Interrupt Processing in Concurrent Processors

Wade Walker

Harvey G. Cragon
University of Texas at Austin

Systems architects are faced with many possibilities for designing interrupt-processing strategies that optimize computer resources and performance. This framework of hardware implementation techniques highlights choices for consideration.

As the complexity of modern processors grows, effective strategies for handling interrupts become increasingly important. To meet the demands of more applications, peripherals, and functions, such strategies must feature different types and levels of interrupts that have been selectively combined. While many strategies are possible, it's a challenge for the designer to determine which interrupt-processing techniques and methods are best for optimal system performance.

To help designers systematically explore options for handling interrupts and help researchers compare interrupt-processing strategies, we offer a taxonomy (or classification) of implementation choices. The approach we've developed broadly classifies interrupt-processing techniques and implementations into six phases.

In preparing this taxonomy, we've examined the strategies used in 15 modern concurrent processors (those that can process more than one instruction at a time), such as the MIPS R4000 and Intel Pentium. We extend our findings, as applicable, to interrupt-processing design decisions in general and survey the different hardware techniques available to designers. We concentrate on concurrent processors because their interrupt-processing systems are more complex than those of nonconcurrent processors, and because the level of concurrency in modern processors is steadily increasing.

What is an interrupt?

Originally, interrupts were used to make I/O processing more efficient by eliminating the need for software polling.[1] Later, interrupts subsumed the function of traps (also known as internal[2] or program[3] interrupts). Still later, the terminology was unified, and interrupts were defined as those events (except branches) that change the normal flow of program execution.[4] Because using this definition is simpler than categorizing interrupts based on cause, which is the technique most processor manufacturers use, we'll retain it for our discussion. We will, however, refer to some interrupts by name—for example, I/O interrupts and page faults. We also assume, for simplicity, that the interrupts we discuss are isolated. (For a description of other interrupts, see the sidebar "Queued and nested interrupts."

When an interrupt occurs during process execution, the processor must stop the currently executing process—however briefly—to handle the interrupt. Unless the interrupted process will not resume (which would be the case with catastrophic interrupts), some state information about the interrupted process is saved. Then the interrupt is processed, the saved process state is restored, and the interrupted process is resumed (see Figure 1).

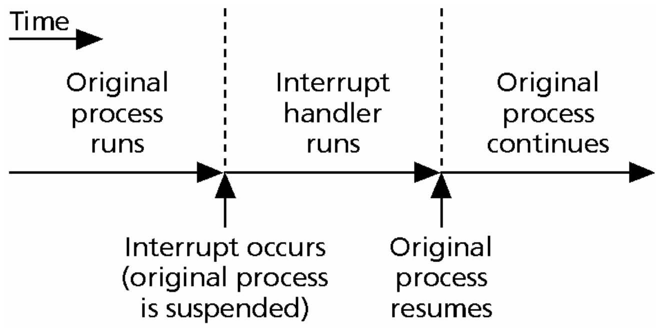
Time
Original process runs
Interrupt handler runs
Original process continues
Interrupt occurs (original process is suspended)
Original process resumes

Figure 1. Typical interrupt processing.

The fundamental information about a process that must be saved for the process to resume is its program counter. In practice, additional information can be saved, but the program counter is essential to most processors, since it defines exactly which instructions of the interrupted process have—and have not—completed execution. Depending on which instructions have completed (and which have not) at the time of interrupt processing, an interrupt can be either precise or imprecise.

What are precise interrupts?

Precise interrupts are a way to guarantee the fundamental requirement of an interrupt-processing system, namely that after a noncatastrophic interrupt occurs, the interrupted process is guaranteed to be able to continue executing correctly. Interrupts are precise if the following three conditions[5] hold:

- All instructions issued before the one indicated by the saved program counter have finished execution and have modified the process state correctly.
- All instructions issued after the one indicated by the saved program counter are unexecuted and have not modified the process state.
- If the interrupt was caused by an instruction, the saved program counter points to that instruction, called the interrupting instruction. This instruction must either be completely executed or completely unexecuted.

Figure 2 illustrates a four-stage pipeline that adheres to these conditions. In this example, instruction 2 has caused an interrupt. Because the program counter points to instruction 2, instruction 1 and all instructions before it must have finished execution and modified the process state correctly; instructions 3, 4, and so on must be completely unexecuted and must not have modified the process state; and instruction 2 must either be completely executed or completely unexecuted.

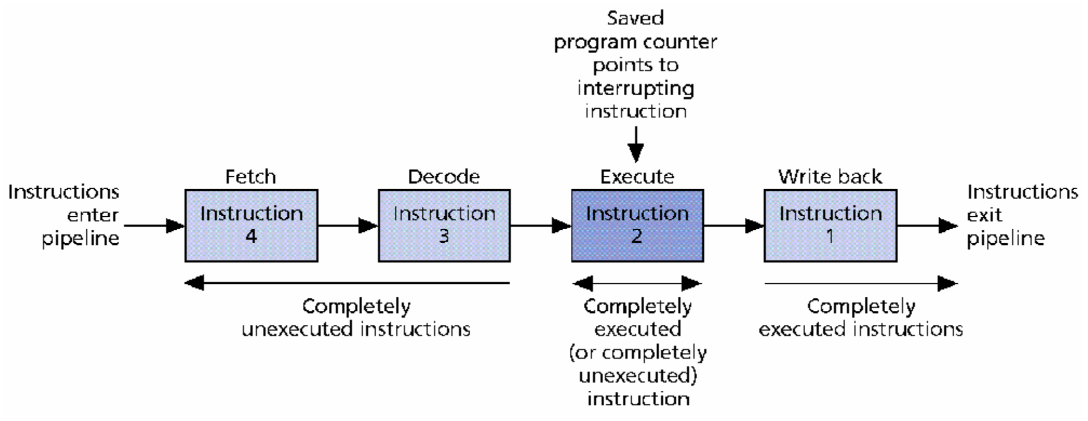
Saved program counter points to interrupting instruction
Instructions enter pipeline
Fetch
Instruction 4
Decode
Instruction 3
Execute
Instruction 2
Write back
Instruction 1
Instructions exit pipeline
Completely unexecuted instructions
Completely executed (or completely unexecuted) instruction
Completely executed instructions

Figure 2. Pipeline example of precise interrupt conditions.

