Precise Interrupts

Mayan Moudgill
IBM T.J. Watson Research Center

Stamatis Vassiliadis
Delft University of Technology

Can we implement interrupts precisely yet avoid performance and/or hardware penalties? We focus on techniques that trade completeness for less expensive or faster implementations.

Normally, a processor fetches and executes instructions from a body of code under the control of the program counter. Occasionally, however, something interrupts the regular execution sequence, and control transfers to a piece of code known as the interrupt handler, whose purpose is to process the interrupt. The interrupt handler takes appropriate action, and then, possibly, allows normal execution to resume.

As an example, consider a low-cost implementation of an architecture that includes an integer divide instruction. (Writers often use the words *architecture* and *implementation* interchangeably. In this article, an architecture specifies the processor's behavior and logical structure; an implementation is the way the hardware captures the specification. There may be more than one way of capturing a processor's behavior and logical structure—that is, there may be different implementations of the same architecture.) The low-cost implementation uses no integer divide hardware. Instead, whenever the processor encounters a divide instruction, it emulates the instruction in software by interrupting normal execution (via an interrupt such as an unimplemented instruction) and transferring control to an interrupt handler. The handler directs the instruction's execution to code that performs the divide using implemented instructions such as shifts and add/subtracts.

To properly process an interrupt, an interrupt handler must identify the interrupting instruction, determine the corrective action, and determine which registers should be used for input and output. While processing the interrupt, the handler must modify the state associated with the program. Finally, after processing the interrupt, it must cue the processor to resume normal execution if appropriate. The hardware must provide mechanisms that enable the interrupt handler to accomplish all these tasks. Most processors do this by implementing precise interrupts.[1, 2]

The definition of a precise interrupt reflects execution in a sequential architecture. In a sequential architecture, instructions issue serially. An instruction runs to completion before the next one issues. When an instruction interrupts, processor hardware immediately transfers control to the interrupt handler. We call an interrupt precise if the machine state at the time of the interrupt is identical to the state that would exist if the implementation were sequential. This state, known as precise state, meets the following conditions:

- All instructions that issued prior to the interrupting instruction have completed.
- No instruction has issued after the interrupting instruction.
- The program counter points to the interrupting instruction.

If all an implementation's interrupts are precise, we say that it follows the precise-interrupt model.

Unfortunately, pipelining, an important mechanism for improving processor performance, interferes with the processor's ability to handle precise interrupts. Techniques that implement precise interrupts on pipelined processors use a large amount of extra hardware, diminish performance, or both. Here, we investigate ways to implement precise interrupts, focusing on techniques that trade completeness for less expensive or faster implementations. To gain some insight into the problem, we create a taxonomy that divides interrupts into four classes. For each class we ask the following questions:

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

We will show that implementing precise interrupts for two of our four classes is simple, one class can possibly be ignored, and techniques exist for implementing the remaining class at a fairly low cost.

Interrupts

There are many types of interrupts, and architectures vary in the interrupts they exhibit. But a subset is present in virtually all processors. The following are the most common interrupts, the situations in which they occur, and the actions the corresponding handlers normally perform:

- *Timer interrupt*. The program has executed for a predetermined time. Timer interrupts usually implement time sharing, for which the interrupt handler executes a context switch to another program, and performance measurement, for which the handler performs a bookkeeping action such as updating a counter.
- *I/O interrupt*. An I/O device has completed a task, such as reading from disk. Possible interrupt handler actions include copying data from or to I/O buffers, scheduling a new I/O request for the device, moving a process from an I/O wait queue to a ready queue, or context switching to the process whose I/O request has completed.
- *Hardware fault*. A fault in the hardware may cause all executing programs to abort and a diagnostic utility to take over.
- *Virtual-memory interrupt*. A memory location being accessed by an instruction is not resident in the translation look-aside buffer (TLB) or, alternatively, is on disk. In the first case, the interrupt handler loads the requisite TLB entry and resumes execution of the interrupting instruction. In the second case, the handler schedules the memory location to be paged in. It usually context switches to another process while waiting for I/O to complete.
- *Unimplemented instruction*. An instruction in the architecture is not implemented in hardware. Instead, the handler must emulate the instruction, using instructions that are implemented.
- *Exception*. Instruction execution results in an error, such as divide by zero, underflow, or overflow. The appropriate corrective action sometimes involves setting the result to a predetermined value. Alternatively, the handler may abort the program, possibly after printing out an error message and trace data.

Interrupts, exceptions, and traps. The terminology for interrupts is confusing; the literature refers interchangeably to interrupts, exceptions, and traps. Here, *interrupt* denotes all interrupts. We reserve the term *exceptions* for interrupts caused by invalid inputs to or erroneous computations of the fixed- and floating-point units.

A trap is another processor behavior sometimes called an interrupt. Typically, in an architecture with several layers of privileges, when control enters the interrupt handler, the privilege level changes to allow maximum privileges to the interrupt handler code. Many operating systems exploit this fact to enable a program to request privileged services. For instance, when a program wishes to perform I/O, it sets up various register values and then uses a special instruction to force an appropriate interrupt. The interrupt handler examines the registers, validates the request, and schedules the appropriate I/O. In a sense, this is a function call to an operating system subroutine, but the program uses an interrupt instead of the regular function call interface because it needs to change privilege levels. Unlike other interrupts, which are unexpected, this interrupt, or trap, is part of the program.

We can implement a trap as an external interrupt or as a special kind of branch. Our treatment of external interrupts covers the first case; in the second case, a trap is not treated as an interrupt at all. Therefore, we ignore traps in the rest of the article.

