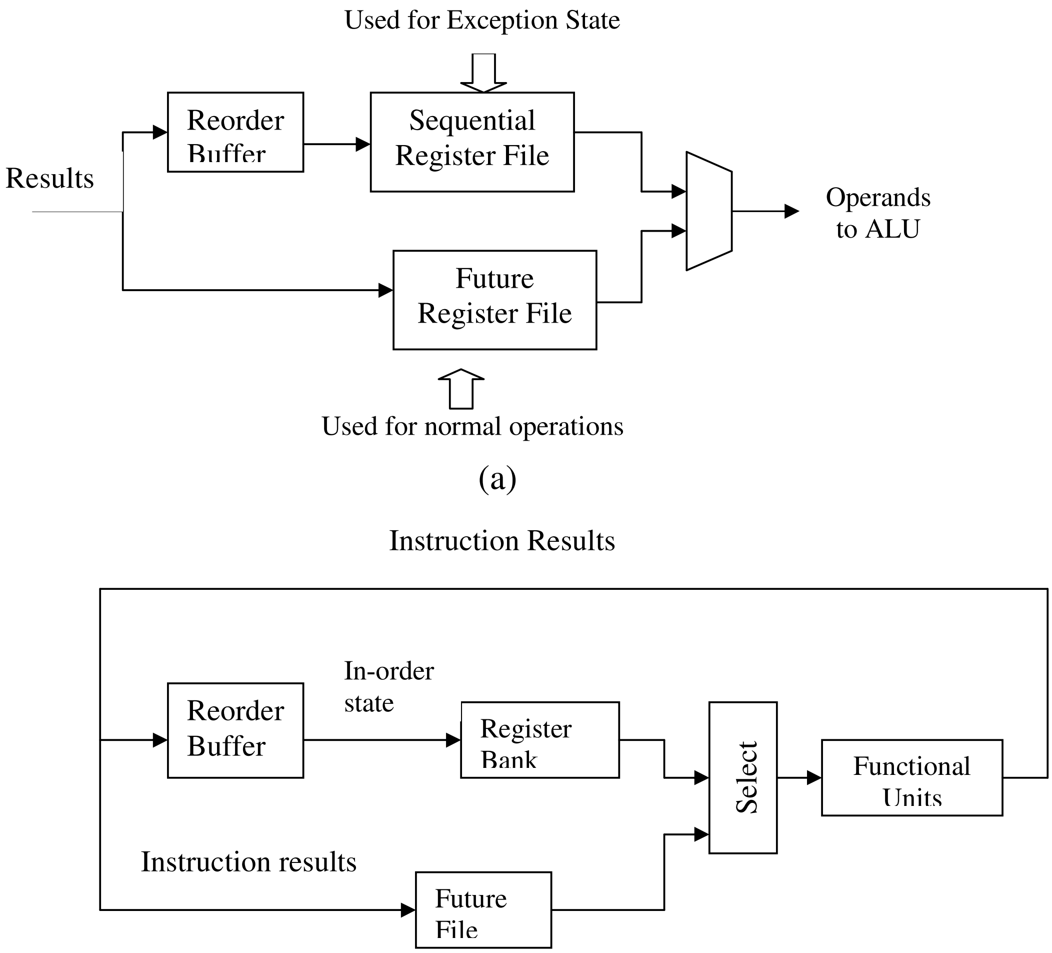
5.4 The Future File (FF)

5.4.1 Scheme Description:

The organization of a system using the future file is shown in figure 5.19. It consists of a model similar to the simple reorder buffer with the addition of an extra register file known as the future file "FF". As in the simple reorder buffer system, the reorder buffer holds look-ahead state and the register bank holds in-order state. In normal operation the future file holds the architectural state, however upon recovery from exception the architectural state is formed by a combination of the future file and the Architectural File "AF" (David, 1997).

In this scheme, there are two register files: the Architectural File, which has in-order results, and the Future File, which has out-of-order results for use as operands and use Reorder Buffer to keep Architectural File in-order.

As each instruction is issued the location in the future file corresponding to its destination register is marked to indicate that it is valid, and a tag is stored corresponding to the instruction which is to write the result (David, 1997).



(a)

(b)

Figure 5.17. Processor organization with a future file

As results arrive from the functional units they enter the reorder buffer and the future file. If the instruction returning the result does not match the future file's tag, the result is discarded; this allows WAW dependencies to be resolved by discarding older versions of the register irrespective of the order that they return from the functional units. If the tag matches the result overwrites it (David, 1997).

When an instruction is being issued and needs to read its operands, it reads the same location in the register file and the future file. If the future file location is marked as invalid the operand is read from the architecture file, otherwise the tag or value is read from the future file. If the future file contains a tag, the instruction issue must stall to wait for the result to arrive. This lookup and flag check replaces the tag comparator lookup used in the reorder buffer (David, 1997).

When an exception occurs the valid results in the reorder buffer before the exception drain into the Architecture File, completing the in-order state. The valid flags in the future file are then cleared. Any instruction, which is now executed, will read from the in-order state in the register bank (David, 1997).

5.4.2 Scheme Analysis

Here we are using two files, the future file (FF) and the architectural file (AF). The future file is (more-or-less) a normal register file. But the main differences with the ordinal files can be summarized as follows:

- While the architecture register file (AF) holds in-order state and is updated in order, the future file (FF) is updated out-of-order. Reorder buffer controls updates to architecture file.
- The future file holds the architectural state.

1. During decode, all operands are read from either the architecture file or the future file, whichever is current.
2. The value read from the future file could be tagged if the instruction producing the value has not been yet completed.

- The reorder buffer manages look-ahead state for eventual retirement to the architectural file.
- While the Future file is managed like an ordinary imprecise pipeline, the Architecture file managed like reorder buffer scheme when head entry valid. On exception Architecture file contains precise state.

There are two views of the algorithm that can be used. The original algorithm maintains the precise state with respect to the program counter. In this work we make some modification to this algorithm by keeping the same advantages with respect to the registers not the PC as the second view does.

To implement the FF scheme, the following are the main steps:

1. Once the instruction is issued, it is transferred to RSR.
2. When instruction is completed, write from RSR to both future register and ROB.
3. Decoder reads from future file (no bypassing).
4. When non-faulting instructions reach the bottom of the reorder buffer, their results are written to the AF.
5. When a faulting instruction reaches the bottom of the reorder buffer,

   a. The FF and the ROB are cleared.

   b. On interrupt, copy AF into FF.

The following modifications can be applied to implement the second view of the algorithm:

1. After an exception is detected, the ROB is emptied from tail to head and RSR is also emptied.
2. After executing all previous instructions and handling the exception, transfer all completed instruction from the FF to RSR and then to ROB.
3. The transferred instructions are those have no exceptions and have no dependencies to the excepted instruction. This can be used to ensure that no error will occur in the register values after this transfer.
4. The transferred instructions are tested as before in order to be committed and transferred to the AF.
5. This improvement can be used to improve the execution time and to increase the speed up of the pipeline by not rolling back all instructions but trying to complete the pipeline from the moment after the exception.

The main experiments that were done follow the second view of the FF algorithm.

Experiments:

The same experiments used in the previous mechanisms, were used here to test the algorithm. Some modification is applied on the original algorithm as mentioned above. The results of the experiments done were as follows:

1. The first, and the second sample of code, and all their variations were applied to the algorithm. The results were as follows:

   a. The instructions that have hazard dependencies have a sharp peak at the instructions with dependencies as shown in Fig. 5.18 and 5.19. Other instructions have a uniform number for the failed tests.

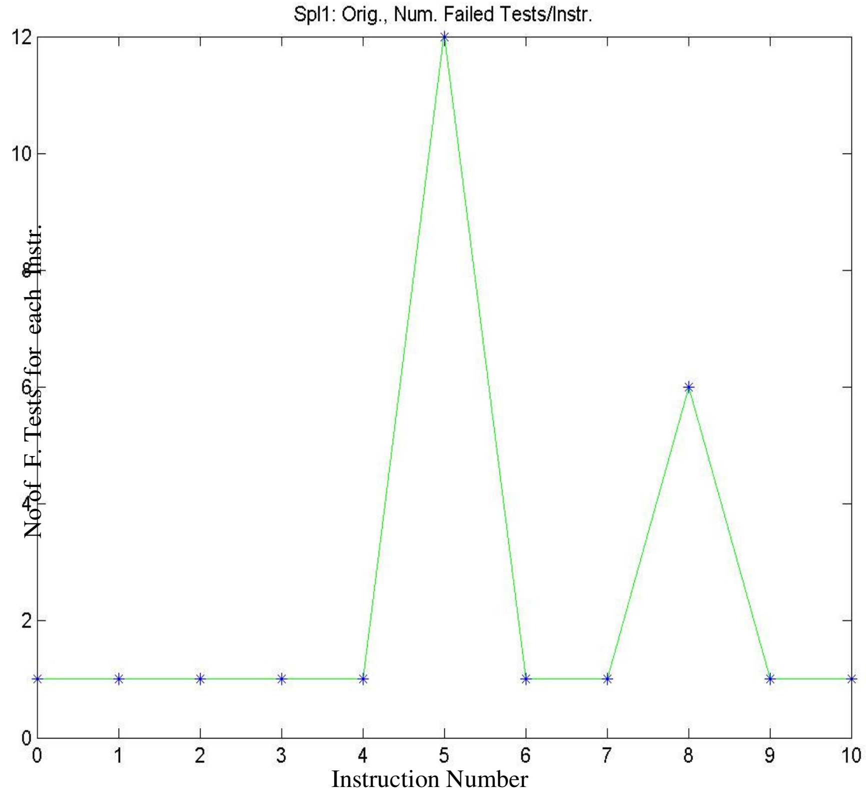
   b. The number of the failed tests and succeeded tests are clearly changed as shown in figure Fig.5.20, Fig. 5.21. Changing the FUs improve the performance significantly. The total execution time also is decreased significantly after increasing the number of the functional units.

2. The RAW dependencies are solved; this increases the number of shifts necessary for each instruction in order to be completely executed.

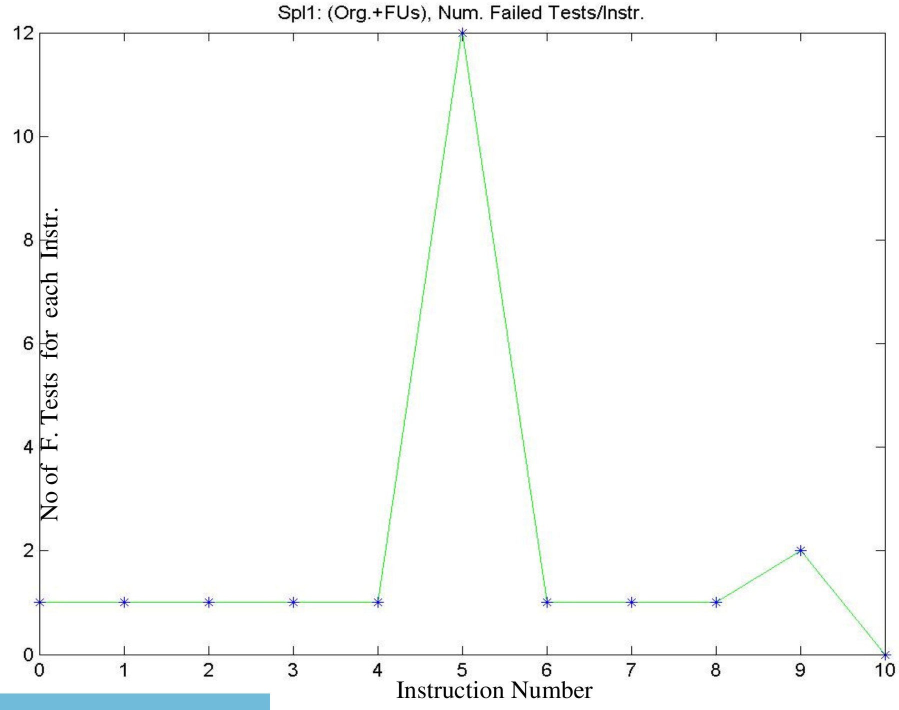
3. When the size of the Future file, ROB is increased, the results were as shown in figure 5.22. As we increase the size of the FF and ROB, the total number of shifts and tests were changed considerably. The change in the number of shifts appears more clearly. The same discussion on the size of both ROB and FF can be applied as that mentioned to the ROB and HB previously.

4. We applied the two samples of codes with their variations using the original flow of the algorithm, the change in the number of tests and shifts were shown in figure 5.23. In general, the same results obtained as in figure 5.22, except the number of shifts, which is changed. This can be justified by remembering that re-executing the completed instructions after the excepted instruction needs more shifts. The nature of the executed instructions affects the results obtained. If the instructions need to be re-executed every time an exception occurs, this will increase the number of tests and shifts, but if the instructions are independent

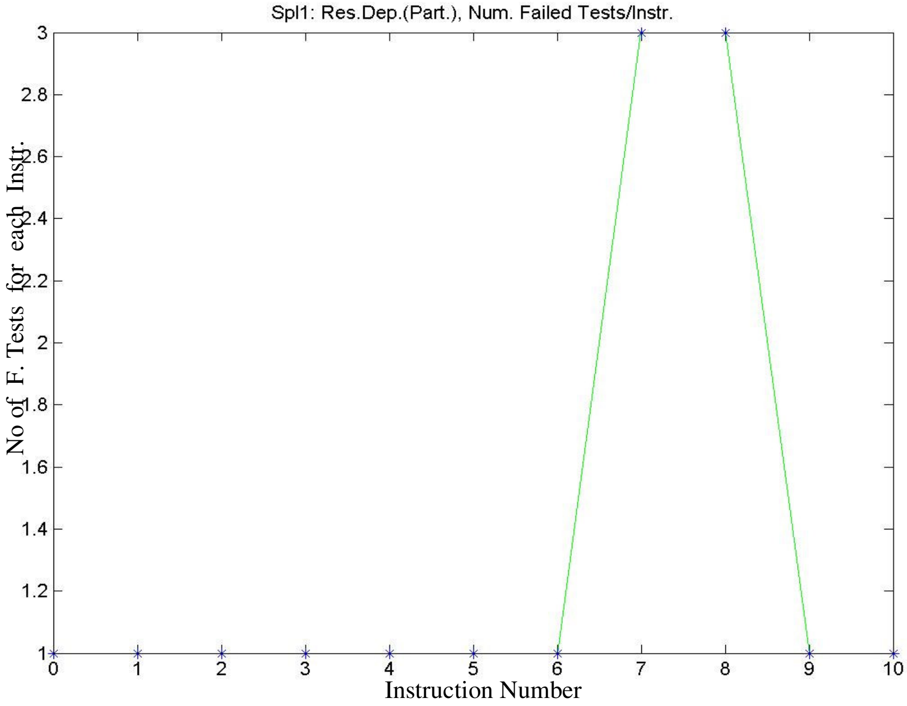
and do not need re-execution the two forms of the flow will give the same results.



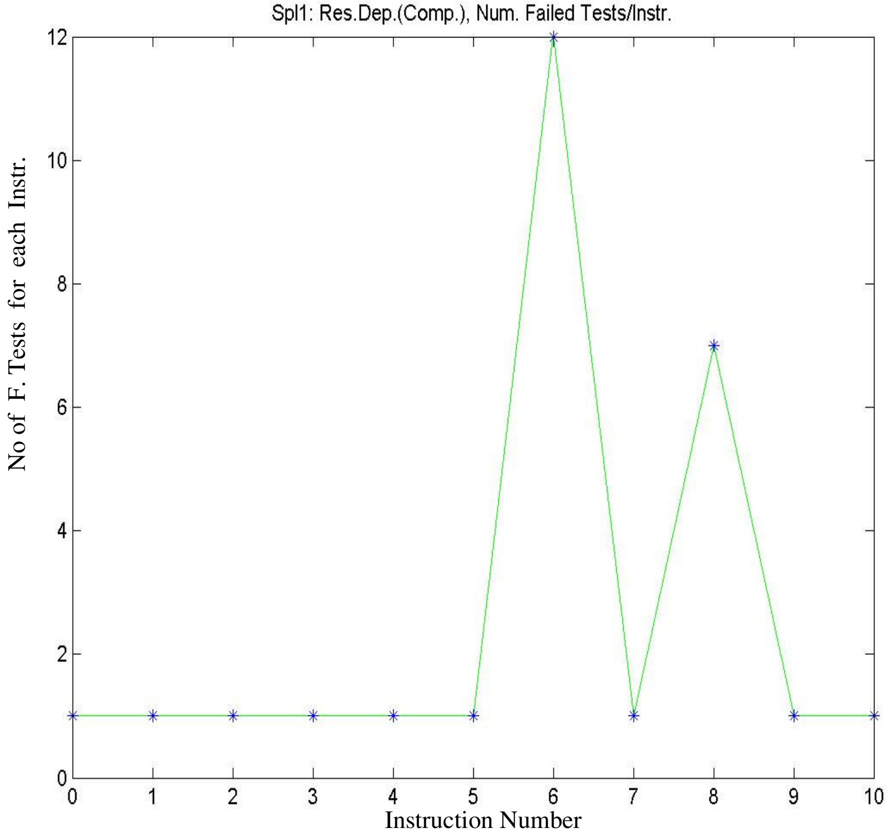
(a) Original Code.