Glossary

Architected register—(or more generally, architected resource) is one whose value is accessible by a machine-language programmer.[1] An example of an architected register is the EAX register in the Intel Pentium.

Implemented register—(or implemented resource) contains internal processor state that is not accessible by a machine-language programmer. Examples of implemented resources include register scoreboards, reorder buffers, latches between pipeline stages, and history buffers.

Internal interrupt—one that is caused by an instruction; an external interrupt is one caused by something external to the processor.

Interrupt handler—a software program invoked by the processor when an interrupt occurs. It is the interrupt handler's responsibility to respond to the interrupt if it is recoverable and to relinquish control of the processor when it is finished.

Interrupting instruction—the instruction that caused an interrupt (if the interrupt is internal). There are no interrupting instructions for external interrupts

Out-of-order execution—description of instructions that execute in a different order from the original program order.

Page fault—an interrupt generated when the processor attempts to access a virtual-memory page that is not resident in main memory.

Pipelining—a form or processor concurrency that involves breaking instruction execution into several stages and sending instructions through the stages in assembly line fashion.

Serial instruction execution—state changes are committed in the same order that the instructions entered the processor. A processor with one pipeline that does no instruction reordering executes instructions serially, although they are not executed sequentially. A serially correct state (of a processor or a process) is a state identical to that which would result from serial instruction execution.

Sequential instruction execution—each instruction is executed completely before the next instruction is started. A concurrent processor can execute instructions sequentially by waiting to issue each instruction until the previous one has completed.

Superpipelining—a form of processor concurrency similar to pipelining but with more stages.

Superscalar—a form of processor concurrency in which multiple instructions can be issued and retired in every clock cycle.

Stack—an area of main memory, pointed to by a processor register, where a processor may store state information.

States:

> **External state**—all state information in the entire system that is not part of the internal state. External state encompasses cache memory, main memory, and secondary memory, but not processor registers.
>
> **Processor state**—(also called an internal state) refers to the values of a processor's registers, both architected and implemented.
>
> **Process state**—all information relevant to a process. A process state may encompass the processor state (if the process is running) as well as partial system state outside the processor, such as values in caches or main memory.

Trap—an instruction that deliberately causes an interrupt, such as an instruction that invokes an operating system routine by causing an interrupt.

Reference

1. K. Venkatramani, "A Semantics-Based Approach for the Design and Verification of Concurrent Processors," doctoral dissertation, University of Texas at Austin, Aug. 1990.

With a precise interrupt, the process state just before interrupt processing is described as *serially correct*. This means that the process state is just as if the program had been executed serially, one instruction at a time. With an imprecise interrupt, on the other hand, the process state just before interrupt processing is not serially correct.

An interrupt example

Precise interrupts, while not absolutely necessary, offer a convenient way to implement features that are all but mandatory on modern processors such as virtual-memory support.

For example, suppose a designer is working on a new superscalar processor. The processor will allow out-of-order instruction issue and completion and is intended for use in a multitasking, virtual-memory environment. The first type of interrupt the designer decides to implement is the page fault to implement virtual memory. When a page fault occurs, the current process is suspended, the necessary page loads, and the process resumes as if the page fault had not occurred. (See Figure 3)

The designer ensures that the original process can resume successfully by making the page fault a precise interrupt, with the faulting instruction completely unexecuted. Therefore, to resume the original process after the memory page loads requires only that instruction fetching resumes with the faulting instruction.

The fundamental requirement of interrupt-processing systems can also be met without making interrupts precise. For example, strategies listed in a later discussion under “instruction continuation” are imprecise because they continue execution from the point where an interrupt occurred without satisfying Smith and Pleszkun’s three conditions.

Implementing interrupt-processing systems

Interrupt-processing systems can be implemented in many ways. Some techniques, like history buffers, are general enough that the designer could make all processor interrupts precise, depending on the desired end result. With other strategies, such as adding special-purpose registers to a processor, the designer might make only one type of interrupt precise—for example, I/O interrupts.

Some commercial processors make only the bare minimum of interrupts precise. This is the strategy chosen by Digital Equipment for its Alpha microprocessor. The Alpha’s page faults are precise interrupts; however, its arithmetic interrupts are imprecise unless the user issues a special instruction to the processor after an arithmetic instruction. This special instruction, TRAPB, prevents any more instructions from issuing until the arithmetic instruction has exited the pipeline, which effectively ensures that the processor state will be serial. The Alpha’s design and implementation were effectively simplified by means of allowing for imprecise interrupts.

The choice of whether to make a particular interrupt precise or imprecise is usually based on difficulty of implementation. Generally, it’s easier to implement precise interrupts than imprecise ones, assuming that both fulfill the fundamental requirement of interrupt-processing systems. However, imprecise interrupts sometimes result in higher performance, since they avoid the overhead of ensuring that Smith and Pleszkun’s three conditions are met.

