On Precise Interrupts

Mayan Moudgill† Stamatis Vassiliadis‡

† T.J. Watson Research Center
I.B.M. Corporation
Yorktown Heights, NY

‡ Electrical Engineering Dept.
T.U. Delft
Delft, The Netherlands

Abstract

Interrupts and in particular precise interrupts constitute an integral part of all computer architectures. Implementing precise interrupts can substantially inhibit the performance of computers. To gain some insight into the problem, we divide common interrupts into four classes, and examine the cost of implementing precise interrupts. Two of these classes, external-critical and external-error, can be implemented cheaply on a pipelined processor, with little or no impact on performance. We propose that interrupts be implemented imprecisely, except during debugging, of a third class of interrupts, internal-error interrupts. Finally, we introduced some techniques that can be used to cheaply implement precise interrupts for the fourth class of interrupts, internal-critical interrupts, but may not apply generally. While the central concern is precision, or lack thereof, we also deal with several peripheral issues that arise when implementing interrupts on aggressive implementations. These include sparse restart, which will arise whenever we weaken the requirements for precision on an out-of-order issue processor, and the impact of parallel (e.g., superscalar) issue.

Key Words - precise interrupts, exceptions, traps, interrupt handlers, pipelining, out-of-order issue processors, superscalar processors, instruction level parallel processors.

1 Introduction

Normally, a processor fetches and executes instructions from some body of code, under the control of the program counter for that program. Occasionally, however, this regular execution sequence is *interrupted*, and control is transferred to a piece of code known as the *interrupt handler*, whose purpose is to process the

interrupt. The interrupt handler takes action appropriate to the interrupt, and then, possibly, resumes normal execution of the program.

As an example, consider a low-cost implementation of an architecture[1] that includes an integer divide instruction. The implementation is designed with no integer divide hardware. Instead, it emulates the divide instruction in software, by interrupting normal execution whenever the processor encounters a divide instruction (via an interrupt such as "unimplemented instruction") and transferring control to an interrupt handler. The handler can direct the execution of the instruction to code which performs the divide using implemented instructions such as shifts and add/subtracts.

To properly process an interrupt, an interrupt handler must be able to identify the interrupting instruction, determine what corrective action is to be taken, which registers should be used for input and output, etc. While processing the interrupt, the interrupt handler must be able to modify state associated with the program. Finally, it must be possible, when desired, to resume normal execution after processing the interrupt. The hardware must provide mechanisms that enable an interrupt handler to accomplish all these tasks. The way by which most processors make it possible for interrupt handlers to accomplish all these functions is by implementing *precise interrupts* [9, 3].

The definition of a precise interrupt is derived from execution on a sequential architecture. In a sequential architecture, instructions are issued serially. An instruction is allowed to run to completion before the next one is issued. If any instruction interrupts, the interrupt is reported immediately. An interrupt on some implementation is said to be precise if the machine state at the interrupt is reported is identical to the state that would have been obtained if the implementation had been sequential. In other words:

1. All instructions issued before the interrupting instruction would have completed.
2. No instruction issued after the interrupting instruction has been issued.
3. The program counter points to the interrupting instruction.

This state is known as a *precise state*. If the state at the time the interrupt handler is entered is precise, the interrupt is said to be a *precise interrupt*. If all interrupts of an implementation are precise, it is said to implement the *precise interrupt model*.

[1] The words architecture and implementation are often used interchangeably: in this paper, an architecture is a specification of the behavior and logical structure of the processor, while an implementation is the way the hardware captures this specification. There may be more than one way of capturing in hardware the behavior/logical structure of a processor –these correspond to different implementations.

It turns out that one of the major mechanisms used to improve processor performance, pipelining, interferes with the ability to handle precise interrupts. Techniques that implement precise interrupts on pipelined processors either use a large amount of extra hardware, or cause a loss of performance, or both. In this paper, we shall investigate ways to implement precise interrupts, focusing on techniques that trade completeness for cheaper or faster implementations. In the process, we investigate the following questions:

- Can some interrupts be implemented precisely, yet still avoid the performance and/or hardware penalty?
- Which interrupts are essential for machine operation? Conversely, which interrupts can be implemented imprecisely, without impairing the ability of the machine to run "correct" programs?
- What benefit can be gained from discarding precision for some interrupts? In particular, since the rest of the interrupts must be implemented precisely, will the implementation still have to incur a similar performance/hardware cost?

To gain some insight into the precise interrupt problem, we create a taxonomy of interrupt that divides interrupts into four classes. We investigate the answers to the questions posed above for each of these classes. It turns out that it is simple to implement precise interrupts for two of the classes, one class can possibly be ignored, and there exist techniques for implementing the remaining class of interrupts at a fairly low cost.

The presentation is organized as follows: First, we review and classify interrupts, and the interrupt handling actions required to support them. We then introduce *interrupt models*—specifications of the machine state at the time an interrupt handler is entered, and of the mechanism to resume normal execution. We outline the actions an interrupt handler would be required to perform in general, and deduce from it the minimal interrupt state specification such that it is possible to write such handlers. Then, we show how the precise interrupt model, captures that minimal state specification. In the next section, we introduce pipelining, and describe implementations of precise interrupts for pipelined processors. We then consider how the precise interrupt model can be implemented on a pipelined processor for each of the classes in our taxonomy. In particular, we emphasize the implementation of those interrupts that are necessary for machine operation. Finally, we deal with miscellaneous issues in implementing interrupts, including the interaction with superscalar issue, and dealing with restart on out-of-order issue processors.

2 Interrupts

There are many different kinds of interrupts, which vary from architecture to architecture. Of these, there is a subset, most of which are present on all processors. Some of these common interrupts, the situations in which they occur, and the actions the corresponding handlers normally perform are listed below.

Timer The program has executed for some pre-determined time. This interrupt is usually used to implement time sharing, in which case the interrupt handler executes a context switch and starts executing some other program, or for performance measurement, in which case the interrupt handler performs some book-keeping action like updating a counter.

I/O An I/O device has completed some task, such as reading from disk. Interrupt handler actions may include copying data from or to I/O buffers, scheduling a new I/O request for that device, moving an process from an I/O wait to ready queue, or context switching to the process whose I/O request has completed.

Hardware Fault There is a fault in the hardware. This may cause all programs in execution to be aborted and some kind of diagnostic utility to be entered.

Virtual Memory Some memory location being accessed by an instruction is not resident in the TLB or, alternatively, is on disk. In the first case, the interrupt handler loads the requisite TLB entry and resumes execution of the interrupting memory access instruction. In the second case, the handler schedules the memory location to be paged in. It will usually context switch to some other process while waiting for the I/O to complete.

Unimplemented Instruction Some instruction in the architecture is not implemented in hardware. Instead, the handler must emulate the instruction using other instructions that are implemented.

Exception The execution of an instruction of the program resulted in an error, e.g., divide by zero, underflow, overflow. The appropriate corrective action must be taken. This will sometimes involve setting the result to a predetermined value. Alternatively, the handler may abort the program, possibly after printing out an error message and trace data.