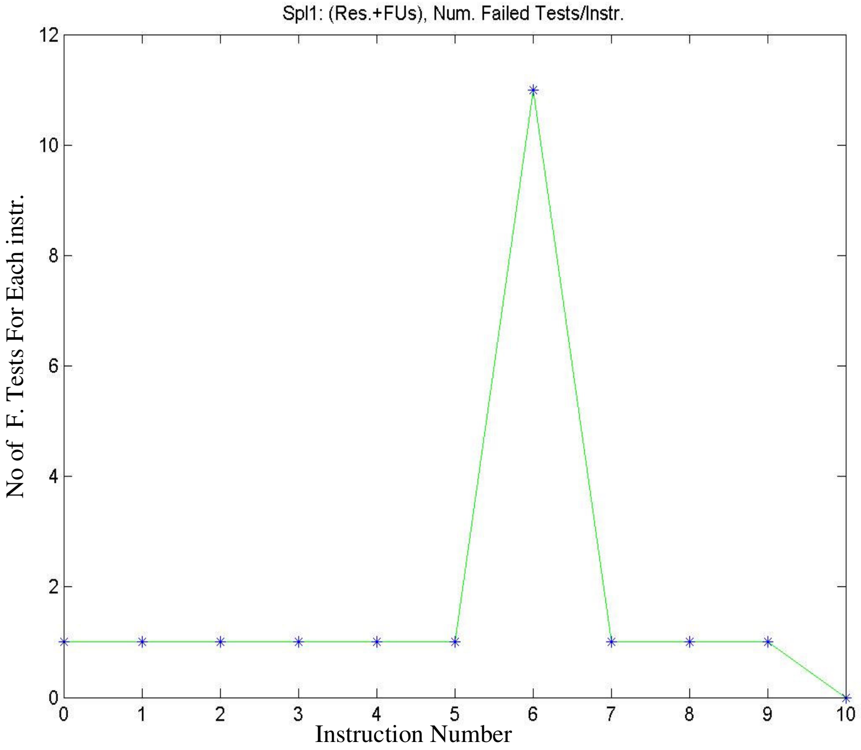
(b) Changing FUs.



(c) Resolve Dependency hazards (partially)

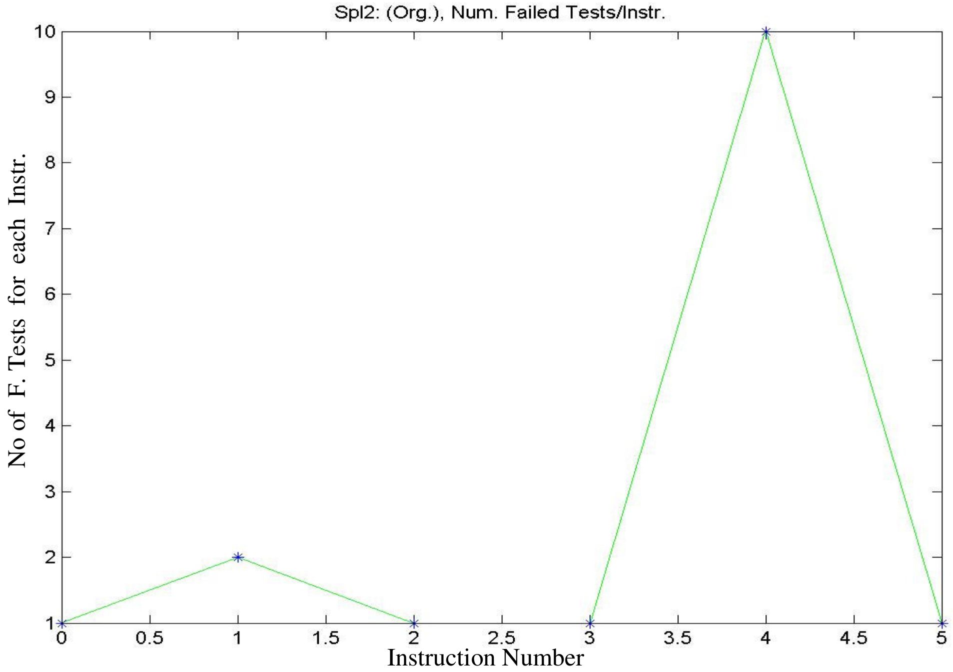


(d) Resolve Dependency hazards (Completely)

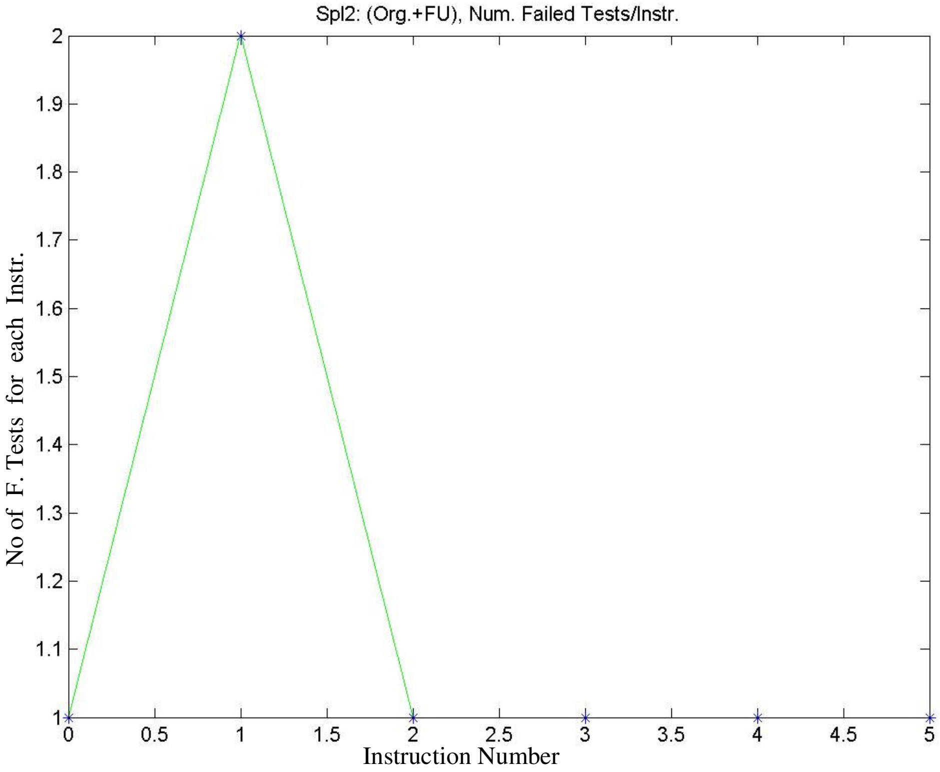


(e) Resolving Dependencies and FUs.

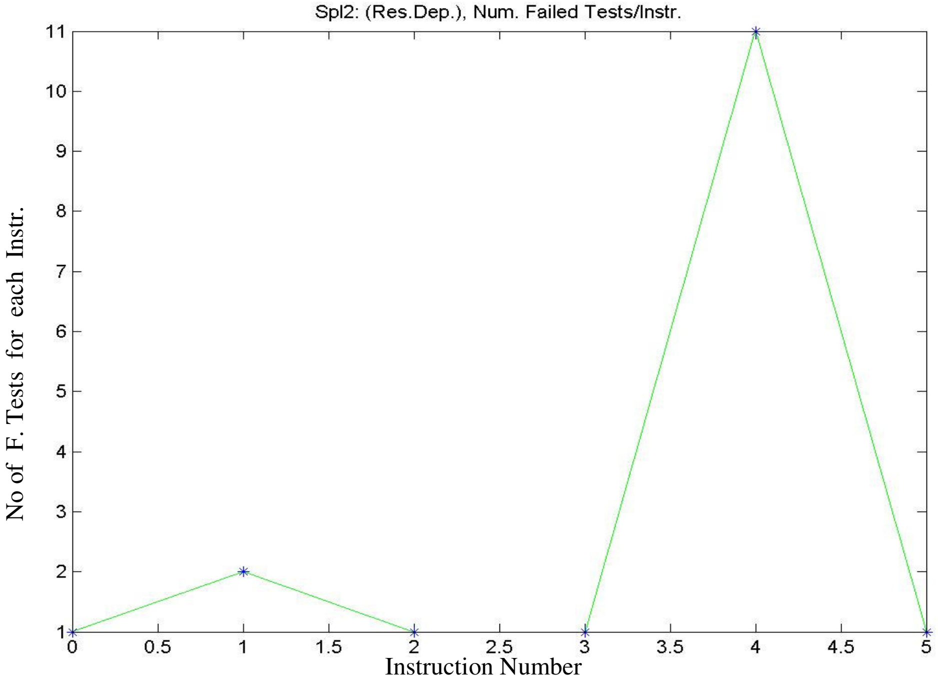
Figure 5.18. FF. Sample 1: the number of Failed Tests per each instruction.



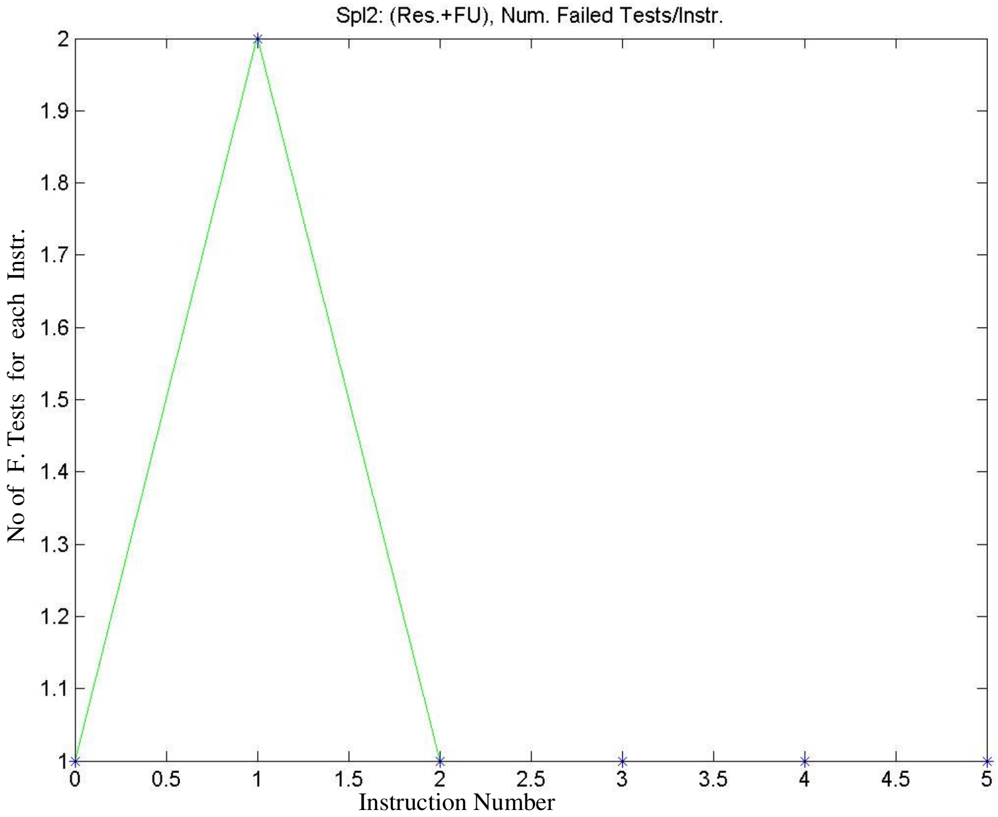
(a) Original Code.



(b) Changing FUs.

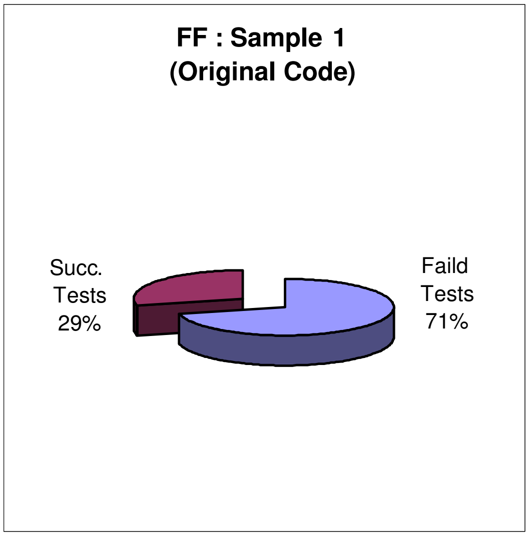


(c) Resolve Dependency hazards.

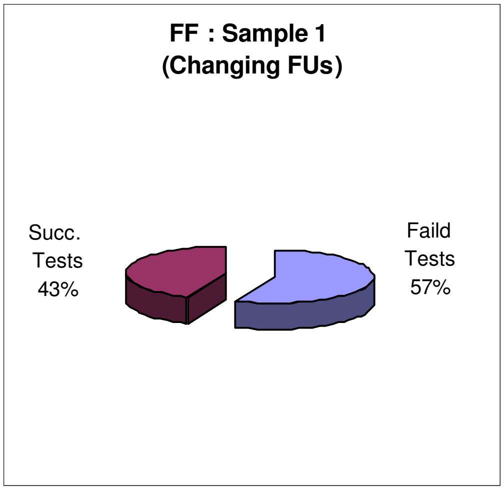


(d) Resolving Dependencies and FUs.

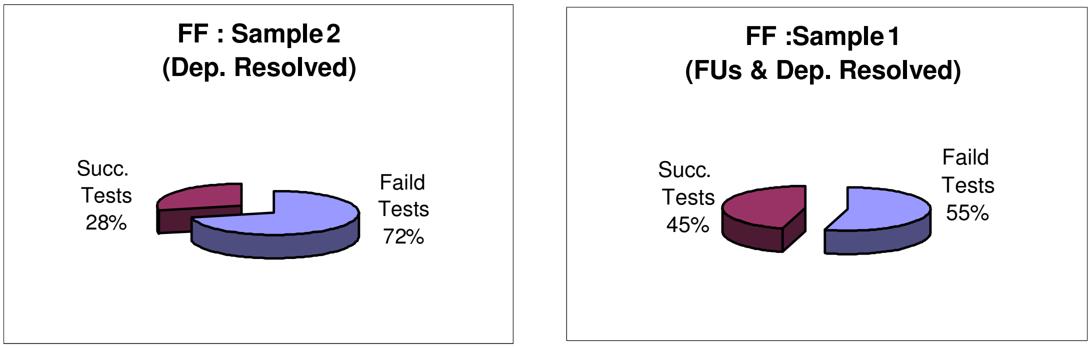
Figure 5.19. FF. Sample 2: the number of Failed Tests per each instruction.



(a) Original Code



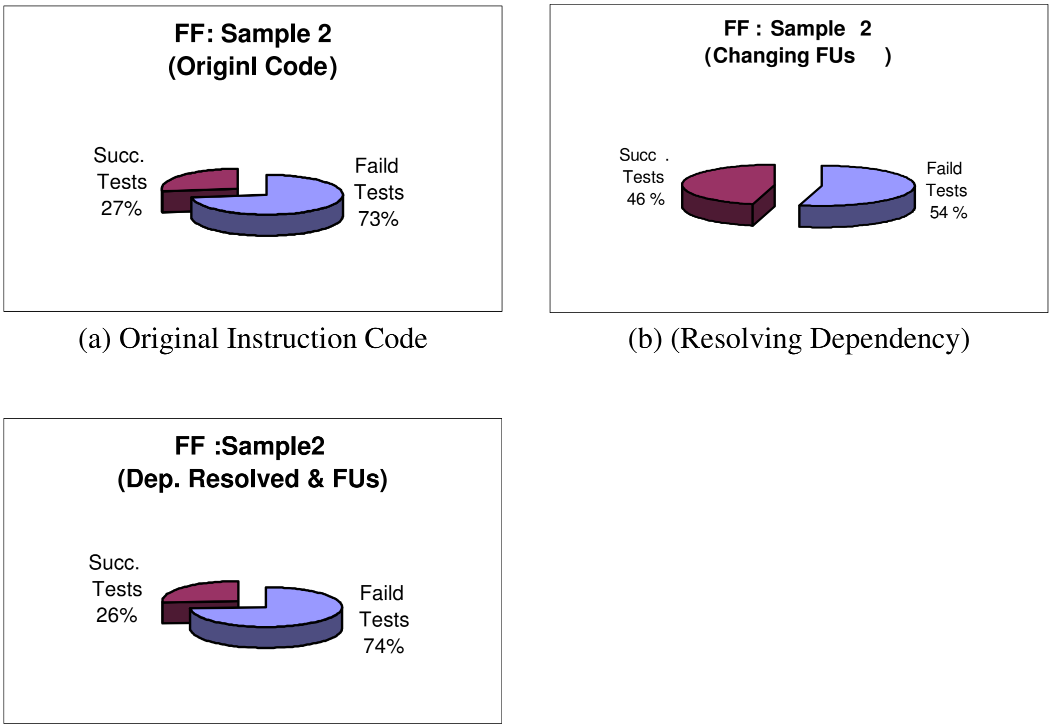
(b) Original Instructions (changing FUs.)



(c) Resolved dependencies (completely)

(d) Changing FUs and Resolving Dependencies

Figure 5.20. FF. Sample 1. The number of failed tests and Succeed tests to the Total number of tests.

