HARDWARE/SOFTWARE COST ANALYSIS OF INTERRUPT PROCESSING STRATEGIES

By

LOAI E. GARALNABI

Bachelor of Computer Science

University of Khartoum

Khartoum, Sudan

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 1997

HARDWARE/SOFTWARE COST ANALYSIS OF
INTERRUPT PROCESSING STRATEGIES

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

J Chandler

Blayne E. Mayfield

Thomas C. Collins

Dean of the Graduate College

PREFACE

The purpose of this work was to study and analyze interrupt processing strategies from a cost point of view. The cost referred to is the architectural cost of implementing a particular interrupt processing strategy. The scope of this study included five strategies. All the strategies under investigation were originally designed to make it possible for pipelined processors to support precise interrupts. To analyze the cost of each strategy, its design and implementation was carefully studied. Based on that it was possible to determine or closely estimate the amount of hardware, and the complexity of software needed to implement each strategy.

On pipelined processors, interrupt processing can be broken down into six phases. Some phases such as detecting the interrupt, running the interrupt handler, and resuming the interrupted process (for precise interrupts), are common for all strategies. The strategies differ in whether they finish pending the instructions once an interrupt has occurred, or they just flush the pipeline. Also they differ in whether they undo state changes or maintain a precise state at all times. Hardware dominates the cost of many of the strategies, except for one, namely Checkpoint Repair, for which the cost varies from being mainly composed of hardware costs to being mainly composed of software costs according to the strategy's implementation.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Mansur Samadzadeh for his guidance and encouragement. I would like also to thank Drs. Mayfield and Chandler for agreeing to serve on my thesis committee.

My deepest appreciation goes to my family and friends for their continuous unselfish support throughout the course of my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I
INTRODUCTION

Interrupts are the events, other than branches, that change (discontinue temporarily) the normal flow of execution [Hennessy and Patterson 90] [Cragon 96] [Hwang 84]. Interrupts are essential to modern computer systems. By utilizing interrupts, the processor(s) and the operating system can interact and react to different events.

Interrupts were first introduced on the UNIVAC I to signal arithmetic overflow [Hennessy and Patterson 90]. Since then, as processors grew more complex, interrupts' duties increased. Some of these duties include handling: I/O device requests, page faults, segmentation faults, and hardware malfunctions.

Although interrupts are essential to the correct functioning of a system, relatively little has been written about them [Walker 92]. This, and the numerous duties assumed by interrupts, has led to a confusing situation of no one acceptable terminology used for interrupts. Interrupt terminology relies mainly on computer manufacturers' creativity, with many using different names to mean the same thing [Hennessy and Patterson 90] [Cragon 96] [Walker 92].

There are three basic types of interrupts: External, Internal, and Software, as described below.

- External interrupts are interrupts that occur outside the processor. With the exception of processor malfunctions, all hardware malfunction interrupts are external.
- Internal interrupts are interrupts inside the processor. They are usually caused by the running process.
- Software interrupts are generated by the running process. They usually occur in predefined points during execution, and will occur at the same location every time the process is run.

Figure 1.1 names some of the existing interrupts classified according to these three categories.

I/O device request	External
page fault	Internal
illegal instruction	Internal
requesting operating system service	Software
traps	Software
power failure	External

Figure 1.1 Interrupts classified using three categories

Interrupts are essential to modern computer systems, but extremely high frequency of occurrence of interrupts is undesirable. Interrupts have to be processed, which implies time overhead. Interrupt processing is a time-consuming process [Torng and Day 93]. Therefore, special attention must be paid to their design and efficient implementation. Interrupts are low level events, thus, the design of an interrupt processing strategy involves modifications to both the hardware and, consequently, the software implementations of a system. The cost of these modifications tends to be high. This study intends to shed some light on the costs of various interrupt processing strategies in terms of their hardware/software decomposition.

CHAPTER II
INTERRUPT PROCESSING

The general idea of interrupt processing is that when an interrupt occurs, control is passed to a code segment known as the interrupt handler. The interrupt handler performs the appropriate action and then possibly returns control to the interrupted process [Moudgill and Vassiliadis 96]. This is a simple view of the interrupt processing process. To allow an interrupted process to resume, the full state of the machine must be saved before an interrupt can be processed [Hennessy and Patterson 90]. Subsequently, the saved state must be restored before the interrupted process can resume execution.

2.1 Precise and Imprecise Interrupts

There is much more involved in resuming an interrupted process than what was mentioned in the above paragraph. For an interrupted process to be restarted, the exact point at which the interrupt occurred, i.e., the instruction at which the interrupt occurred, has to be known. For a process to resume execution correctly, the following conditions have to be satisfied [Cragon 96] [Walker and Cragon 95] [Smith and Pleszkun 88] [Moudgill and Vassiliadis 96].

- All instructions issued before the interrupted instruction have finished execution and modified the process state.
- All instructions issued after the interrupted instruction are unexecuted or have not modified the process state.
- The saved program counter points to the interrupted instruction.

The above conditions ensure that the process state at the time of the interrupt is consistent with the state that would exist if the instructions were executed serially, i.e.,

one instruction at a time. If the above conditions hold at the time of the interrupt, then that interrupt is precise, otherwise it is imprecise.

Processes interrupted by an imprecise interrupt are usually terminated. This is due to the fact that the process state at the time the interrupted instruction was issued, could not be reconstructed correctly. Therefore, the process could not continue execution correctly [Wang and Emnett 93]. This implies that imprecise interrupts do not require any special design.

It is the designers choice on which interrupts to be implemented as precise or imprecise. The choice is based on deciding which interrupts are essential to the machine's ability to run programs correctly and which are not [Moudgill and Vassiliadis 96].

2.2 Interrupt Processing on Uniprocessors

On uniprocessors, instructions are executed sequentially. No instruction is executed until the prior instruction has completed execution, that is, there is no instruction overlapping. This implies that the process state is always consistent with the conditions of a precise state. This greatly simplifies the processing of interrupts [Walker 92].

When a process is interrupted, the appropriate interrupt handling routine is invoked to service the interrupt. To ensure the ability to restart the interrupted process, its state is saved. Since the process is executed sequentially, minimal information is included in the process state. In addition to the used registers, the most important pieces of information would be the program counter and the status register. The program counter will provide the location in the instruction stream to restart the interrupted process. This information is pushed onto a stack. When the interrupt handler is done, the stack is popped and the process state is restored [Comer and Fossum 88].

2.3 Interrupt Processing on Pipelined Processors

As the complexity of processors increases, the complexity of interrupt processing strategies increase accordingly. They have to "meet the demands of more applications, peripherals, and functions" [Walker and Cragon 95].

Pipelining is a sophisticated technique used to increase processor performance. It is a system implementation and execution technique that allows the staging and overlapping of instructions during execution [Hennessy and Patterson 90]. Pipelining presents the designers of the interrupt processing strategies with a number of challenges. The overlapping of instructions means that there is an increased amount of state information that needs to be saved and restored in case of occurrence of an interrupt. Also, since instructions are in different stages in the pipeline, the occurrence of an interrupt means that some instructions need to be partially or completely re-executed in order to restore the processor's state [Walker 92].

Some pipelines allow for out-of order execution and completion of instructions. This permits shorter instructions to complete before longer instructions that have been issued earlier. This mechanism greatly enhances the performance of a processor [Torng and Day 93], but it clearly allows for the possibility of violation of the second condition of precise interrupts, as explained in Section 2.1.

In pipelined processors, interrupt processing is not a one-step process [Walker 92] [Walker and Cragon 95]. It consists of six phases (or steps). In each phase, a choice among different tasks to be accomplished has to be made. These choices categorized by the phase in which they might be implemented, are listed below.

Phase 1: Detect the interrupt.

- The interrupt is automatically detected by the processor.
- The interrupt is not automatically detected. Polling is used.

Phase 2: Finish pending instructions.

- Run instructions to completion.
- Nothing is done.

Phase 3: Undo state changes.

- Insure that there are no state changes to be undone.

- Completely undo changes to the state.
- Partially undo changes to the state.
- Ignore changes to the state.

Phase 4: Save process state.

- Save the state.
- State is not saved.

Phase 5: Run the interrupt handler.

- No interrupt handler to be run.
- Interrupt vector.
- Interrupt register.
- Software polling.

Phase 6: Resume the interrupted process.

- Complete instruction re-execution.
- Partial instruction re-execution.
- Instruction continuation.
- Process terminated.

The purpose of phases 2 and 3 is to enforce the conditions of the precise state. Phase 2 ensures that all instructions issued before the interrupted instruction have completed execution and modified the process state. Phase 3 ensures that any instruction issued after the interrupted instruction has not modified the process state.

An implementation choice made in one phase can affect the choices in other phases, i.e., the choices are not independent [Walker 92] [Walker and Cragon 95]. For each design choice made, there are several implementation methods to accomplish that choice. These choices also determine the cost of the interrupt processing strategy as a whole [Wang and Emnett 93].

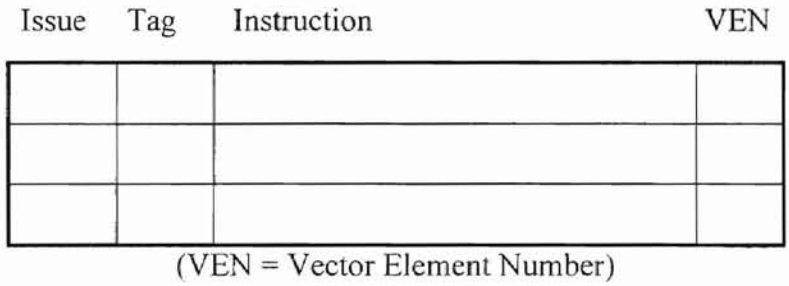
Since interrupt processing on uniprocessors is almost trivial, it will not be considered in this study. Also, since processes interrupted by an imprecise interrupt are usually terminated, there is no point in further considering them. The focus in this study will be on strategies that allow pipelined processors to support precise interrupts.

CHAPTER III
INSTRUCTION WINDOW

3.1 Description

The instruction window (IW) [Torng and Day 93] is a mechanism that enables pipelined systems, which allow out-of-order instruction execution and completion, to support precise interrupts. The IW consists of a set of registers called cells. The IW acts as a staging area for instructions entering the pipeline. An instruction enters the IW when it is fetched and remains there until it finishes execution.

Each cell in the IW consists of at least three fields: issue, instruction, and tag. The issue field is a one-bit field that indicates whether an instruction has been issued. The instruction field holds a copy of the instruction itself. The tag field contains a unique identification of the instruction in that cell. There is an optional field called the vector element number (VEN) that is used by processors that support vector instructions. The structure of the IW is shown in Figure 1.



(VEN = Vector Element Number)

Figure 3.1 The Instruction Window (IW) (Source: [Torng and Day 93])

The IW support three basic operations called Fill, Issue, and Remove/Update, as described below.

- Fill: The Fill operation pushes the newly fetched instruction into the topmost empty cell and assigns it a unique tag. If vector instructions are supported, the VEN field is set to the number of vector elements remaining to be processed, otherwise it is set to 1.

- Issue: At the beginning of each machine cycle, the topmost unissued instruction is checked. If there are no data dependencies and an appropriate functional unit is available, that instruction is issued and its issue bit is set.

- Remove/Update: In this operation the IW makes use of a new parameter: No Return Point (NRP). The NRP is a point in the pipeline after which an instruction has to complete execution and deposit its result, in other words it can not be aborted. When an instruction passes the NRP, its tag is passed back to the IW, where it is identified. If the VEN field is equal to 1, the instruction is removed from the IW, otherwise the VEN field is decremented by 1.

In the case of an interrupt, the following actions are taken by the processor. All instructions that have passed their NRP are allowed to complete and the rest are aborted. When the instructions that are allowed to complete their execution have deposited their results, the processor state is saved. The IW is part of the information to be saved. The appropriate interrupt handler is then fetched and executed.