2.1 Interrupts, Exceptions and Traps

The terminology for interrupts is rather confusing: they are referred to, interchangeably, as interrupts, exceptions and traps in the literature. In this paper, we shall exclusively use interrupts to refer to all interrupts. We shall reserve exceptions for those interrupts that are caused by invalid inputs to or erroneous computations of the fixed and floating point units.

There is another kind of processor behavior that is sometimes called an interrupt. Typically, on an architecture with several layers of privileges, when the interrupt handler is entered, the privilege level is reset to allow the interrupt handler code the maximum privileges. Many operating systems take advantage of this fact to enable a program to request privileged services. For instance, when an program wishes to do I/O, it may set up various registers values and then force an interrupt (by using a special instruction) of the appropriate kind. The interrupt handler examines the registers, validates the request, and schedules the appropriate I/O. In some sense, this is a function call to an operating system subroutine, which uses an interrupt instead of the regular function call interface because it needs to change privilege levels. This kind of interrupt is part of the program, instead of being unexpected like the other interrupts. An "interrupt" used this way is usually called a *trap*.

It is possible to implement traps exactly as one does external interrupts, or as a special kind of branch. The first case is subsumed by our treatment of external interrupts, while in the second case a trap should not be treated as an interrupt at all. Therefore, we shall ignore traps in the rest of the paper.

2.2 Classification

Interrupts can be divided into different classes. The interrupts in these classes, described below, arise under different circumstances, and are of differing importance. These divisions are central to our discussion on interrupts.

We roughly divide interrupts into two categories based on the source—those directly caused by the execution of some instruction of the program, and those caused by some agency other than an instruction of the program. We call the first category *internal* interrupts, and the second *external* interrupts. Exceptions, unimplemented instruction and virtual memory interrupts are internal interrupts, while timer, I/O, and hardware interrupts are external interrupts.

Another categorization of interrupts can be done on the basis of use. First, interrupts are used by the processor to report an erroneous condition and cause the program to enter some kind of recovery routine.

	Critical	Error
External	Timer I/O	Hardware Fault
Internal	Virtual Memory Unimplemented Instruction	Exception

Table 1: A Taxonomy of Interrupts

The default recovery routine typically either ignores the interrupt, or sets the output to a default value such as 0 or NaN, or aborts the program. However, it can be arbitrarily complex. Exceptions belong to this category. Second, interrupts are used to communicate with the operating system. Some of the ways the interrupts we consider interact with the operating system are:

- I/O interrupts allow I/O devices to asynchronously interrupt programs, and thereby avoid the overhead of having programs continuously poll the I/O device status.
- Timer interrupts allow the operating system to preempt programs, and thus share resources.
- Virtual memory interrupts, such as TLB-misses or page faults, cause the operating system to load the appropriate TLB entry or bring in the appropriate page to memory.

We call those interrupts whose primary use is for error handling *error* interrupts, and those whose primary use is for communicating with the operating system *critical* interrupts.

Table 1 shows how the interrupts can be divided based on our classification. As we will show later, all interrupts in a quadrant impose similar kinds of restrictions on the hardware. Further, each quadrant differs from the others in how it effects the implementation of interrupts.

3 Interrupt Models

Normally, control flows through a program under the control of the program counter. One consequence of this regular flow of control is that the compiler (or programmer) knows what values are in which register, which instructions generated those values, how control could have reached those instructions, etc. Unlike a normal program, an interrupt handler has no way of anticipating where it will be called from and under what conditions it will be invoked. However, the interrupt handler may still need to access information about the interrupted program to properly process the interrupt. The architecture must specify the assumptions an

interrupt handler can make about when it will be invoked, and what guarantees the machine state will satisfy at that point. This state is known as the *interrupt state* specification.

Further, when the interrupt handler has completed processing, it must be able to transfer control back to the program with as little disruption as possible. The architecture must define some *restart mechanism* which will allow the interrupt handler to resume normal execution. The interrupt state specification and the restart mechanism together define the *interrupt model* for the architecture.

In this section, we first deduce the minimal set of assumptions the architecture must guarantee so that it is possible, in general, to write an interrupt handler. We consider one interrupt model, the *precise interrupt* model, and show that it exactly captures these assumptions.

3.1 The General Interrupt Handler

After a precise interrupt, control transfers to the interrupt handler for that interrupt, which performs certain actions. This section considers the actions an interrupt handler *must* be able to accomplish, in general. >From these, we deduce the interrupt state that must be presented by an implementation to an interrupt handler so that it is able to correctly process the interrupt. As shall be shown later, internal interrupts are more complex to handle than external interrupts. This discussion assumes that the interrupt handler is for an exception, on a pipelined implementation in which, at the point where an interrupt is reported, there are potentially instructions preceding the interrupting instruction that are still in execution, and instructions succeeding the interrupting instruction that have completed.

When an interrupt handler is entered, it may need to read various register and memory values to determine the precise cause of the interrupt and decide on the appropriate corrective action. Instructions issued after the interrupting instruction should not have modified this state. For instance, if a floating point multiply caused an exponent overflow, then the handler may need to read the input values to the multiply. Therefore, no subsequent instruction should have modified them. Since, in general, there is no way of determining exactly what values are needed by an interrupt handler, a safe assumption would be that when an interrupt handler is entered, the state should not have been modified by any instruction issued after the interrupting instruction.

Further, it is necessary to wait until all prior instructions have completed—to ensure that none of them interrupt. If any of these preceding instructions did interrupt, then some other interrupt handler should be executed first. That handler may then alter the state so that the subsequent interrupt would not occur. In

the absence of some mechanism to "speculatively" execute handler code, it is necessary to wait until it is known that all prior instructions will not interrupt.

So far we have stated that an interrupt handler should be able to inspect the state unmodified by any instructions issued after the excepting instructions. This could be implemented by having two copies of the state—the in-order state and the future state. The *in-order* state provides the state required by the interrupt handler for the interrupting instruction. The *future* state is the most recent machine state, potentially containing modifications by instructions issued after the excepting instruction. At a first glance, it would seem that it would then be unnecessary to abort all instructions issued after the excepting instruction. After running the interrupt handler (using the in-order state), execution could resume where it had left off on detecting the interrupt (using the future state).

However, consider the case where the interrupt handler modified some registers or memory. The interrupt handler expects that instructions issued after the excepting instruction would execute in the modified state. This is illustrated using the following code fragment:

```
(1) r3 := r1 * r2
(2) r4 := r1 + r5
```

Assume that `(1)` had an exception, but that `(2)` had completed before the exception is reported. Further, assume the handler modified `r1`. Now, a reasonable expectation is that `(2)` should be re-executed using the new value of `r1`. One way of implementing this would be for the interrupt handler to take the responsibility for determining and re-executing such completed subsequent instructions in the modified state. A simpler method is to always abort instructions issued after the excepting instruction, and re-execute them after interrupt processing.