Classification. Our division of interrupts into four classes is central to our discussion. Figure 1 illustrates the classification. All interrupts in a quadrant impose similar hardware restrictions. Further, each quadrant differs from the others in implementation effects.

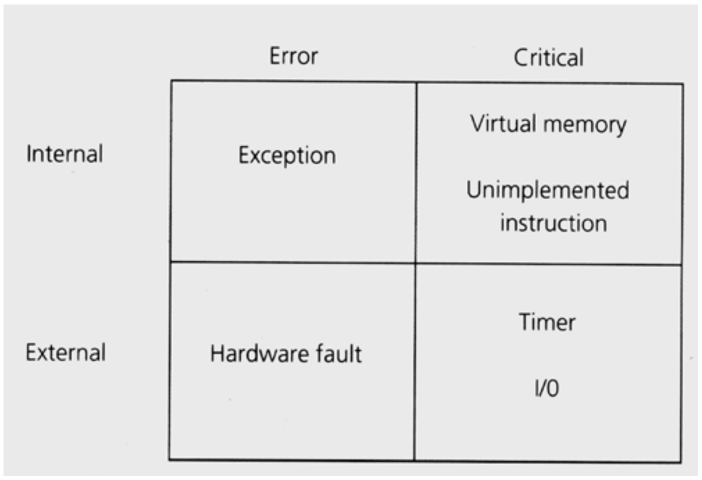


Figure 1. Interrupt taxonomy.

First, we divide interrupts into two categories based on their source: internal interrupts, directly caused by the execution of a program instruction, and external interrupts, caused by an agency other than a program instruction.

We also categorize interrupts according to use. First, the processor uses error interrupts to report an erroneous condition and cause the program to enter a recovery routine. The default recovery routine typically ignores the interrupt, sets the output to a default value such as 0 or *NaN*, or aborts the program. However, the recovery routine can be arbitrarily complex.

Second, the processor uses critical interrupts to communicate with the operating system. For example:

- I/O interrupts allow I/O devices to interrupt programs asynchronously, thereby avoiding the overhead of having programs continuously poll 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 the appropriate page into memory.

Interrupt models

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

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

General interrupt handler. Now we consider the actions an interrupt handler must perform after a precise interrupt. From these, we deduce the interrupt state an implementation must present to the interrupt handler so that it can correctly process the interrupt. In this discussion we assume an interrupt handler for an exception on a pipelined implementation. We also assume that when the exception occurs, preceding instructions may still be executing, and succeeding instructions may have completed.

When an interrupt handler takes control, it may need to read various register and memory values to determine the cause of the interrupt and decide on the appropriate corrective action. For instance, if a floating-point multiply caused an exponent overflow, the handler may need to read the multiply's input values. Since we cannot determine exactly what values an interrupt handler needs, a safe assumption is that when the handler takes over, instructions issued after the interrupting instruction should not have modified the state.

Further, interrupt handling must wait until all prior instructions have completed—to ensure that none interrupt. If a preceding instruction does interrupt, another interrupt handler should execute first. That handler may alter the state so that the subsequent interrupt cannot occur.

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

- All instructions that issue before the excepting instruction should be complete before control enters the interrupt handler.
- The state should appear as it would if no instruction issued after the excepting instruction.
- The address of the excepting instruction must be available to the interrupt handler.

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

The characteristics listed earlier for the precise-interrupt model are identical to the requirements of the general interrupt handler. Thus, implementing general-purpose interrupt handlers is possible on any processor that uses the precise-interrupt model.

Pipelining

We can implement the precise-interrupt model simply on a nonpipelined, sequential-architecture implementation. But modern processors use pipelining to improve performance, complicating implementation of precise interrupts. Consider the following:

1. r3 := r1 * r4
2. r4 := r1 + r5
3. r6 := r4 * r8

Assume that these are floating-point instructions. Assume also that a multiply takes nine cycles to complete, while an add takes five, spending six and two cycles respectively in the floating-point unit. Finally, assume that results are available in the cycle following the final execution stage. As Figure 2a shows, on a sequential architecture the three instructions take 23 cycles. By pipelining, we can overlap execution of the various instructions, allowing the code fragment to complete in 12 cycles (see Figure 2b). Notice that instruction 3 uses the result of 2 and so must wait an extra cycle for the result to become available.

Pipelining creates a problem. In Figure 2b, instruction 2 completes before the multiply. Suppose that instruction 1 causes an exponent overflow in its eighth cycle. When the interrupt handler takes control, the state will contain modifications caused by an instruction following the interrupting instruction—a violation of the precise-interrupt model. The completion of instructions in an order different from their program order is known as out-of-order completion. Out-of-order completion gives rise to situations in which subsequent instructions (that is, instructions issued after an interrupting instruction) have completed. Implementing precise interrupts requires a mechanism to undo the effects of subsequent instructions.

We will consider several mechanisms for implementing precise interrupts on pipelined implementations. As we have indicated, the main source of difficulty is the order of completion, not the order of issue. For simplicity, unless otherwise stated, we assume implementations that issue one instruction at a time. Further, we assume that instructions issue in order—that is, in their program order.

In-order completion. Clearly, if instructions completed in the order they issued, we could handle an interrupting instruction by allowing it to reach its last pipeline stage and then preventing the completion of all subsequent instructions. This scheme guarantees precise state. The original Mips implementation used a similar scheme.[3]

However, forcing in-order completion can degrade performance, as Figure 2c shows. Executing the three instructions takes 16 cycles because the shorter add must idle for four cycles to ensure that it completes after the multiply.