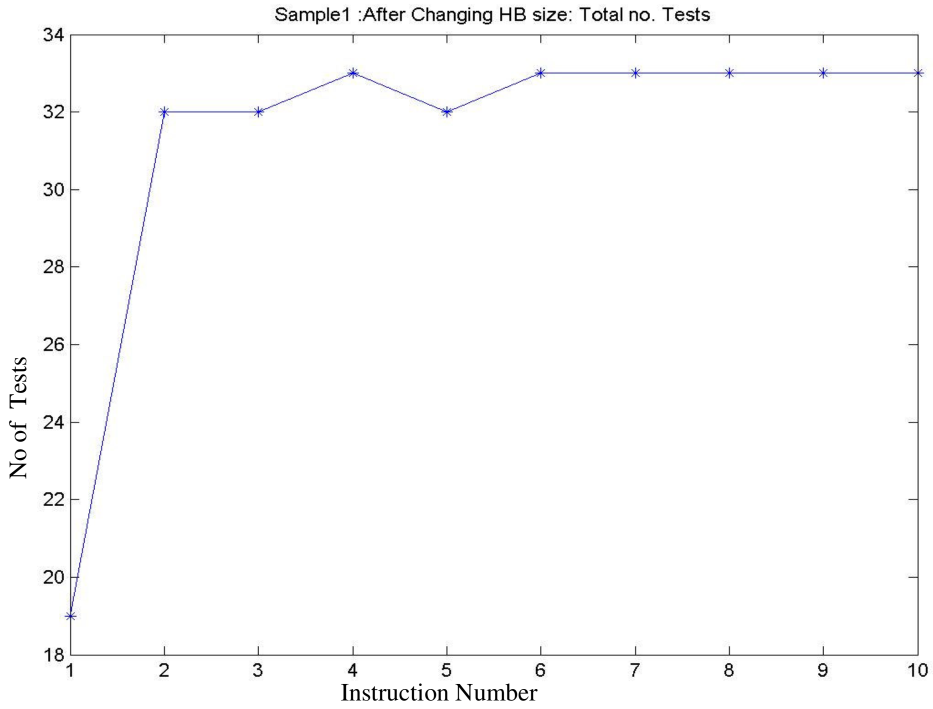


(a) Original Instruction Code

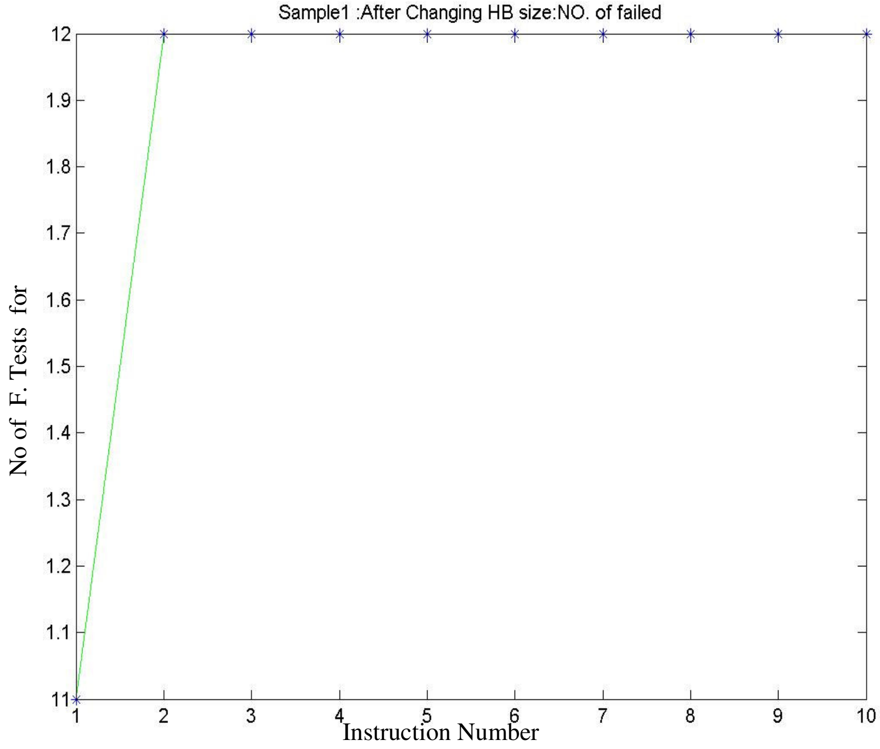
(b) (Resolving Dependency)

(c) Changing FUs and Resolving Dependencies

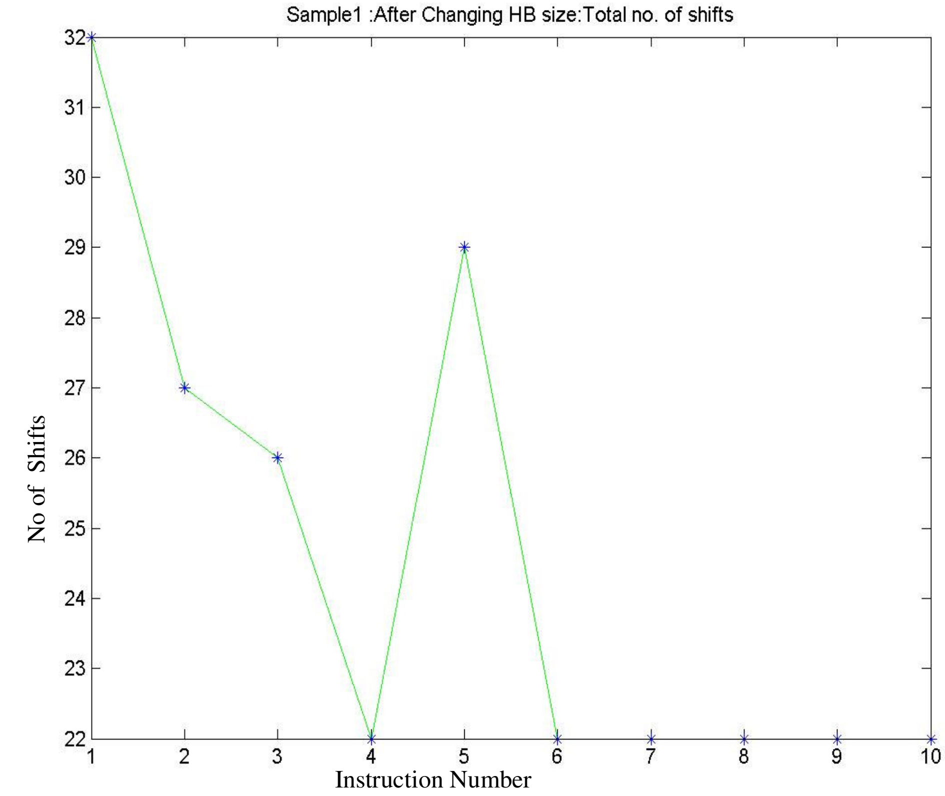
Figure 5.21. FF. Sample 2. The number of failed tests and Succeed tests to the Total number of tests.



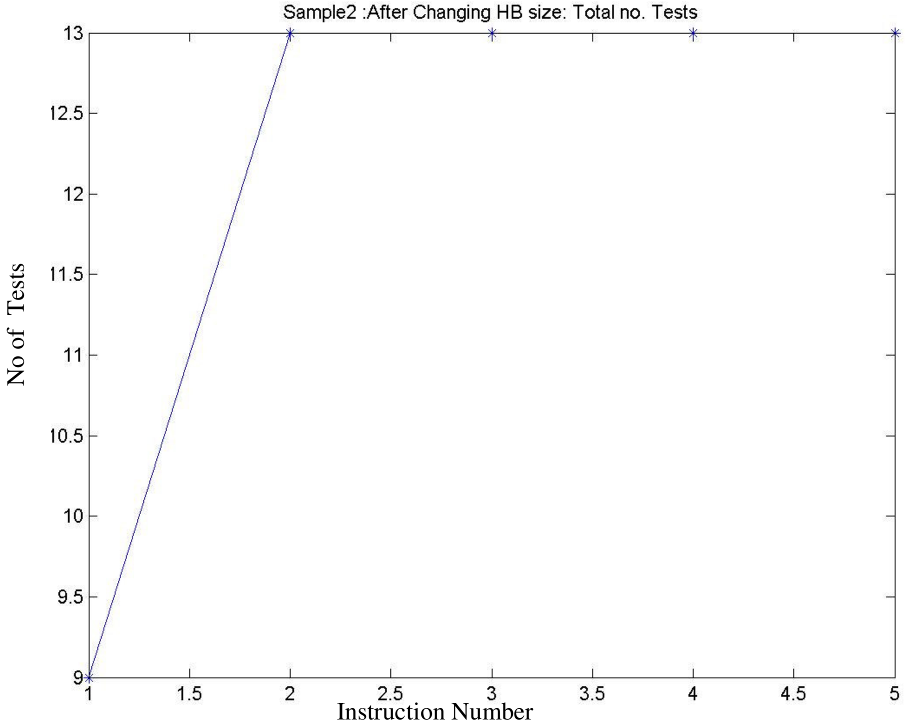
(a) The Total number of tests as the size of FF changes: Sample 1



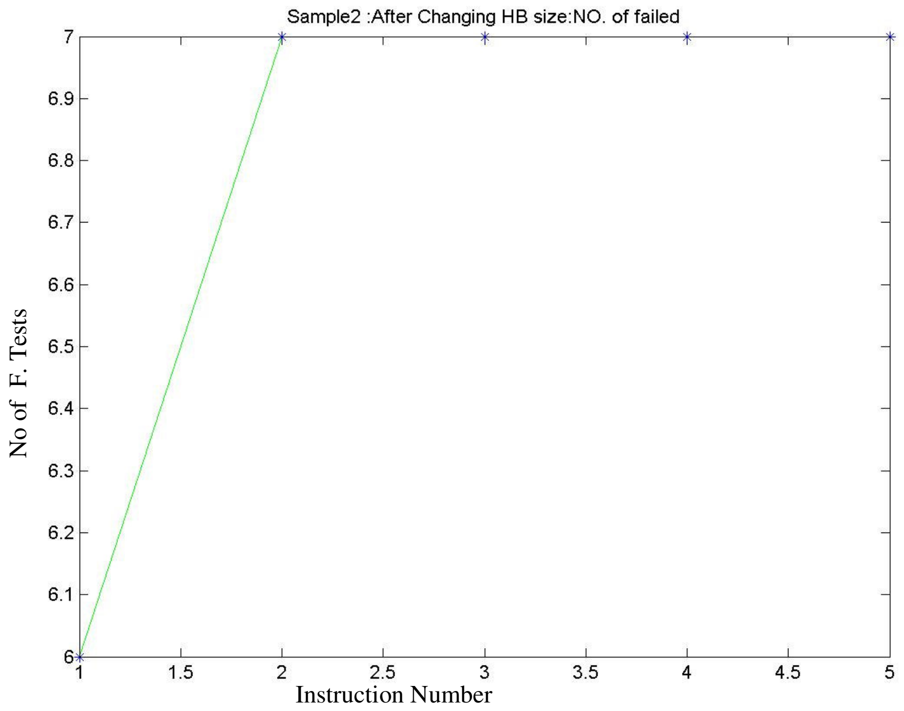
(b) The number of failed tests as the size of FF changes: Sample 1



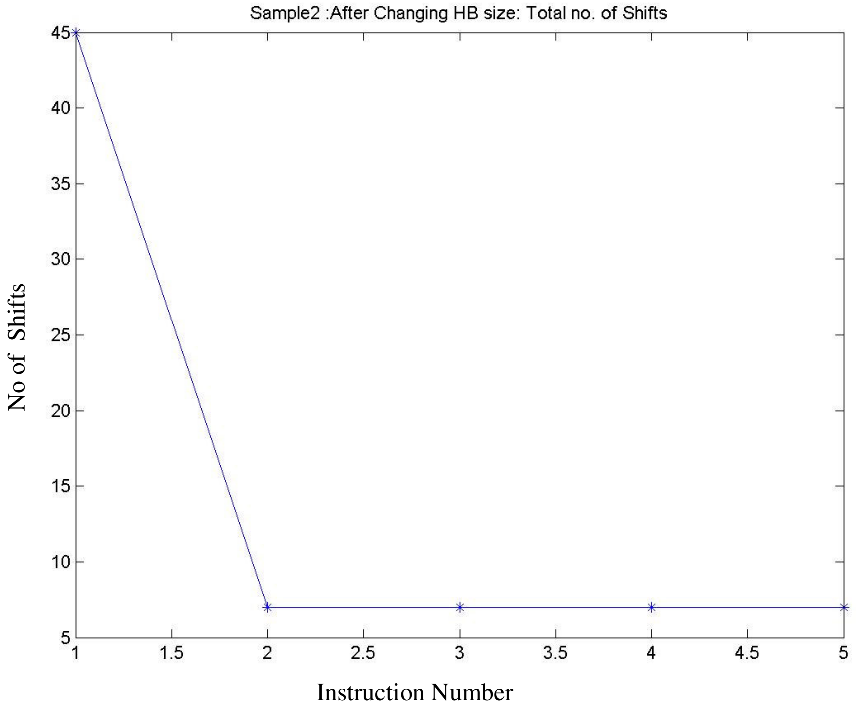
(c) The Total number of shifts as the size of FF changes: Sample 1



(d) The Total number of tests as the size of FF changes: Sample 2

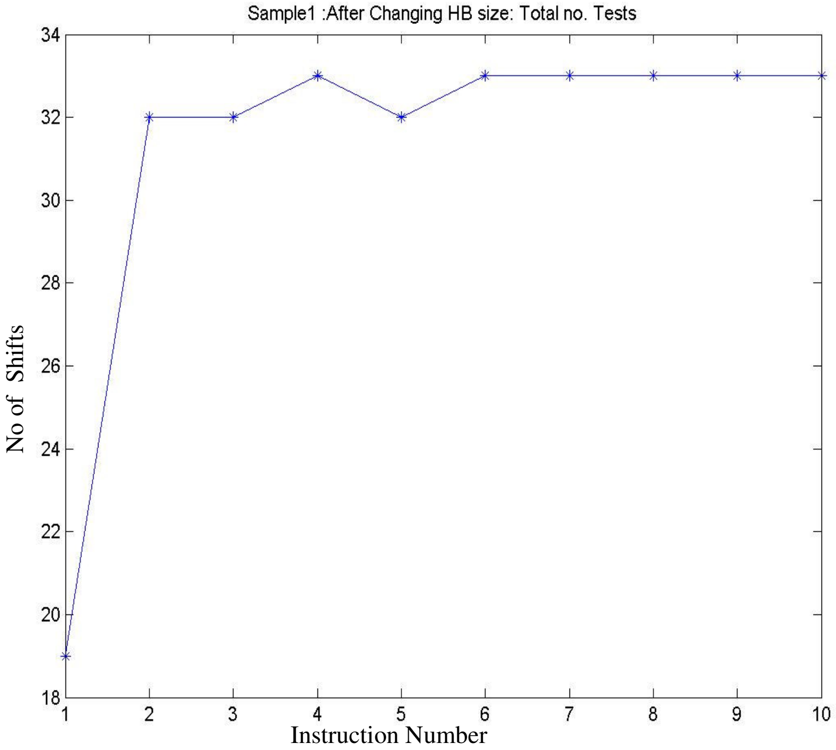


(e) The number of failed tests as the size of FF changes: Sample 2

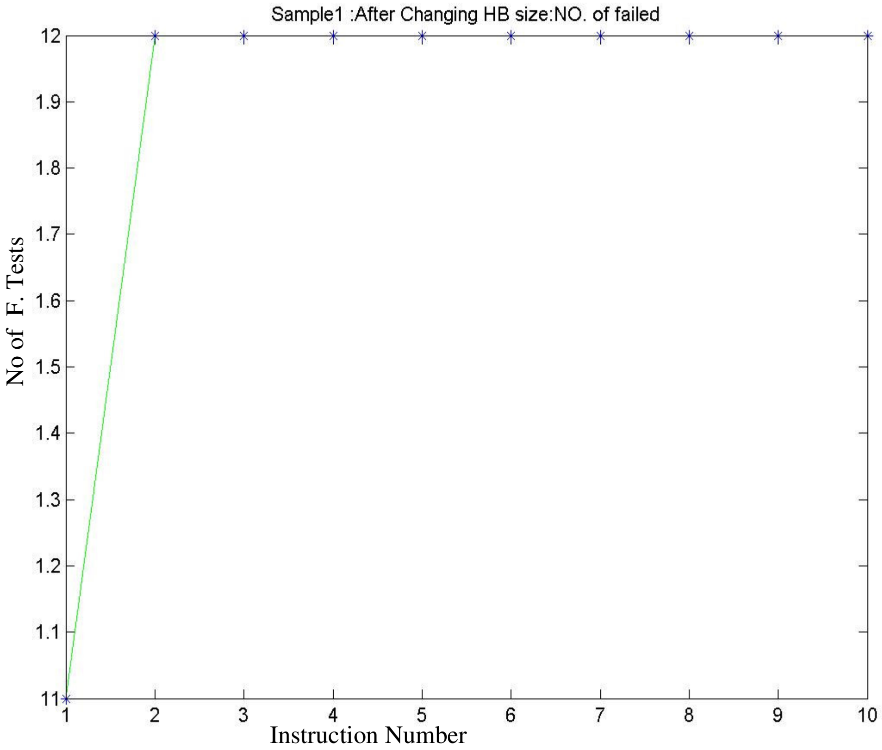


(f)The Total number of shifts as the size of FF changes: Sample 2

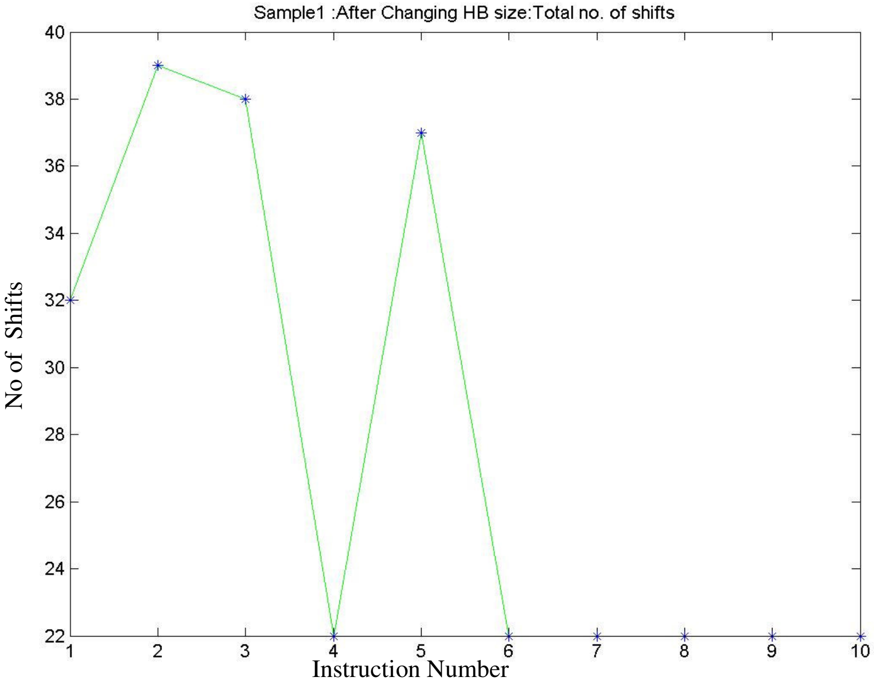
Figure 5.22. FF. After increasing the size of the ROB, FF.