Thus, in the general case, cleanly supporting interrupt processing requires that the interrupt state satisfy the following requirements:

1. All instructions issued before the excepting instruction should be complete before we enter the interrupt handler.
2. The state should be as if no instruction issued after the excepting instruction issued.
3. The address of the excepting instruction must be available to the interrupt handler.

We have not yet discussed a restart mechanism. If the interrupt state satisfies the conditions above, there is an obvious restart mechanism—after processing the interrupt, branch to either the interrupting instruction

(and re-execute it in the new state, in which it should not cause an interrupt), or the succeeding instruction.

Notice that the *precise interrupt model* conditions are identical to those deduced from the requirements of the general interrupt handler. Thus, it is should be possible to implement general-purpose interrupt handlers on any implementation that satisfies the precise interrupt model.

4 Pipelining

The precise interrupt model can be trivially implemented on a non-pipelined implementation of a sequential architecture. However, modern processors are pipelined to improve performance. The pipelining complicates the implementation of precise interrupts. Consider the following code fragment:

```
(1) r3 := r1 * r4
(2) r4 := r1 + r5
(3) r6 := r4 * r8
```

Assume that the instructions are floating point, and that a multiply takes 9 cycles to complete, while an add takes 5, of which 6 and 2 cycles respectively are spent in the floating point unit. Also, assume that results are bypassed. As shown in Fig. 1(a), on a sequential architecture, the three instructions take 23 cycles. By pipelining, the execution of the various instructions can be overlapped, allowing the code fragment to be completed in 12 cycles (see Fig. 1(b)). Notice that (3) uses the result of (2), and so must wait an extra cycle for the result to become available.

Pipelining creates a problem. In the example in Fig. 1(b), instruction `(2)` completes before the multiply. If `(1)` were to cause an exponent overflow in its 8th cycle, then at the time the interrupt handler were entered it would contain modifications caused by an instruction issued after the interrupting instruction—a violation of the precise interrupt model. The phenomenon whereby the instructions of a program complete in an order different from their order in the program is known as *out-of-order* completion. An implementation which supports out-of-order completion will always have situations where instructions issued after an interrupting instruction have completed. To implement precise interrupts, there must exist some mechanism that can undo the effects of these subsequent instructions.

We shall consider several mechanisms for implementing precise interrupts on pipelined implementations. As indicated above, the main source of difficulty is the order of completion, not the order of issue. For simplicity, unless otherwise stated, we assume that the implementations will issue at most one instruction

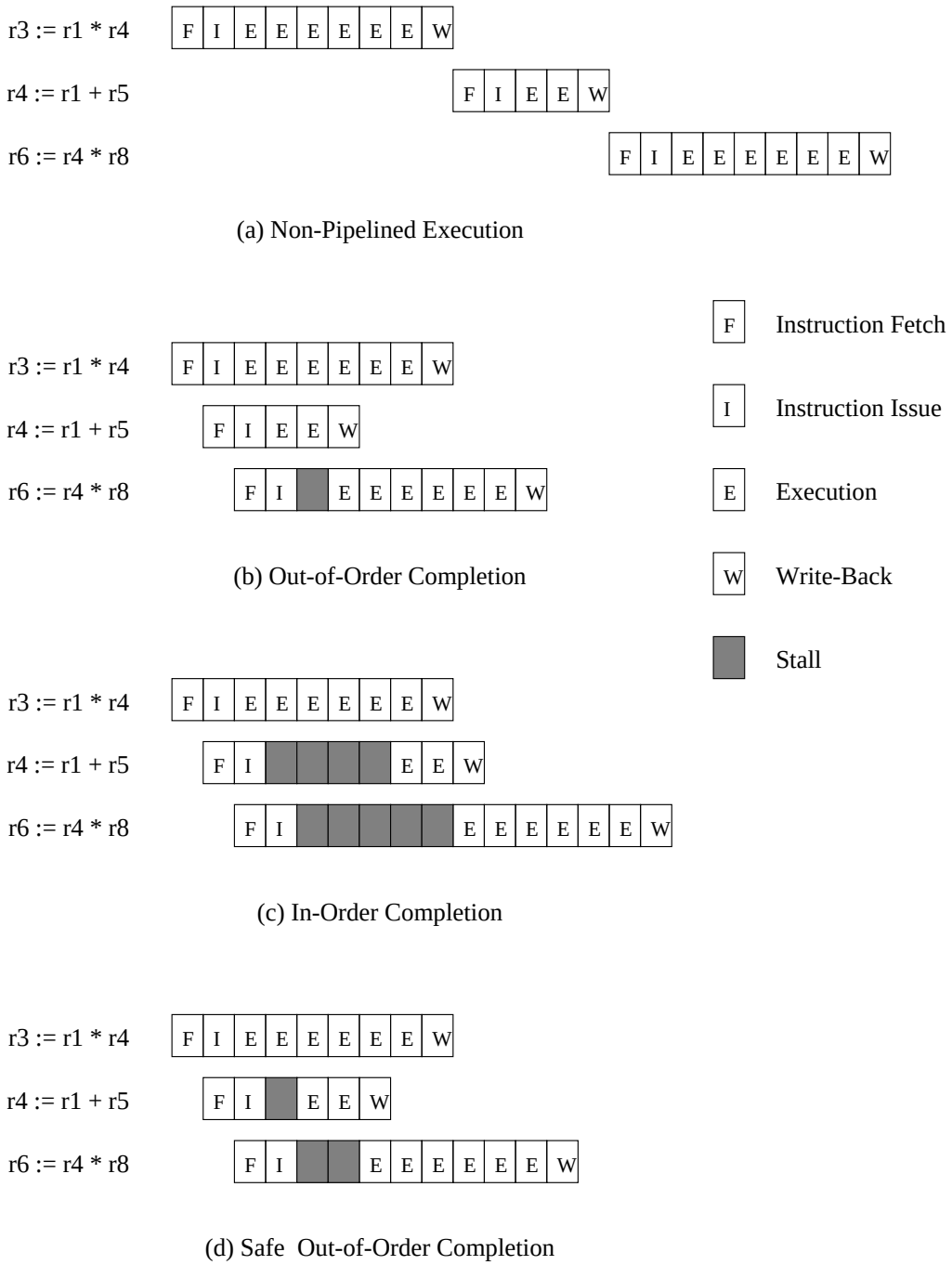


Figure 1: Instruction Execution

at a time. Further, we shall assume that the instructions are issued in-order, i.e. in the same order as they occur in the program.

4.1 In-order Completion

Clearly, if instructions completed in the same order that they issued, an interrupting instruction could be handled by allowing the instruction to reach its last pipeline stage, and then squashing the completion of all subsequent instructions from the program. It is easy to verify that this scheme will guarantee a precise state. A similar scheme that forced in-order completion was used in the original MIPS implementation [8].