Of course, the only reason the add must wait for the multiply is that the multiply might cause an interrupt. If we could prove 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 the multiply could cause are exponent overflow and underflow. We can guarantee that a multiply will not overflow if the sum of the multiplicands' exponents is one less than the maximum representable. A similar guarantee holds for underflow. (These conditions are sufficient but not necessary; even if they are not satisfied, an overflow or underflow may not occur.)

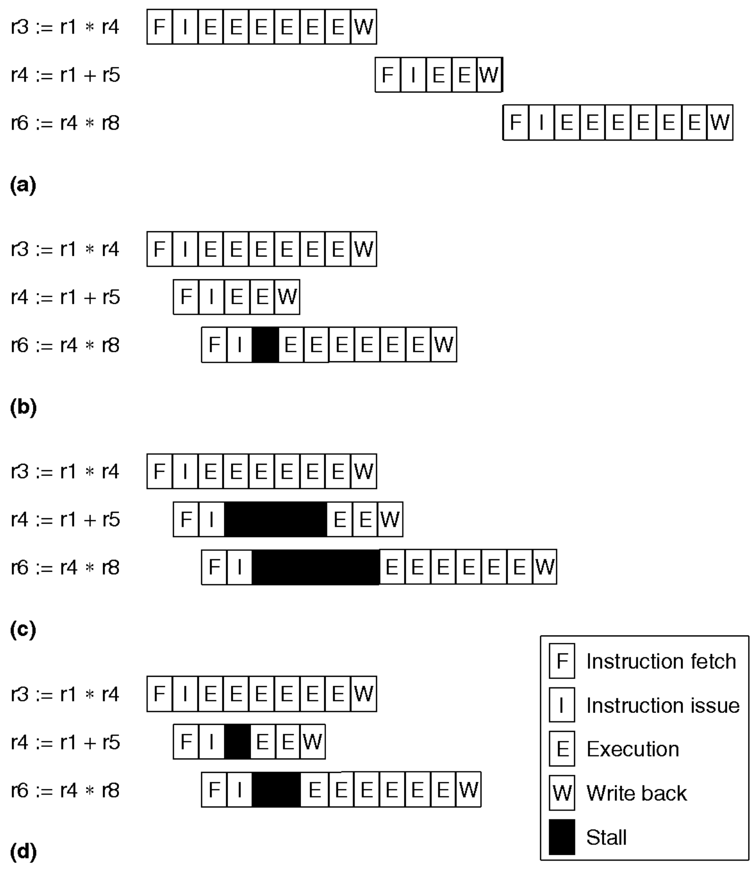


Figure 2. Instruction execution: nonpipelined (a), out-of-order completion (b), in-order completion (c), and safe out-of-order completion (d).

We can add hardware that checks for these guarantees. If it finds that an instruction satisfies the guarantees, it allows subsequent instructions to complete out of order with respect to that instruction. Otherwise, the implementation falls back on in-order completion. This scheme could produce the 13-cycle speedup shown in Figure 2d (assuming a guarantee not to interrupt). If an interrupt were possible, the hardware would use in-order completion, taking 16 cycles. The Pentium uses this optimization for several of its floating-point instructions.[4]

Out-of-order completion. One mechanism that can undo the effects of instructions that complete after an interrupt is the history buffer. This is the mechanism the MC88110[5] uses to implement precise interrupts.

The history buffer is a FIFO queue. Every time the processor fetches an instruction, it allocates the instruction a slot at the bottom of the history buffer. When the instruction completes, the register value it overwrites (the old value) is preserved in the instruction's allocated slot. Instructions leave the top of the history buffer as they complete. An instruction can interrupt only when it reaches the top of the history buffer. The processor restores the precise register state by using the original register values preserved in the history buffer.

Figure 3 illustrates the use of a history buffer to recover the precise register state. This example shows three sequential instructions. As each instruction issues, it enters the bottom of the history buffer. The instruction issued at cycle 3 completes and updates its output register, r4, at cycle 6. The buffer saves 13, the old value of r4. The instruction issued earlier, at cycle 2, is still executing when this happens—an example of out-of-order completion. Finally, the first instruction completes. If it completes successfully, the top two instructions retire from the history buffer, since they are both complete.