The information in the IW defines the precise point at which the interrupted process can resume. At the completion of the interrupt handling procedure, the execution is resumed from the instruction at the top of the IW. Since all the instructions that have completed out-of-order had no data dependencies, and the process knows exactly which instructions were in the pipeline, there is no need to undo any state changes.

As an example, consider the sequence of instructions in Figure 3.2 (this example has been adapted from [Torng and Day 93]).

```
1.    ADD R0, R1, R0
2.    MUL R4, R5, R4
3.    ADD R2, R3, R2
4.    ADF VR0, VR1, VR0
5.    ADD R6, R7, R6
```

Figure 3.2 Example code

ADD is an integer addition operation, MUL is an integer multiplication operation, and ADF is a floating point addition operation. Both integer and floating point addition operations need 3 cycles to execute. The integer multiplication operation requires 6 cycles

to execute. Three functional units are available: An integer add unit, a floating point add unit, and an integer multiplication unit. The NRP is set to be at the end of the final cycle of execution. An assumption is made that the processor fetches one instruction per cycle.

Assuming there is at least 4 cells in the IW, an instruction is fetched and written to the IW in every cycle as long as there is room in the IW. At the end of cycle 4, the contents of the IW are as shown in Figure 3.3.

Issue	Tag	Instruction	VEN
1	1	ADD R0, R1, R0	1
1	2	MUL R4, R5, R4	1
1	3	ADD R2, R3, R2	1
0	4	ADF VR0, VR1, VR0	2

(VEN = Vector Element Number)

Figure 3.3 The Instruction Window after cycle 4 for the example code in Figure 3.2

There are no data dependencies between those instructions, therefore an instruction is issued each cycle depending on the availability of an appropriate functional unit. The processing of those instructions is shown in Figure 3.4, where F stands for fill, I for issue, Ex for execute, and D for deposit.

	1	2	3	4	5	6	7	8	9
ADD	F	I	Ex	Ex	Ex	D			
MUL		F	I	Ex	Ex	Ex	Ex	Ex	Ex
ADD			F	I	Ex	Ex	Ex	D	
ADF				F	I	Ex	Ex	Ex	D
ADD						F	I	Ex	Ex

(F = Fill, I = Issue, Ex = Execute, D = Deposit)

Figure 3.4 Instruction process for the first 9 cycles for the example code in Figure 3.2

Note that at the end of cycle 5, instruction 1 passes the NRP mark. At this point its tag is sent back to the IW to be matched. By the end of cycle 6, the tag has been matched, the instruction is removed, and the rest of the instructions are moved to fill the topmost cells in the IW. This makes room for a new instruction, so at the end of cycle 6 the contents of the IW will be as shown in Figure 3.5.

Issue	Tag	Instruction	VEN
1	2	MUL R4, R5, R4	1
1	3	ADD R2, R3, R2	1
1	4	ADF VR0, VR1, VR0	2
0	5	ADD R6, R7, R6	1

(VEN = Vector Element Number)

Figure 3.5 The Instruction Window at the end of cycle 6 for the example code in Figure 3.2

Let an interrupt occur during cycle 8. At the beginning of cycle 9, instruction 4 has passed the NRP. Therefore its tag is sent back to the IW, where its VEN field is reduced by 1. All other instructions are aborted. The contents of the IW that are going to be saved as part of the process state are shown in Figure 3.6.

Issue	Tag	Instruction	VEN
0	2	MUL R4, R5, R4	1
0	4	ADF VR0, VR1, VR0	1
0	5	ADD R6, R7, R6	1

(VEN = Vector Element Number)

Figure 3.6 The Instruction Window to be saved for the example code in Figure 3.2

3.2 Implementation

Implementation of the IW includes all the stages of the pipeline. It is not limited to the actions to be taken in the case of occurrence of an interrupt. These modifications to be implemented in the pipeline simplify interrupt processing in case an interrupt occurs.

The Fill operation does not greatly differ from the normal fetch operation in the pipeline. Extra work has to be done to locate the cell to receive the instructions. After that, a tag is generated and assigned to the tag field associated with the cell. In processors that support vector operations, there is a need for a register to hold the initial number of vector elements to be processed. From this register, called Vector Length Register [Torng and Day 93], the initial value of the VEN field can be obtained.

More complicated actions are involved in the implementation of the Issue operation. These mainly arise from the need to resolve the data dependencies that might exist during the issuing of any instruction. At first glance, one might question the relevancy of this to the interrupt processing strategy. On the other hand, one of the most important features of the IW is that it allows out-of-order execution and completion of

instructions. This implies that an instruction that cannot issue due to unresolved data dependencies, should not obstruct the issuance of subsequent instructions that are ready to issue. There are several dependency-resolution algorithms. One such algorithm is Tomasulo's algorithm [Sohi 90].

Tomasulo's algorithm is a hardware algorithm that was first introduced in the floating point unit of the IBM 360/91 [Tomasulo 67]. Tomasulo's algorithm depends on waiting stations also known as reservation stations. Each functional unit is associated with a reservation station. An instruction that has unavailable operands or unresolved dependencies is dispatched to the reservation station of the proper functional unit. The instruction waits there monitoring the result bus until all its operands are available and its data dependencies are resolved. Then it can be dispatched to the functional unit for execution [Sohi 90]. In the meantime, the issuing process of subsequent instructions can continue with no delays. The structure of each reservation station is shown in Figure 3.7.

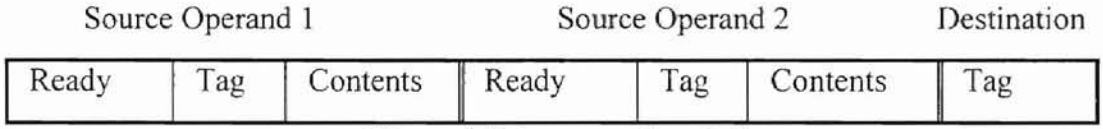


Figure 3.7 A reservation station

If the register containing a source operand of an issuing instruction is busy, that is, it is the destination register for an active instruction, the tag of that register is obtained and the instruction is dispatched to a reservation station. The ready bit of that source operand is set to indicate that it is unavailable. The instruction also obtains a tag for the destination register.

When an instruction completes execution, its result along with the tag of the destination register appear in the result bus. The registers update their busy bits according to this information.

When a source operand is available, its content is copied into the reservation station and the ready bit for that operand is reset. When all the operands are available, the instruction is dispatched to the functional unit. The reservation station is then freed to be

used by subsequent instructions [Sohi 90] (more details on this algorithm and others can be found in [Weiss and Smith 84] and [Tomasulo 67]).

The Remove/Update operation depends mainly on the implementation of the No Return Parameter (NRP) [Torng and Day 93]. During normal execution, the NRP could be set to a fixed stage. This is used to indicate at which point the instruction tag is sent back to the IW, where the tag is matched and the instruction is removed or updated as described in Section 3.1. In the case of the occurrence of an interrupt, the need arises for a flexible NRP that depends on the type of the interrupt detected. The flexible NRP provides a variable response time, according to the nature of the interrupt, by varying the amount of instruction processing that is discarded in order to achieve a faster response time [Torng and Day 93].

The flexible NRP could be implemented through a code segment associated with each interrupt type. The code segment will direct the processor as to which stages of the pipeline to abort. For example, if the NRP for an interrupt type is set at the final cycle of execution, any functional unit that has not signaled to transmit its result in the result bus is flushed. The processor then will proceed to complete whatever instructions are left in the pipeline at that point. Each different interrupt type is associated with a different code segment that enforces its NRP.

3.3 Hardware/Software Cost Analysis

The initial cost arises from building the IW structure itself. Let the IW have n cells, where $n > 1$, then

- Size in bits of the issue field = 1
- Number of issue bits = n * (1 bit per cell) = n
- Size in bits of the tag field = *log* n bits (To ensure uniqueness)
- Number of tag bits = n * (*log* n bits per cell) = n * *log* n
- Each instruction field = size of instruction (system dependent)
- Number of bits for instruction fields = n * (size of instruction)
- size in bits of the VEN field = size of Vector Length Register
- Number of bits for VEN fields = n * (size of Vector Length Register)

The second major cost arises from the implementation of the data dependency resolution algorithm. Intuitively, this depends on the type of algorithm chosen. Referring back to Tomasulo's algorithm, a reservation station has to be build in association with each functional unit. Let there be n functional units and m registers in the system. Since each register could be a destination register, we have

- Number of bits for registers = m * (size of register)
- Size of register = size of original register + busy bit + size of tag
- Size in bits of the busy bit = 1
- Size in bits of the tag field = *log* n

Since each functional unit needs a reservation station associated with it, we have the following equations.

- Number of bits for reservation stations = n * (size of reservation station)
- Size of reservation station = size of fields of source operand 1 + size of fields for source operand 2 + destination tag
- Size of fields for source operand = size of ready field + size of tag + size of contents field
- Size in bits of the ready field = 1
- Size in bits of the tag field = *log* n
- Size in bits of the contents field = size of a register (without the busy bit and tag)
- Size in bits of destination tag = *log* n

Since each register is tagged, each register needs to be associated with a combinational circuit to carry out the tag matching [Sohi 90]. Similar combinational circuits need to be associated with the IW to carry out the issue bit and tag matching during the Issue and Remove/Update operations respectively [Torng and Day 93].

Software does not constitute a great part in the implementation of this strategy. For each different type of interrupt there is an appropriate interrupt handling routine. In addition, each interrupt type is associated with a code segment that implements its NRP.

It can be inferred from the above discussion that hardware is the major contributor to the cost of implementing the IW strategy. Software constitutes a small but not trivial part of the cost. It would be safe to state that the cost of implementing the IW strategy is about 90% hardware and 10% software.

CHAPTER IV
CHECKPOINTING

4.1 Description

Out-of-order execution and completion of instructions is a mechanism that can enhance the performance of a processor. It is a further step towards fully utilizing the concept of concurrent processing. Unfortunately, it introduces complications with regard to interrupt processing. These complications are mainly a result of the fact that the machine state (the contents of registers and memory) is not always precise.

Checkpointing [Hwu and Patt 87] is a mechanism that assists in processing interrupts in processors that allow out-of-order execution. It does not force the machine state to be precise, rather it saves the machine state at preselected points of execution called checkpoints. In the case of an interrupt occurring, the machine state is repaired to the state saved at the last checkpoint which is known to be precise. Instructions are then executed sequentially until the instruction that caused the interrupt is reached. This way the precise state for any instruction can be restored.

When the machine state is repaired to a previous state, useful work is lost. Re-executing instructions to reach the interrupted instruction implies redundancy and extra time overhead. A question arises: why not establish a checkpoint at every instruction? This way no useful work is discarded and no instructions are re-executed. Hwu and Patt addressed this question as follows [Hwu and Patt 87]:

> On the one hand, since checkpointing is an overhead function, its cost in time and additional hardware should be kept as small as possible. This means no more checkpoints than absolutely necessary. On the other hand, repair to the last checkpoint involves discarding useful work. The further apart the checkpoints, the more useful work gets thrown away.

4.2 Checkpoint Mechanism

4.2.1 Definitions

The following are definitions of the terminology used to describe the checkpoint algorithm [Hwu and Patt 87]. Let c be the maximum number of simultaneously active checkpoints allowed by the checkpoint algorithm.

- An instruction is *active* if it has been issued but not yet finished execution.
- A processor's state at one instruction is *consistent* if it is precise. Hwu and Patt made a distinction between consistent and precise states [Hwu and Patt 87]. For the objectives of this study, the difference is irrelevant.
- The *E-repair range* of a checkpoint consists of the sequence of instructions between this checkpoint and the next checkpoint.
- A processor's state at one instruction is called a *potential precise state* if some of the instructions issued before this instruction are still active. The condition, for a precise state, that instructions issued after this instruction cannot affect the state, also holds.
- *Active(t)* is the set of consecutive checkpoints that have active instructions in their E-repair ranges at time t. $Active_i(t)$ is the ith element of the set, where i ranges from 1 to c. The nearest active checkpoint is at position 1.
- *Potent(t)* is the set of potential precise states at time t. A potential precise state is maintained for each active checkpoint. Hence at time t, $Potent_i(t)$ holds the potential precise state maintained for the checkpoint at $Active_i(t)$, where i ranges from 1 to c.
- A *logical space* is a copy of the processor's state. It can be the machine state or a potential precise state of one of the checkpoints. Since the checkpoints are pre-determined, there will be a known number of logical spaces needed by the checkpoint mechanism.