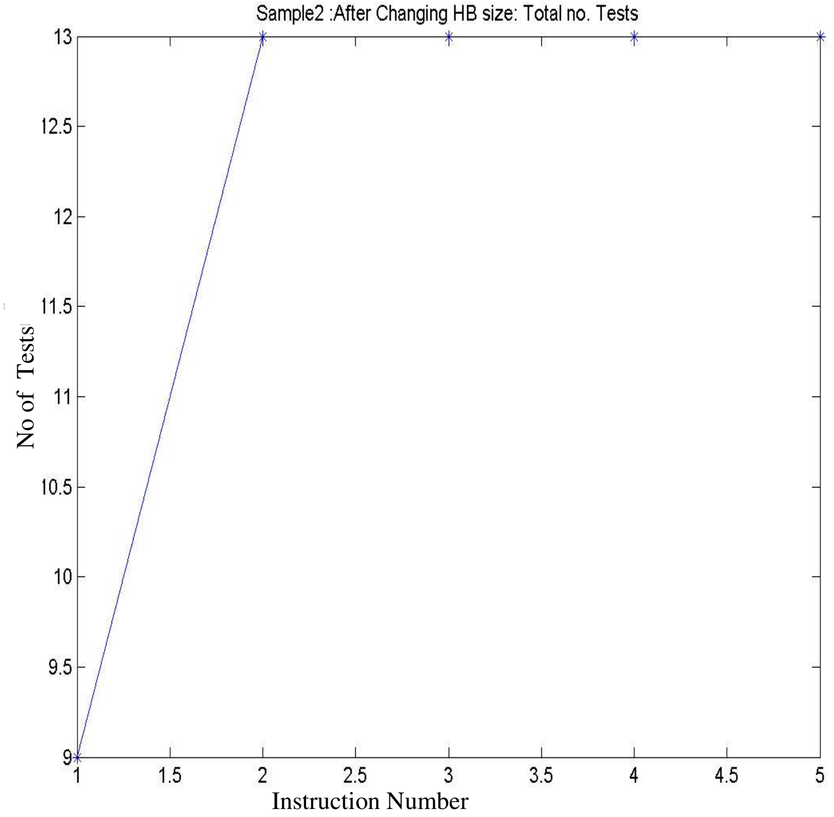
(a)The Total number of tests as the size of FF changes: Sample 1



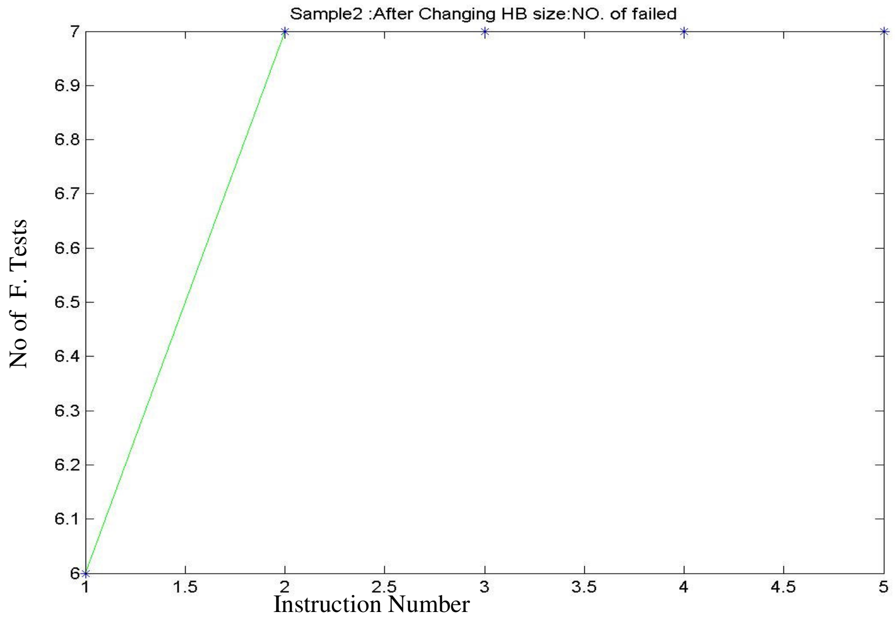
(b) The number of failed tests as the size of FF changes: Sample 1



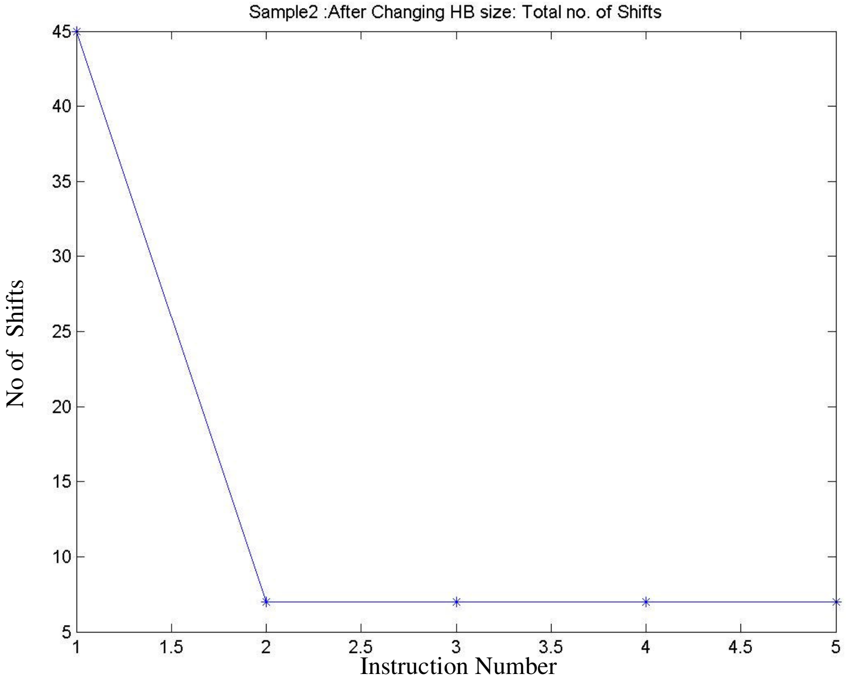
(c) The Total number of shifts as the size of FF changes: Sample 1



(d) The Total number of tests as the size of FF changes: Sample 2



(e) The number of failed tests as the size of FF changes: Sample 2



(f)The Total number of shifts as the size of FF changes: Sample 2

Figure 5.23. FF. After increasing the size of the ROB, FF using the 1st view of the algorithm.

5. The number of the functional units can affect the performance of the technique, as well as the dependency resolving, positively.

Finally, the main disadvantages of the Future File scheme:

1. The cost in terms of hardware is a duplicate register bank (the future file itself), and the validity and tag logic.
2. The future file cannot, by itself, solve RAW dependencies and so is unsuitable for out-of-order execution without the addition of extra hardware (David, 1997).

The key advantages offered by the future file can be summarized as follows:

1. Associative lookup in the reorder buffer is no longer required.
2. State saving is easy.
3. No extra bypassing.

5.5 Summary

The out of order schemes for handling the precise interrupts were implemented and examined. After analyzing all schemes, we can see that the ROB technique is not efficient without bypass paths. These forwarding paths maintain the preciseness of the processors when they are interrupted and solve the RAW and WAW dependencies. Although HB scheme reduces the amount of storage needed, it takes one cycle per entry to restore from the HB since there is only one result bus. In the Future File scheme there are two files, so the hardware cost is increased because of the duplicated register banks. Also there is no extra bypassing, but it cannot by itself solve the RAW dependencies.

6. The Proposed Approach

This chapter implements the proposed scheme, which is a compromised approach between the In Order completion and the Out of Order Schemes for handling precise interrupts in pipelining system.

6.1 Scheme Description

The same RSR and logic control can be used as that used in the In Order Completion approach. There is logic on the result bus that checks for exception conditions in instructions, as they complete. This control information identifies the functional unit that will supply the result and the destination register of the result. It is also marked “*valid*” with a validity bit. Each clock period, the control information is shifted down one stage toward stage one. When it reaches stage one, if its program counter is the current program counter to be committed, it is used during the next clock period to control the result bus so that the functional unit result is placed in the correct result register.

In this proposed scheme, the scenario is changed. Reserving the entries in the RSR necessary for each instruction, which are not necessarily consecutive stages, but occupying the first unused entries in the RSR to keep it executing in order can accomplish this. The RSR contents are shown in table 6.1.

This makes it possible for more instructions to be executed at the same time, i.e., to increase the pipeline throughput. So, in this way, it is possible for a short instruction to be placed in the result pipeline in stage *i, i –1,i –2,..* , Where i_s are the first unused stages in RSR.

Table 6.1. Result Shift Register – Sample 2

(a) Result Shift Register - Sample 1 (James *et al.,* 1988).

Stage	**Functional Unit Source**	**Destination Register**	**Valid**	**Program Counter**
1			0	
2	Integer Add	0	1	7
3			0	
4			0	
5	FLPT ADD	4	1	6
. .	. .	. .	. .	. .
N			0	

Shift Upward

Instruction In stage 1 writes the result-bus

(b) Sample 2

Stage	**FU/OP**	**Destination Register**	**Valid**	**Program Counter**
1		↕		
2	LD (1)		0	A
3	LD (2)		0	B
4				
5				
6				
7				
8		Reserved by MULTD		
9				
10				
11				
12				
13	MULTD			C
14				

Shift Upward

Instruction In stage 1 writes the result-bus

These modifications necessary to reserve the entries in the RSR, causes complexity in handling instructions completed out of order, as in the Out of Order

Completion, these instructions must be kept in the RSR to be loaded to the result bus only when their PC is the current program counter to be committed.

6.2 The Proposed Scheme Algorithm

To implemented the proposed scheme, we can follow the following steps:

- Initialization:

1. Initialize the RSR

- ***Instruction Issue***:

1. If there is no RAW dependency and there is enough space in RSR for the next instruction, then it is issued.
2. Otherwise, stall the instruction issue.

- Instruction Completion:

1. If the instruction caused an exception, mark exception bit in the RSR for that instruction
2. If the completed instruction has not the correct PC, stall the instruction until instruction with the correct PC is completed. Otherwise commit instruction.
3. At the commit stage, check the exception bit:
    1. If there is no fault:
        a. Commit state (the value is written to the register file).
        b. The entry removed from the RSR. And all reserved entries for that instruction are flushed.
    2. If the exception bit is set:
        a. Issue is stopped in preparation for the interrupt.

b. Squash subsequent instructions in RSR

c. All further writes into the register file are inhibited.

d. The in-order state of the register file is restored.

6.3 Scheme Analysis

Many experiments were done to test the scheme to handle the precise interrupts. The same variations used in previous chapters were applied here. The results of the experiments were as follows:

C- The first sample of code is applied to this suggested approach:

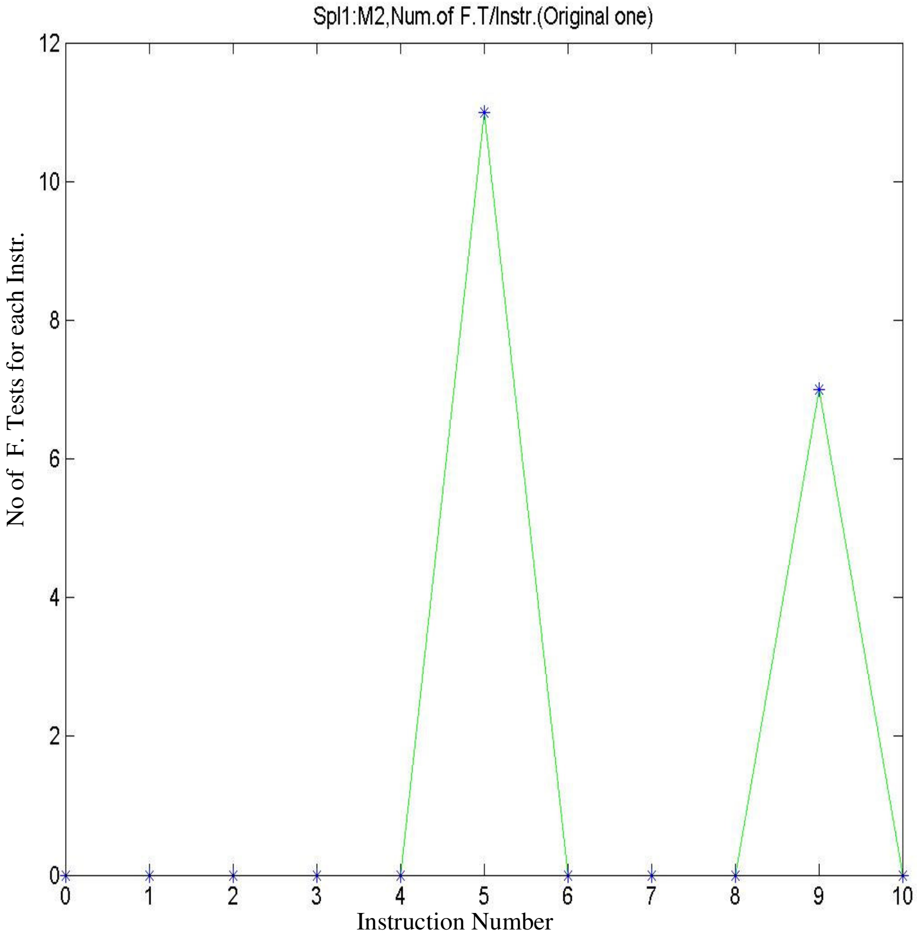
1. The total number of shifts and tests were changed, as we reorder instructions to resolve hazards, and change the FUs. The percentage for the failed tests and succeeded tests was as shown in Fig.6.2.
2. Fig. 6.1 shows how is the number of shifts and failed tests, is changed after each modification. As we change the number of functional units, we get better performance. The same happens if we resolve the dependencies among instructions and if we combine the two experiments together.

B- The second sample of code is applied. The results were as follows:

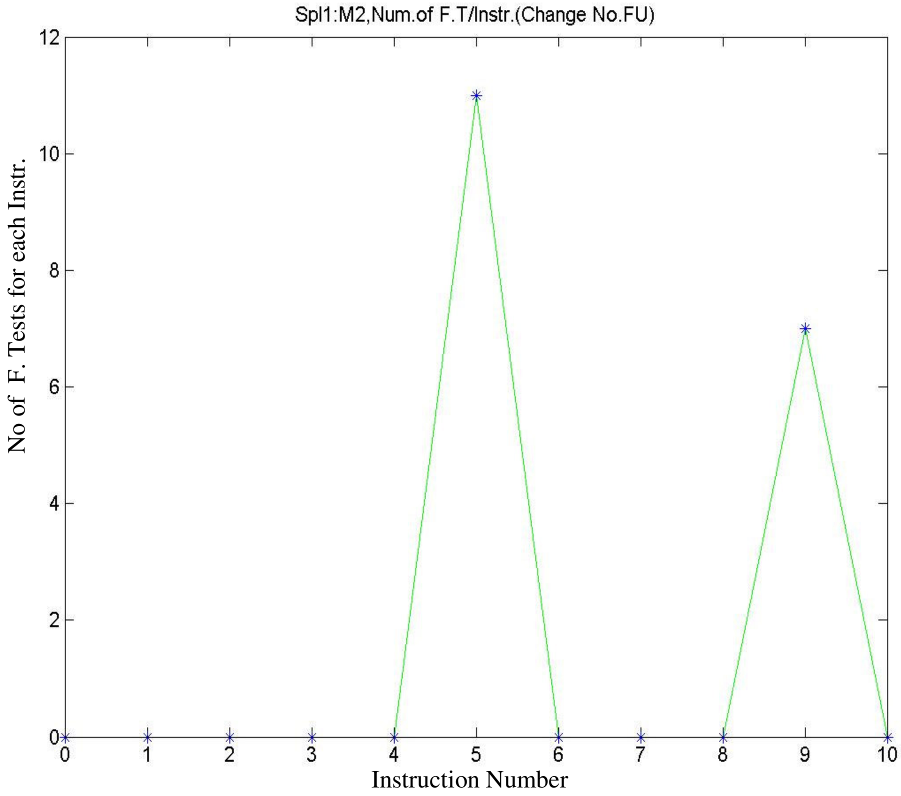
1. The total number of tests was the same without any modification while the total number of shifts is increased as we tried to reorder instructions to resolve dependencies. The percentage for the failed tests and succeeded tests was shown in Fig.6.3.
2. Fig. 6.4 shows that, as we increase the number of functional units or having no dependencies, we get better performance. This better performance appears in the

increase in the number of succeeded tests that are done and decrease in the failed ones.

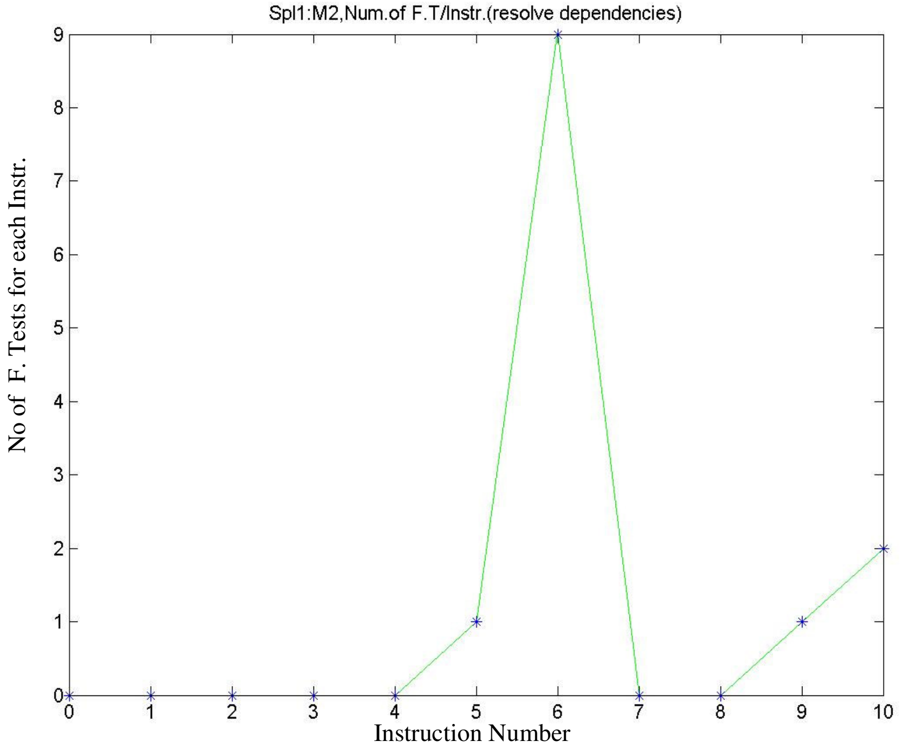
C- Several values of the size of RSR are examined. When we applied both samples, the results were as shown in figure 6.5. As we increase the size of the RSR, we get better results in the total number of tests. This improvement is slightly changed when we expanded the RSR to the double size of the minimum size needed, which is the longest execution time among all instructions used.