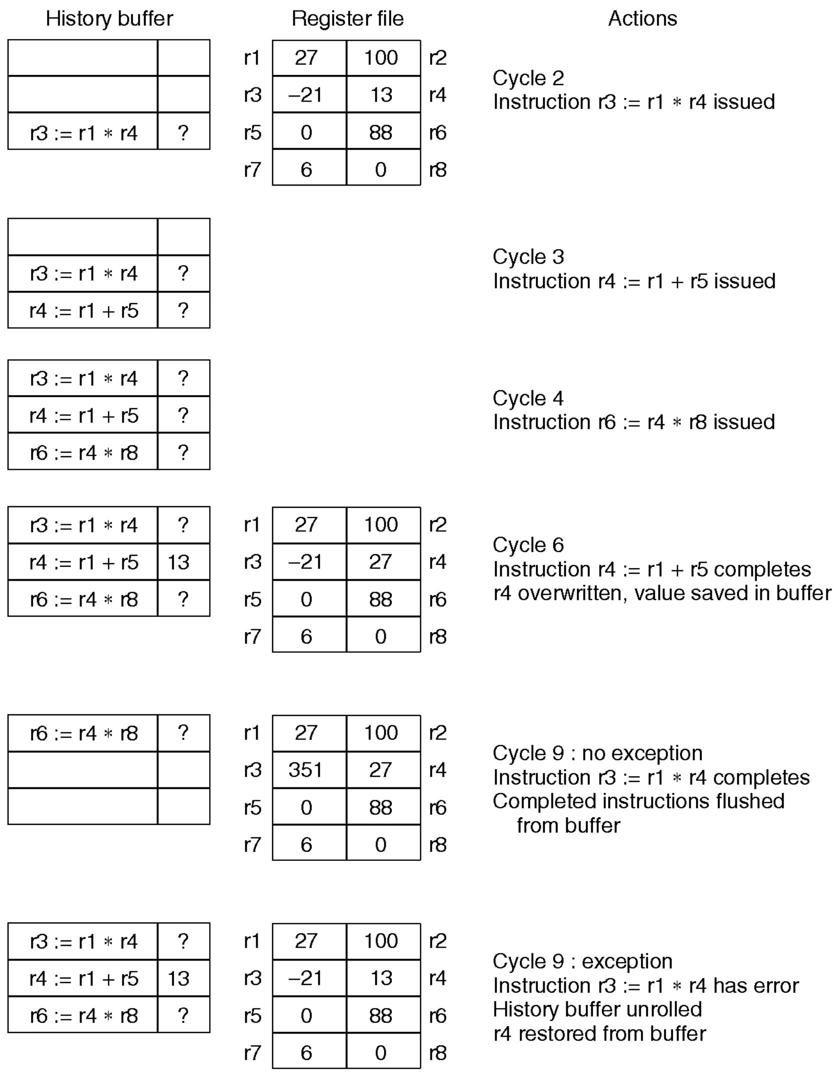


Figure 3. History buffer.

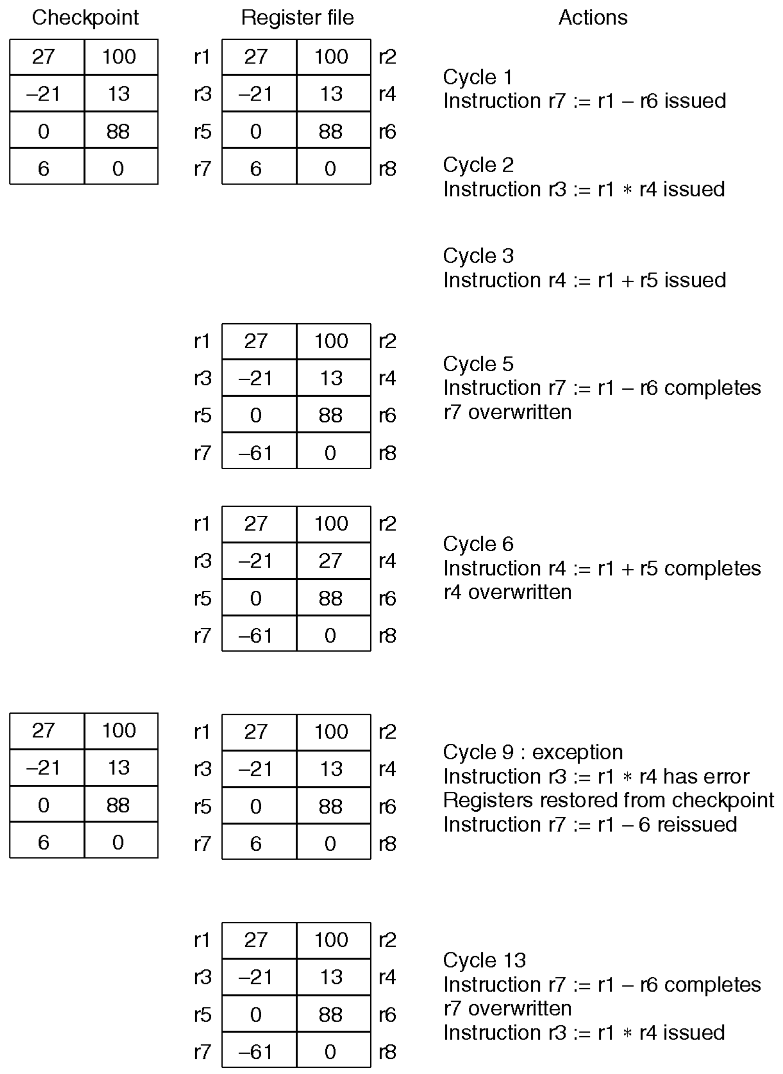
On the other hand, if the instruction issued at cycle 2 causes an exception, the processor uses the history buffer to restore the state when that instruction issued—in other words, the state at cycle 2. The appropriate processor logic accomplishes this as follows: First, it examines the bottommost instruction. Since that instruction has not yet completed, it examines the next lowest instruction. That instruction has completed. Therefore, the logic restores the saved value, 13, to r4, the register overwritten by the completed instruction. Then, the logic examines the next instruction in the buffer. Processing all instructions in the buffer restores the register file to the required precise state.

Most out-of-order-completion, precise-interrupt schemes described in the literature, such as the future file, in-order buffer, and reorder buffer,[1] incorporate the idea of keeping multiple copies of any overwritten register. These mechanisms recover precise state by discarding all values written after the interrupting instruction and restoring the register state from the remaining values. They differ in implementation cost and time needed to restore precise state. Wang and Emmett[6] examine these trade-offs in detail.

An alternative to recovering precise state at any instruction is checkpoint retry.[7] In this scheme, as execution proceeds, hardware chooses certain instructions as checkpoints. The implementation ensures that at least one valid checkpoint always exists and that recovering precise state at a valid checkpoint is possible. At an interrupt, the machine returns to the state of the most recent valid checkpoint. Then execution resumes from the checkpoint, this time sequentially, until it reencounters the interrupting instruction. The interrupt state at this point is precise.