4.2.2 Data Structures

The following is a description of the different data structures maintained by the checkpoint algorithm in order to accomplish its tasks [Hwu and Patt 87]. All the arrays are of size c.

- *Current* is the logical space holding the current machine state.
- *Backup* is an array of logical spaces holding the potential precise states. At time t, $Backup_i$ holds $Potent_i(t)$, where i ranges from 1 to c.
- *Count* is an array of counters. At time t, $Count_i$ holds the number of active instructions in the E-repair range of $Active_i(t)$, where i ranges from 1 to c.
- *Except* is an array of flags. At time t, $Except_i$ indicates whether an interrupt has occurred in the E-repair range of $Active_i(t)$, where i ranges from 1 to c.
- *Ident* is a decrementing counter that is used to uniquely identify a checkpoint. At time t, Ident holds the identification number of $Active_i(t)$, where i ranges from 1 to c.

4.2.3 Checkpoint Algorithm

Initially, all elements of the arrays Count and Except are set to zero, all the elements of the array Backup are initialized to Current, and the decrementing counter Ident is initialized to -c. The arrays are indexed from left to right. That is, if $i > j$, then location i is to the left of location j in any of the arrays.

The main idea behind this algorithm is to maintain a potential precise state for every active checkpoint. The algorithm manages those potential states until they evolve into precise states. These precise states can then be used to restore the machine state in case of the occurrence of an interrupt.

The operations performed by the algorithm are Issue, Check, Deliver, and Repair, as explained below.

- Issue: When a new instruction is issued, the value of Ident is attached to the new instruction to identify to which checkpoint range it belongs. The value of $Count_i$ is incremented.

- Check: This operation is performed when establishing a new checkpoint. After the last instruction in a checkpoint's E-repair range is issued, the arrays (i.e., Backup, Count, and Except) are shifted one position to the left. Current is copied into $Backup_1$, and $Count_1$ and $Except_1$ are reset to zero. Finally, Ident is decremented by 1 to identify the new checkpoint. The algorithm should always maintain at least one precise state. Therefore, if at this point $Count_c$ is not zero, this implies that all the states are still potential. Hence, the instruction issue is stalled until $Count_c$ is zero.
- Deliver: When an instruction finishes execution, its result is written to Current. Now the potential states have to be updated also. To do so, the following sequence of operations are executed,
  - Subtract the current value of Ident from the checkpoint identification value attached to the instruction. The result of this subtraction is used as an index i to reference the arrays. Write the execution result to $Backup_k$, where k = 1 to i.
  - Decrement $Count_{i+1}$.
  - If an interrupt has occurred, set $Except_{i+1}$ to 1.
- Repair: This operation is performed when an interrupt occurs. As a result of this operation, all active instructions are discarded and the machine state is restored to a previous precise state. This is done by copying the contents of $Backup_c$ into Current. The next step is to execute instructions sequentially until the interrupted instruction is reached. Then the operation Check is performed c times to reset all the positions in Backup to become identical to Current, and all the values in Count and Except are reset to zero. At this point, the machine state is precise and the system is ready to invoke the interrupt handler.

The running of this algorithm is depicted in the flow chart of Figure 4.1.

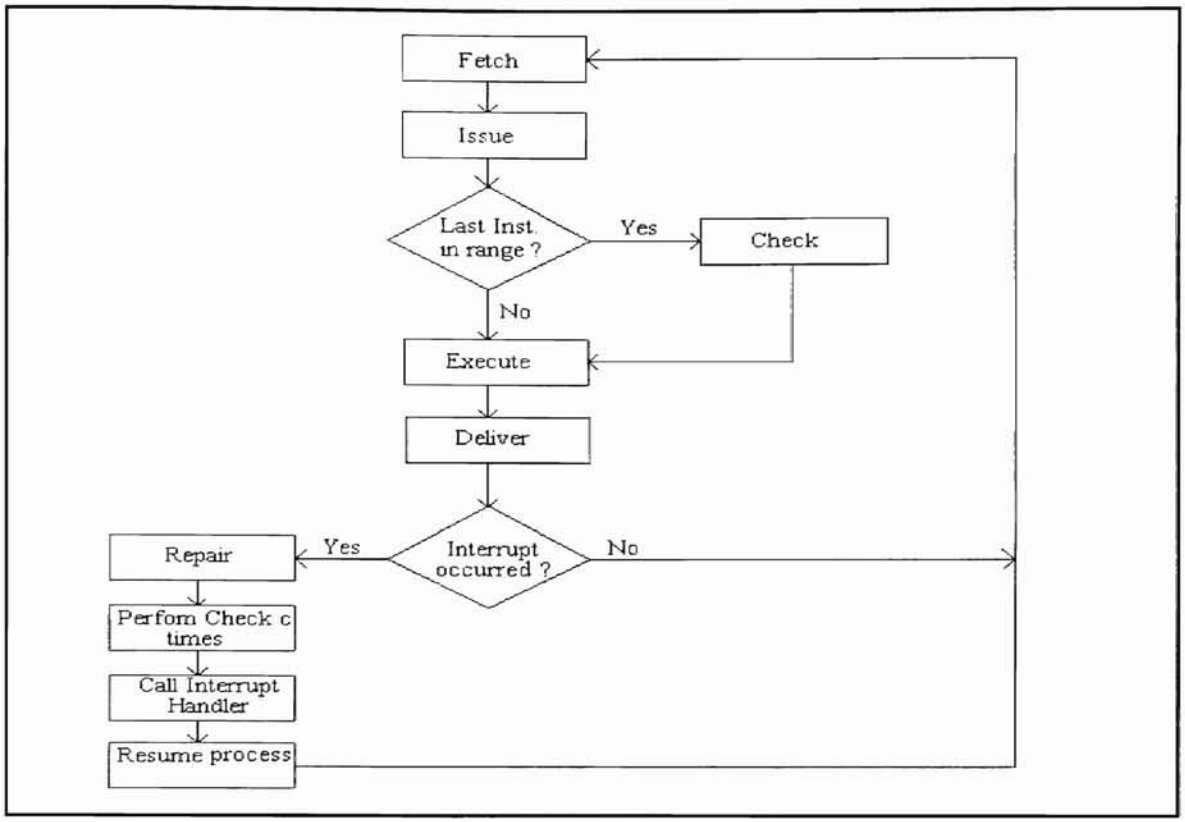


Figure 4.1 Flow Chart of Checkpoint Algorithm

As an example, consider the set of instructions in Figure 4.2, this example was given in [Hwu and Patt 87]. This code segment basically traverses a linked list, multiplying the value in each location by a known factor.

```
A: loop : LD R0, R1(0)
B:        MUL R0, R2
C:        ST R1(0), R0
D:        LD R1, R1(1)
E:        If(R1 != nil) goto loop
```

Figure 4.2 Example Code

In this figure, MUL is a floating point multiplication that needs 4 cycles to execute. LD and ST are memory load and store operations respectively, both of which need 2 cycles to execute. The branch operation needs only one cycle to execute.

Assume the initial machine state is as given in Figure 4.3.

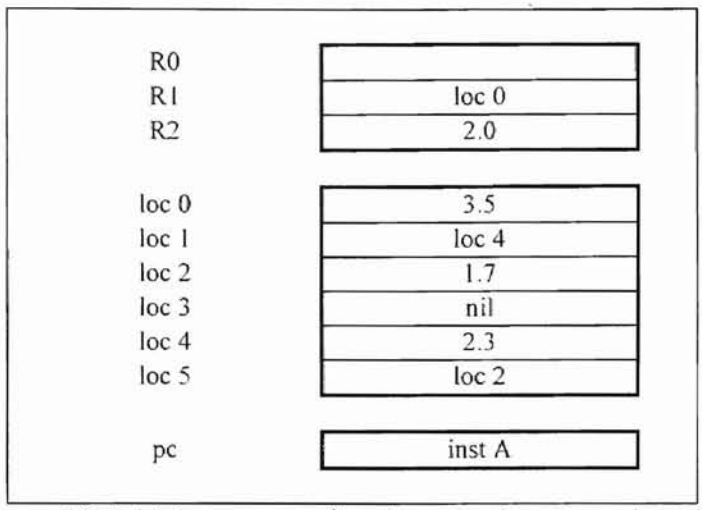


Figure 4.3 Initial contents of registers and memory locations for the example code in Figure 4.2

The pipeline is a typical one consisting of 4 stages: fetch, issue, execute, and deposit. It makes use of a branch prediction mechanism to enhance the performance of the pipeline. For the purpose of this example, the assumption is made that all branch predictions were correct. A data dependency algorithm is used to resolve dependencies between instructions during the issue stage. More details about the data dependency algorithms can be found in Section 4.2. The execution of the example program in Figure 4.2 is depicted in Figure 4.4.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
F	A1	B1	C1	D1	E1	A2	B2	C2	D2	E2	A3	B3	C3	D3	E3	A4	B4	C4
I		A1	B1	C1	D1	E1	A2	B2	C2	D2	E2	A3	B3	C3	D3	E3	A4	B4
Ex			A1		B1	D1		A2 E1	C1	B2	D2		A3 E2	C2	B3	D3		A4 E3
Ex				A1		B1	D1		A2	C1	B2	D2		A3	C2	B3	D3	
Ex							B1					B2					B3	
Ex								B1					B2					B3

Figure 4.4 Execution of the example code in Figure 4.2 for the first 18 cycles

The checkpoints for this example are selected to be before A1, between E1 and A2, and between E2 and A3, as shown in Figure 4.5. The initial state of all the relevant data structures is shown in Figure 4.6.

Figure 4.5 Locations of checkpoints along the instruction stream for the example code in Figure 4.2

For this example, the maximum number of simultaneous active checkpoints is chosen to be two, because it has been proven theoretically that this is the minimum number needed to implement this algorithm effectively [Hwu and Patt 87]. Therefore, c is equal to 2, which implies that the initial value of Ident will be -2. Based on this, the algorithm needs three logical spaces, two for the potential precise states and one for the machine state.

Referring back to Figure 4.4, in cycle 5 instruction E1 is issued (E1 is instruction E in the first iteration of the example program in Figure 4.2), which is the last instruction in the E-repair range of checkpoint -2. At this point the code has loaded the contents of loc 0 into R0. Now, the operation Check is to be performed, and the contents of $Backup_1$, $Count_1$, and $Except_1$ are shifted into $Backup_2$, $Count_2$, and $Except_2$, respectively. A copy of Current is placed in $Backup_1$, and $Count_1$ and $Except_1$ are reset to zero. Finally, the decrementing counter Ident is decremented to -3. Figure 4.7 shows a snapshot of the system at the end of cycle 6.

At the end of cycle 14, the only instruction active in the E-repair range of checkpoint -3 is C2, as shown in Figure 4.8. C2 will finish execution at the end of the next cycle. At that point, Deliver will be performed as follows,

- The value in Ident(-4) is subtracted from the checkpoint identification of C2(-3) to obtain the index value 1.
- The result of C2 will be written to Current and $Backup_1$.
- $Count_2$ will be decremented.

Assume an interrupt occurred during the execution of C2. The results will not be written to the logical spaces and $Except_2$ will be set to 1, as shown in Figure 4.9. At this point, Repair will restore the machine state to a precise state. The precise state is the one in $Backup_2$. $Backup_1$ is not precise since B3, C3, and D3 are still active. Repair will replace the contents of Current by those of $Backup_2$, as shown in Figure 4.10.