However, forcing in-order completion can result in performance degradation, as can be seen in Fig. 1(c). It takes 16 cycles to execute the three instruction sequence, because the shorter add has to idle for 4 cycles, just to ensure that it completes after the multiply.

Of course, the only reason that the add has to wait for the multiply is that the multiply might cause an interrupt. Now, if it could be proved that the multiply would not cause an interrupt, the add could finish out-of-order with respect to the multiply. Assume that the only interrupts that could be caused by the multiply are exponent overflow and exponent underflow. It is guaranteed that a multiply will not overflow if the sum of the exponents of the multiplicands is one less than the maximum representable. Similarly, for underflow.[2] An implementation could add hardware that would check for these guarantees, and if it found that some instruction satisfied the guarantees, allow subsequent instructions to complete out-of-order with respect to that instruction. Otherwise, it would fall back on in-order completion. This could result in the speedup shown in Fig. 1(d): 13 cycles (assuming guaranteed not to interrupt). If an interrupt were possible, the hardware could fall back on in-order completion, taking 16 cycles. The Pentium implements this optimization for several of its floating point instructions [1].

4.2 Out-of-Order Completion

As mentioned earlier, to implement precise interrupts in an implementation with out-of-order completion requires mechanisms that can undo the effects of the various instructions. One mechanism that accomplishes this is the *history buffer*. This is the mechanism used to implement precise interrupts in the MC88110 [11].

The history buffer is a FIFO queue. Every time an instruction is fetched, it is allocated a slot in the bottom of the history buffer. When the instruction completes, the value in the register it overwrites (i.e., the old

[2]The conditions are sufficient but not necessary; even if they are not satisfied an overflow (or underflow) may not occur.

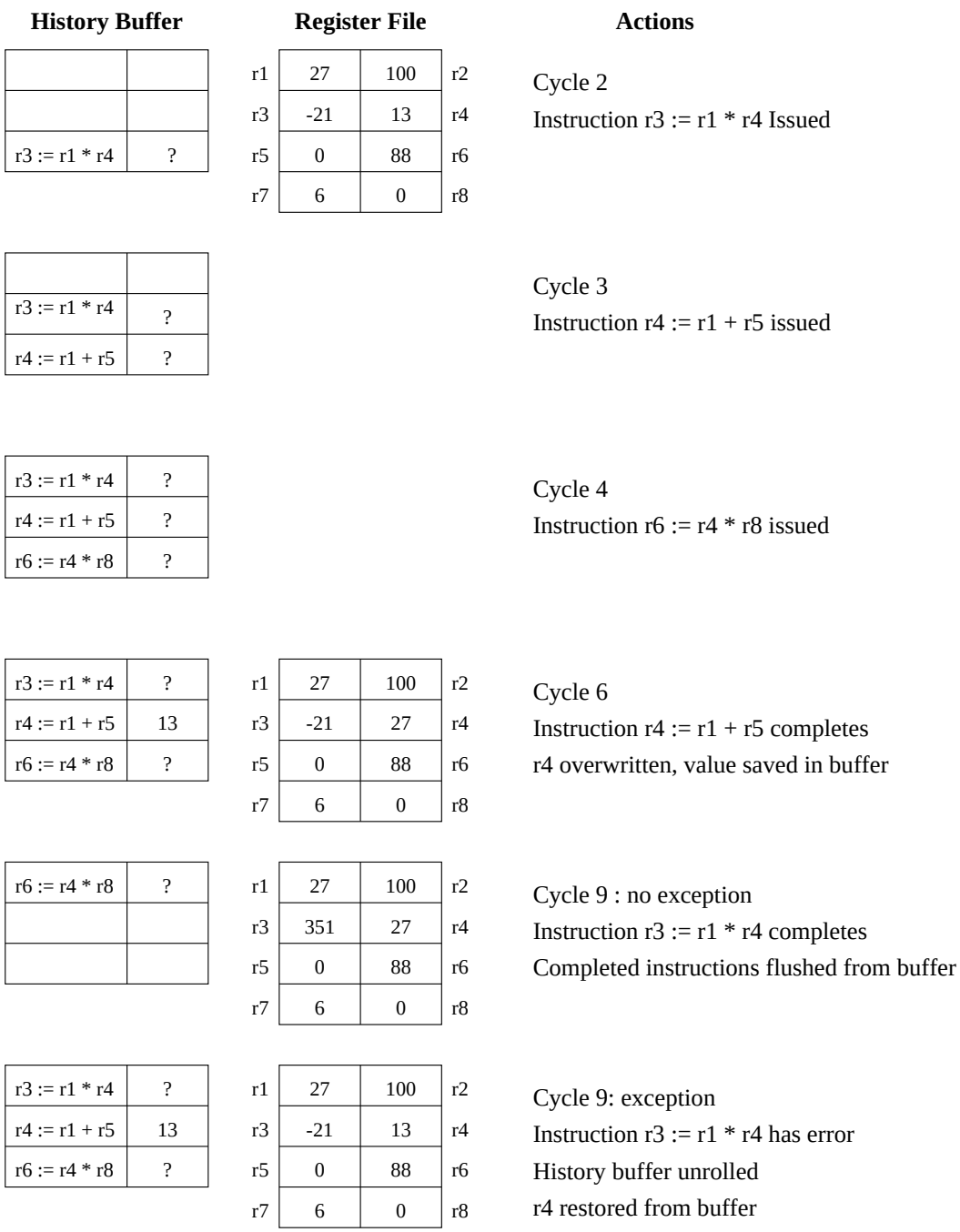


Figure 2: History Buffer

value) is preserved in the slot allocated to the instruction. Instructions are removed from the top of the history buffer as they complete. Any interrupt caused by an instruction is reported only when it gets to the top of the history buffer. The precise register state is restored by using the original values of the registers preserved in the history buffer. The simplest way to accomplish this is by "unwinding" the history buffer, i.e., by going through the history buffer bottom-up, and restoring the old register value of any instruction that completed and modified a register.

Fig. 2 illustrates the behavior of a history buffer, and shows how it could be used to recover the precise register state. This example shows three instructions being issued, one after another. As each instruction is issued, it is added to the bottom of the history buffer. In the example, the instruction issued at cycle 3 completes and updates its output register, `r4` at cycle 6. The old value of `r4`, `13`, is saved in the history buffer for the instruction. Note that the instruction issued earlier, at cycle 2, is still executing when this happens—an example of out-of-order completion. Finally, this instruction completes. If it completes successfully, the top two instructions from the history buffer can be retired, since they are both complete. On the other hand, if it causes an interrupt, then the history buffer is used to restore the state to what it was when that instruction was issued; i.e., the state is restored to the state at cycle 2. This is accomplished in the following manner: first the bottom-most instruction is examined. Since it has not yet completed, the next lowest instruction is examined. This instruction has completed. Therefore the saved value, `13`, is restored to the register which was overwritten by this instruction, `r4`. Then the next instruction in the buffer is examined, and so on. When all instructions in the buffer have been processed, the register file has been restored to the required, precise, state.