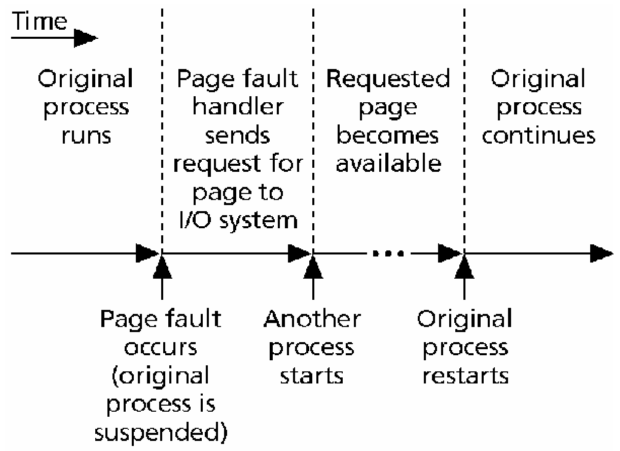
Time
Original process runs
Page fault handler sends request for page to I/O system
Requested page becomes available
Original process continues
Page fault occurs (original process is suspended)
Another process starts
Original process restarts

Figure 3. Page-fault interrupt processing.

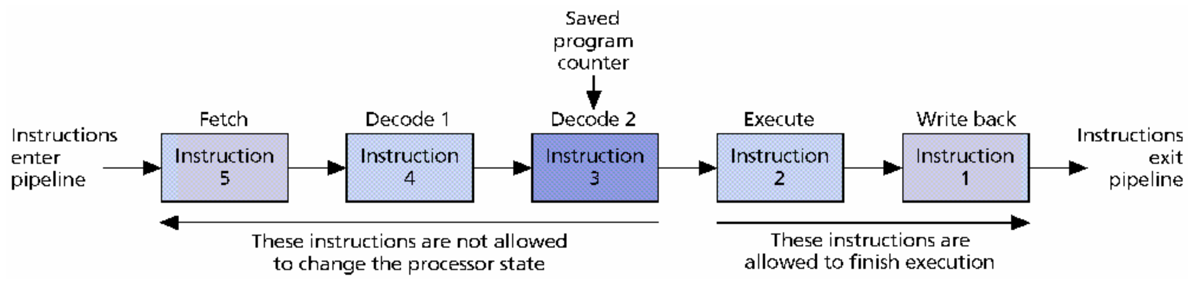

Figure 4 Processor state serialization with in-order instruction execution.

Serializing the processor state

To implement precise interrupts, the processor state must be serially correct before the processor state can be saved and interrupt handling can begin. The processor state can be serialized in one of two ways. (Serializing the processor state means making the state of the processor serially correct, either by not allowing it to become serially incorrect or by fixing it if it is serially incorrect.) The first and easiest method, allowing instructions before the interrupting instruction to complete execution and cancelling those after the interrupting instruction, works only on a processor that does not allow out-of-order instruction issue or completion; moreover, if the interrupt is internal, the processor state must not be changed by any instructions issued after the interrupting instruction (see Figure 4).

The interrupt shown in Figure 4 is internal, with instruction 3 being the interrupting instruction. If instruction 3 had been designed to cause an interrupt—such as a debugging instruction—then, depending on the processor, the saved program counter could have pointed to instruction 4. If the interrupt had been external, the saved program counter could have been placed at the discretion of the processor designer as long as it and all the instructions issued after it had not yet changed the processor state.

A second technique to serialize processor state involves adding extra interrupt-processing hardware to the processor. This technique can work regardless of out-of-order issue and completion and whether or not an instruction issued after an interrupting instruction has caused a processor state change. Figure 5 shows a processing case that benefits from additional hardware.

The instructions shown in Figure 5 have been reordered by the processor. Instructions 1 and 5 have already modified the processor state, and instruction 3 has caused a page fault. Extra interrupt-processing hardware is required to undo the state changes caused by instruction 5 so that processing can later resume from instruction 2 (or, alternatively, to ensure that instruction 5 doesn't cause any permanent state changes until all previous instructions have completed). When instruction 3 is reached again after the new memory page loads, a page fault will not occur.

Saving the process state

The interrupt-processing hardware we describe generally affects the *processor* state only, because an external state rarely needs modification to ensure that a process resumes. We use the broader term *process* state instead because it is theoretically possible to change state external to the processor during interrupt processing. (An example of an interrupt-processing strategy that causes external state changes is a checkpointing strategy that checkpoints main memory periodically and resumes a process from this checkpoint state after interrupt processing.)

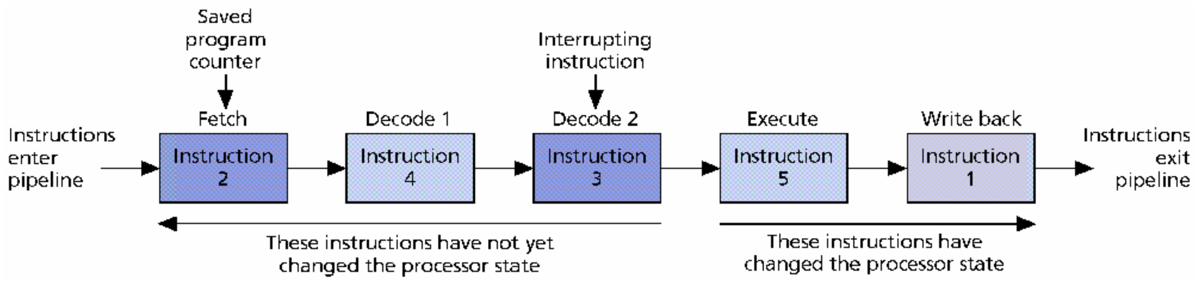

Figure 5. Processor state serialization with out-of-order instruction execution.

Queued and Nested Interrupts

Including interrupts that occur simultaneously (or during the processing of other interrupts) in our taxonomy would have made the taxonomy unwieldy. Because queued and nested interrupts affect most interrupt-processing strategies in the same way, it's simpler to discuss them separately. A *queued* interrupt occurs while another interrupt is being processed but is not processed itself until that first interrupt is completed. A *nested* interrupt occurs while another interrupt is being processed, and it preempts processing of the first interrupt until the nested one completes.

Table A lists the processor designer's choices for allowing queuing, nesting, or both. The *current* interrupt refers to the interrupt now being processed; the *new* interrupt refers to the interrupt that occurs during the processing of the current interrupt, regardless of levels.