The next step is to execute instructions A2 and B2 sequentially, in order to arrive at the precise state before the interrupted instruction. Check is then performed twice to reset the contents of all the positions in Backup to become identical to Current. The system is now ready to invoke the interrupt handler. A snapshot of the state of the system, just before invoking the interrupt handler, is shown in Figure 4.11.

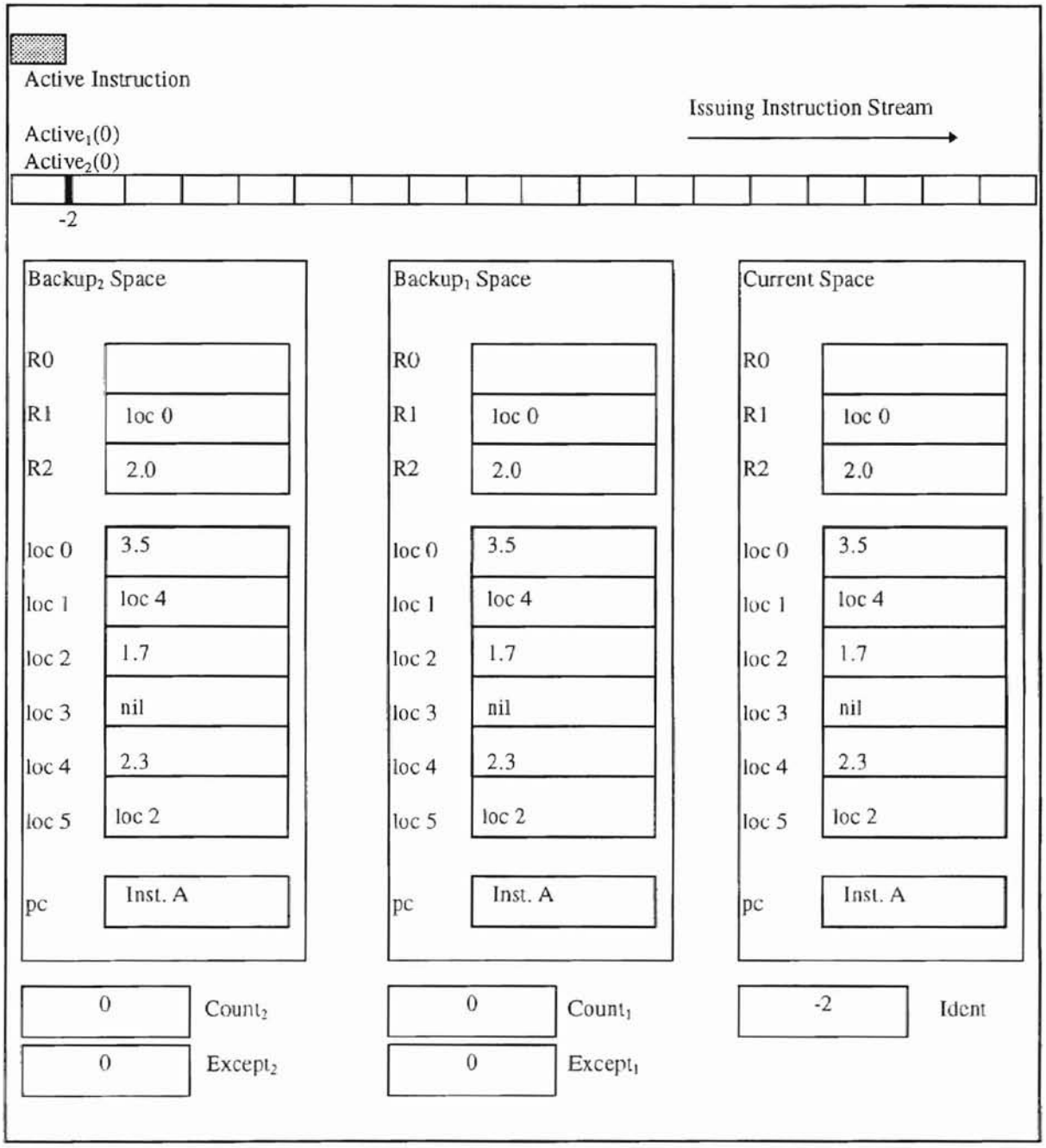


Figure 4.6 Initial state for the example code in Figure 4.2

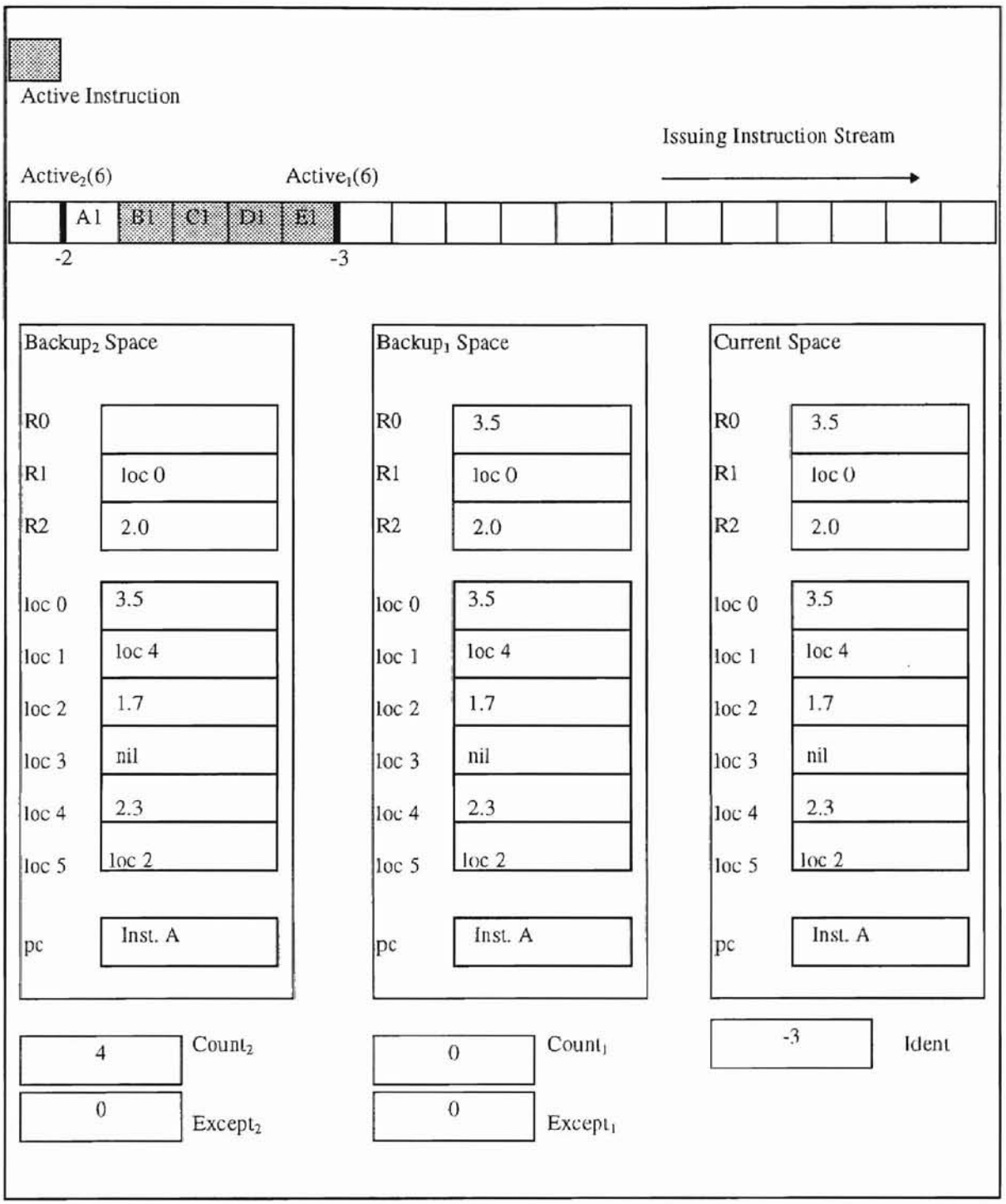


Figure 4.7 State at the end of cycle 6 for the example code in Figure 4.2

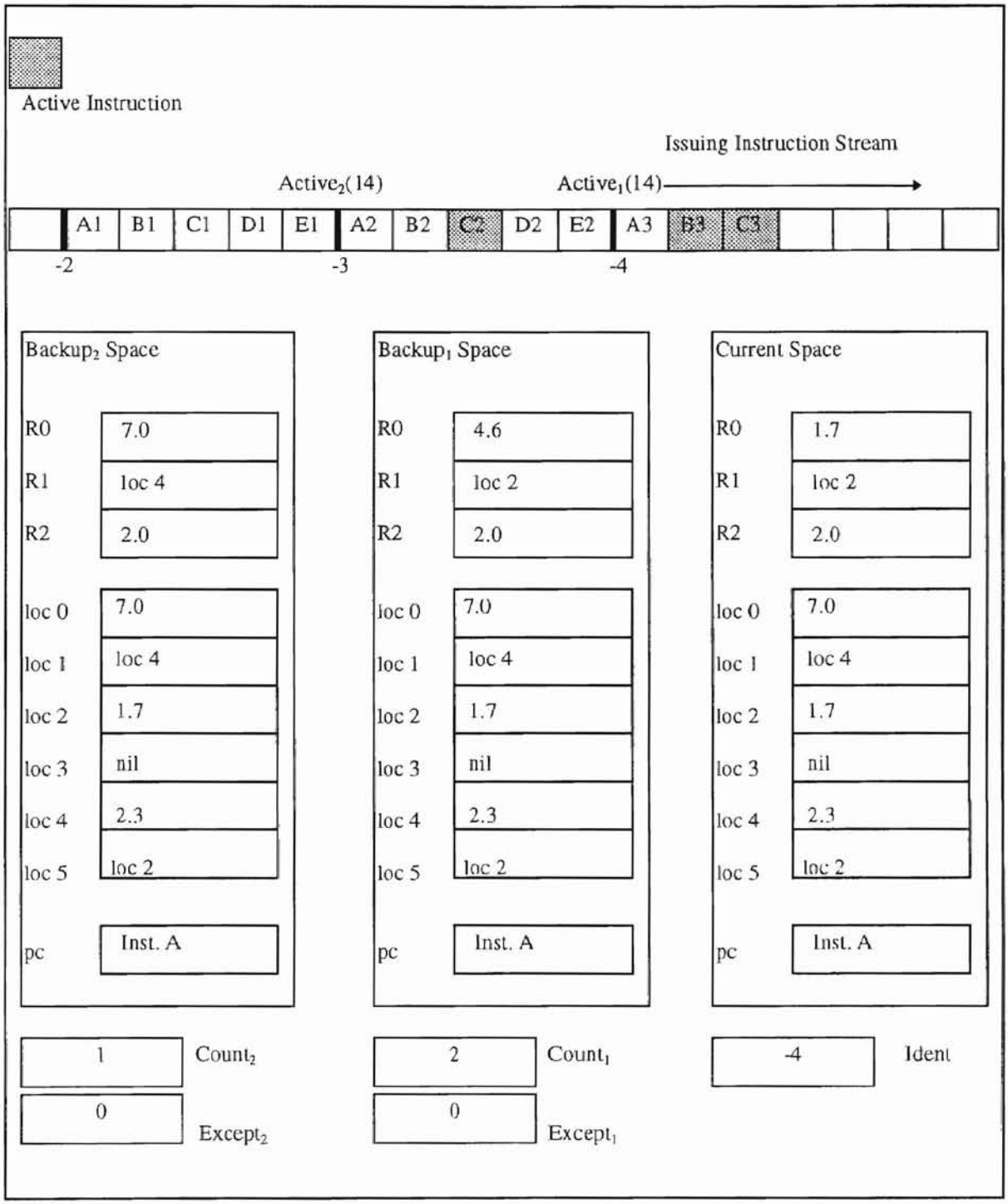