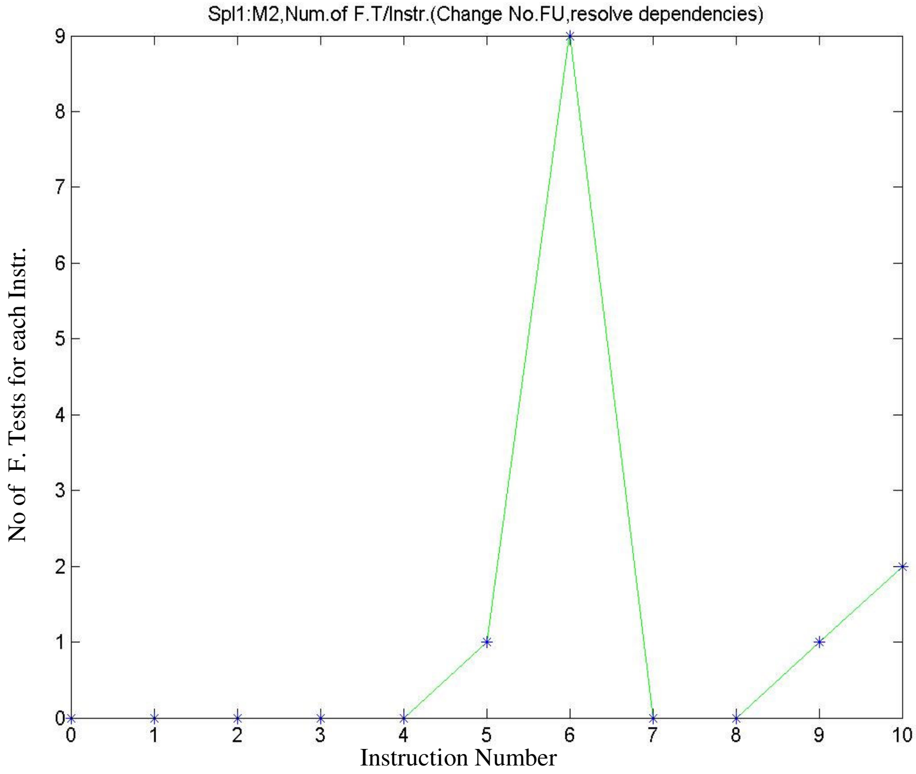
(a) Original Code.



(b) Changing Number of FUs.

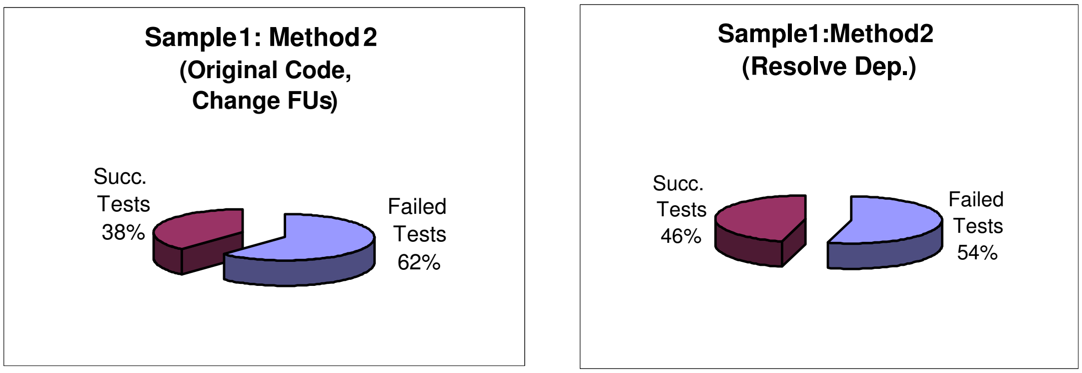


(c) Resolve all Dependency hazards.



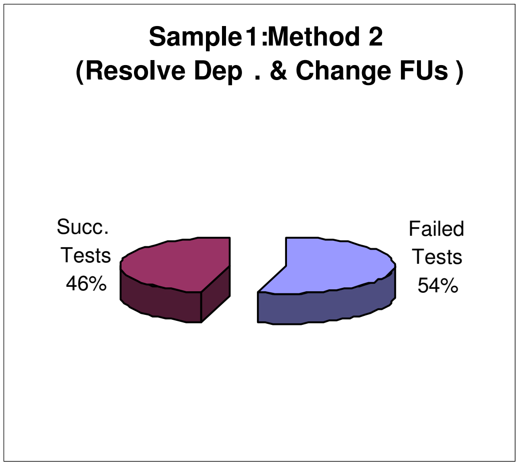
(d) No dependencies, change FUs

Figure 6.1. Proposed Approach. Sample 1, the number of failed tests per each instruction



(a) Original Instruction Code, Change FUs

(b) (Resolving Dependency)



(c)Changing FUs. and Resolve Dep.

Figure 6.2. Proposed Approach. Sample 1. The number of failed tests and Succeed tests to the Total number of tests.

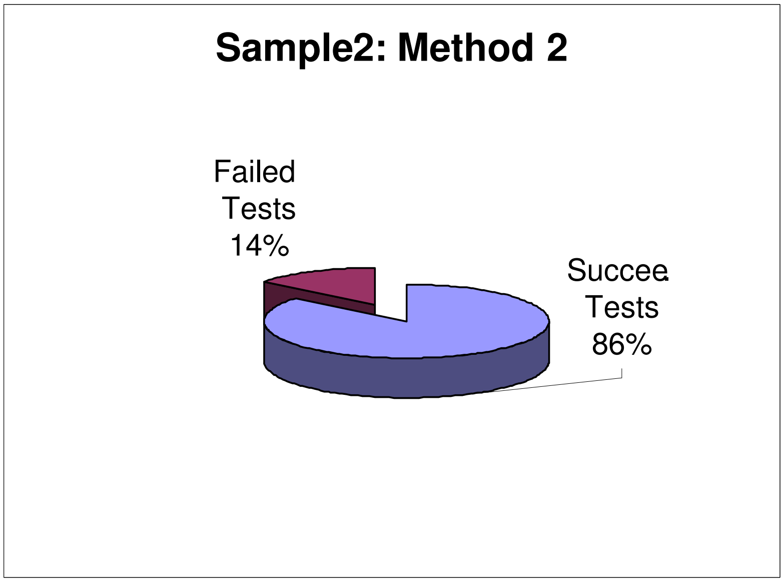
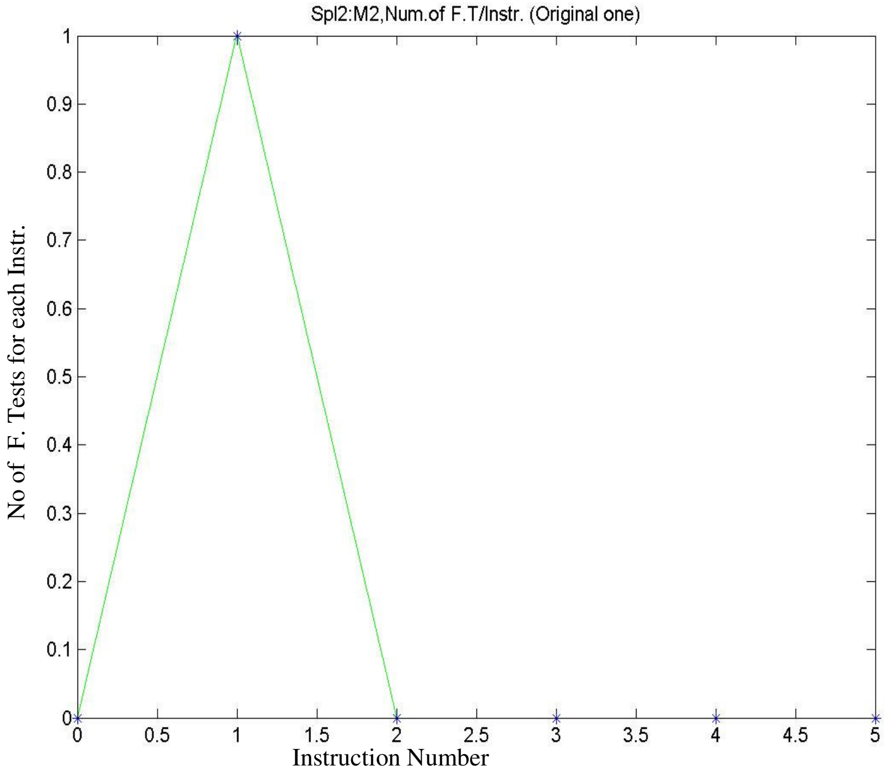
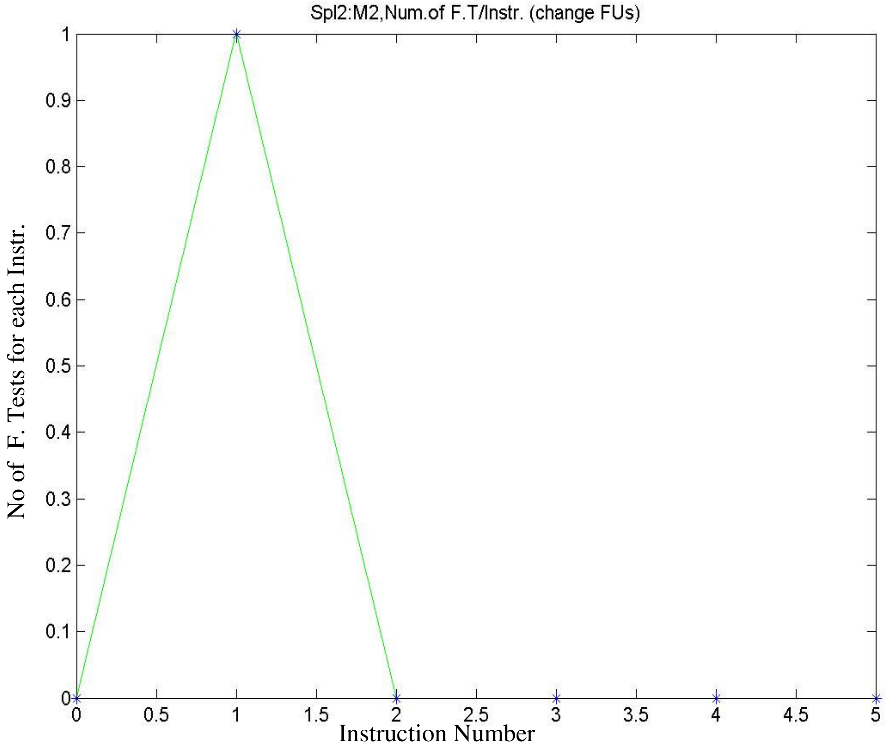


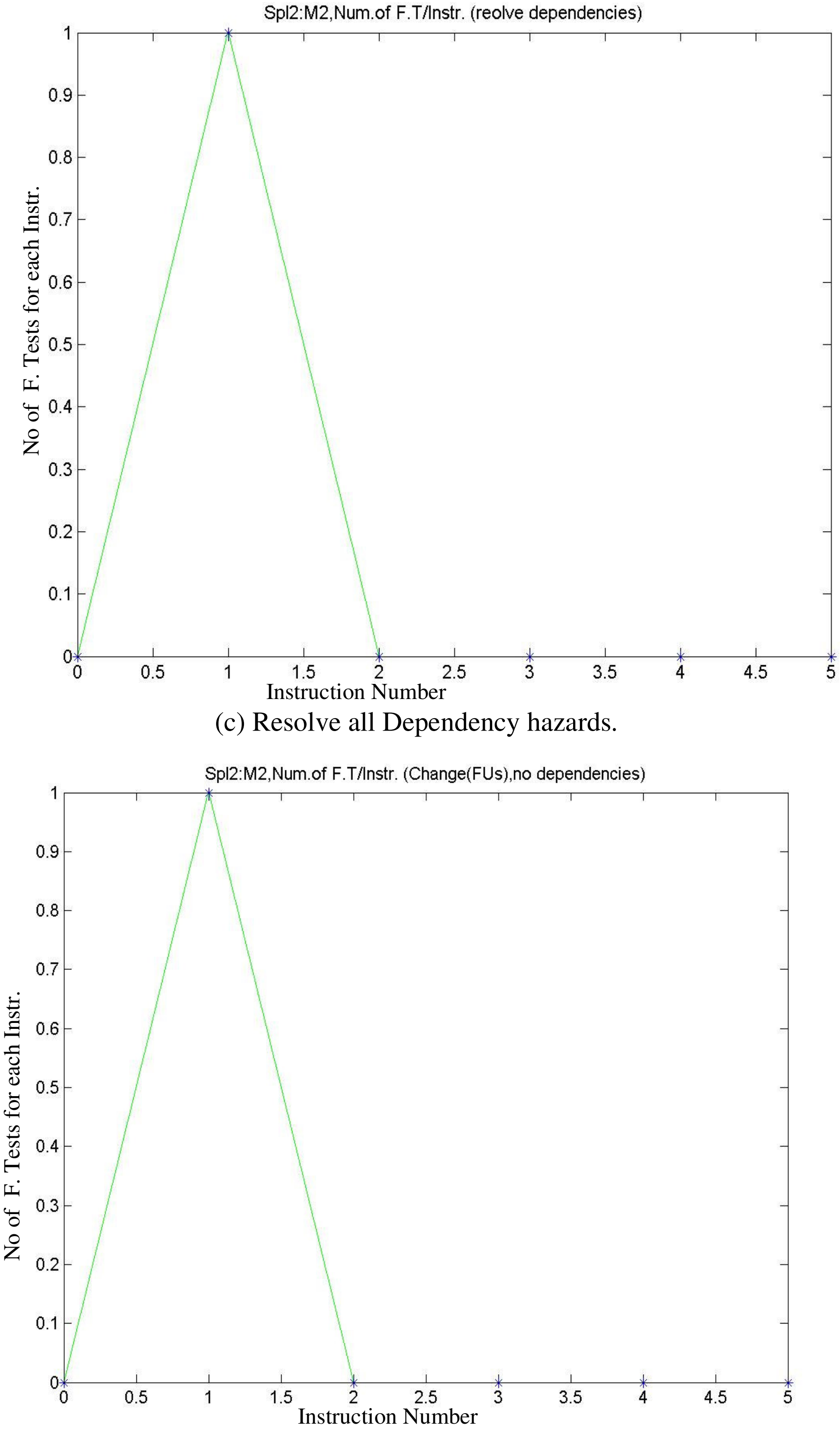
Figure 6.3. Proposed Approach. Sample 2. The number of failed tests and Succeed tests to the Total number of tests.



(a) Original Code.



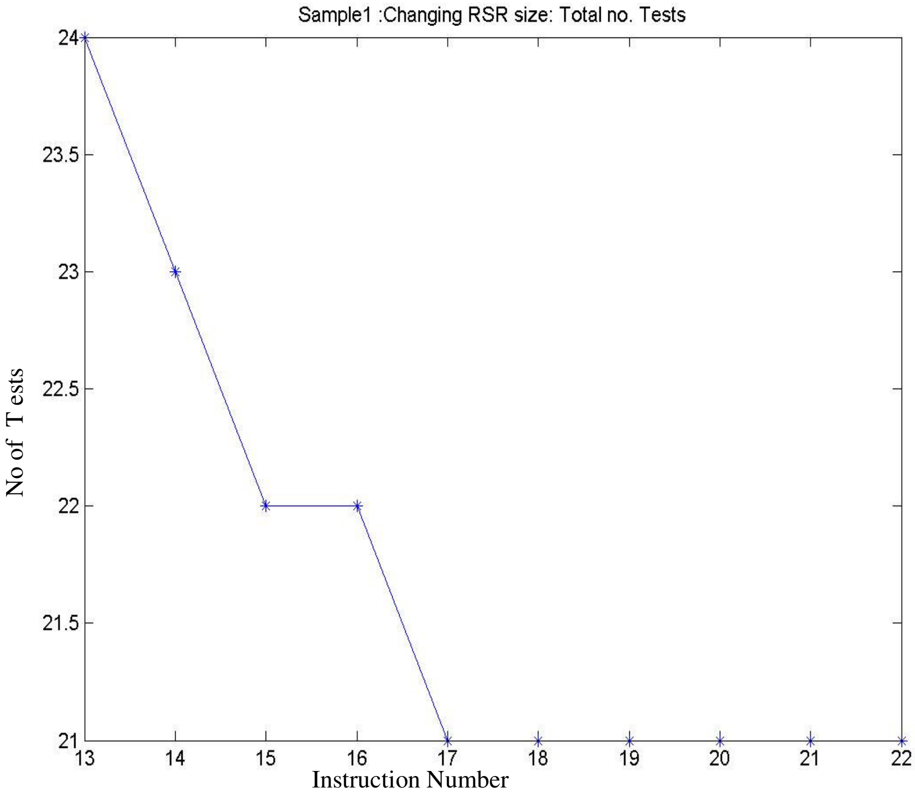
(b) Changing Number of FUs.