In other words, if an *external* interrupt occurs while processing another interrupt, the external interrupt can be safely queued until later, since the current interrupt can finish processing without having to process the new one. But if an *internal* interrupt occurs while processing another interrupt, it must be nested because the current interrupt cannot finish processing without first processing the new one.

Most concurrent processors avoid the problem of unnecessarily providing many levels of interrupts by disabling new interrupts during interrupt processing (for external interrupts) or by stopping execution and signaling an error if another interrupt occurs (for internal interrupts). This saves them from having to handle nested interrupts.

Table A. Queued and nested interrupts.

Current interrupt	New interrupt	Options
External	External	Queue or nest
External	Internal	Nest
Internal	External	Queue or nest
Internal	Internal	Nest

In any processor that allows interrupts, the interrupt-processing system saves enough process state information for the original process to resume after an interrupt occurs. The process state is saved in two different places: The process state that is not part of the processor state is usually stored by the operating system in main or secondary memory, and the processor state is usually saved directly by the processor. The time to service interrupts depends heavily on how much process state is saved and where. To illustrate this, we consider two processors, Motorola's MC68030 and MC88100. We assume that the process states stored by the operating system in either case are about the same size.

When an interrupt occurs, the MC68030 serializes its processor state and saves the program counter and other registers on the stack before running the interrupt handler. After the interrupt handler is run, the registers are restored from the stack. Although serializing the processor state requires extra time, in this case it is worth it because serialization allows a much smaller state to be saved to the stack than if the processor state was not serialized. This is because saving a serial state usually requires just a program counter; saving a nonserial state requires saving more complex information that specifies which instructions have been executed and which have not. Because stack operations are relatively slow, saving a small state can reduce interrupt-processing time despite the additional time spent to serialize the processor state.

The MC88100, on the other hand, continuously saves register values in shadow registers. When an interrupt occurs, the state of the shadow registers is "frozen" and not allowed to change while the interrupt handler is run. After the interrupt handler runs, the registers are restored from the values preserved in the shadow registers.

Because the processor state is continuously saved inside the processor rather than to the stack when an interrupt occurs, interrupt handling on the MC88100 is more time-efficient than on the MC68030. However, the lower latency of interrupt processing is gained at the expense of additional chip real estate to accommodate numerous shadow registers.

Hardware possibilities

Interrupt-processing hardware performs one of two functions: saves the processor state or serializes the processor state. In most processors, the processor state is first serialized, then saved. In theory, serialization would not be required if enough processor state were saved; however, the more processor state is saved, the longer the interrupt-processing latency tends to become.

State-saving hardware

In choosing a technique, designers will consider at least five kinds of state-saving hardware.

Stacks. The most common type of state-saving hardware is a simple stack. The program counter and associated information are saved on the stack when an interrupt occurs. Because external memory access is relatively slow, only the program counter and some registers are usually saved.

Shadow registers. In a shadow register implementation, some or all of the normal processor registers are associated with counterparts known as shadow registers. By “normal” we mean architected and implemented registers. Each time a shadowed register’s value changes, the normal register’s old value is saved in the shadow register. When an interrupt occurs, the normal register values can be restored from the shadow registers to serialize the processor state. Shadow register values cannot be changed by an instruction until the processor is assured that the instruction will not cause an interrupt.

The Intel 80486 processor, for example, uses shadow registers to save processor state (although they are not specifically named in the literature). For page faults, some of the processor registers’ values are restored to what they were before they were changed by the interrupting instruction. This is accomplished through several dedicated shadow registers. Shadow registers are also used in the MC88100.

If the shadow register concept were extended, a shadow register would be associated with every processor register, both architected and implemented. A processor could then save its entire state so that no state serialization would be required when an interrupt occurred. This type of hardware is called *extended shadow registers*.

Although we are unaware of any machine that saves its state so comprehensively, extended shadow registers offer an attractive alternative for processors in which state serialization is difficult, such as superscalar processors. The drawback of this technique, however, is that there can be many shadow registers, especially since even the latches between pipeline stages must be saved to completely obviate state serialization, and this requires a lot of space on the chip.

Shadow stacks. These are functionally identical to shadow registers, except that each shadow register is replaced by a shadow stack. Each shadow stack serves the same purpose as an individual shadow register. Each time the value of a shadowed register is changed, the old value is pushed onto the shadow stack. Normal register values can be restored from the shadow stack when an interrupt occurs. Shadow stacks have an advantage over shadow registers in that, since more than one old value of a register may be saved in a shadow stack, nested interrupts can be accommodated by a shadow stack mechanism.

One hardware implementation we classify as a shadow stack is a *program-counter stack mechanism*.[6] Shadow stacks are provided for three processor registers whose values might have to be restored for certain types of interrupts.

Checkpointing hardware. This type of hardware can save external state changes for interrupt processing (the other implementations work only for processor state changes). Checkpointing saves either the entire external state or all the changes made to the external state during a specific time period. Because the amount of external state can be large, checkpointing generally saves the state either to primary or secondary memory. Hwu and Patt[7] elaborate on this technique in out-of-order execution processors.

Auxiliary processor. This is simply a second processor that handles the interrupt while the main processor waits (as with an internal interrupt) or continues processing (as with an external interrupt). We call this state&hyphen;saving hardware because the main processor state is saved in situ while the auxiliary processor is processing the interrupt. The auxiliary processor concept is not new, dating back at least to Keller.[8] An auxiliary processor and the extended shadow registers discussed earlier are logically equivalent because both save the entire processor state. Only the location where the state is saved is different. With extended shadow registers the state is saved in special registers; with an auxiliary processor, the state is saved in situ.

State-serializing hardware

After deciding what type of state-saving hardware they need, designers will consider at least some of the following state-serializing hardware.