Most of the out-of-order completion, precise interrupt schemes described in the literature, such as the future-file, in-order buffer and reorder buffer mechanisms [9] are based on the idea of keeping around multiple copies of any overwritten register. The precise state is recovered by the discarding all values written after the interrupting instruction, and restoring the register state from the remaining values. They basically differ in implementation cost and the time taken to restore the precise register state. These tradeoffs are examined in [12].

An alternative to being able to recover the precise state at *any* instruction is checkpoint-retry [7]. In this scheme, as execution proceeds, certain instructions are chosen as checkpoints. The implementation ensures that there is always at least one valid checkpoint, and that it is possible to recover the precise state at a valid checkpoint. At an interrupt, the machine state is restored to the most recent valid checkpoint. Then execution is resumed from the checkpoint, this time sequentially, till the interrupting instruction is re-encountered.

The interrupt is then reported. The interrupt state at this point is precise.

Consider the example in Fig. 3: the checkpoint is established just before the first instruction issued. Each instruction is executed normally, updating the register file when it completes (as in cycle 5), potentially out-of-order (as in cycle 6). When the second instruction causes an interrupt, instead of the state being directly restored to the precise state for the second instruction, execution resumes at the checkpoint. This is accomplished by restoring the register file from the checkpointed state, and resuming execution at the instruction following the checkpoint. Thus, the first instruction is (re-)executed, with pipelining turned off. When control reaches the interrupting instruction again, the register state is the required, precise, state.

Instructions, e.g. stores, can modify memory, as well as registers. It would be much more difficult to undo memory operations, and so a different mechanism is used to support precise memory state. An operation which modifies memory, instead of writing directly to memory, initially writes to a *store buffer*. Any subsequent load instructions will look in the store buffer as well as the cache for the value to be loaded, with an address match in the store buffer taking precedence over a cache hit. A buffered store is released to memory only after all instructions preceding the store have completed.

5 Optimizing the Implementation

Implementing precise interrupts through in-order completion leads to performance degradation, since it reduces the amount of pipelining possible. Implementing precise interrupts and out-of-order completion requires a significant amount of hardware. Worse, the extra hardware can add to the cycle time of the machine, resulting in performance degradation. To gain some insight on how we can reduce the cost of implementing interrupt handling in the presence of out-of-order completions, we shall consider the requirements of the different classes of interrupts.

External-critical interrupts can be implemented, cheaply and efficiently, by simply halting all further instruction issue once an external interrupt has been detected, and waiting for the pipeline to drain before entering the interrupt handler. This results in low-cost precise external-critical interrupts.

External-error interrupts may be implemented exactly as external-critical interrupts. However, consider the case of a hardware fault—it may not be possible to drain the pipeline after such an interrupt. Moreover, a precise interrupt state may not even be the desired state after a hardware fault; the desired response could be to freeze the machine state at the point the interrupt is reported, so that an service processor can diagnose the cause of the problem. In either of these case, it is possible to implement external-error interrupts at low

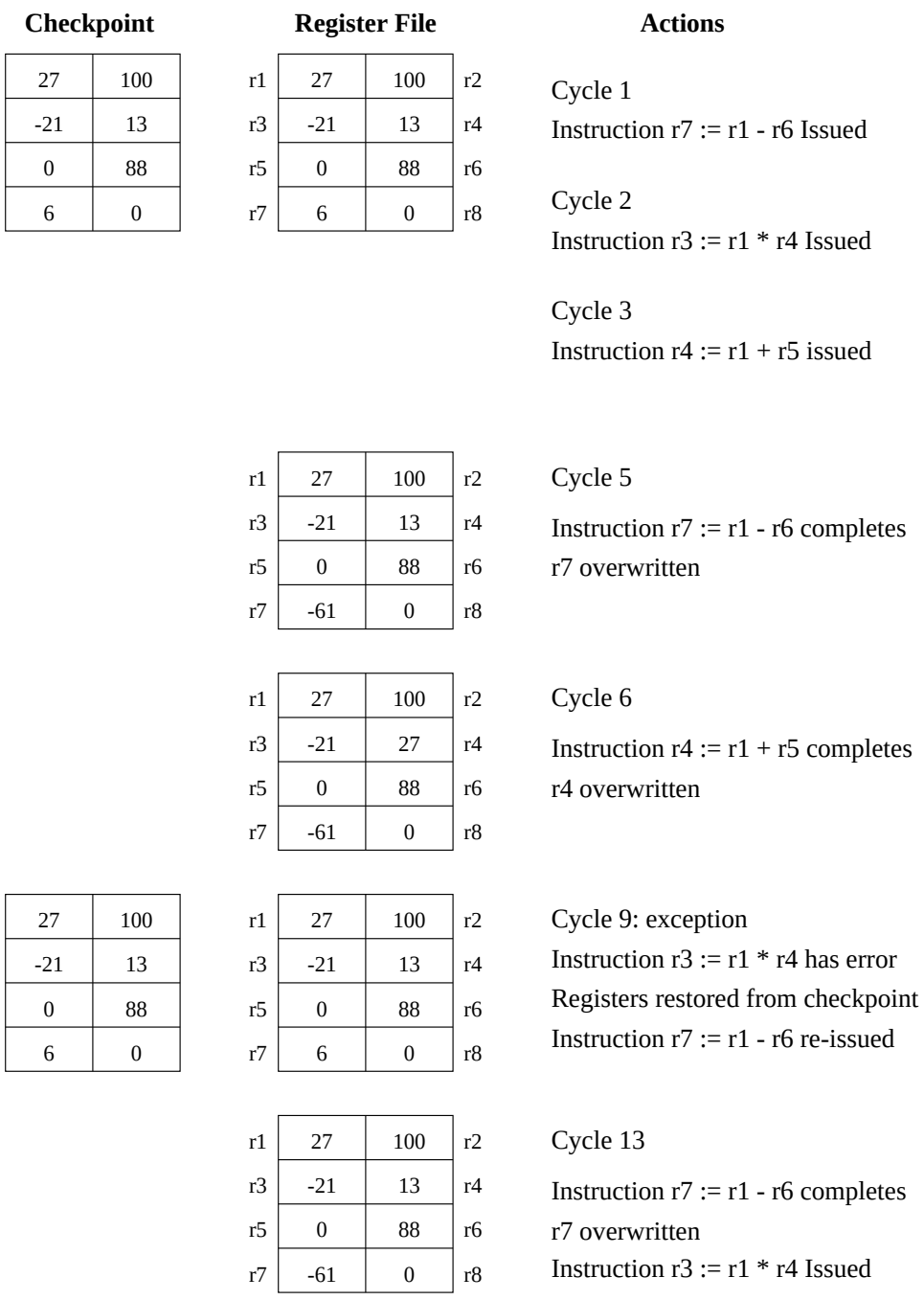


Figure 3: Checkpoint-Retry

cost, by draining or freezing the pipeline.