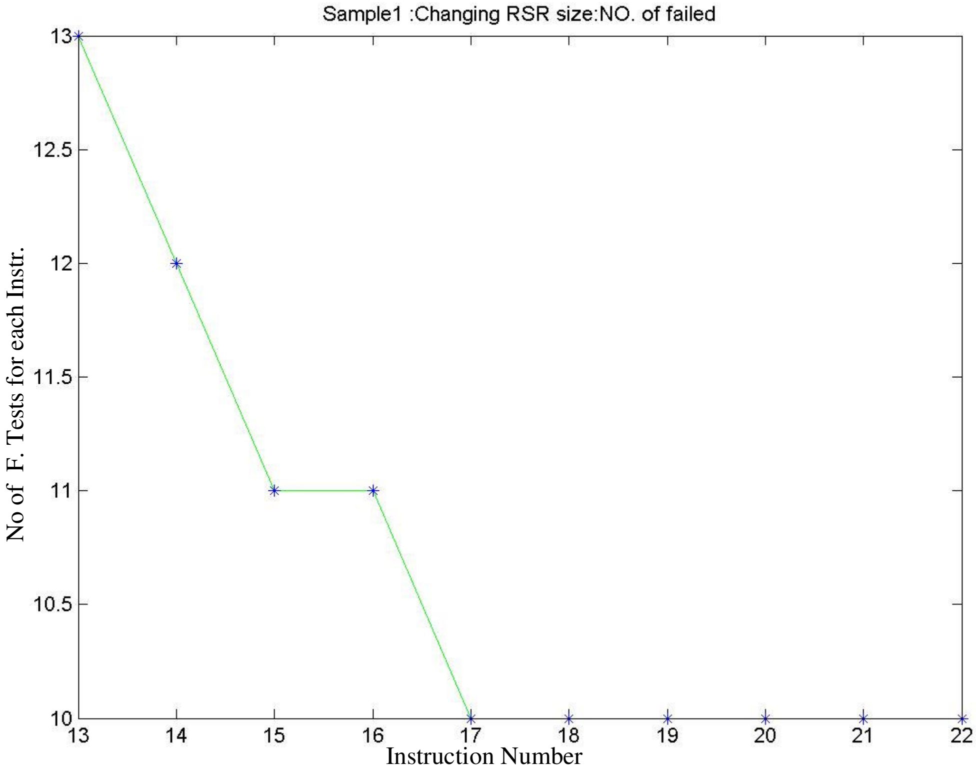
(c) Resolve all Dependency hazards.

(d) Reordering the Instructions to resolve dependencies and changing FUs.

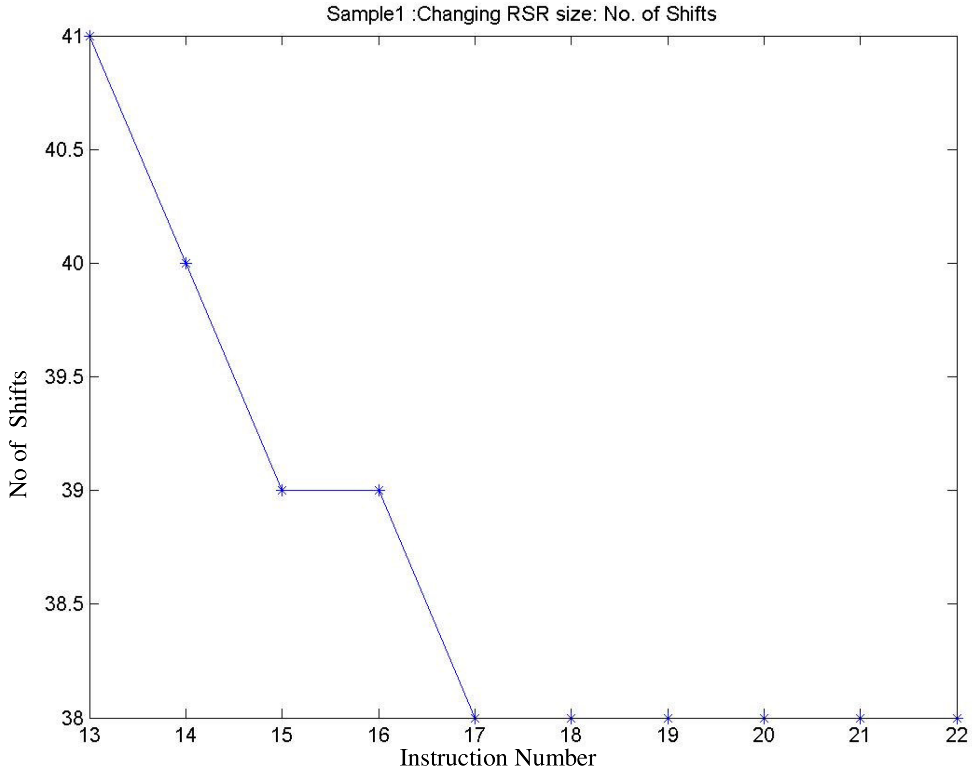
Figure 6.4. The Proposed Approach. Sample 2, the number of failed tests per each instruction



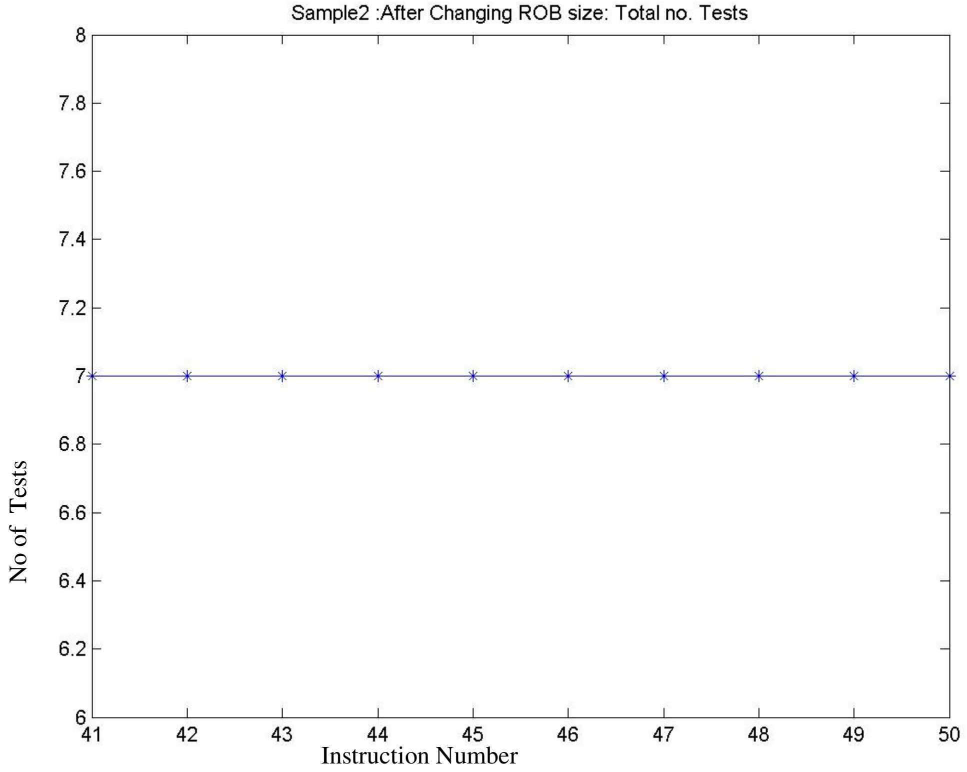
(a) The Total number of tests as the size of RSR changes: Sample 1



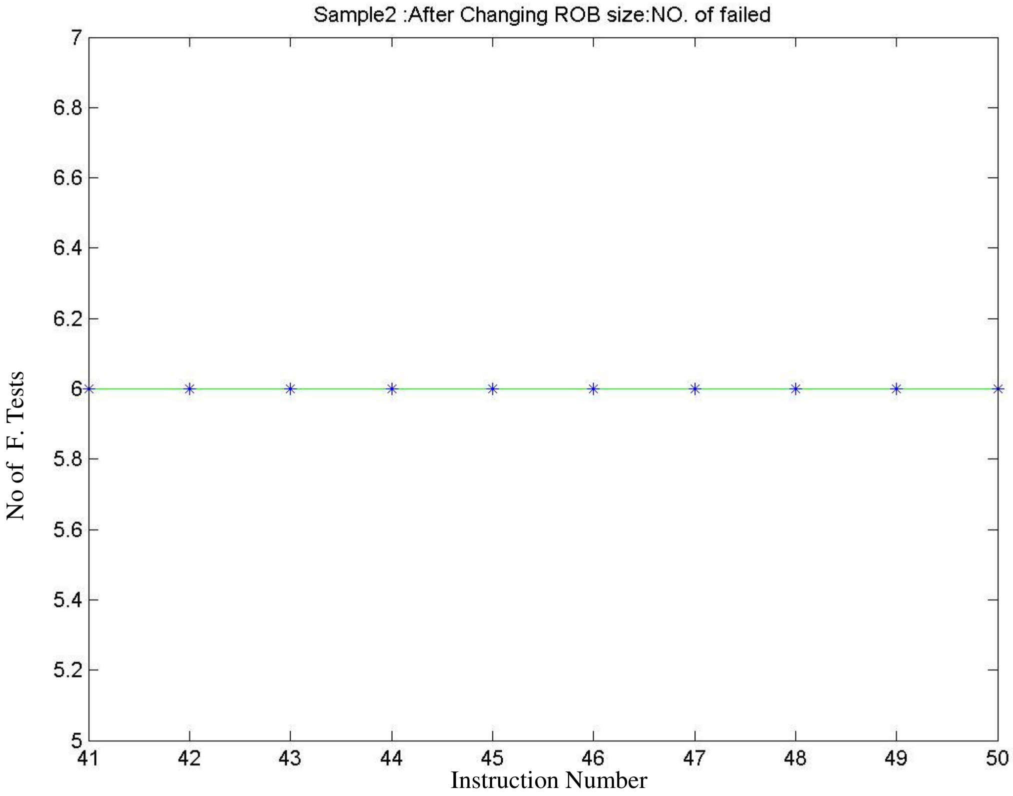
(b) The number of failed tests as the size of RSR changes: Sample 1



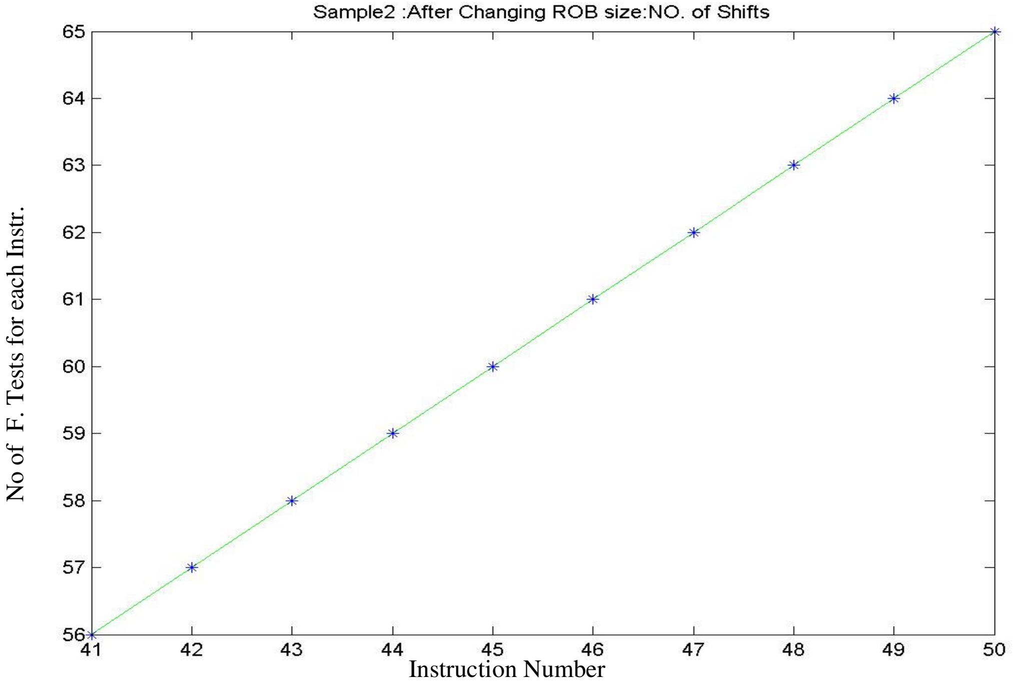
(c) The Total number of shifts as the size of RSR changes: Sample 1



(d) The Total number of tests as the size of RSR changes: Sample 2



(e) The number of failed tests as the size of RSR changes: Sample 2



(f)The Total number of shifts as the size of RSR changes: Sample 2

Figure 6.5. The Proposed Approach. After increasing the size of the RSR.

The precise interrupt is still accurately handled in this compromised approach. The process state can be easily restored, since no instruction stores its state and commits its results out of order. It increases the speed up of the pipeline and improves the parallelism of the whole execution of the program code without reordering the committing of the instructions to keep the In-Order Completion of instructions with less hardware and logic control than that used in the Out of Order Completion schemes.

Instructions are enqueued in the register file; only when their operands are available, in program order the same as that in the Out of Order Completion Scheme. When instruction is successfully de-queued, its results are committed in order. The computed results that are computed out of order is held in the RSR file until previous instructions, finishing later, have updated the register file. Since exceptions are tested at the commit stage, exceptions are handled in program order. When the committing instruction interrupted, all later instructions are squashed.

This scheme has better results than the In Order Completion while keeping the same advantages of being simple and easy to be implemented (Disregarding the complexity of keeping the uncommitted instructions in the RSR, until the prior instructions completed execution). The nature of instructions and dependencies among them affect the results obtained too.

In this approach independent instructions can be executed without waiting for the in-order completion. This can be done if these instructions are safe from exceptions, or any pipelining hazards. This is a costly procedure, compared to the In Order Completion scheme, since it is difficult to find such instructions. Also, if any previous unexecuted (uncommitted) instruction has an exception, these out-of-order executed instructions should be re-executed unless their committed results don't affect the process state.

The improvement of this compromised approach reduces the effect of the In Order Completion approach disadvantages but still not deleting them. Because we have a fixed size of the RSR, long executed instructions will reserve most of the RSR entries preventing others from being issued. Also, if the space is not enough for a particular long executed time instruction then it will wait for other instructions to be freed from the RSR, preventing other instructions also from being issued. Sometimes we need to handle the bad fragmentation in the RSR table, which needs special management. Keeping the in-order committing of instructions limit the performance too.

In this scheme there is no need to resolve WAR. Since no instructions written before a previous read operation has time to commit, so no read operation would obtain an incorrect value. The architectural model proposed guarantees this facility, since the operand registers are read at the time an instruction is issued. Also, there is a single result bus that returns results to the register file. This bus may be reserved at the time an instruction is issued or when an instruction is approaching completion. This is the same as all Out of Order Completion schemes, so there was no need to handle this problem in this scheme or any of the previous out of order schemes.

To solve the RAW dependencies, we added an entry in the RSR file to store *DestRegOld* to get the values of the operands when it is needed to avoid waiting for the value to be committed, which can compensate the bypassing paths used in ROB and HB. When the interrupt is detected, the old values are restored. This useful feature requires more read ports to get the *DestRegOld* value stored in the RSR file, the same as the History Buffer scheme.

6.4 Summary

When we compare our scheme with the other used schemes, we should consider the amount of hardware required in each scheme. In contrast to other schemes, the proposed scheme requires just one table to handle the interrupts precisely. The Out of Order Completion schemes use more files than the proposed scheme. The ROB scheme requires two files: the ROB and the RSR files. The HB scheme requires: the RSR and the HB files, while the FF scheme uses three files: in addition to the FF and RSR files, it requires AF file too.

Our suggested approach necessitates an increase in the number of the read ports the same as that required in the HB scheme and ROB scheme with bypassing; in order to resolve the RAW dependencies. The number of comparators needed in the Reorder Buffer scheme with bypassing is no longer needed. The proposed scheme uses the same mechanism used in the History Buffer and Future File schemes to compensate the use of these comparators.

7. Precise Interrupt Handling in a VLIW Processors

In this chapter we introduce the reorder buffer with future file and history buffer methods after extending them for a Very Long Instruction Width (VLIW) processor. The final scheme, which is discussed, is the Current State Buffer scheme.

7.1 Handling Precise Interrupts in VLIW Processors

In a VLIW processor, interrupt handlers should not modify any of source registers in the interrupted operation because subsequent operations with respect to the original program order that use the same source register would need to be re-issued and re-executed after resumption. The processor cannot re-execute those operations because they may have already updated the processor state.

Debugging requires precise interrupt in any architectural model. Since the performance is less critical during debugging, the debugger can execute the original (unscheduled) code in program order.

The same schemes used in chapter four and chapter five can be used here with some modification to be applied on VLIW processors. Consider the sample code in Table 7.1, scheduled for a 2-issue VLIW machine with functional units as shown. The latencies of the **ADD, SUB** and **DIV** operations are 1, 1 and 2 cycles, respectively. This example illustrates the operation of both the reorder buffer with future file and the history buffer schemes since each buffer has similar operation mechanisms. The **DIV** operation in Mop2 is assumed to cause a trap, so it will not be re-executed after return from the interrupt handler.

Table 7.1. A sample code for a 2-issue VLIW processor.

	ALU Unit	DIV Unit
Mop1	ADD (R2)	NOP
Mop2	ADD (R1)	DIV (R3)
Mop3	ADD (R4)	NOP
Mop4	ADD (R1)	NOP
Mop5	ADD (R4)	NOP

7.2 Reorder Buffer with Future File

The reorder buffer keeps enough information about MultiOps and updates the processor state in scheduled program order. MultiOps in the scheduled code update the processor state sequentially, but the operations of the original program may not complete in original program order. The reorder buffer structure is shown in Table7.2.

Table7.2. Reorder buffer.