Architecture. The simplest type of state-serializing hardware is a "clever" architecture, one designed such that instructions complete serially and don't cause any processor state changes that would need to be undone if an interrupt occurred. (This implementation was illustrated in Figure 4.) Many of the processors we examined achieve this architecture by making the last pipeline stage the only one that could cause processor state changes, such as the Sun Sparc, Motorola MC88000, DEC 21064, and Advanced Micro Devices 29000.

Result shift registers. In a result shift register implementation,[5] instruction issue is forestalled, if necessary, to ensure that processor state changes occur serially. Because instruction results are not committed until the processor is certain that the instruction didn't cause an interrupt, there are never any processor state changes to undo.

Reorder buffer. In this implementation, a special memory area called the reorder buffer stores instruction results. Instructions are allowed to issue and/or complete out of order, but the results are committed, in order, by the reorder buffer as they become available and only after the processor is certain that the instruction hasn't caused an interrupt. The reorder buffer[5] can be viewed as a generalization of the result shift register.

Register update unit. A register update unit9 lets a serial processor state be maintained like a reorder buffer. It also resolves dependencies using an extended version of Tomasulo's dependency resolution algorithm.[10]

History buffer. In this implementation, a special memory area called the history buffer stores old processor state information as it is replaced by new state information. When an interrupt occurs, the old processor state information is read from the history buffer to restore the processor to its state just before the interrupt occurred. History buffers, which are used in Motorola's MC88110, are described in more detail by Smith and Pleszkun[5] and by Ullah and Holle.[11]

Future file. Two separate register files are maintained—the future file, which is updated continuously as the processor operates, and the architectural file, which is updated in order by a reorder buffer. When an interrupt occurs, the architectural file is copied into the future file,[5] thus effectively backing up the processor state to a serially correct point before the interrupt.

Interrupt-processing phases

Now that we've surveyed the predominant techniques available, let's examine interrupt processing in the context of what we have defined as phases. Our purpose is to see what tasks are involved with each phase, then to learn which implementation choices are available to the designer. Note that design choices for different phases are not independent—choices made in one can limit those made in later phases.

The phases are as follows:

- Detect the interrupt.
- Finish pending instructions.
- Undo process state changes.
- Save the process state.
- Run the interrupt handler.
- Resume the interrupted process.

Figures 6 through 11 illustrate these phases. Each lists the possible hardware implementations to fulfill that phase and, where applicable, lists processor examples that use that particular technique. Many branches of the taxonomy diagrams have no processor listed. This just means that none of the relatively narrow set of concurrent processors we examined use the technique.

We assume a pipelined architecture as the basis for the discussion that follows. For a processor whose functional units are not pipelined, interrupt handling will likely be simpler in some cases than what we discuss. More information on this taxonomy can be found in the literature.[12]

Phase 1: Detect an interrupt

The source of a detected interrupt (see Figure 6) can be automatically identified by the processor or identified later by the interrupt handler. Identifying the source of the interrupt is done in phase 5. In this taxonomy, the detection of an interrupt within an instruction means that an interrupt can be detected while an instruction is actually inside one of the pipeline's stages. Interrupts, such as page faults, that are detected within an instruction are usually internal. External interrupts, on the other hand, are usually not detected until the processor advances all the instructions in the pipeline from one stage to the next. This is referred to as detecting interrupts between instructions, and all the processors we examined do this.

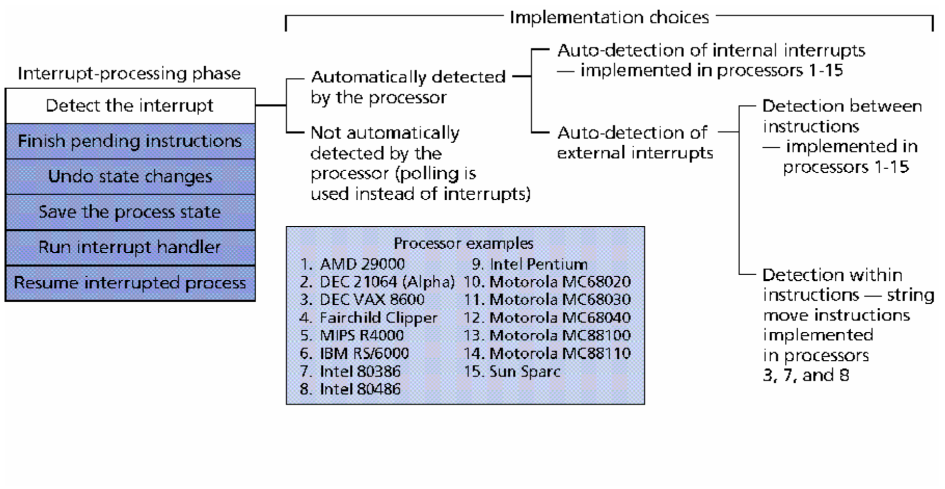
Implementation choices
Interrupt-processing phase
Detect the interrupt
Finish pending instructions
Undo state changes
Save the process state
Run interrupt handler
Resume interrupted process
Automatically detected by the processor
Not automatically detected by the processor (polling is used instead of interrupts)
Auto-detection of internal interrupts — implemented in processors 1-15
Auto-detection of external interrupts
Detection between instructions — implemented in processors 1-15
Detection within instructions — string move instructions implemented in processors 3, 7, and 8
Processor examples
1. AMD 29000
2. DEC 21064 (Alpha)
3. DEC VAX 8600
4. Fairchild Clipper
5. MIPS R4000
6. IBM RS/6000
7. Intel 80386
8. Intel 80486
9. Intel Pentium
10. Motorola MC68020
11. Motorola MC68030
12. Motorola MC68040
13. Motorola MC88100
14. Motorola MC88110
15. Sun Sparc

Figure 6. Phase 1: Detect the interrupt.

Phase 2: Finish pending instructions

Here, the processor either completes the pending instructions (those that execute before the interrupt handler is run if precise interrupts are to be implemented) or does nothing. If the interrupted process will not resume, the instructions issued before the interrupting instruction can be discarded instead of executed. (See Figure 7.)