Figure 4.8 State at the end of cycle 14 for the example code in Figure 4.2

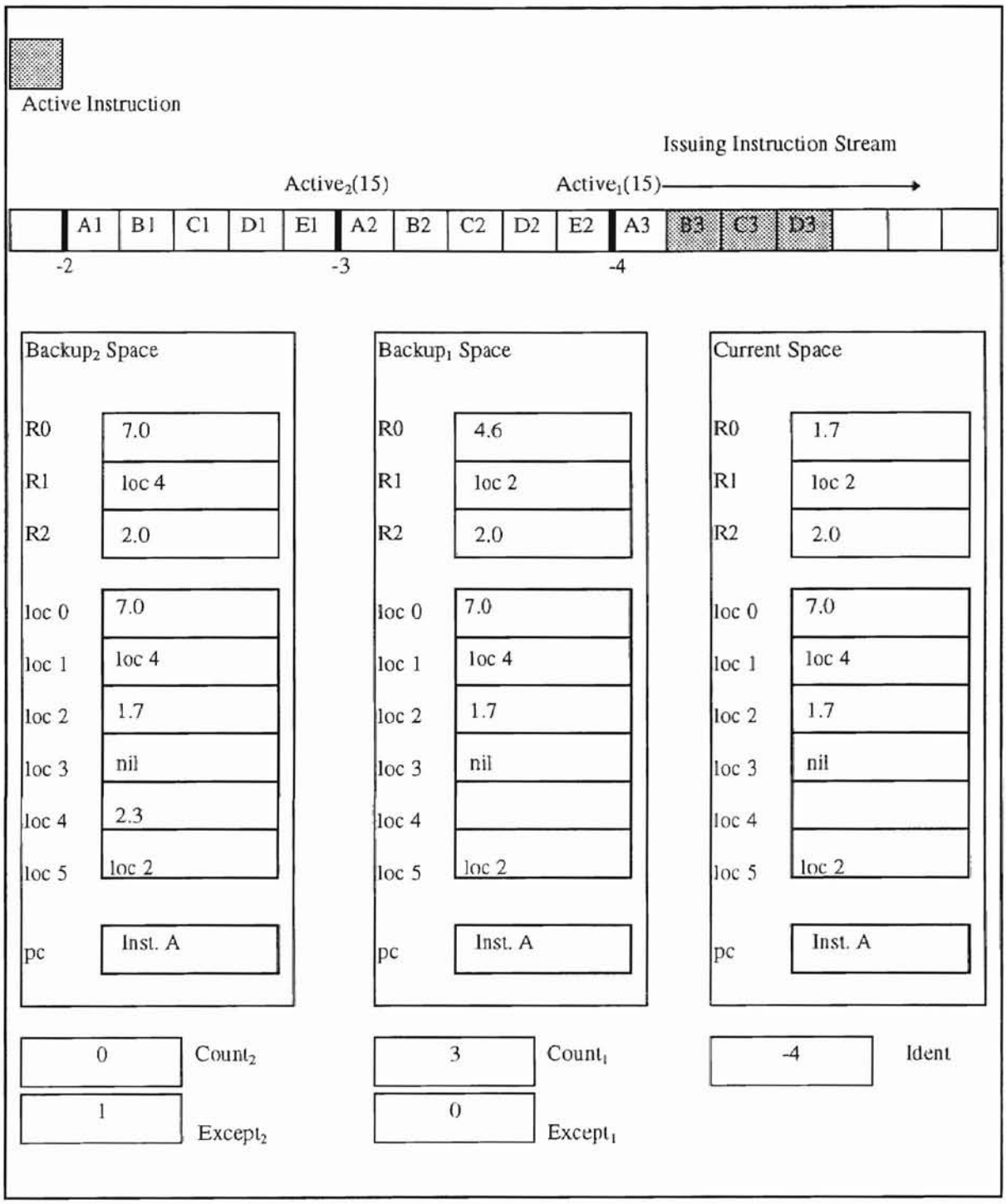


Figure 4.9 State at the end of cycle 15 before Repair for the example code in Figure 4.2

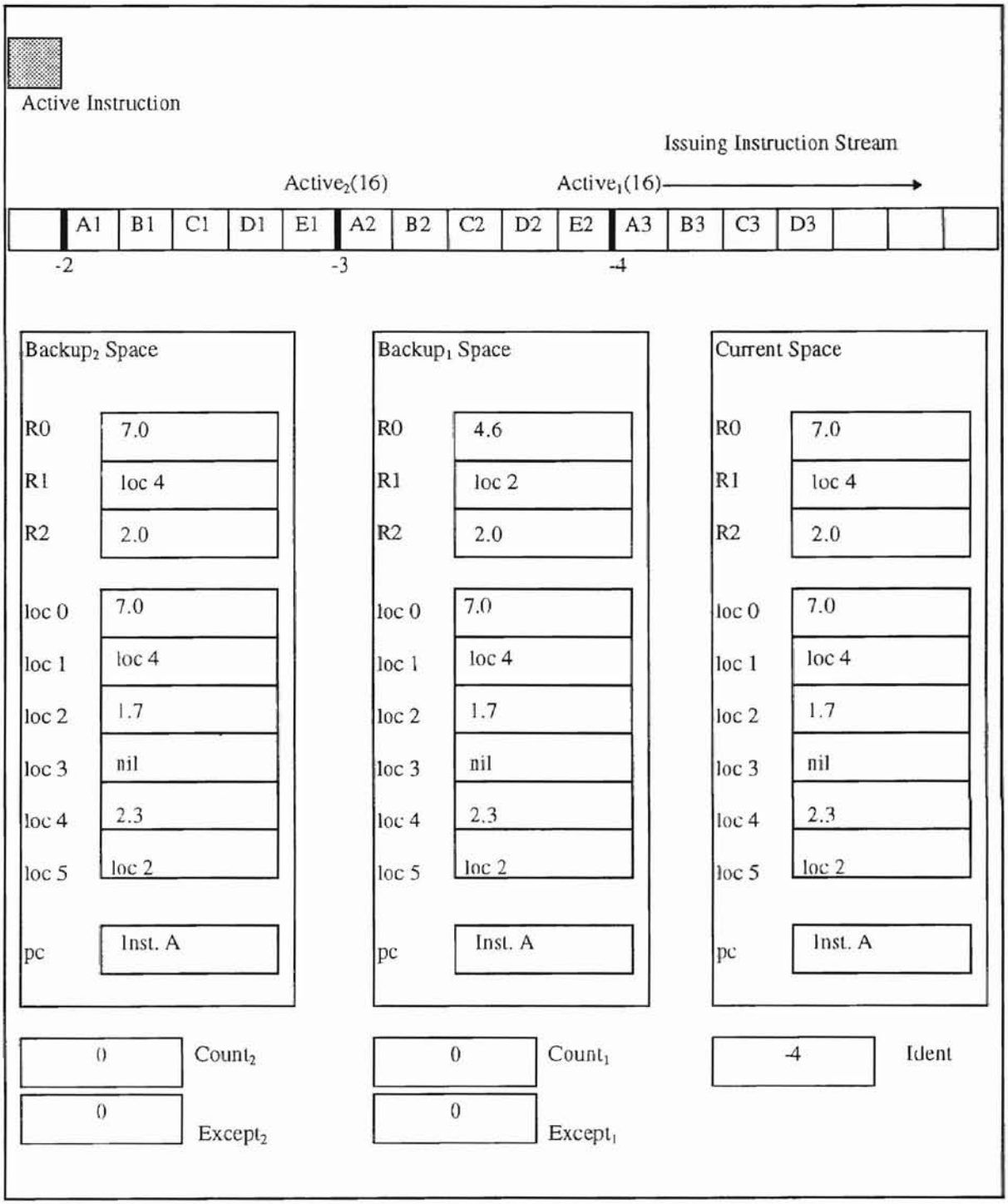


Figure 4.10 State at the end of cycle 16 after Repair for the example code in Figure 4.2

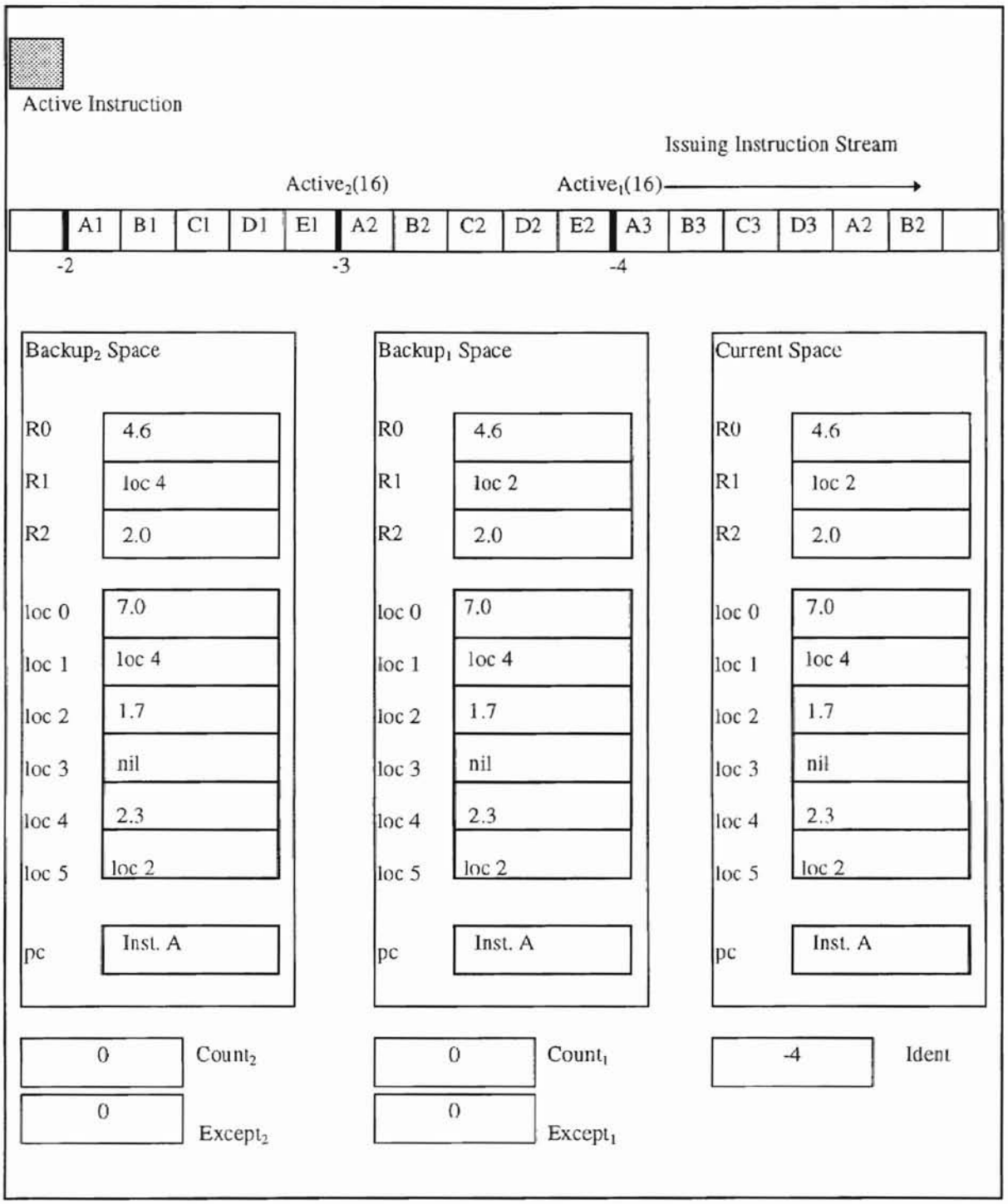


Figure 4.11 State before running interrupt handler for the example code in Figure 4.2

4.3 Hardware/Software Cost Analysis

The cost of this strategy can be divided into two parts: the cost to implement the algorithm operations and the cost of maintaining the data structures used by the algorithm.