In the example in Figure 4, the hardware establishes a checkpoint just before the first instruction issues. Each instruction executes 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 directly restoring precise state for the second instruction, the processor resumes execution at the checkpoint. It accomplishes this by restoring the register file from the checkpointed state and resuming execution at the instruction following the checkpoint. Thus, the first instruction re-executes with pipelining turned off. When control reaches the interrupting instruction again, the register state is precise.

Instructions such as stores can modify memory as well as registers. Undoing memory operations is much more difficult, so we use a different mechanism to support precise memory state. An operation that modifies memory, instead of writing directly to memory, initially writes to a store buffer. Subsequent load instructions 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. The buffer releases a store to memory only after all instructions preceding the store have completed.



Checkpoint
Register file
Actions
27 100 / –21 13 / 0 88 / 6 0
r1 27 100 r2 / r3 –21 13 r4 / r5 0 88 r6 / r7 6 0 r8
Cycle 1
Instruction r7 := r1 – r6 issued
Cycle 2
Instruction r3 := r1 ∗ r4 issued
Cycle 3
Instruction r4 := r1 + r5 issued
r1 27 100 r2 / r3 –21 13 r4 / r5 0 88 r6 / r7 –61 0 r8
Cycle 5
Instruction r7 := r1 – r6 completes
r7 overwritten
r1 27 100 r2 / r3 –21 27 r4 / r5 0 88 r6 / r7 –61 0 r8
Cycle 6
Instruction r4 := r1 + r5 completes
r4 overwritten
27 100 / –21 13 / 0 88 / 6 0
r1 27 100 r2 / r3 –21 13 r4 / r5 0 88 r6 / r7 6 0 r8
Cycle 9 : exception
Instruction r3 := r1 ∗ r4 has error
Registers restored from checkpoint
Instruction r7 := r1 – 6 reissued
r1 27 100 r2 / r3 –21 13 r4 / r5 0 88 r6 / r7 –61 0 r8
Cycle 13
Instruction r7 := r1 – r6 completes
r7 overwritten
Instruction r3 := r1 ∗ r4 issued

Figure 4. Checkpoint retry.

Optimizing the implementation

Implementing precise interrupts through in-order completion degrades performance by reducing the amount of pipelining possible. Implementing precise interrupts with out-of-order completion requires a significant amount of hardware. Worse, the extra hardware can add to the machine's cycle time, thus degrading performance. To reduce the cost of interrupt handling with out-of-order completion, we must consider the requirements of the different classes of interrupts.

We can handle external-critical interrupts inexpensively and efficiently by halting all further instruction issue once an external interrupt has been detected and waiting for the pipeline to drain before invoking the interrupt handler.

We can handle external-error interrupts the same way. In the case of a hardware fault, however, draining the pipeline after such an interrupt may not be possible. Moreover, a precise-interrupt state may not even be appropriate after a hardware fault. The correct response might be to freeze the machine state at the interrupt point so that a service processor can diagnose the cause of the problem. In either case, we can implement external-error interrupts inexpensively.

More ambitious implementations try to work around hardware faults, possibly using retry, to handle soft (intermittent) hardware faults in a user-transparent manner. In such machines, hardware fault handlers are complicated to implement. Handling hardware faults with retry mechanisms resembles internal-critical interrupts and perhaps should be classified with them. Of course, additional logic (and possibly microcode) is necessary to retry instructions after a fault and, if the fault persists after repeated retries, to report it as a hard (uncorrectable) failure.

So far, we have shown that we can implement external interrupts efficiently and inexpensively (except in machines that retry hardware faults). Only internal interrupts appear to require expensive mechanisms. Clearly, we must somehow implement internal-critical interrupts such as virtual-memory interrupts, since they are necessary for running any program. But what about internal-error interrupts?

An internal-error interrupt, by definition, occurs only in the run of an erroneous program, either because of unanticipated data or because the program itself is wrong. If the program is running in a mode other than debugging, the interrupt handler probably will abort the program or ignore the interrupt. If the hardware does not provide the appropriate behavior, such as reporting an interrupt, the programmer must insert checks for data that may cause interrupts and must add appropriate handling code. For example, the following program contains code ensuring that division cannot cause a divide-by-zero interrupt:

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

Furthermore, instead of relying on the interrupt handler to fix the potential divide by zero, the program contains repair code to do so. Such code can be simple, merely reporting the violation and then exiting the program. Or it can be sophisticated, including code to scale appropriate variables and thereby work around the problem.

Implementations with imprecise internal-error interrupts can invoke a handler, which then reports the interrupt. Because the interrupts are imprecise, however, the handler can take no recovery action. It either resumes program execution without modifying the machine state, possibly after logging the error, or aborts execution.

Of course, precise internal-error exceptions are necessary in some situations, especially debugging. Debugging requires implementation of precise interrupts, possibly at the cost of performance. One method is to turn off all pipelining. Our justification for this is that performance is not critical during debugging. Moreover, the code is compiled so as to maintain the original (source program) order, thereby inhibiting pipelining; further loss of pipelining due to precise interrupts does not significantly add to performance degradation.

Except in debugging, we believe that internal-error interrupts are not necessary to a machine's operability. Thus, if ignoring these interrupts results in a more efficient implementation, we should not implement them in hardware. Instead, we can leave the burden of anticipating and dealing with errors to the programmer if necessary.

The only interrupts not yet discussed are internal-critical interrupts. As mentioned, we must implement these interrupts, since they are necessary to the functioning of all programs. For example, in a virtual-memory machine, if the processor cannot correctly handle virtual-memory interrupts, it cannot execute any program.