Phase 3: Undo process state changes

In this phase, the processor undoes any process state changes that were caused by instructions required to be completely unexecuted by Smith and Pleszkun's second precise-interrupt condition. If precise interrupts are *not* being implemented, this phase is unnecessary. (See Figure 8.)

Some processors only partially undo process state changes in some cases. There are two possible reasons for this. A particular processor might not need to make interrupts entirely precise to ensure that the interrupted process runs correctly after resumption; or the interrupted process can't (or won't) resume, so precise interrupts are unnecessary. Phases 2 and 3 together accomplish what was earlier called state serialization.

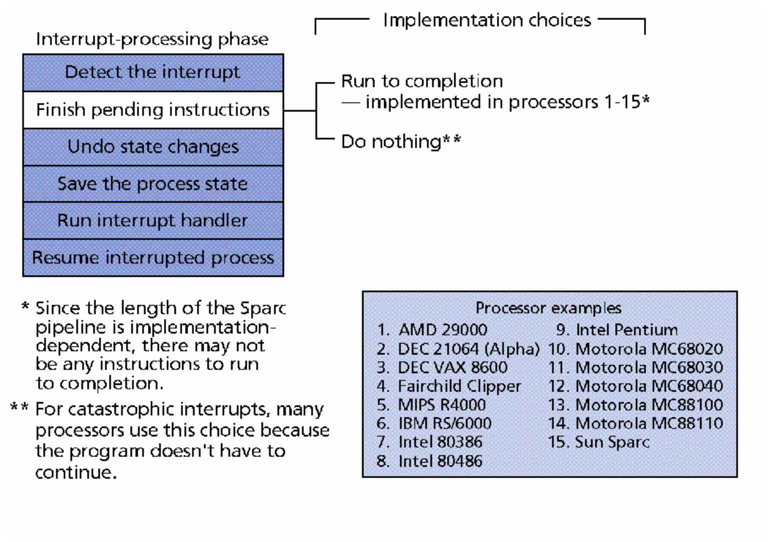

Figure 7. Phase 2: Finish pending instructions.

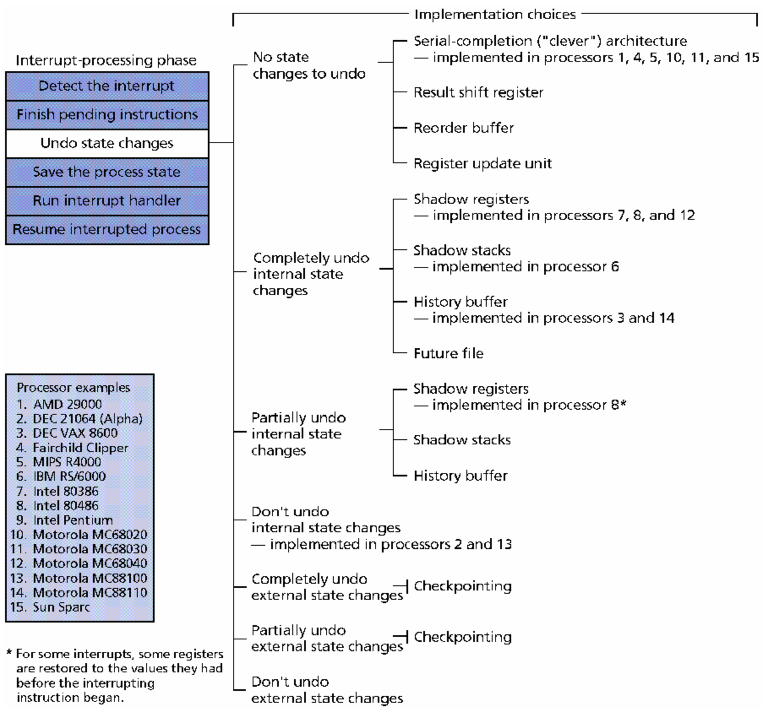
Implementation choices
Interrupt-processing phase
Detect the interrupt
Finish pending instructions
Undo state changes
Save the process state
Run interrupt handler
Resume interrupted process
No state changes to undo
Serial-completion ("clever") architecture — implemented in processors 1, 4, 5, 10, 11, and 15
Result shift register
Reorder buffer
Register update unit
Completely undo internal state changes
Shadow registers — implemented in processors 7, 8, and 12
Shadow stacks — implemented in processor 6
History buffer — implemented in processors 3 and 14
Future file
Partially undo internal state changes
Shadow registers — implemented in processor 8*
Shadow stacks
History buffer
Don't undo internal state changes — implemented in processors 2 and 13
Completely undo external state changes
Checkpointing
Partially undo external state changes
Checkpointing
Don't undo external state changes
Processor examples
1. AMD 29000
2. DEC 21064 (Alpha)
3. DEC VAX 8600
4. Fairchild Clipper
5. MIPS R4000
6. IBM RS/6000
7. Intel 80386
8. Intel 80486
9. Intel Pentium
10. Motorola MC68020
11. Motorola MC68030
12. Motorola MC68040
13. Motorola MC88100
14. Motorola MC88110
15. Sun Sparc
* For some interrupts, some registers are restored to the values they had before the interrupting instruction began.

Figure 8. Phase 3: Undo process state changes.

Phase 4: Save the process state

Here, the processor saves the state that will be restored in phase 6. Saving occurs after pending instructions are allowed to complete in phase 2 and state changes are undone in phase 3. The program counter saved in this phase generally points to the first instruction that will be loaded into the processor after the interrupt handler completes, but this can vary in individual implementations.

About half of the processors cited in Figure 9 (see next page) use stacking for state saving. This is feasible because the process state was already serialized in phases 2 and 3, so there is not much state to save. The state, while not explicitly saved in the auxiliary processor branch of the taxonomy, is preserved in situ as described earlier. The only branch in which absolutely no state saving or preservation takes place is "no process resumption" branch.