Referring to the example given in Section 4.2, the value of c, the maximum number of simultaneously active checkpoints, was 2. Therefore, the system had to maintain 3 logical spaces. Assuming that the machine state is not large, this allows for all the logical spaces to be physically implemented. This means that the system will need triple the usual number of registers and related hardware. The Count and Except arrays can also be implemented using hardware. In order to perform the Check and the Repair operations, concurrent data transfer paths between those logical spaces are needed, as shown in Figure 4.12 [Hwu and Patt 87].

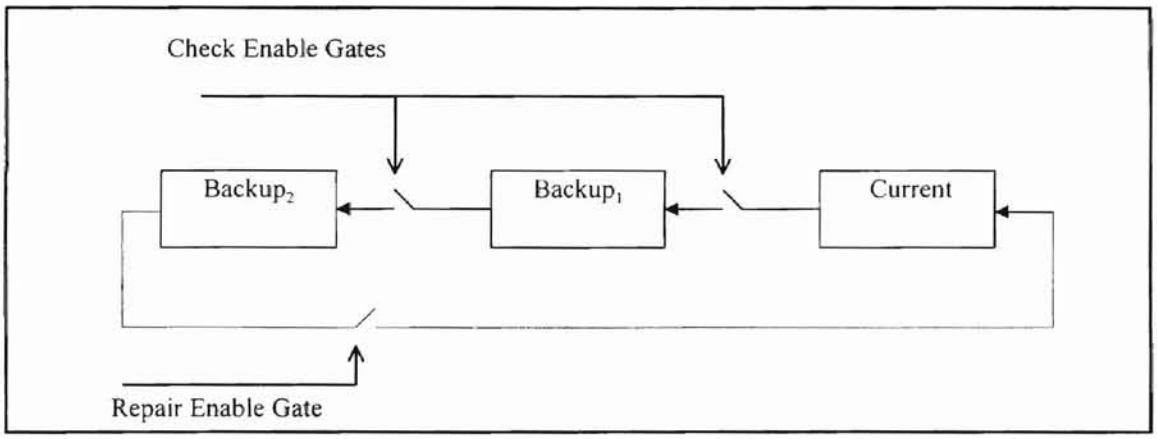


Figure 4.12 Physical implementation of Logical Spaces and the associated data paths

If the system supports a larger number of logical spaces or the machine state is very large, hardware implementation will be impractical because of the prohibitive cost. The practical solution would be to store the logical spaces in secondary storage. This implies that managing the logical spaces would be more of a task handled by software. Based on this argument, the implementation of the operations, and hence their cost, vary according to the implementation of the logical spaces.

The Issue operation does not differ much from the usual issue operation. The instruction has to be modified to carry with it a value to identify to which checkpoint's E-repair range it belongs, this would be the value of the decrementing counter Ident at the time the instruction is issued. The cost analysis for the data dependency resolution algorithm is similar to that discussed in Section 3.3. The Check operation, on the other hand, provides a bigger challenge. The main complexity arises from the shift operation on the arrays, especially the Backup array. If the logical spaces are physically implemented as shown in Figure 4.11, the Check operation has to enable the data transfer paths to move the contents of one location to another. If the logical spaces were stored in secondary storage, then shifting them requires a series of move operations. Since each location in this array has the same size as the machine state, the complexity of transferring all this information would be bound by the size of the machine state and the value of c.

The Deliver operation involves the execution of a subtraction operation and then a reference to the arrays. The referencing implies the calculation of an effective address. This implies that this operation is implemented through software regardless of the way the logical spaces are implemented.

If the logical spaces were physically implemented, the Repair operation has to enable the repair data transfer path between $Backup_c$ and Current. If they are stored in secondary storage, the new contents of Current are loaded from $Backup_c$. The re-execution of the instructions is part of the normal pipeline operations. The interrupt handler is the last piece of software needed to conclude the interrupt processing.

According to the above discussion, the cost of the algorithm depends mainly on the maximum number of simultaneous active checkpoints supported by the algorithm, and whether the logical spaces are implemented physically or not. Therefore, the cost of this strategy is variable. If the logical spaces are physically implemented, then the hardware portion of the cost dominates the cost of this strategy. As the number of active checkpoints increases, it would be practical to store the logical spaces in secondary storage. Hence, more software and less hardware, would be needed to manage the potential precise states.

CHAPTER V

STRATEGIES USING RESULT SHIFT REGISTERS

5.1 Result Shift Register

The main obstacle in interrupt processing is obtaining a precise state prior to the processing of an interrupt. Out-of-order execution and completion of instructions is the main cause of imprecise states. An imprecise state is a state that has been modified in an out of order fashion, hence, one obvious course of action would be to develop a method that controls the order according to which instructions modify the processor's state. The result shift register is a mechanism that does just that by controlling the order at which instructions deposit their results. Figure 5.1 shows the structure of one result shift register.

Stage	Functional Unit	Destination Register	Valid	Program Counter
1				
2				
3				
4				
5				
. . .				
N				

Figure 5.1 Result Shift Register (Source: [Smith and Pleszkun 88])

When an instruction is issued, it reserves a stage in the result shift register and writes some control information in the proper fields of that stage [Smith and Pleszkun 88]. The stage field shows how far the instruction is from completing execution. That is, if an instruction requires i cycles to complete, stage i will be reserved for that instruction if it is available. If the stage is not available, the issuing is stalled until the next cycle. The valid bit indicates if this stage is reserved, and the rest of the fields are self-explanatory.

At end of each cycle, the control information of each instruction is moved one step up towards the top of the result shift register. For the instruction at the top, the result bus is reserved for the functional unit, shown in the functional unit field, in order to deposit its result in the proper register, indicated by the destination register field.

By controlling which stage an instruction reserves when issued, the result shift register controls when an instruction can deposit its results. All the interrupt processing strategies discussed in this chapter are built based on the use of a result shift register, with some modifications, to accomplish their task.

5.2 Reorder Buffer

The reorder buffer strategy allows instructions to execute out of order, but it controls the order at which instructions modify the process state. This way, the reorder buffer utilizes the performance enhancement provided by the out-of-order execution of instructions, but at the same time ensures a precise state all the time [Smith and Pleszkun 88].

The reorder buffer with its associated result shift register are shown in Figure 5.2.

The reorder buffer is a circular queue with head and tail pointers. Entries are inserted in and deleted from the buffer similar to any circular queue. When an instruction is issued, two entries are made: one in the result shift register and one in the reorder buffer. The entry in the reorder buffer will be inserted at the location pointed to by the tail pointer at that time.

In the result shift register, the fields for the destination register and program counter are replaced by a tag field. The tag field holds the location of the corresponding

entry in the reorder buffer, that entry contains the destination register and the program counter fields. In addition, each entry in the reorder buffer will hold the result of the instruction once it completes. Also, there is an exception field for each entry that is set in case the instruction at that entry causes an interrupt.

Stage	Functional Unit	Valid	Tag
1			
2			
3			
4			
5			
. . .			
N			

Result Shift Register

Entry Number	Destination Register	Result	Exceptions	Valid	Program Counter
1					
2					
3					
4					
5					
. . .					

Reorder Buffer

Figure 5.2 Reorder Buffer and the associated Result Shift Register (Source : [Smith and Pleszkun 88])

When an instruction reaches the top of the result shift register, that is, it finished execution, the contents of the tag field are used to locate the corresponding entry in the reorder buffer. The result of the instruction is written, not to the destination register, but to the designated field in that entry, and that entry is marked as valid. If the instruction has caused an interrupt, the exception field is set accordingly. The result of an instruction is written to the destination register only if it is at the head of the reorder buffer, and it is

valid. Since the instructions enter the reorder buffer in the order of their issuance, the reorder buffer allows the processor to maintain a precise state at all time. The instructions can finish execution out of order, but the process state is only modified in order.

Since only the instruction at the head of the reorder buffer is allowed to write its result, only at that point will an instruction be checked for interrupts. If that instruction indicates an interrupt, the issuing process is stopped. Since the process state is precise, no extra work needs to be done. The program counter field indicates the precise location where the interrupted process can resume.

Consider the example code in Figure 5.3.

```
ADD R0, R0, R1
MUL R2, R2, R3
ADD R4, R4, R5
```

Figure 5.3 Example Code

ADD is integer addition and needs 2 cycles to finish execution. MUL is integer multiplication and needs 4 cycles to finish execution. The execution of this code is shown in Figure 5.4.

	1	2	3	4	5	6	7	8
ADD	F	I	Ex	Ex	D			
MUL		F	I	Ex	Ex	Ex	Ex	D
ADD			F	I	Ex	Ex	D	

(F = Fetch, I = Issue, Ex = Execute, D = Deposit)

Figure 5.4 Execution of the example code in Figure 5.3

At the end of cycle 4, the first ADD instruction finishes execution and exits the result shift register. Since its entry is at the head of the reorder buffer, being the first to issue, its result is written to its destination register and its entry is discarded. At the end of cycle 5, the result shift register and the reorder buffer are as shown in Figure 5.5. At the end of

cycle 6, the second ADD has finished execution and is ready to deposit its result. Its result will be placed in entry 3 of the reorder buffer. However, since it is not at the head of the buffer, this result will not be written to the destination register until MUL, in entry 2, has finished execution and deposited its result, hence preserving the precise state.

Stage	Functional Unit	Valid	Tag
1	Integer ADD	1	3
2	Integer MUL	1	2
3			
4			
5			
.			
.			
.			
N			

Result Shift Register

Entry Number	Destination Result	Result	Exceptions	Valid	Program Counter
1					
2	R2			0	2
3	R4			0	3
4					
5					
.					
.					
.					

Reorder Buffer

Figure 5.5 Reorder Buffer and the associated Result Shift Register at the end of cycle 5 for the example code in Figure 5.3

During the normal flow of the pipeline, an instruction ready to issue may need some of the results being held in the reorder buffer, and, in such case, have to wait until those results are written to the appropriate registers. To avoid this, the reorder buffer strategy uses bypass data paths. The bypass paths allow issuing instructions to read their operands values from the reorder buffer before they are written to registers. This will not

affect the precise state of the process since these instructions will not be allowed to modify the process state out of order.

5.3 History File

The history file strategy does not differ much from the reorder buffer strategy. In fact, this strategy was developed as an enhancement of the reorder buffer strategy [Smith and Pleszkun 88]. The primary difference is that this strategy does not guarantee a precise state at all time. As in the reorder buffer strategy, it allows instructions to execute out of order, but it also allows the process state to be modified out of order. On the other hand, it retains enough information to attain a precise state when needed [Smith and Pleszkun 88].

The history file is a circular queue that is associated with a result shift register. The basic structure is similar to that used in the reorder buffer strategy, except for some minor differences, as shown in Figure 5.6.

When an instruction is issued, an entry is made at the proper stage in the result shift register. Another entry is made at the tail of the history file, and the location of this entry is saved in the tag field of the corresponding entry in the result shift register. The old value of the destination register for this instruction is saved in the old value field of that instruction's entry in the history file. This value will be used, in case of an interrupt occurring, to undo the change made by this instruction.

When an instruction reaches the top of the result shift register, it writes its result to the destination register and exits the result shift register. Using the tag field, the corresponding entry in the history file is located and marked as valid. If an interrupt has occurred, the exception field is set accordingly.

A valid instruction at the head of the history file is checked for exceptions. If none occurred, the old value of the destination register is not needed any more and that entry is removed from the file. On the other hand, if an interrupt has occurred, the issuing process is halted. All active instructions are allowed to complete execution until the result shift register is empty. To retain a precise state, all changes made to the process state by instructions issued after the interrupted one have to be undone. Therefore, starting at the

tail of the history file and working backwards to the head, at each entry the register indicated by the destination register field is reloaded by the value in the old value field of that entry, and the entry is then discarded. In other words, the state changes are undone starting with the most recent one and working backwards until the interrupted instruction. At this point of time, the process state is precise and once it is saved, the interrupt handler can be invoked. The program counter entry at the head of the history file indicates the precise point to restart an interrupted process.

Stage	Functional Unit	Destination Register	Valid	Tag
1				
2				
3				
4				
5				
. . .				
N				

Result Shift Register

Entry Number	Destination Register	Old Value	Exceptions	Valid	Program Counter
1					
2					
3					
4					
5					
. . .					