If we choose to implement only internal-critical interrupts precisely, several methods are possible. We can use one of the already-discussed techniques, optimized to handle only internal-critical interrupts. For instance, we can modify in-order completion to implement only virtual-memory interrupts. In that case, the implementation would not allow instructions to complete out of order with respect to memory operations but would allow them to complete out of order with respect to other operations.

Such approaches may have an impact on processor performance. On the other hand, the next three techniques we discuss exploit certain properties of internal-critical interrupts to implement these interrupts efficiently. We illustrate the techniques with virtual-memory interrupts, the most common internal-critical interrupts, but the techniques apply to all interrupts in this class.

Direct implementation

The simplest way to deal with internal-critical interrupts is to integrate the interrupt-handler design with the hardware design. Such an interrupt handler uses a nonstandard interface, different from the interface for user-written interrupt handlers, and probably contains encoded, implementation-specific knowledge. This interrupt handler is unlikely to be portable across implementations. An extreme example of this approach is a handler (such as for a TLB miss) implemented in microcode.

The Cyber 200 processor[8] uses a different approach, based on similar reasoning, to implement virtual-memory handlers. At a virtual-memory interrupt, the processor saves its entire state, including the machine-dependent state information of partially completed instructions, in an invisible exchange package. After processing the interrupt, the handler uses this information to restart the machine. The partially completed instructions restart from the interrupt point. This approach guarantees that the virtual-memory handler will never alter any inputs to the executing instructions. Therefore, unlike a general interrupt handler, the virtual-memory handler has no reason to abort the partially completed instructions and allows them to complete.

But this approach suffers several drawbacks. First, freezing, saving, and restoring partially completed operations must be possible. This means that there must be paths to the intermediate pipeline latches, so that they can be saved. Second, the interrupt handler must be aware of the number of intermediate latches. Thus, a change of the processor's implementation may force a rewrite of the interrupt handler. Third, the interrupt handler is nonportable. These drawbacks are not present in the following schemes.

Restricting precision

We can implement imprecise exceptions as follows: When an instruction interrupts, if any instruction issued after the interrupting instruction has completed, all instructions between the interrupting instruction and the last completed instruction run to completion. Control transfers to the interrupt handler in this state. Normal execution can resume after the last completed instruction.

Some imprecise interrupts are guaranteed to be precise. The fact that an instruction will interrupt is determined in the *n*th cycle of its execution. If this *n* is no larger than the number of cycles necessary to execute the shortest instruction, that interrupt is precise. No instruction issued after the interrupting instruction can have completed, so only 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 four cycles to complete. Assume that a load takes seven cycles. Now, consider a TLB miss on the load in the following:

1. r3 := ld r4
2. r4 := r1 + r5

If the appropriate logic detects the TLB miss in the fourth cycle, the add will still be executing, so the processor can squash it, thereby obtaining precise state. If the TLB miss is detected in the sixth cycle, however, the add will have completed. If it is detected in the fifth cycle, handling it becomes a little complicated. The add will be in the process of writing its results to the register file. Depending on the implementation, the processor may be able to intercept the write, generating precise state.

Thus, we can implement internal-critical interrupts precisely yet inexpensively if they are detected in less time than the shortest instruction takes to complete. Fortunately, the common internal-critical interrupts meet this condition. Logic can detect an unimplemented instruction while decoding the instruction, typically in the second pipeline cycle. A TLB miss is more of a problem, but we can provide hardware to detect it in the pipeline's first execution stage. The RS/6000 FPU[9] and the Alpha[10] use this technique.

Discarding precise interrupts

The restricted-precision technique just described imposes the constraint that internal-critical interrupts 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 use the same number of cycles as an add operation, which simply adds two registers together.

Another drawback is that the technique makes it almost impossible to implement interrupt handlers for interrupts detected late, such as exponent underflow. A suggested use of the underflow interrupt is to implement denormalization with an interrupt handler called when a floating-point number is smaller than the smallest normal floating-point number. Instead, with the imprecise-interrupt technique, we must implement denormalization in hardware.

Imprecise interrupts make the writing of interrupt handlers difficult because they allow instructions issued after the interrupting instruction to complete and thereby overwrite a value the interrupt handler needs to recover from the interrupt. Typically, the only values needed by an interrupt handler are the inputs to the interrupting instruction and possibly status or control register values. That is one reason for aborting instructions issued after the interrupting instruction.

The other reason is that if the interrupt handler modifies the state, it may be necessary to reexecute the subsequent instructions in the modified state. Again, a typical interrupt handler, such as the denormalizing handler, modifies only a restricted portion of the machine state: the instruction's output register. However, no instruction that used the interrupting instruction's output register value could have completed; if the processor encountered such an instruction, it would cease to issue instructions until the value became available. (Nor could the processor issue any instruction modifying the output register value; even without an interrupt, that would update the output register in the wrong order.) Thus, in an in-order-issue pipelined implementation, an interrupt handler that modifies only the output register of the interrupting instruction does not need to reexecute instructions issued after the interrupting instruction.

These observations suggest an alternative interrupt handler interface, suitable for an implementation with imprecise interrupts. When an instruction interrupts, the interrupt mechanism gives the interrupt handler the input values to the interrupting instruction and some control information, as well as the addresses of the interrupting instruction and the last completed instruction. The interrupt handler can safely modify only the interrupting instruction's output register and resumes execution after the last completed instruction. The ROMP processor uses a similar approach[11] to handle imprecise memory interrupts.