Phase 5: Run the interrupt handler

This phase involves two tasks: The processor identifies the interrupt handler that needs to be run, then runs it. (See Figure 10.)

Some interrupts (for example, a page fault) might be designed so that no software interrupt handler is required. The necessary page could be loaded into main memory by specialized hardware or microcode rather than by a software interrupt handler. However, we are unaware of any processors that do this.

In an interrupt vector approach, starting addresses of the interrupt handlers are stored in a table where the processor can access them when an interrupt occurs. Special-purpose hardware identifies the cause of the interrupt and loads the correct interrupt vector into the program counter. Interrupt registers or devices could be polled in the interrupt handler invoked here, although this polling does not determine which interrupt handler to run.

When an interrupt occurs in an interrupt register implementation, identifier bits are set in a register by the interrupting device to indicate the source of the interrupt. The processor or the interrupt handler tests this register and takes the appropriate action. The IBM System/360 and System/370 use this technique for external interrupts.

Another implementation possibility for external interrupts is software polling. An interrupt handler is invoked in software polling when an external interrupt occurs. The interrupt handler must then poll the I/O devices to discover which one of them caused the interrupt.

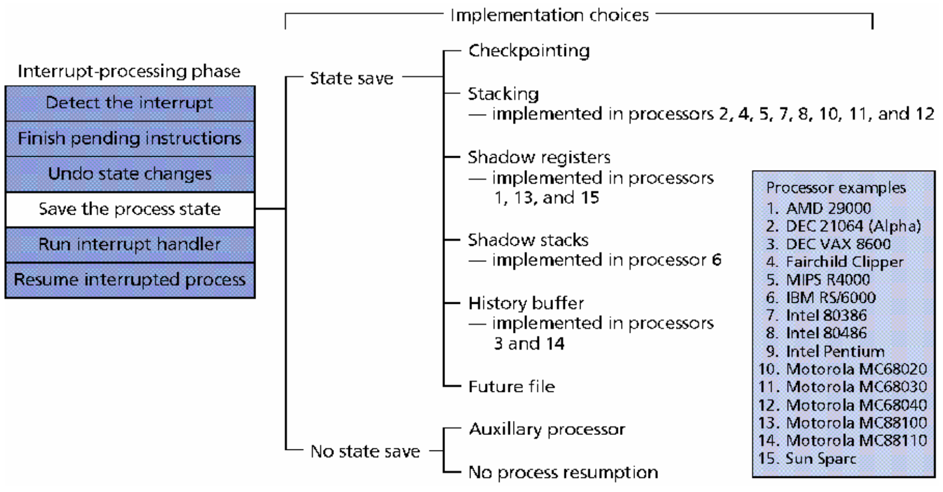

Figure 9. Phase 4: Save the state of the process.

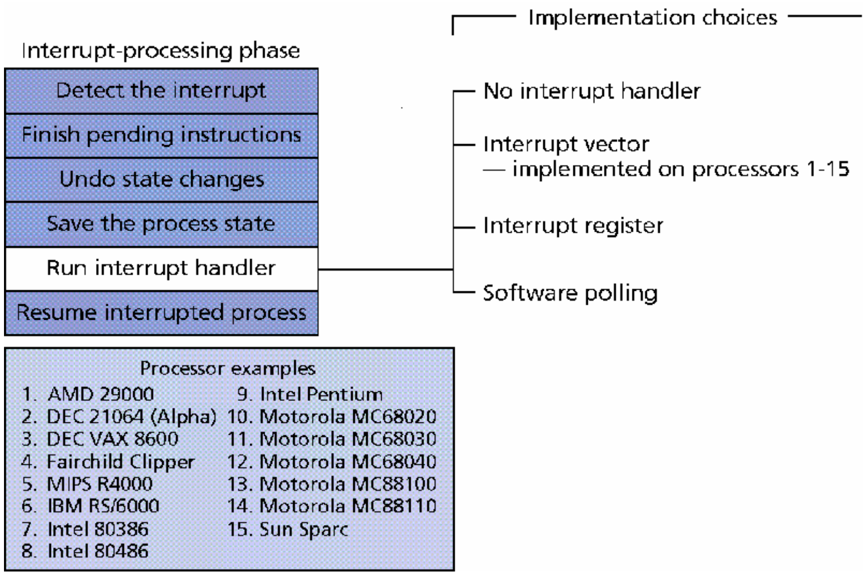

Figure 10. Phase 5: Run the interrupt handler.

Phase 6: Resume interrupted process

The interrupted process resumes from the process state that was saved in phase 4 (see Figure 11). The instructions that are being either completely or partially re-executed in this taxonomy are those that were required to be completely unexecuted by Smith and Pleszkun's second condition, but which were already far into the pipeline when the interrupt occurred.

In complete re-execution, all these instructions are restarted from the beginning of the pipeline. In partial re-execution, these instructions are backstepped either an integral number of pipeline stages (atomic stage backstep) or are some number of clock cycles less than a full pipeline stage (partial stage backstep). Partial stage backstep is possible only in a pipeline with a stage that sometimes requires more than one clock cycle to complete.