	PC	**Exe1…Exen**	**Dest1…Destn**	**Results1…Resultn**	**Excpt1…Excptn**
Head →					
Tail →	. . .	. . .	. . .	. . .	. . .

The *PC* field is the program counter address of a MultiOp. *Exen* is a one-bit location that is set when the nth operation is executed. *Destn* is the destination register number of the nth operation. *Resultn* is the result generated by the nth operation. *Excptn* is the exception conditions generated by the nth operation if one is generated. The reorder buffer is a circular buffer consisting of Head, Tail and OpSequence pointers.

Head and Tail pointers point to the head and tail entries in the buffer, respectively. OpSequence pointer provides an index to the required operation field. The maximum buffer length is the longest latency operation plus one.

At MultiOp issue, the PC of the MultiOp is placed into the *PC* field of the reorder buffer entry pointed by tail pointer, and the destination register numbers are written into the *Dest* fields. When an operation executes without an exception, the result is written into the future file, its *Exe* bit is set and its result field is updated. The future file is a replica of the architectural register file. If an operation caused an exception, the exception status is recorded in the *Excpt* field. At each cycle, the entry pointed to by the head pointer is examined. The results are written into the architectural register file if all *Exe* bits of nonempty operation fields, i.e. excluding NOPs, are set in the entry. If an exception occurs in one of the operations, the whole MultiOp is said to be at the interrupt boundary. Then instruction issue is stopped, and all pipelines are flushed. The architectural register file loads its contents into the future file. The PC value of the MultiOp and the exception bits within the instruction are saved as part of the processor state to identify the source of the exception, and the reorder buffer contents are discarded. The first excepting operation, in left-to-right order in a MultiOp, will be reported if more than one operation in the MultiOp cause exceptions. A store buffer is required to buffer the writes by the store operations into the cache until they are removed from the reorder buffer.

7.3 History Buffer

The history buffer method can be extended in a similar way as the reorder buffer method for VLIW. The operating principle for the extended history buffer method is the same as the original history buffer scheme. It uses a history buffer to keep the old values of registers, rather than the latest values of registers as in the reorder buffer. The buffer structure is the same as the reorder buffer without a future file, except that the **Result** field is replaced with the **Old** field that retains the old value of a register.

7.4 Current-State Buffer

The current-state buffer is a new interrupt-handling scheme for a VLIW processor that signals interrupts immediately using a modest buffer suggested in. It supports both the Less-than-or-Equals and Equals scheduling models. It detects and signals interrupts as soon as they occur. The current-state buffer does not re-execute the operations that have already completed before the interrupt is taken when the program resumes. It relies on compiler scheduling support but requires only simple hardware. The hardware consists of a buffer, called current-state buffer, and a mask register that is shown in Figure 7.1. Each buffer entry consists of a **PC** field for each MultiOp, an **Exe** bit and **Excpt** bits for each operation.

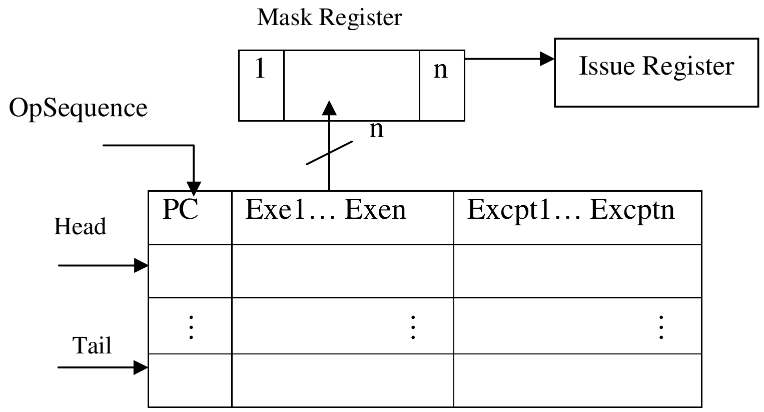


Figure 7.1. The current-state buffer and mask register.

The PC value of a MultiOp to be issued next is put into the **PC** field of the buffer entry pointed to by the tail pointer. Then the tail pointer is incremented. Each operation in the MultiOp is issued with an attached buffer address, an operation identifier, and exception tag. When an operation is executed, the tags attached to each operation access the associated buffer entry. Then its computed result is written into the register file, and its **Exe** bit is set in the current-state buffer if there is no exception. On each cycle, the entry at the head of the buffer is examined. The entry will be discarded by incrementing the head pointer if all **Exe** bits of nonempty operation fields are set. An exception will be signaled immediately to the exception logic if an operation causes an exception.

1. The excepting operation's exception bits are set in the buffer, the issue is stopped and all pipelines are flushed.
2. The processor state, the relevant portion of memory and the current-state buffer contents between the head and tail pointers are saved. The saved buffer contents identify the source of the excepted operation.
3. After return from the interrupt handling routine, previously executed and completed operations are not re-executed, the interrupt handler routine returns by a special return instruction, which switches to a special mode.

In the special mode, the processor, the memory state and the current-state buffer contents are resumed, except the PC register. The buffer contents are examined starting from the head pointer. If all operations in a MultiOp completed execution, that MultiOp is not fetched. However, pipeline bubbles are inserted in order to honor dependence latencies of scheduled operations if there are MultiOps previously issued from the current-state buffer and being still executed in the pipelines. Otherwise, no pipeline bubbles are needed. If one or more operations in a MultiOp are still incomplete, the PC value of the entry is loaded into the PC register and the associated MultiOp is fetched and brought into the issue register. The **mask register** is loaded with the **Exe** bits of the entry, masking the issue register so that previously executed operations are not re-issued. This modified MultiOp is then issued to the functional units. This process continues until the head pointer passes the tail pointer. (This is detected by the buffer control logic.) As soon as this happens, the buffer control logic pops up the PC register, either from the stack or a shadow PC register, and the processor resumes execution from the PC.

7. 5 Compiler Support

Anti-dependence between two operations may cause a problem in the current-state buffer when the sink operation of anti-dependence completes before the source operation of anti-dependence does. If the source operation excepts, the interrupt handler may read an incorrect value of the source register on which the anti-dependence occurs. Such anti-dependence is called an *unsafe* anti-dependence. The compiler can treat unsafe anti-dependencies between operations as if they are flow dependencies. It must guarantee that the destination register of the sink operation is not modified until the source operation completes.

The compiler modifies only unsafe anti-dependencies. Safe anti-dependencies remain in the code. In the pass-scheduling phase of the compiler, unsafe dependencies are converted into flow-dependencies.

However, there is a drawback in treating unsafe anti-dependencies as flow-dependencies. The scheduling times tend to increase because of NOPs that have to be inserted to increase the distance between two anti-Head dependent operations. This could increase the schedule length and degrade the performance, particularly in Equals scheduling model because it is likely that Equals model produces more unsafe anti-dependencies than Less-than-or-Equals model.

7.6 Schemes Experiments

The operation of the reorder buffer with future file and the history buffer for the same example illustrated in table 7.1 is shown in table7.3.

Table7.3. Execution steps in the Reorder and History Buffer

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 0	H →	1	R2	0	0	-	-	-
	T →							

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
		1	R2	1	0	-	-	-
Cycle 1	H →	2	R1	0	0	R3	0	0
	T →							

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 2	T →	1	R2	1	0	-	-	-
	H →	2	R1	1	0	R3	0	0
		3	R4	0	0	-	-	-

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 3		4	R1	0	0	-	-	-
	T H →	2	R1	1	0	R3	0	1
		3	R4	1	0	-	-	-

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 4+n	H →	2	R1	0	0	-	-	-
	T →							

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 5+n		2	R1	1	0	-	-	-
	H →	3	R4	0	0	-	-	-
	T →							

		PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
Cycle 6+n	T →	2	R1	1	0	-	-	-
		3	R4	1	0	-	-	-
	H →	4	R1	0	0	-	-	-

Cycle 7+n

	PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
	2	R1	1	0	-	-	-
H →	3	R4	1	0	-	-	-
T →	4	R1	0	0	-	-	-

Cycle 8+n

	PC	Dest1	Exe1	Exp1	Dest2	Exe2	Exp2
	2	R1	1	0	-	-	-
H →	3	R4	1	0	-	-	-
T	4	R1	1	0	-	-	-

The operation of the current state buffer for the same example is the same as that shown in table7.3 except after the exception is detected. Table7.4 shows the results after cycle n+4.

Table7.4. Execution steps in the current-state buffer

Cycle 4+n

	PC	Exe1	Exp1	Exe2	Exp2
H →	4	0	0	-	-
T →					

Cycle 5+n

	PC	Exe1	Exp1	Exe2	Exp2
	4	1	0	-	-
H →	5	0	0	-	-
T →					

Cycle 6+n

	PC	Exe1	Exp1	Exe2	Exp2
	4	1	0	-	-
H →	5	1	0	-	-
T →					

Figure 7.2 shows the comparison between the three previous schemes performed on the same sample of code.

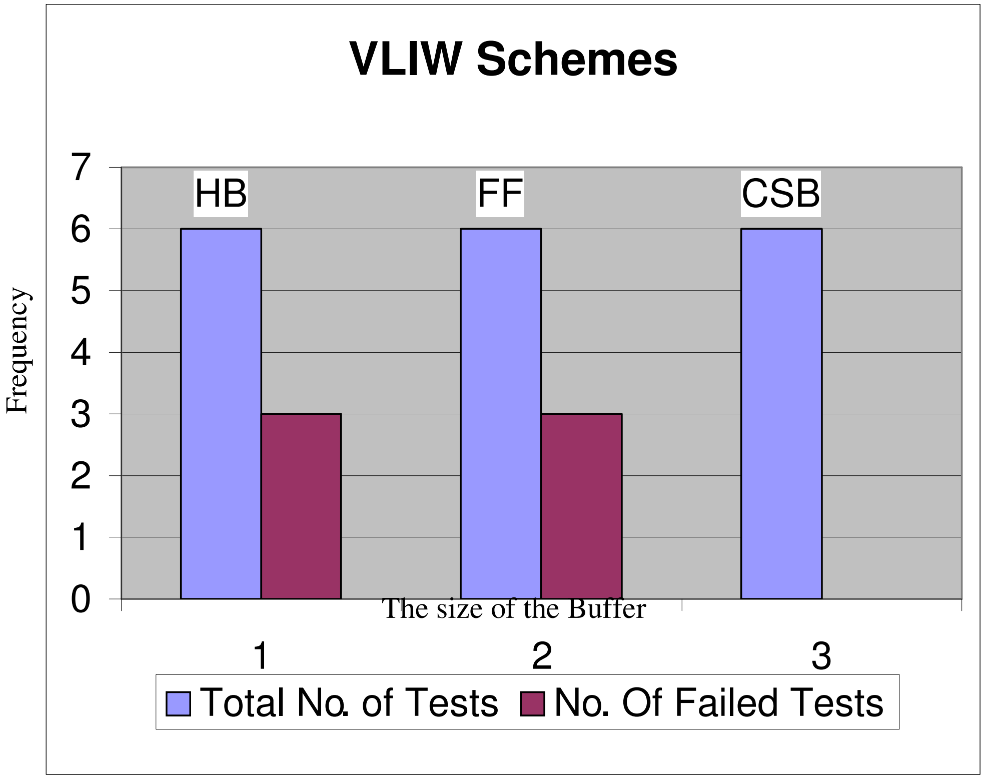


Figure 7.2. The comparison diagram between the three schemes

7.7 Summary

In this chapter, three schemes for handling precise interrupts in VLIW processors were implemented and analyzed. The algorithms written for implementing these schemes give the same results obtained in. The Current State Buffer gives the best results over all the other two schemes.

8. Conclusion and Future Work

In the thesis, we have implemented and investigated five schemes for handling the precise interrupts in pipelining systems and a new scheme is suggested. Another study is done for the VLIW processors. Obtained results show the following considerations and conclusions.

8.1 Preface

Some observations are appropriate after introducing the different schemes. The average time to produce the finished results is faster if each stage is specialized than if one general-purpose stage does it all, even if the total time to produce each result, start to finish, is lengthened. There is an implication that each stage takes the same amount of time, and as a result of this constant time per stage, we get results on regular basis (Henry *et al*., 1989). We used specialized stages in all schemes discussed in the thesis, except the suggested approach.

The second important point is that there are some practical limitations on practical depth of a pipeline, which arise from:

1. *Pipeline latency.* The fact that the execution time of each instruction does not decrease, adds limitations on pipeline depth;
2. *Imbalance among pipeline stages.* Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage;

3. *Pipeline overhead.* Pipeline overhead arises from the combination of *pipeline register delay* (setup time plus propagation delay) and *clock skew.*

So, once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, since there is no time left in the cycle for useful work.

Another remarkable notice is that, the change of the FUs sometimes becomes more costly than the better performance obtained. When the execution stage is extended, the pipeline is extended too. In addition to the longer (and possibly more frequent) stalls, the longer pipeline requires additional forwarding hardware. It also requires more complex hazard detection to find dependencies in the additional stages. The major benefit to a longer pipeline is that each stage may be shorter than the case where there is a shorter pipeline. This means that the clock cycle can be shorter, allowing more instructions to be issued in a fixed time. Of course, the added stalls might put away this benefit, but the hope is that at least some speedup will be left.

8.2 The Comparison Between Schemes

First of all, we compare the different schemes used to handle precise interrupts in scalar processors, with the architecture discussed fully in chapter 4.

Before comparing the different mechanisms used throughout the thesis, we should take into consideration the fact that the performance gained by adding a new mechanism has to be balanced against the amount of hardware required to implement the mechanism, and that implementation's potential impact on the cycle-time of the processor. Both of these are extremely difficult to quantify without an actual implementation, and even with such an implementation it is sometimes difficult to isolate the impact caused exclusively by the new mechanism, or by its interactions with other processor elements.

We built our own simulators to verify the results obtained since we did not find any ready used simulators. The only existing simulators, which concerned in pipelining did not handle interrupts at all and just mentioned that it is another heavy work to be done. Even though, those uncompleted pipelined simulators, were built with the full corporation between several universities in the United States of America and the IBM Company.

So, the comparisons of our scheme with other schemes that implement precise interrupts were more qualitative than quantitative.

In the In Order Completion, the in-order state is always available in the register file, thus it is restored immediately (unlike the other schemes). But this scheme suffers from stalling instructions at the issue stage, the decrease in performance and long program execution time.

The key features of the Reorder Buffer, that can be stated after the discussion and analysis of the results in chapter 5, are that: The Reorder Buffer requires complex bypass paths, which may increase register read latency. If we use it without bypassing, ROB not much better than In Order Completion. With a relative small number of ROB entries, implementing precise interrupts causes relatively little performance losses. The reason of why some performance loss remains may be that a pending store will cause all further load or store to stall at issue. Finally, the Reorder Buffer scheme requires rollback logic to put old results back in register file on exceptions.

In the History Buffer scheme, the in-order state is not always available in the register file, thus it requires rollback logic to put old results back in register file on exceptions. This is because instructions complete out-of-order, while the register file updated immediately, the hardware cleans-up on exception, so it is used to undo instructions. History Buffer has fewer interconnections than bypassing ROB, but requires another read ports to register file with a total of three. The final feature is that bypassing is slightly more complex here than that in Reorder Buffer.

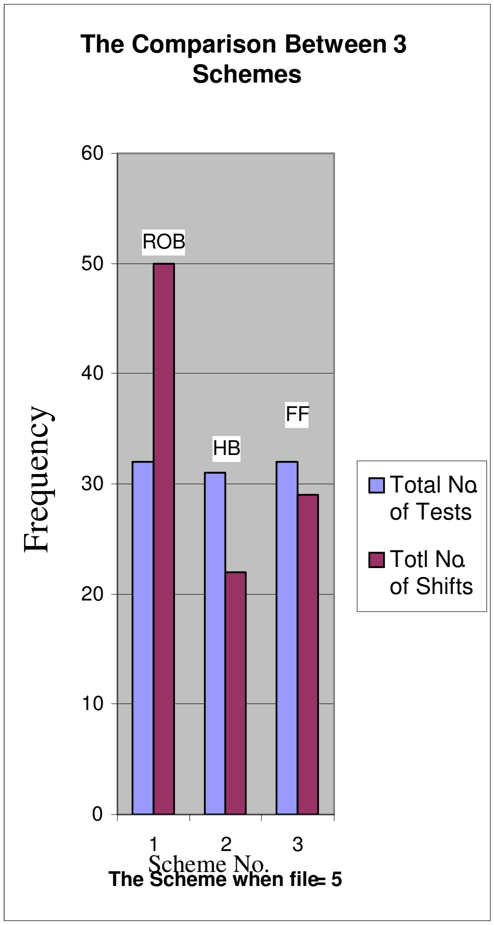
The Future File scheme features can be summarized as being fast, need no bypass paths, with simple rollback, but duplicates of register files are needed which are, the Future File and the Architecture File. The future file removes the need for the associative lookup of the reorder buffer while still not incurring cost in saving state. In a system with a future file, the recovery from an exception is simple since it is only necessary to drain the reorder buffer and clear a set of flags (David, 1997).

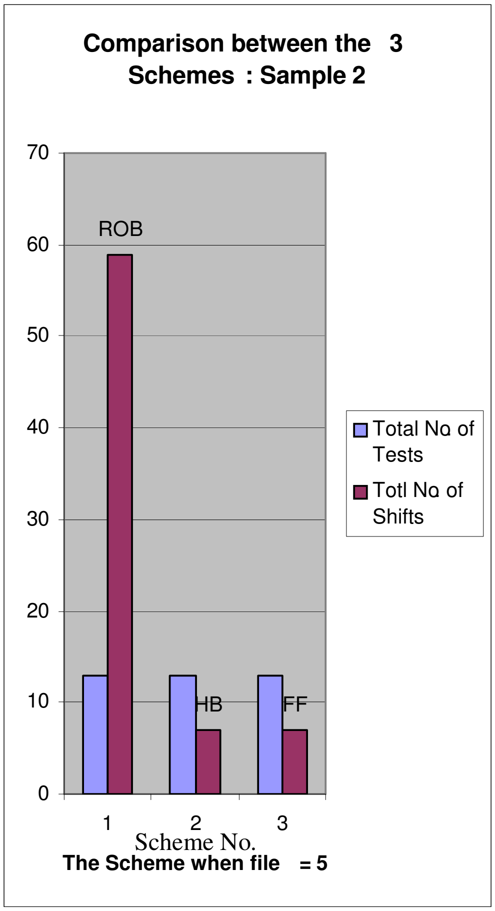
The suggested scheme is a compromised technique between the features of Out of Order Completion and that of the In Order Completion schemes. Although this technique reduces the effect of the In Order Completion approach disadvantages, it also requires the committing of instructions only in order, the same as the Out of Order Completion schemes. Another limitation is the size of the RSR, which may force long instructions to wait until the required space is available. In contrast to other schemes, the compromised one requires less hardware and control logic to handle the interrupts precisely than all other schemes.