Augmenting interrupt hardware to provide inputs to the interrupt handler allows implementation of handlers for internal-critical interrupts even though the machine state is imprecise when interrupts occur. These handlers read only the interrupting instruction's input values and modify only the output register. Moreover, the scheme imposes no constraints on how early an exception must be reported. Most interrupt handlers for exceptions that can only be detected late, including the one that implements denormalized numbers, can use this interface. Thus, this technique not only implements all but internal-error interrupts inexpensively, it also implements most internal-error interrupt handlers.

Related issues

Several peripheral issues arise in handling interrupts on aggressive implementations. These include sparse restart, which occurs whenever we weaken the precision requirements on an out-of-order-issue processor, and the impact of parallel (for example, superscalar) issue. Other problems, even on less aggressive processor designs, include recursive interrupts, multiple simultaneous interrupts, and memory interrupts.

Sparse restart. So far a single address has been sufficient to resume normal execution. It can be the address of the interrupting instruction or of the last completed instruction. After processing the interrupt, the handler resumes normal execution by branching to the provided address or 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 after it have completed. We call this situation a dense restart. By contrast, in a sparse restart there is an address prior to which all instructions have completed, and another (different) address past which no instruction has completed. Between the two addresses are uncompleted instructions mingled with completed instructions. Of course, all restarts are dense on an architecture with precise interrupts; by definition, no instructions after the interrupting instruction can have completed, and all prior instructions must have.

Avoiding sparse restarts is difficult when a processor issues instructions out of order while implementing imprecise interrupts. Even in implementations that issue instructions in order, we must take special care to ensure a dense restart state. For example, when defining an imprecise exception, we may have to allow instructions issued after the interrupting instruction but not completed at the time of interrupt detection to run to completion because an even later instruction has already completed.

Figure 5 illustrates how sparse restarts cause problems. When control enters the interrupt handler, the shaded instructions have all issued and completed, while the unshaded ones have not. To properly resume execution after interrupt handling, the processor must execute the unshaded instructions. A mechanism that can selectively execute these instructions is necessary.

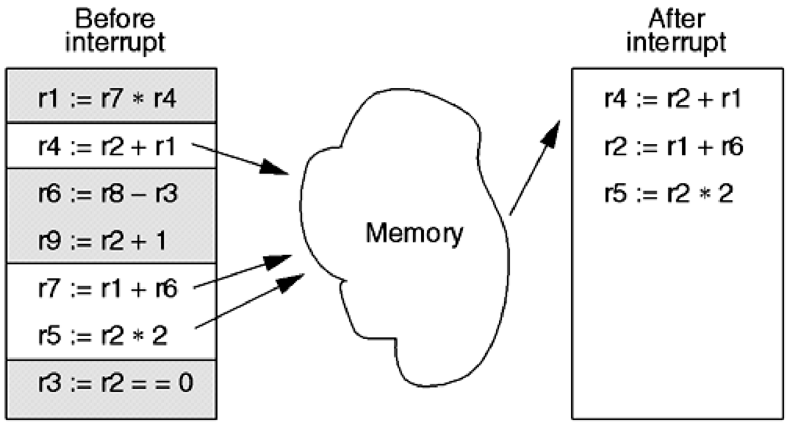


Figure 5. Sparse restart.

The restart mechanism illustrated in Figure 5, suggested by Torng and Day,[12] assumes that instructions issue from an instruction window. On an interrupt, the processor saves uncompleted instructions in the window (as well as the address of the next instruction to be fetched into the window). The processor resumes normal instruction execution by reloading the instruction window (and the fetch address) from the saved state. Only the previously unexecuted instructions have been reloaded. Now when execution resumes, none of the previously executed instructions will reissue.

As indicated earlier, a general interrupt handler can modify an arbitrary state and thereby force reexecution of instructions that appeared after the interrupting instruction in the program but executed before or in parallel with it. Assume that the interrupting instruction in Figure 5 is the second instruction, r4 := r2 + r1. Now, assume that the interrupt handler modified r2. In that case, the interrupt handler should use the new value of r2 to reevaluate all instructions that appear after the interrupting instruction and use r2. For instance, the fourth instruction, r9 := r2 + 1, must reexecute, as must the last instruction in the window, r3 := r2 = = 0. The proposed sparse-restart mechanisms, however, will not accomplish this. Thus, with a sparse-restart mechanism, the interrupt handler must either be careful while modifying register values (other than the interrupting instruction's output register values) or ensure that all affected instructions reexecute.

Parallel issue. Superscalar and VLIW (very long instruction word) implementations issue more than one instruction per cycle. An obvious complication is that when an instruction causes an interrupt, the logic must determine which instructions that issued in parallel actually preceded the interrupting instruction and therefore should run to completion. On a VLIW processor, a reasonable way to circumvent this problem is to interrupt at the boundary of an instruction packet (a collection of instructions issued simultaneously), instead of at a particular instruction. Thus, if any instruction in a packet interrupts, we say that the packet as a whole has interrupted.

Another problem arises from the ability to issue more than one instruction at a time. For example, consider a case in which an architecture's shortest instruction takes four cycles. A single-issue implementation guarantees that any instruction will complete at least five cycles after the previous instruction—four to execute plus one because it issued one cycle later. In a parallel-issue implementation, the time decreases by one because the instruction can issue in the same cycle as the previous instruction. This affects techniques that implement precise memory interrupts in the presence of imprecise interrupts. A parallel-issue implementation may need to detect interrupts one cycle earlier than a single-issue implementation.