There exist more ambitious implementations which try to work-around some hardware faults, possibly using retry, so that "soft", i.e., intermittent, hardware faults are handled in a manner transparent to the user. On such machines, hardware fault handlers are complicated to implement. They share features with internal-critical interrupts, and perhaps should be classified with them. Of course, additional logic[3] is required to retry instructions after a fault and, if the fault persists after repeated retries, to report it as a "hard", i.e., uncorrectable, failure.

>From the previous discussion, it appears that external interrupts (excluding machines that retry hardware faults) can be implemented efficiently and cheaply. The expensive mechanisms, discussed above, appear to be required only for the internal interrupts. It is clear that internal-critical interrupts such as virtual memory must be implemented somehow, since they are necessary for running any program. But what about internal-error interrupts?

An internal-error interrupt, by definition, should only arise in the run of an erroneous program, either because of unanticipated data, or because the program itself is wrong. If the program is being run in any mode other than debugging, it is quite likely that the action taken by the interrupt handler will be to abort the program, or to ignore the interrupt. If the desired behavior is different from that provided by the hardware, e.g., if the hardware ignores an interrupt, and the desired behavior is that the interrupt should be reported, the burden is on the programmer to insert checks for data in the program that may cause interrupts, and to add appropriate "handling" code. This is illustrated in the example in Fig 4, where the program contains code that will ensure that the division cannot cause a divide-by-zero interrupt. Furthermore, instead of relying on the interrupt handler to fix-up the potential divide by zero, the program contains code, denoted as "repair code", to do so. In general, such code can be simple, merely reporting the violation (divide by zero in this case) and then exiting the program, or as sophisticated, including code to scale appropriate variables, and thereby work around the problem.

[3] and possibly microcode

```
if( b == 0 ) {
   /* repair code */
   z = ...
}
else {
   z = a/b;
}
```

Figure 4: Protected Code

Note that implementations having imprecise internal-error interrupts can invoke handlers, which can then report the interrupt; however, since the interrupts are imprecise, the handler can, in general, take no recovery action. It will either resume program execution without modifying the machine state, possibly after logging the error, or abort execution.

Of course, there are situations where it is necessary to have precise internal-error exceptions, principally while debugging. In that case, there must be some mechanism for implementing precise interrupts, possibly at the cost of performance. One solution is to turn off all pipelining. This can be justified on the basis that, during debugging, performance is not critical; further, the code is compiled in a manner that maintains the original (source-program) order, thereby inhibiting pipelining, and the further loss of pipelining due to precise interrupts does not result in a significant additional performance degradation.

The viewpoint adopted here is that internal-error interrupts are not necessary for the operability of the machine. Thus, if not implementing these interrupts can lead to a more efficient implementation, the hardware should not implement them. Instead, the burden of anticipating and dealing with errors can be left to the programmer, if necessary.

The only interrupts that remain to be discussed are internal-critical interrupts. As mentioned above, it is necessary to implement these interrupts, since they are required for the functioning of even "correct" programs. For example, in a virtual memory machine, if the processor cannot correctly handle virtual memory interrupts, it becomes impossible to execute *any* program.

If the processor chooses to implement only internal-critical interrupts precisely, several optimizations are possible. One of the techniques discussed above could be used, optimized to handle only internal-critical interrupts. For instance consider modifying in-order completion to implement only virtual memory interrupts. In this case instructions would not be allowed to complete out-of-order with respect to memory

operations, but would be allowed to complete out-of-order with respect to other operations.

Such approaches may still have some, though smaller, impact on the performance of the processor. It turns out that there are certain properties of internal-critical interrupts that can be exploited to allow these interrupts to be implemented efficiently. These result in different techniques, which are discussed in the next three sections. These techniques are illustrated using virtual-memory interrupts, since they are the most common internal-critical interrupt; however they are applicable to all internal-critical interrupts.

6 Direct Implementation

The simplest way of dealing with internal-critical interrupts is to integrate the design of the interrupt-handler with the design of the hardware. Such an interrupt-handler will use an interface completely different from that used by other interrupt handlers, and will probably have implementation-specific knowledge encoded into it. It is unlikely that such an interrupt handler will be portable across implementations. The extreme example of this approach would be to implement the handler (such as the handler for a TLB-miss) in microcode.

A different approach, based on similar reasoning, is that used in the CYBER 200 [2] to implement virtual memory handlers. At a virtual memory interrupt, the entire state of the processor, including the machine-dependent state information of partially completed instructions, is saved in an *invisible exchange package*. After processing the interrupt, the machine is restarted from this information. The partially completed instructions, in particular, are restarted from the point at which they were interrupted. The reason why this approach is adequate is that virtual memory handler is guaranteed, for reasons discussed below, to never alter any of the inputs to the instructions in execution. Therefore, unlike the case of a general interrupt handler, there is no reason to abort the partially completed instructions, and they can be allowed to completion.

This approach suffers from several drawbacks. First, it must be possible to "freeze", save, and restore partially completed operations. This implies that there must be paths to the intermediate latches of the pipeline, so that they can be saved. Second, the interrupt handler must be aware of the number of intermediate latches. Thus, if the implementation of the processor changes, it may force a re-write of the interrupt handler. The interrupt handler is, of course, non-portable. These drawbacks are not present in the schemes discussed below.

7 Restricting Precision

In some sense, the "natural" interrupt state to report is that derived by the following sequence: when some instruction reports an interrupt, the processor allows all instructions issued before the last *completed* instruction, as well as those issued before the interrupting instruction to complete. Such an interrupt is known as an *imprecise* exception. The main benefit of imprecise exceptions is that they avoid the overhead required to restore precise state.

Note that if some instruction issued after the interrupting instruction has completed by the time the interrupt is detected, then all instructions between the interrupting instruction and the last completed instruction will run to completion. This is necessary for restarting. By completing all instructions till the last completed instruction, execution can be resumed after that instruction.

Another point to note: some imprecise interrupts are guaranteed to be precise. The fact that an instruction will interrupt will be determined in the nth cycle of execution of the instruction. If this n is no larger than the number of cycles take to execute the shortest instruction, then that interrupt for that instruction is precise—it is impossible for any instruction issued after the interrupting instruction to have completed, so only those instructions issued before the interrupting instruction will run to completion.

More concretely, assume that the shortest instruction in the architecture is an add, which takes 4 cycles to complete. Assume that a load takes 7 cycles. Now, consider a TLB miss on the load in the following code sequence:

```
(1) r3 := ld r4
(2) r4 := r1 + r5
```

If the TLB miss is detected in the 4th cycle, then the add will still be in execution, and so it is possible to squash it, thereby obtaining a precise state. If the TLB miss is detected in the 6th cycle, however, then the add will have completed. If it is detected on the 5th cycle, it becomes a little complicated to handle. The add will be in the process of writing its results to the register file. Depending on the implementation, it may be possible to intercept the write, generating a precise state.