History File

Figure 5.6 History File and the associated Result Shift Register (Source: [Smith and Pleszkun 88])

5.4 Future File

The future file strategy is composed of a little of each of the previous strategies discussed in this chapter. It uses the same concept as the history file, allows instructions to execute and modify the process state out of order, but retains information to attain a precise state when needed. The strategy uses two register files: the future file and the architectural file. The strategy maintains the architectural file always in a precise state, using a reorder buffer and a result shift register associated with the file. This file will be used, when required, to restore a precise state. The future file, on the other hand, can be updated out of order. Issuing instructions read their operands values from the future file, thus avoiding waiting for the results to be written to the registers. The organization of the future file strategy is shown in Figure 5.7. The result shift register and the reorder buffer have the same structure as in the reorder buffer strategy.

When an instruction is issued, two entries are made: one in the result shift register and one in the reorder buffer. The mechanism behaves exactly as in the reorder buffer strategy, except when results are ready to be written to registers. When an instruction reaches the top of the result shift register, its result is written to both the future file and the reorder buffer. The tag field is used to mark the corresponding entry in the reorder buffer as valid. When a valid instruction is at the head of the reorder buffer, its result is written to the architectural file. This way, the architectural file is guaranteed always to contain the precise state.

When an instruction at the head of the reorder buffer indicates the occurrence of an interrupt, the issuing process is stopped. All active instructions are allowed to complete until the result shift register is empty. Then using the destination register field in the entries between the head and tail of the reorder buffer, the register values in the architectural file are used to restore the precise state to the future file. The program counter field at the head of the reorder buffer indicates the precise point to resume the interrupted process.

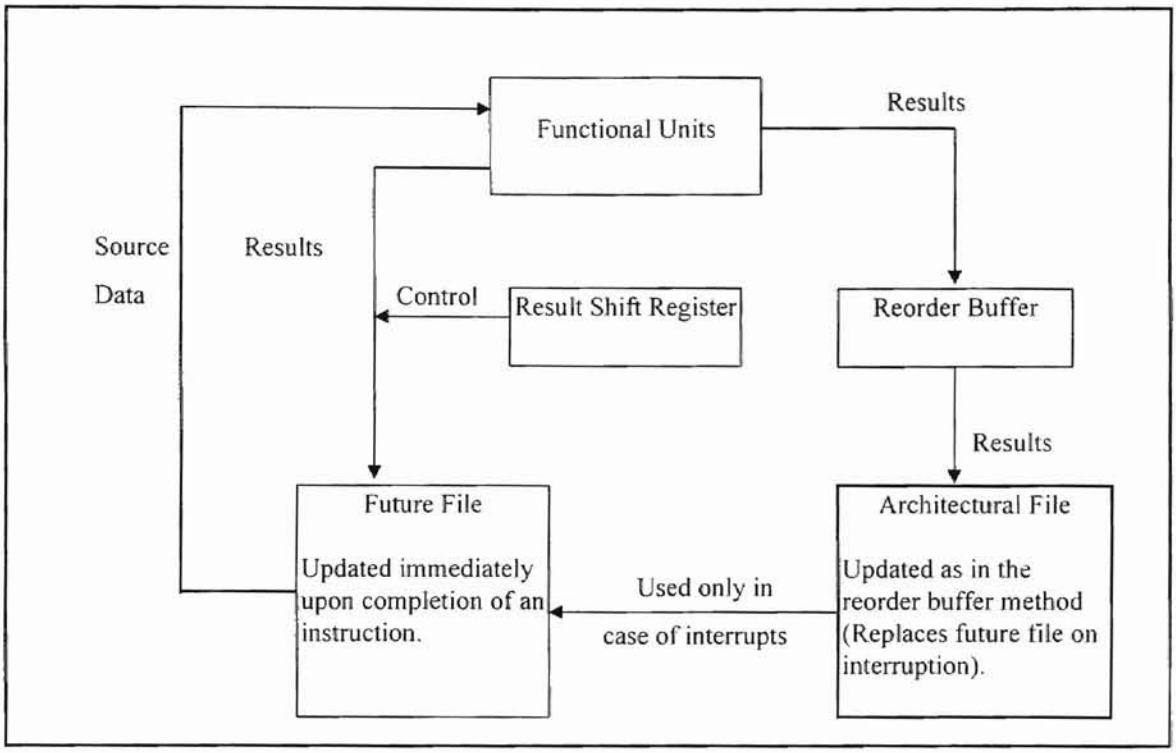


Figure 5.7 Future file organization

5.5 Hardware/Software Cost Analysis

5.5.1 Reorder Buffer

The first task at hand is to determine the cost of implementing the result shift register and the reorder buffer used in this strategy. Referring back to Figure 5.2, let the result shift register have n stages. To allow for the accommodation of all types of instructions, n should be equal to the largest number of cycles needed by one instruction. As for the reorder buffer, let there be m entries. The value of m should be equal to or greater than the maximum number of instructions that can be in the pipeline at one time. Then, for the result shift register we have

- Size in bits of the stage field = *log* n
- Number of bits for stage fields = n * (*log* n bits per entry) = n * *log* n
- Size in bits of the functional unit field = *log* (number of functional units)
- Number of bits for functional unit fields = n * *log* (number of functional units)
- Size in bits of the valid field = 1

- Number of bits for valid fields = n * (1 bit per entry) = n
- Size in bits of the tag field = *log* (size of reorder buffer) = *log* m
- Number of bits for tag fields = n * (*log* m bits per entry) = n * *log* m

As for the reorder buffer, we have the following

- Size in bits of the entry number field = *log* m
- Number of bits for entry number fields = m * (*log* m bits per entry) = m * *log* m
- Size in bits of the destination register field = *log* (number of registers)
- Number of bits for destination register fields = m * *log* (number of registers)
- Size in bits of the result field = size of register (which is system dependent)
- Number of bits for result fields = m * (size of register)
- Size in bits of the exceptions field = *log* (number of interrupt types)
- Number of bits for exceptions fields = m * *log* (number of interrupt types)
- Size in bits of the valid field = 1
- Number of bits for valid fields = m * (1 bit per entry) = n
- Size in bits of the program counter field = size of the program counter register
- Number of bits for program counter fields = m * (size of program counter register)

Two registers are needed to hold the head an tail pointers of the reorder buffer. When inserting a new entry, the tail is incremented, and when an entry is removed, the head is decremented. This implies the need for a small piece of code to implement those operations.

When an entry in the result shift register needs to locate its corresponding entry in the reorder buffer, tag matching has to be performed. This can be achieved through the use of combinational circuits associated with each entry in the reorder buffer.

In order to enhance the performance of the system, implementing bypass paths is crucial [Smith and Pleszkun 88]. The structure of this method is shown in Figure 5.8.

The combinational circuits are associated with all entries of the reorder buffer. They can be used to compare the operand register designator and the destination register of each entry. When a match is found, the multiplexer is used to route the data to the operand register output latch.

It can be inferred from the above discussion that the cost of implementing the reorder buffer strategy is dominated by hardware costs. Apart from the small code to implement the queue operations and the interrupt handler code itself, hardware accomplishes all the other operations. It would be safe to state that the cost of implementing the reorder buffer strategy is about 90% hardware and 10% software.

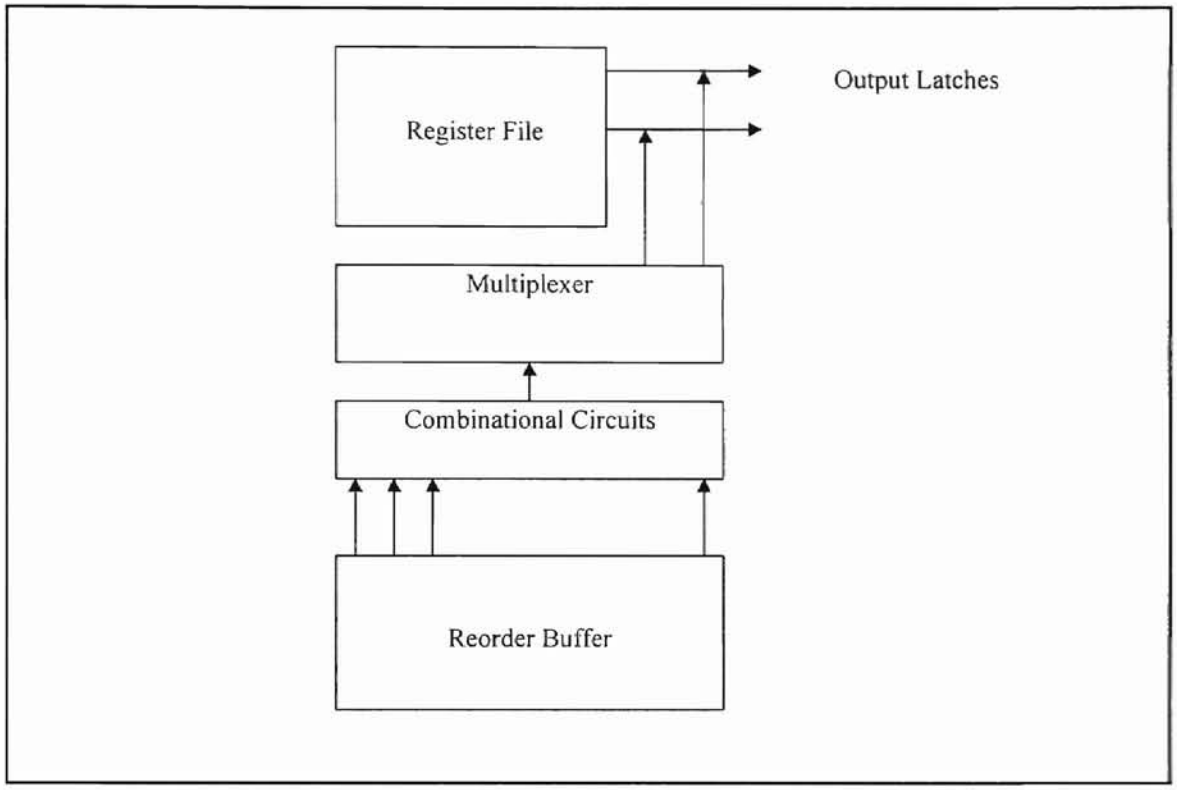


Figure 5.8 Implementation of the Bypass Paths in the Reorder Buffer Strategy

5.5.2 History File

Implementation of the history file and its associated result shift register does not differ greatly from that of the reorder buffer strategy. Comparing Figures 5.2 and 5.6, it is noticeable that the differences are minor.

The result shift register in the history file strategy has an extra field: the destination register. We have, similar to the discussion in Subsection 5.5.1,

- Size in bits of the destination register filed = *log* (number of registers)
- Number of bits for destination register fields = n * *log* (number of registers)

As for the history file, it has a field to hold the old value of a register. This field replaces the field that holds the result of an instruction in the reorder buffer. Since both fields have a size equal to that of a register, the size of this field is the same in both strategies. The history file and the reorder buffer have the same size.

Since the history file strategy allows instructions to modify the state in any order, there is no need for bypass paths as in the reorder buffer. The extra hardware to implement this mechanism is no longer needed. However, in case of interrupts, the history file strategy needs to undo state changes, something not needed in the reorder buffer strategy. To accomplish this, a code segment is needed to loop through the entries of the history file, from tail to head, determining at each entry the destination register and moving the contents of the old value field to replace the current contents of that register. The complexity of such code depends on the size of the history file.

It can be concluded that hardware constitutes the main portion of the cost of implementing the history file strategy. The amount of hardware needed by this strategy is not as much as that used by the reorder buffer strategy, also it uses more software. Therefore, it can be stated that the cost of implementing this strategy is about 75% hardware and 25% software.