The following points can summarize the above comparison

- The reorder buffer must complete all outstanding register updates that come before the faulting instruction prior to being invalidated. This could take many cycles, especially since some of the instructions in question will not have completed.
- There is no way around this when using a future file, as it holds only the most recent values after the exception.

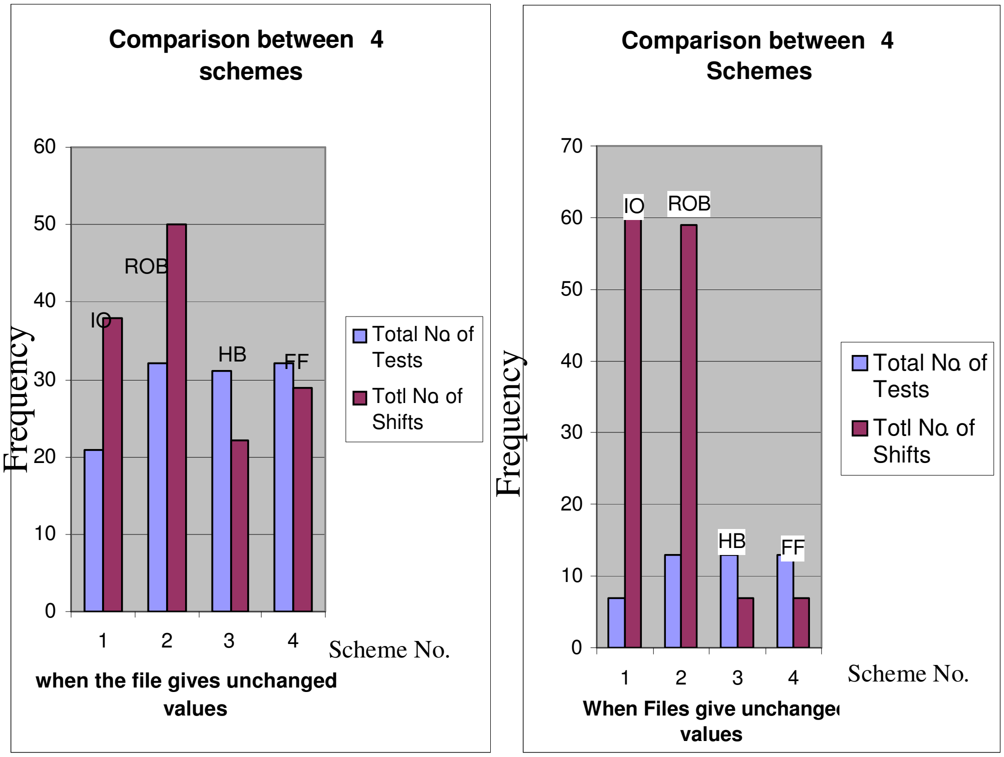
When a future file is not used, we can provide the reorder buffer with the ability to invalidate all entries that were allocated after the instruction. Since the most recent value is provided by the associative lookup, we don't need to wait on outstanding, valid instructions.





(a) The Total number of shifts/Tests in the three schemes when the size = 5: Sample 1

(b) The Total number of shifts/Tests in the three schemes when the size = 5: Sample 2



(c) The Total number of shifts/Tests in the three schemes when values become stable: Sample 1

(d) The Total number of shifts/Tests in the three schemes when values become stable: Sample 2

Figure 8.1. Comparison between the schemes: "Compromised Approach (IO)", ROB, HB, and FF schemes.

Figure 8.1, shows the comparison between the four main schemes used for the scalar processor mentioned in chapter four. The first graphs (a, b) restricted when the size of the buffer used in the algorithm is 5 at which the results become steady in all these schemes. Since the minimum size of the RSR should be at least the same as the largest pipeline stage, the comparison is repeated but now with a size of the buffer equal to the value where the results can not be changed even if the size is increased more.

As shown from the figure, the better improvement regarding the total number of the tests is obtained in the case of the suggested scheme. The total number of shifts was the maximum in this scheme, this can be understood when we remember that the scheme uses only one buffer, and every time an instruction is issued multiple shifts are done, especially when trying to get empty places to this newly issued instruction.

Finally, as shown from the previous analysis in this chapter and the previous ones, the suggested technique gives the best results. The Reorder Buffer technique is a good one but its drawbacks can be solved easily in the History Buffer. The advantages of he HB are used in the proposed technique to solve the ROB drawbacks.

An important improvement to this method can be accomplished by using an intelligent compiler, to do the proper re-arrangements and optimization to reorder instructions to get the minimum dependencies among instructions.

Secondly, the thesis discusses the handling of precise interrupts in VLIW processors, as shown in chapter seven, the Current State Buffer scheme gives better results than all other schemes used.

8.3 Extensions

8.3.1 Handling other State Values:

Most architectures have state information other than what we have assumed in the architectural model in the thesis. Such state information is the page and segment table, interrupt mask conditions, etc.

In architectures that use condition codes, the condition codes are state information. Extensions to Reorder Buffer, History Buffer, and the Future File can be used to handle such information state.

8.3.2 Linear Pipeline Structure

An alternative to the parallel functional unit organizations used in the discussion previously is the linear pipeline organization. Linear pipelines provide a more natural implementation of the register-storage architectures like the IBM 370 (James et al., 1988). In general, reordering instructions after execution is not as significant as issue in such organizations because it is natural for instructions to stay in order as they pass through the pipe. Even if they finish early in the pipeline, they proceed to the end where exceptions are checked before modifying the process state. Hence, the pipeline itself acts as a sort of reorder buffer (James *et al.*, 1988).

Linear pipelines often have several bypass paths connecting intermediate pipeline stages. A complete set of bypasses is typically not used, rather there is some critical subset selected to maximize performance while keeping control complexity manageable (James *et al.*, 1988).

8.3.3 Vectors

Implementing precise interrupts in pipelined vector architecture is more difficult than for a scalar architecture. When considering precise interrupts with respect to vector instructions, preciseness must be carefully defined. Unlike the scalar instructions described thus far, vector instructions do not produce a single result and change the system state as they complete. Rather, they produce a series of results that change the system state over the course of many clock periods. The sequential architectural model, as applied to vectors, requires that one vector instruction complete its last result before the next begins producing results (James *et al*., 1988).

There are two primary classes of vector architectures: those with vector registers, and those with memory-to-memory vector operations. For vector register architectures, we extend our earlier methods for maintaining scalar register precisely (James *et al*., 1988).

Various methods for implementing precise interrupts can be extended for vector registers, but the cost is a doubling of the number of hardware registers plus some additional control hardware to keep track of the “current” pointers (James *et al*., 1988).

8.3.4 In-Line Interrupt Handling

To improve the performance of the handling of interrupts and to get better results, we can use any of the two methods of in-lining the interrupt handler within the reorder buffer as mentioned in (Amer *et al.,* 2000). Both of the schemes exploit the property of a reorder buffer: instructions are brought in at the tail, and retired from the head. If there is enough room between the head and the tail for the interrupt handler to fit, we essentially inline the interrupt by either inserting the handler before the existing user-instructions, or after the existing user-instructions. Inserting the handler routine instructions after the user-instructions, the *append* scheme*,* is similar to the way that a branch instruction is handled: the PC is redirected when a branch is predicted taken, similarly in this scheme, the PC is redirected when a TLB miss is encountered. Inserting the handler instructions before the user-instructions, the *prepend* scheme, uses the properties of the head and tail pointers and inserts the handler instructions before the user-instructions. (Amer *et al*., 2000)

The two schemes differ in their implementations, the first scheme being easier to build into existing hardware. To represent our schemes in the following diagrams, we are assuming a 16-entry reorder buffer, a four-instruction interrupt handler, and the ability to fetch, en-queue, and retire two instructions at a time. To simplify the discussion, we assume that instruction state is held in the ROB entry, as opposed to being spread out across ROB and reservation-station entries. A detailed description of the two in-lining schemes follows:

1. ***Append in-line mode***: The hardware responds by checking to see if the handler would fit into the available space. Assuming the handler is four instructions long, it would fit in the available space. The hardware turns off user-instruction fetch, sets the processor mode to INLINE, and begins fetching the first two handler instructions. These have been en-queued into the ROB at the tail pointer as usual. For example, the Alpha's TLB-write instructions modify the TLB state once they have finished execution and not at instruction-commit time. In many cases, this does not represent an inconsistency, as the state modified by such handler instructions is typically trans- parent to the application; for example, the TLB contents are merely a hint for better address translation performance.
2. ***Prepend in-line mode***: The hardware checks to see if it has enough space, and if it does, it saves the head and tail pointer into temporary registers and moves the head and tail pointer to number of instructions before the current head. At this point the processor is put in INLINE mode, the PC is redirected to the first instruction of the handler, and the first two instructions are fetched into the pipe. Eventually, when the last handler instruction fills the TLB, the flag of the excepted instruction can be removed and the exceptional instruction may re-access the TLB. This implementation effectively does out-of-order committing of

handler instructions, but again, since the state modified by such instructions is transparent to the application, there is no harm in doing so. (Amer *et al*., 2000)

The two schemes presented differ slightly in the additional hardware needed to incorporate them into existing high performance processors. Both the schemes require additional hardware to determine if there are enough reorder buffer entries available to fit the handler code. Since the *prepend* scheme exploits the properties of the head and tail pointers, additional registers are required to save the old values of the head and tail pointers. As we shall see later, incorporating these additional registers will allow for the *prepend* scheme to out-perform the *append* scheme by 20-30%. There are a few implementation issues concerning the in lining of interrupt handlers. They include the following:

1. The hardware knows the handler routine length.
2. There should be a privilege bit per ROB entry.
3. Hardware needs to signal the exceptional instruction when the handler is finished.
4. After loading the handler, the “return from interrupt” instruction must be killed, and fetching resumes at **nextPC**, which is unrelated to **exceptionalPC**.
5. In-lined handler instructions shouldn’t affect the state of user registers.
6. The hardware might need to know the handler’s register requirements.
7. Branch mispredictions in user code should not flush handler instructions. (Amer *et al.,* 2000)

8.3.5 Register File Extension

When using the Reorder Buffer we can extend the register file used. Here, in the Register File Extensions, the register file holds the values of the specification registers of the machine. We still denote the set of registers by R. We denote the value of the register *r* 2 R during cycle *T* by *R*[*r*] *T*: *data*. We assume that all registers have a common width. We denote the set of possible values of a register by W (*R*) (Daniel, 2001).

The register file is extended with a *producer table*. The producer table records which instruction in the machine writes its results to a given register. For that purpose, the producer table contains two data items for each register. The first is a valid bit. We denote the value of the valid bit of register *r* during cycle *T* with *R*[*r*] *T*: *valid*. If it is set, there is no instruction currently executing with the register as destination. If it is not set, there is such an instruction. In this case, the second item, a reorder buffer tag, points to the last instruction with the register as destination. We denote the value of this tag by *R*[*r*] *T*: *tag* (Daniel, 2001).

The reservation stations act as queue for the instructions and their source operands. We give each reservation station a number. We denote the values in reservation station number *rs* during cycle *T* by *RS* [*rs*] *T*. Each reservation has a full bit *RS* [*rs*]: *full*. It indicates that the reservation station is in use. In addition to that, we store the tag of the instruction in the reservation station in *RS* [*rs*]: *tag* (Daniel, 2001).

We support instructions with an arbitrary number of source operands. Let *x* denote the number of a source operand. For each source operand, we store a valid bit *RS* [*rs*]:*op* [*x*]: *valid*. If the bit is set, the value of the operand is stored in *RS* [*rs*]:*op* [*x*]: *data*. If it is not set, we store the tag of the instruction producing the value in *RS* [*rs*]:*op* [*x*]: *tag* (Daniel, 2001).

8.4 Future Work

There are many other interesting issues related to implementing precise interrupts. The extensions presented in this chapter can be implemented and tested. The handling of memory faults can be studied also.

8.5 Summary

Different schemes used to handle precise interrupt in both scalar and VLIW processors. The comparisons of our scheme with other schemes that implement precise interrupts were more qualitative than quantitative.

The suggested scheme is a compromised technique between the features of Out of Order Completion and that of the In Order Completion schemes. Although the Reorder Buffer technique is a good one, its drawbacks can be solved easily by using the History Buffer. The advantages of the HB scheme are used in the proposed technique.

If we consider the amount of hardware required in each scheme, we can find that the proposed scheme requires just one table to handle the interrupts precisely. The Out of Order Completion schemes use more files than the proposed scheme. The ROB scheme requires two files: the ROB and the RSR files. The HB scheme requires: the RSR and the HB files, while the FF scheme uses three files: in addition to the FF and RSR files, it requires AF file too.

The proposed scheme tries to reduce the effect of the In Order Completion approach disadvantages. It requires committing of instructions the same as the Out of Order Completion. The limitation on the size of the RSR forces long instructions to wait until the required space is available. It necessitates an increase in the number of the read ports the same as that required in the HB scheme and ROB scheme with bypassing; in order to resolve the RAW dependencies. The number of comparators needed in the Reorder Buffer scheme with bypassing is no longer needed. The proposed scheme uses the same mechanism used in the History Buffer and Future File schemes to compensate the use of these comparators.

REFERENCES

Amer, Jaleel., and Bruce, Jacob. 2000. *Improving the Precise Interrupt Mechanism of Software-Managed TLB Miss Handlers***,** Electrical and Computer Engineering University of Maryland at College Park.

Barry, Wilkinson. 1996. *Computer Architecture design and performance*, second edition, Prentice Hall Europe.

Daniel, Kroning. 2001. *Formal Verification of Pipelined Microprocessors*, Ph.D. Thesis, University of Saarland,

Daniel, Kroening., Silvia M. Mueller., and Wolfgang J. Paul. 2001. *Rigorous Correctness Proof of Tomasulo Scheduler Supporting Precise Interrupts*, Dept. 14: Computer Science, University of Saarland,

David, Gilbert. 1997. *Dependency and Exception handling in an Asynchronous microprocessor*, Ph.D. Thesis, University of Manchester, Manchester, UK.

Henry, Levy., Richard, Eckhouse., Jr. 1989. *Computer Programming and Architecture – VAX*, 2nd Edition, Library of Congress cataloging-in-publication data.

James, Smith., Gurindar, Sohi. Aug. 20, 1995. *The Microarchitecture of Superscalar Processors*.

Mayan, Moudgill. 1996. *Precise Interrupts*, 0772-1732/96, IEEE., IBM T.J. Watson Research center, Stamatis Vassiliadis. Delft University of Technology,

Schmalz, M.S. 1998. *Organization of Computer Systems*, Francisco, CA: Morgan Kaufman.

V.C., Pierguido., Caironi., Lorenzo, Mezzalira., Sami, Mariagiovanna. 1996. *Context Reorder Buffer: an Architectural Support for Real-Time Processing on RISC Architectures*, Politecnico di Milano, Proceedings of the 8th Euromicro Workshop on Real-Time Systems.

Sang-Joon, Nam., In-Cheol, Park., and Chong-Min, Kyung. March 1999. *Fast Precise Interrupt Handling Without Associative Searching in Multiple Out-Of-Order Issue Processors*, IEICE Trans. INF. & SYST. Vol. E82-D, No.3.

UMD-SCA-2000-02 ENEE 446: Digital Computer Design. Fall 2000. *An Out-of-Order RiSC-16- Tomasulo + Reorder Buffer = Interruptible Out-of-Order*, ENEE 446: Digital Computer Design. Prof. Bruce Jacob.

Wen-mei, Hwu. 1999. *Computer Microarchitecture: Hardware and Software*. IMPACT UIUC ECE 411 and NTU CA 718-Q: ©All Rights Reserved.

معالجة الاعتراضات المضبوطة في أنظمة (Pipeline)

إعداد

أمل " محمد بركات" محمود الدويك

المشرف

الدكتور سامي ابراهيم سرحان

ملخص

ان معالجة الإعتراضات المضبوطة Precise Interrupts في انظمة Pipelining تعتبر من ابرز الابحاث واهمها في مجال التطوير وتحسين الكفاءة في علم عمارة الحاسوب بشكل خاص وتطوير الاجزاء المادية في الحواسيب بشكل عام. إن معظم الأجهزة المتداولة حالياً تدعم التنفيذ العشوائي وغير المرتب للتعليمات مما يؤدي الى تنفيذ بعض التعليمات قبل تلك التي تسبقها مسببة تغييرًا في الحالة التسلسلية للعملية المعالجة وبالتالي ظهور الاعتراضات المضبوطة.

تهدف الرسالة الى معالجة مثل هذه الحالات حال وقوعها أثناء تنفيذ التعليمات المختلفة. وقد تم اقتراح طريقة جديدة للمعالجة وهي عبارة عن دمج لخصائص الطريقتين الاساسيتين وهما الانتهاء المرتب للتعليمات In Order Completion والانهاء العشوائي للتعليمات Out of Order Completion.

بعد دراسة وتحليل الطريقة المقترحة ومقارنتها بالطرق المستخدمة حاليا تبين ان الطريقة المقترحة تتصف بمميزات وفعالية تفوق الطرق الاخرى المقارنَة.