Thus, internal-critical interrupts can be implemented precisely yet cheaply if they satisfy the condition mentioned above: i.e. are detected in a time less than that taken by the shortest instruction to complete. As it turns out, the common internal-critical interrupts satisfy this condition. An unimplemented instruction can be detected while the the instruction is being decoded, which is typically in the second cycle of the pipeline.

A TLB-miss is more of a problem, but it is still possible to implement it so that a TLB-miss can be detected in the first execute stage of the pipeline. This technique is the one adopted by the RS/6000 FPU [6] and the Alpha [5].

8 Discarding Precise Interrupts

One constraint the technique described in the previous section imposes on the implementation of internal-critical interrupts is that they be detected early. This makes implementation of operations such as load register indexed difficult. The register indexed load first adds two register values together, and then looks up the TLB. Yet, it must be implemented using the same number of cycles as an add operation, which simply adds two registers together. Another drawback is that this technique makes it almost impossible to implement interrupt handlers for interrupts that are detected late such as exponent underflow. One suggested use of the underflow interrupt is to implement denormalization by using an interrupt handler that is called when an floating point number is smaller than than the smallest normal floating point number. Instead, with technique outlined above, this denormalization must be implemented in the hardware.

Imprecise interrupts make the writing of interrupt handlers difficult because they allow instructions that were issued after the interrupting instruction to complete and thereby overwrite some value needed by the interrupt handler to recover from the interrupt. Typically, the only values needed by an interrupt handler will be the inputs to the interrupting instruction, and possibly status or control register values.

The other reason for aborting instructions that were issued after the interrupting instruction, in general, is that, if the interrupt handler modified the state, it may be necessary to re-execute the subsequent instructions in the modified state. Again, a typical interrupt handler, e.g. the denormal handler described above, will modify only a restricted portion of the machine state: the output register of the instruction. However, no instruction that used the output register value of the interrupting instruction could have completed; if such an instruction were encountered, the processor would cease to issue instructions till the value became available.[4] So, in an in-order issue pipelined implementation, an interrupt handler that modifies only the output register of the interrupting instruction does not need to re-execute instructions issued after the interrupting instruction.

These observations suggest an alternative interrupt handler interface, suitable for an implementation with

[4]Nor could any instruction that modified the output register value be issued; even in the absence of an interrupt, that would have resulted in the output register being updated in the wrong order.

imprecise interrupts. When an instruction interrupts, the interrupt mechanism provides the interrupt handler the input values to the interrupting instructions and some control information, as well as the addresses of the interrupting instruction and the last completed instruction. The interrupt handler may only safely modify the output register of the interrupting instruction, and will resume execution after the last completed instruction. A similar approach is used by the ROMP processor, described in [4], to handle imprecise memory interrupts.

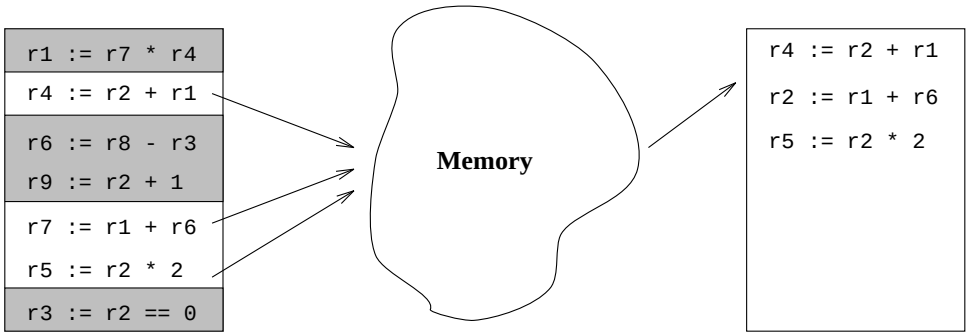
The approach of augmenting the interrupt hardware to provide the inputs to the interrupt handler will allow implementation of handlers for the internal-critical interrupts, e.g. unimplemented instructions and virtual memory, even though the machine state at the point interrupts are reported is imprecise. The interrupt handlers for these categories of interrupts read only the input values to the interrupting instruction, and modify only the output register. Also, the scheme does not impose any constraints on how early an exception must be reported. Most interrupt handlers for exceptions that can only be detected late, including the interrupt handler which uses exponent underflow to implement denormalized numbers, can be implemented using this interface. Thus, not only does this technique allow all but internal-error interrupts to be implemented cheaply, it also allows most internal-error interrupt handlers to be implemented.

9 Miscellaneous

Sparse Restart So far a single address has been sufficient to resume normal execution. This address can be the address of the interrupting instruction or that of the last completed instruction. After processing the interrupt the handler resumes normal execution by branching to the provided address, or to the next address.

Using a single address to determine the restart point requires that *all* instructions prior to that address have completed, and that none of the instructions of the instructions after the restart address have been completed. We shall cause such a situation a *dense restart*. In a *sparse restart*, there is some address prior to which all instructions have completed, and another (different) address past which no instruction has completed. The instructions in between the two addresses contain uncompleted instructions intermingled with completed instructions. Of course, all restarts are dense on an architecture with precise interrupts; by definition, none of the instructions after the interrupting instruction can have completed, and all prior instructions must have.

While implementing imprecise interrupts, sparse restarts can arise when an implementation issues instructions out-of-order. In fact, it is difficult to avoid sparse interrupts. Even on implementations where instructions are issued in-order, special care must be taken to ensure a dense restart state. For example, as shown above when defining an imprecise exception, it may be necessary to allow instructions that have been



(a) Before Interrupt (b) After Interrupt

Figure 5: Sparse Restart

issued after the interrupting instruction, but have not completed at the time the interrupt is detected, to run to completion, because there is some even later instruction that has already completed.

To illustrate how sparse restarts may cause a problem, examine the code fragment in Fig. 5. When the interrupt handler is entered, shaded instructions have all been issued, while the unshaded ones have not. To properly resume execution after interrupt handling, the unshaded instructions must be re-executed. Some mechanism that can selectively execute these instructions is necessary.

The restart mechanism illustrated in Fig. 5 is that suggested by [10]. In this approach, it is assumed that the implementation issues instructions out of an instruction window. On an interrupt, those instructions in the instruction window that have not completed are saved (as is the address of the next instruction to be fetched into the window). Normal instruction execution is resumed by reloading the instruction window (and the fetch address) from the saved state. Note that only the previously unexecuted instructions have been reloaded. Now when instruction resumes, none of the previously-executed instructions will get reissued.

Alternative mechanisms are possible. One possibility is to have a “mask” register. Before an instruction is issued, the leading bit in the mask is examined. If it is 1, the instruction is suppressed. Either way, the mask is shifted, with a 0 being shifted in. The mask is loadable. On an interrupt, the interrupt handler will be provided the address of the first uncompleted instruction, and a vector of all completed instructions after that point. To resume, the handler jumps to the given address after loading the mask register with vector.

The exact mechanism used will probably depend on the processor design. Presumably, the sparse restart mechanism will be chosen so that it meshes with other aspects of the design. For instance, if the implementation does not use a instruction window, it would be impossible to use the restart mechanism described

above.