Recursive interrupts. The interrupt handler itself may cause an interrupt. A processor state, such as the cause of the interrupt and the return address, is associated with the interrupt. If the handler processes the second interrupt immediately, this state will be overwritten, possibly with disastrous consequences. Usually, when control first enters the interrupt handler, all further interrupts are blocked. The first thing the handler does is save the interrupt state. Then other interrupts can pass to the handler. While it is saving the state, the interrupt handler must ensure that it causes no other interrupts, such as memory faults.

Multiple interrupts. Several interrupts can arrive simultaneously—for instance, a timer interrupt and an exception. Interrupts are usually prioritized, and the processor must invoke interrupt handlers accordingly. One technique is to process the least important interrupt first. As soon as the interrupt handler turns off blocking, the next least important interrupt occurs, interrupting the current interrupt handler, and so on. This way, the most important interrupt is completely processed first.

Memory interrupts. The interrupt mechanism must ensure that the interrupt handlers for virtual-memory interrupts (such as a TLB miss or a page not in memory) must not themselves cause an interrupt of the same kind. It is impossible to recover from this 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 entries for the interrupt handler code and data pages are locked in the TLB, and the pages themselves are locked in main memory.

Conclusion

As we have seen, implementing precise interrupts can inhibit the benefits of pipelining. The major cause of this interference is that fully exploiting pipelining introduces out-of-order completion. To implement precise interrupts, instructions must appear to complete in order. We can achieve this 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 second solution exploits pipelining fully, but it uses a large amount of hardware. Further, it may still decrease performance by increasing cycle times.

Our examination of the cost of implementing precise interrupts for each interrupt class has shown that we can implement external-critical and external-error interrupts inexpensively on a pipelined processor, with little or no performance impact. We propose implementing the third class, internal-error interrupts, imprecisely, except during debugging. Finally, several techniques are available for inexpensively implementing precise interrupts for the fourth class, internal-critical interrupts, but these may not apply generally.

We believe that treating each class of interrupts separately, depending on its design constraints, is the correct approach. In particular, since the general interrupt handler interface embodied in the precise-interrupt model is incompatible with aggressive pipelined processors, designers should implement precise interrupts only when necessary or easy. Otherwise, they should adopt weaker, less general techniques. Several recent microprocessor designs[9, 10] reflect this belief. However, it is possible that the technique adopted in these designs—preserving precise interrupts for all but internal-error interrupts—is still too restrictive. The alternative mechanism—augmenting the architecture so that it preserves the inputs to the interrupting instruction and provides them to the handler—may give the processor designer and the interrupt handler programmer more flexibility.

References

1. J.E. Smith and A.R. Pleszkun, “Implementation of Precise Interrupts in Pipelined Processors,” *Proc. 12th Ann. Int’l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp. 36-44.
2. *IBM System/370 Extended Architecture Principles of Operation*, IBM Corp., Poughkeepsie, N.Y., 1983.
3. S.A. Przybylski et al., “Organization and VLSI Implementation of MIPS,” *J. VLSI and Computer Systems,* Vol. 1, No. 2, Fall 1984, pp. 170-284.
4. D.Alpert and D. Avnon, “Architecture of the Pentium Microprocessor,” *IEEE Micro,* >Vol. 13, No. 3, June 1993, pp. 11-21.
5. N. Ullah and M. Holle “The MC88110 Implementation of Precise Exceptions in a Superscalar Architecture,” *Computer Architecture News,* Vol. 21, No. 1, 1993, pp. 15-25.
6. C.-J. Wang and F. Emmett, “Precise Interruptions in RISC Pipelines,” *IEEE Micro*, Vol. 13, No. 4, Aug. 1993, pp. 36-43.
7. W.M. Hwu and Y.N. Patt, “Checkpoint Repair for Out-of-Order Execution Machines,” *Proc. 14th Ann. Int’l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., 1987, pp. 18-26.
8. *CDC CYBER 200 Model 205 Computer System Hardware Reference Manual,* Control Data Corp., Arden Hills, Minn., 1981.
9. G.F. Grohoski, “Machine Organization of the IBM RISC System/6000 Processor,” *IBM J. Research and Development,* Vol. 34, No. 1, Jan. 1990, pp. 37-58.
10. *Alpha Architecture Handbook*, Digital Equipment Corp., Maynard, Mass., 1992.
11. *IBM RT PC Hardware Technical Reference*, IBM Corp., Austin, Tex., 1986.
12. H.C. Torng and M. Day, “Interrupt Handling for Out-of-Order Execution Processors,” *IEEE Trans. Computers*, Vol. 42, No. 1, Jan.1993, pp. 122-127.

Mayan Moudgill works in the VLIW Development Group at IBM’s T.J. Watson Research Laboratory. His interests include compilers, instruction level parallelism, and architectures.

Moudgill received the BTech degree from the Indian Institute of Technology, Kanpur. He received MS and PhD degrees in computer science from Cornell University.

Stamatis Vassiliadis is a professor in the Electrical Engineering Department of Delft University of Technology in the Netherlands. He has also served on the faculties of Cornell University and the State University of New York. Previously, he worked for IBM in the Advanced Workstations and Systems Laboratory in Austin, Texas, the Mid-Hudson Valley Laboratory in Poughkeepsie, New York, and the Glendale Laboratory in Endicott, New York. He has received many awards for his work in computer system design. Among his research interests are computer architecture, hardware design and functional testing, parallel processors, and software engineering.

Vassiliadis received the DrEng degree in electronic engineering from the Polytechnic of Milan, Italy, and a PhD in computer science from the University of Namur, Belgium.

Send correspondence about this article to Stamatis Vassiliadis, TU Delft, Electrical Eng. Dept., Mekelweg 4, 2628 CD Delft, the Netherlands; stamatis@duteca.et.tudelft.nl.