5.5.3 Future File

A large portion of the cost of implementing the future file strategy arises from the need to support two register files, the future file and the architectural file. This implies the need for twice the number of registers and related hardware normally needed to implement a register file. Also, since this strategy uses the mechanisms of a reorder buffer strategy to maintain the precise state in the architectural file, all costs of that strategy are also included.

In the case of an interrupt, the values in the future file are replaced by the ones from the architectural file. A code segment, similar to that used in the history file strategy, is needed to traverse the reorder buffer from tail to head, at each entry moving the contents of the destination register from the architectural file to the corresponding register in the future file. Such code will also manage queue operations, but the complexity of such code mainly depends on the size of the reorder buffer.

From the above discussion, it would be safe to state that the cost of implementing the future file strategy is about 80% hardware and 20% software.

CHAPTER VI
COMPARATIVE EVALUATION

Interrupts are a fact of life for computer systems, without them systems cannot function correctly or at least as efficiently. Interrupt processing strategies can be generally classified as overhead operations. And, as an obvious conclusion that can be inferred from this study, they are expensive.

During the course of this study, five interrupt processing strategies were studied. Their designs and implementations were discussed, and based on that their costs were investigated in terms of their hardware and software components. The strategies share some similarities and have a number of differences. The main similarity is that all strategies need to detect the interrupt as their first step. Also all strategies need to run an interrupt handler; and, since strategies that support precise interrupts were studied, the interrupted processes have to be resumed. Hence, these similarities were not offered any special attention in this study.

Table 6.1 provides the conclusion of this study. It provides the cost of each strategy studied in terms of the percentage of hardware and software costs involved. The percentages provided are not exact figures, their main purpose is to provide a predictive look at the decomposition of the total cost, without pinpointing the actual boundary between the hardware and software costs of each strategy. Other categories in the table, mainly present the commonalities and differences among the strategies under consideration.

Strategy	Hardware cost	Software cost	Finish pending instructions	Undo state changes	Save process state	Location of state changes	Memory requirements
IW	approximately 90%	approximately 10%	Only the instructions that have passed their NRP are allowed to complete.	No state changes to be undone	The IW is part of the process state to be saved.	State changes are written to the process state not to the IW.	Extra memory is needed to maintain and save the IW
Check-point Repair	Dominant if the logical spaces (copies of the process state) are implemented physically.	Dominant if the logical spaces (copies of the process state) are stored in secondary storage.	All active instructions are aborted. Instructions between the chosen checkpoint and the interrupted instruction are re-executed.	State changes that occurred between the chosen checkpoint and the interrupted instruction are discarded.	c copies of the process state are saved, where c is the maximum number of simultaneous active checkpoints allowed by the algorithm.	State changes are written to the process state and to any backup copy as determined by the Check operation.	Large amount of memory is needed to maintain all the backup copies of the process state.
Reorder Buffer	approximately 90%	approximately 10%	Active instructions are aborted.	No state changes to be undone.	The reorder buffer is not part of the process state to be saved.	The state changes are written to the reorder buffer. They are later written to the process state.	Space is needed to maintain the result shift register and the reorder buffer.

Table 6.1 Summary of the five interrupt processing strategies discussed in this study (continued on next page)

Strategy	Hardware cost	Software cost	Finish pending instructions	Undo state changes	Save process state	Location of state changes	Memory requirements
History File	approximately 75%	approximately 25%	Active instructions are allowed to complete.	The values of the affected registers are restored using the old values stored in the history file.	The history file is not part of the process state to be saved.	State changes are written directly to the process state. A record of the old values is kept in the history file.	Space is needed to implement the result shift register and the history file.
Future File	approximately 80%	approximately 20%	Active instructions are allowed to complete.	The values of the affected registers in the future file are restored using the values of the corresponding registers in the architectural file.	The process state to be saved consists of the future file.	The state changes are written both to the future file and the reorder buffer. The values in the reorder buffer are later written to the architectural file.	Space is needed to implement two register files, the architectural file and the future file. More space is needed for the result shift register and the reorder buffer.

Table 6.1 Summary of the five interrupt processing strategies discussed in this study

If an ideal strategy were to be designed, it would have to introduce only a minimum time overhead, plus it has to be relatively inexpensive. The main sources of time overhead are the time needed to redo the aborted instructions once the process is resumed, and the time needed to undo the state changes. Running the interrupt handler is also a source of time overhead, but since this is a process that depends on the size of the interrupt handler and cannot be minimized, it requires no further consideration. Hence, to minimize the overhead time, a strategy has to allow all active instructions to complete execution once an interrupt is detected. Also it has to be able to maintain a precise state at all time, hence avoiding to undo any state changes. In the latter case, the operations involved in maintaining a precise state at all time are another source of time overhead.

From the cost point of view, the question arises, what is an inexpensive strategy? The answer to this question involves a large number of parameters, hardware cost, design cost, implementation cost, human resources, etc. Since the designer has access to all the real life details involved in implementing an interrupt processing strategy, the responsibility of determining the answer to this question falls on the designer's shoulders. Hopefully, this study will be a helpful reference for designers to answer this question.

CHAPTER VII
SUMMARY AND FUTURE WORK

The goal of this study was to provide system designers with a document that allows them to roughly predict the cost of several different interrupt processing strategies or of newly devised ones.

This thesis report can help the designers in making an informed choice among the available strategies.

In Chapter II, general information and definitions related to interrupt processing were discussed. This chapter presented different types of interrupts. It also presented an introductory looks at how interrupts are processed on uniprocessors compared to pipelined processors. It also presents the concept that processing an interrupt on a pipelined processor is not a one-step process. It showed that interrupt processing involves six phases (steps), that in each phase certain tasks have to be completed, and that they collectively complete the processing of an interrupt.

Chapter III introduced the first of the interrupt processing strategies discussed in this study: the Instruction window (IW). The design and implementation of the IW was studied, and based on that an analysis of its cost was presented.

Similarly, Chapter IV introduced another strategy, the Checkpoint Repair. Along analogous lines to the previous chapter, its design and implementation were studied, and its cost was analyzed.

Chapter V grouped three interrupt processing strategies together. The three strategies (The Reorder Buffer, History File, and Future File) all depend on a data structure named the result shift register to accomplish their goals. All these strategies were studied and their costs were analyzed.

Chapter VI presented the conclusions of this study. A table of the results of the cost analysis for all the strategies under consideration was presented. In addition, the

characteristics of these strategies were put in perspective. They were presented in order to show the commonalties and differences among these strategies.

Future work to extend this study can address the effects different memory management algorithms can have on interrupt processing. For example, the impact cache implementations and the use of virtual memory addressing can be addressed. Also, since interrupt processing involves switching among processes, the type of scheduling algorithm used can be a significant factor in the cost of interrupt processing that requires further study.

REFRENCES

[Brey 87] Bary B. Brey, *The 8086/8088 Microprocessor: Architecture, Programming, and Interfacing*, Merill Publishing Company, Columbus, OH, 1987.

[Collyer 96] Geof Collyer, "Setting Interrupt Priorities in Software via Interrupt Queueing", *The USENIX Association Computing Systems*, Vol. 9, No. 4, pp. 119-130, Spring 1996.

[Comer and Fossum 88] Douglas Comer and Timothy V. Fossum, *Operating System Design Vol. I: The XINU Approach*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988

[Cragon 96] Harvey G. Cragon, *Memory Systems and Pipelined Processors*, Jones and Bartlett Publishers, Inc., Boston, MA, 1996.

[Hennessy and Patterson 90] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[Hwang 84] Kai Hwang, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.

[Hwu and Patt 87] Wen-Mei Hwu and Yale Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines", *IEEE Transactions on Computers*, Vol. C-36, No. 12, pp. 1496-1514, December 1987.

[Moudgill and Vassiliadis 96] Mayan Moudgill and Stamatis Vassiliadis, "Precise Interrupts", *IEEE Micro*, Vol. 16, No. 1, pp. 58-67, February 1996.

[RISC 94] *RISC System/6000: PowerPC System Architecture Manual*, Morgan Kaufman Publishers, Inc., San Francisco, CA, 1994.

[Smith and Pleszkun 88] James E. Smith and Andrew R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Transactions on Computers*, Vol. C-37, No. 5, pp. 562-573, May 1988.

[Tomasulo 67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. Res. Develop.*, Vol. 11, No. 1, pp. 25-33, January 1967.

[Torng and Day 93] H. C. Torng and Martin Day, "Interrupt Handling for Out-of-Order Execution Processors", *IEEE Transactions on Computers*, Vol. C-42, No.1, pp. 122-127, January 1993.

[Walker 92] Wade A. Walker, "A Taxonomy of Interrupt Processing Strategies in Pipelined Microprocessors", Masters Thesis, University of Texas at Austin, Austin, TX, December 1992.

[Walker and Cragon 95] Wade Walker and Harvey G. Cragon, "Interrupt Processing in Concurrent Processors", *IEEE Computer*, Vol. 28, No. 6, pp. 36-46, June 1995.

[Wang and Emnett 93] Chia J. Wang and Frank Emnett, "Implementing Precise Interruptions in Pipelined RISC Processors", *IEEE Micro*, Vol. 8, No. 4, pp. 36-43, August 1993.

[Weiss and Smith 84] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers", *IEEE Transactions on Computers*, Vol. C-33, No. 11, pp. 1013-1022, November 1984.

APPENDIX A

GLOSSARY

ACTIVE	An instruction that has been issued but has not finished execution yet.
ACTIVE(t)	The set of consecutive checkpoints that have active instructions in their E-repair ranges at time t.
BACKUP	An array of logical spaces holding the potential precise states.
CONSISTENT	Another terminology for a precise state
COUNT	An array of counters holding the number of active instruction in the E-repair ranges of the active checkpoints.
CURRENT	The logical space holding the current machine state.
DATA PYPASS PATHS	A mechanism used by the reorder buffer strategy to allows issuing instructions to read their source operands from the reorder buffer instead of the register file.
E-REPAIR RANGE	A sequence of instructions between a checkpoint and the next checkpoint in a checkpoint repair algorithm.
EXCEPT	An array of flags that indicates the occurrences of interrupts in the E-repair ranges of the active checkpoints.
FUTURE FILE	An interrupt processing strategy that, maintains two register files: architectural file and future file. The architectural file is used to restore a precise state to the future file.
IDENT	A decrementing counter that is used to uniquely identify a checkpoint.
HISTORY FILE	An interrupt processing strategy that, with the help of a result shift register, allows imprecise state to exist, but retain enough information in the history file to retain a precise state.

IW	Instruction Window, an interrupt processing strategy that enables systems, which allow out-of-order execution and completion of instructions, to support precise interrupts
LOGICAL SPACE	A copy of the processor's state.
NRP	No Return Point, a point in the pipeline after which an instruction cannot be aborted.
POTENT(t)	The set of potential precise states at time t.
POTENTIAL PRECISE STATE	A processor's state at one instruction where some of the instructions issued before this instruction are still active.
REORDER BUFFER	An interrupt processing strategy that, with the help of a result shift register, enables the processor to maintain a precise state at all time.
RESULT SHIFT REGISTER	A data structure used by several interrupt processing strategies to to monitor the flow of instruction execution in the pipeline.
VEN	Vector Element Number, a field in the IW that indicates the number of vector elements remaining to be processed by an instruction.

VITA

Loai E. Garalnabi

Candidate for the Degree of

Master of Science

Thesis: HARDWARE/SOFTWARE COST ANALYSIS OF INTERRUPT PROCESSING STRATEGIES

Major Field: Computer Science

Biographical:

Personal Data: Born in Khartoum, Sudan, October 10, 1972, son of Elhadi Garalnabi and Egbal Osman.

Education: Received Bachelor of Science degree in Computer Science from University of Khartoum in January 1994. Completed the requirements for the Master of Science degree at the Computer Science Department, Oklahoma State University in May 1997.

Experience: System Programmer, Computer & Electronics World, U.A.E. August 1994 - November 1994; Lab Consultant, Computing and Information Services, Oklahoma State University, November 1995 - August 1996; Teaching Assistant, Computer Science Department, Oklahoma State University, August 1996 - May 1997.