As indicated earlier, a general interrupt handler can modify arbitrary state, and thereby force re-execution of those instructions that appeared the interrupting instruction in the program, but were executed before, or in parallel with, the interrupting instruction. Assume that the interrupting instruction in Fig. 5 is the second instruction, `r4 := r2 + r1`. Now, assume that the interrupt handler modified `r2`. In that case, all instructions that appear after the interrupting instruction and use `r2` should be reevaluated using the new value of `r2`. For instance, in the example, the fourth instruction, `r9 := r2 + 1`, must be re-executed, as must the last instruction in the window, `tt r3 := r2 == 0`; this, however, will not happen with the proposed sparse restart mechanisms. Thus, if a precise restart mechanism is being used, the interrupt handlers either must be careful while modifying register values (other than those of the output register of the interrupting instruction), or it must ensure that all effected instructions are re-executed.

Parallel Issue Superscalar and VLIW implementations issue more than one instruction per cycle. One obvious complication that arises is that when an instruction causes an interrupt, it becomes necessary to determine which of the instructions that were issued in parallel actually preceded the interrupting instruction, and therefore should be allowed to run to completion. On a VLIW, one reasonable way to circumvent this problem is to interrupt at the instruction "packet" (i.e. collection of instructions issued simultaneously) boundary, instead of at any particular instructions; thus, if any one instruction in a packet interrupts, the packet as a whole is deemed to have interrupted.

There is another problem created by the ability to issue more than one instruction at a time. This is best illustrated with an example—consider the case where the shortest instruction on an architecture takes 4 cycles. Then, on a single issue implementation, it is guaranteed that any instruction will complete at least 5 cycles after the previous instruction; 4 for executing the instruction, plus 1 since it is issued one cycle later. On a parallel issue implementation, this time is reduced by 1, since the instruction could be issued in the same cycle as the previous instruction. This impacts techniques such as those described above that implement precise memory interrupts in the presence of imprecise interrupts. Interrupts may now need to be detected one cycle earlier than on a single issue architecture.

Recursive Interrupts The interrupt handler may itself cause an interrupt. There is some state in the processor associated with the interrupt, such as the cause of the interrupt and the return address. If the second interrupt is processed immediately, this state will be overwritten, with possibly disastrous consequences.

Usually, when an interrupt handler is first entered, all further interrupts are blocked. The first thing the interrupt handler does is save the interrupt state. It then allows other interrupts to be passed to it. The interrupt handler must ensure that it causes no other interrupts while it is saving the state, such as memory faults.

Multiple Interrupts It is possible to have several interrupts arrive simultaneously—for instance, a timer interrupt and an exception. Interrupts are usually prioritized, and the relevant interrupt handlers must be invoked according to the priority of the interrupt. One possible technique is to first issue the least important interrupt. As soon as the interrupt handler turned off blocking (see previous paragraph), the next least important interrupt is issued, interrupting the current interrupt handler, and so on. This way, the most important interrupt gets completely processed first.

Memory Interrupts One restriction that the interrupt mechanism *must* ensure is that the interrupt handler for virtual memory interrupts (such as TLB-miss or page-not-in-memory) must not themselves cause an interrupt of the same kind. It would not be possible to recover from this disastrous recursion. There are two ways of avoiding this situation: real-mode or locking. In real-mode, all memory references by interrupt handlers use real addresses, and therefore do not pass through the virtual memory subsystem. In locking, the TLB-entry for the interrupt handler code and data pages are locked in the TLB, and the pages themselves are locked in main memory.

10 Conclusions

High-performance processors gain a large part of their performance by pipelining operations. Unfortunately, implementing precise interrupts can inhibit the benefits of pipelining. The major cause of this interference arises from the fact that exploiting pipelining fully introduces out-of-order completion. To implement precise interrupts, instructions must appear to complete in-order. This can either be achieved by implementing in-order completion, and thereby limiting pipelining, or by adding hardware to undo effects of instructions that complete out-of-order with respect to the interrupting instruction. The last solution can exploit pipelining fully, but it can use a large amount of hardware. Further, it may still decrease performance, by increasing cycle times.

To gain some insight into the problems, we divided common interrupts into four classes, and examined the cost of implementing precise interrupts for them. Two of these classes, external-critical and external-error,

can be implemented cheaply on a pipelined processor, with little or no impact on performance. We propose that interrupts be implemented imprecisely, except during debugging, of a third class of interrupts, internal-error interrupts. Finally, we introduced some techniques that can be used to cheaply implement precise interrupts for the fourth class of interrupts, internal-critical interrupts, but may not apply generally.

We believe that separating interrupts into classes and treating every class separately depending on the constraints of the design is the correct approach. This perception has been echoed in several recent microprocessor designs [6, 5]. In particular, since the general interrupt handler interface embodied in the precise interrupt model is incompatible with aggressive pipelined implementations of processors, it is necessary to implement precise interrupts only when necessary or easy, and otherwise adopt weaker, less general, techniques for interrupt handling. However, it is possible that the technique adopted in these designs, that of preserving precise interrupts for all but internal-error interrupts is still too restrictive. The alternative mechanism, of augmenting the architecture so that it preserves the inputs to the interrupting instruction and provides them to the handler, may provide more flexibility, both to the processor designer, and the interrupt handler programmer.

The subject of the paper is implementing interrupts on pipelined processors. While the central concern was precision, or lack thereof, we also deal with several peripheral issues that arise when implementing interrupts on aggressive implementations. These include sparse restart, which will arise whenever we weaken the requirements for precision on an out-of-order issue processor, and the impact of parallel (e.g., superscalar) issue. Other problems, which arise even on less aggressive processor designs, include dealing with multiple simultaneous interrupts and recursive interrupts.

References

[1] D. Alpert and D. Avnon. Architecture of the pentium microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[2] Control Data Corporation. *CDC CYBER 200 Model 205 Computer System Hardware Reference Manula*. Arden Hills, MN, 1981.

[3] International Business Machines Corporation. *IBM System/370 Extended Architecture Principles of Operation*. Poughkeepsie, NY, 1983.

[4] International Business Machines Corporation. *IBM RT PC Hardware Technical Reference*. 1986.

[5] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992.

[6] G.F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[7] W.M. Hwu and Y.N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, June 1987.

[8] S.A. Przybylski, T.R. Gross, J.L. Hennessy, N.P. Jouppi, and C. Rowen. Organization and VLSI implementation of MIPS. *Journal of VLSI and Computer Systems*, 1(2):170–284, Fall 1984.

[9] J.E. Smith and A.R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[10] H.C. Torng and M. Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, January 1993.

[11] N. Ullah and M. Holle. The MC88110 implementation of precise exceptions in a superscalar architecture. *Computer Architecture News*, 21(1):15–25, March 1993.

[12] C.-J. Wang and F. Emmett. Precise interruptions in RISC pipelines. *IEEE Micro*, 13(4):36–43, August 1993.