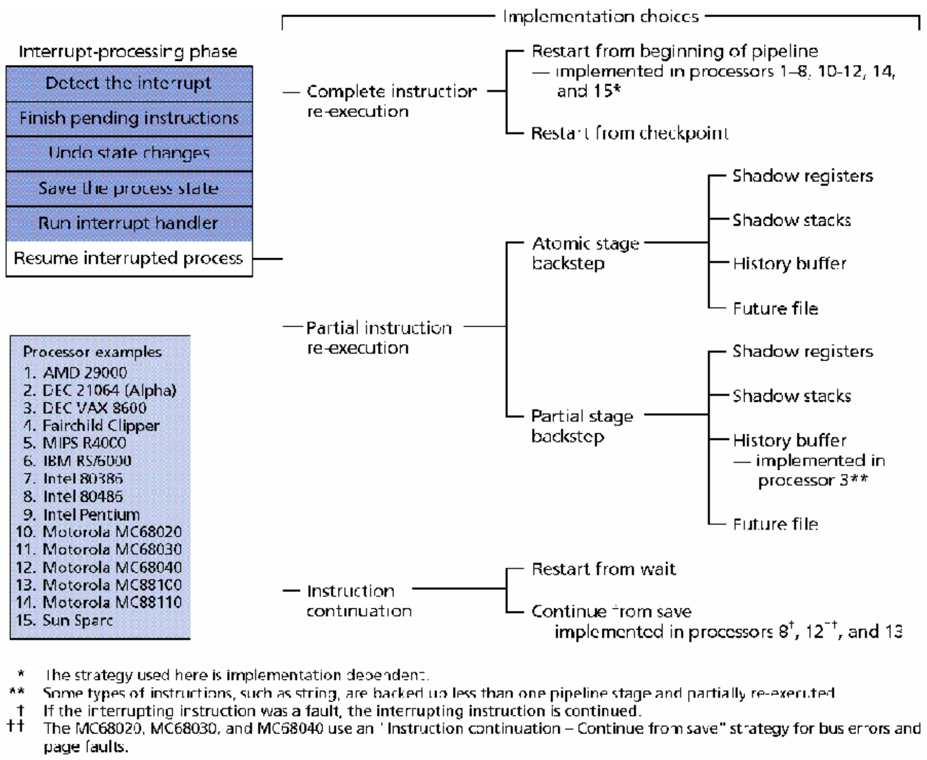
Implementation choices
Interrupt-processing phase
Detect the interrupt
Finish pending instructions
Undo state changes
Save the process state
Run interrupt handler
Resume interrupted process
Complete instruction re-execution
Restart from beginning of pipeline — implemented in processors 1–8, 10-12, 14, and 15*
Restart from checkpoint
Partial instruction re-execution
Atomic stage backstep
Shadow registers
Shadow stacks
History buffer
Future file
Partial stage backstep
Shadow registers
Shadow stacks
History buffer — implemented in processor 3**
Future file
Instruction continuation
Restart from wait
Continue from save implemented in processors 8†, 12††, and 13
Processor examples
1. AMD 29000
2. DEC 21064 (Alpha)
3. DEC VAX 8600
4. Fairchild Clipper
5. MIPS R4000
6. IBM RS/6000
7. Intel 80386
8. Intel 80486
9. Intel Pentium
10. Motorola MC68020
11. Motorola MC68030
12. Motorola MC68040
13. Motorola MC88100
14. Motorola MC88110
15. Sun Sparc
* The strategy used here is implementation dependent.
** Some types of instructions, such as string, are backed up less than one pipeline stage and partially re-executed
† If the interrupting instruction was a fault, the interrupting instruction is continued.
†† The MC68020, MC68030, and MC68040 use an "Instruction continuation – Continue from save" strategy for bus errors and page faults.

Figure 11. Phase 6: Resume interrupted process.

For instruction continuation, work previously done by the processor need not be redone. There are two methods by which continuation can occur. The first is by an auxiliary processor that handles interrupts so that the main processor is stopped when the interrupt occurs and restarted after the interrupt is handled. In the second method, continuation occurs when the process state saved by the processor in phase 3 is so complete that no work need be redone when the interrupted process resumes.

Interrupt-processing caveats

With respect to commercial processors, any attempt to categorize their interrupts in accordance with our approach might show that for a given processor, different types of interrupts use different processing strategies.

For example, on many processors, some instructions—such as those for moving memory blocks around—take a long time to execute. If it took 2,000 processor cycles to execute such an instruction, an interrupt at the 2,000th cycle could cause the processor to redo 1,999 work cycles. Consequently, most processors have a specific mechanism to continue instructions like this from an intermediate step in the calculations, rather than having to restart the whole instruction (for example, Intel's Repeat String instruction). However, most types of interrupts on a given processor do not have this capability, as it is usually either unnecessary or too costly to implement.

We can therefore conclude that, to be completely accurate, each possible processor interrupt (or type of interrupt) should be classified on an individual basis. As a whole, processors tend to resist being conveniently classified because of the kinds of special cases just described.

Conclusion

As systems designers evaluate the sometimes-perplexing array of hardware options, it might appear that the task of designing interrupt-processing strategies becomes increasingly complex. Processors are more concurrent today, and a processor's performance and ease of design depend on the choice of interrupt-processing strategy. When (and if) new processor implementations, such as VLIW (very long instruction word), come into wider use, the design choices will be entirely different from those of today's processors. Our taxonomy approach, which encompasses several possible hardware techniques (both old and new) can help lessen the difficulty of design, although our approach does not—cannot, in fact—address all situations for designers sorting through myriad interrupt-processing techniques and implementations. However, our approach does capture the essence of interrupt-processing possibilities and, in that way, it can help designers of next-generation computers.

Acknowledgments

We thank Vijay Kumar Garg for his helpful comments while reviewing the thesis on which this article is based. We also thank our anonymous reviewers for their detailed, scholarly, and very helpful comments.

This work was supported by a Microelectronics Development Fellowship and an Engineering Doctoral Fellowship, both from the University of Texas at Austin.

References

1. G.J. Meyers, *Advances in Computer Architecture,* 2nd ed., John Wiley and Sons, New York, 1982.
2. D.J. Kuck, *The Structure of Computers and Computations,* John Wiley and Sons, New York, 1978.
3. J.-L. Baer, *Computer Systems Architecture,* Computer Science Press, Rockville, Md., 1980.
4. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, Palo Alto, Calif., 1989.
5. J.E. Smith and A.R. Pleszkun, “Implementation of Precise Interrupts in Pipelined Microprocessors,” *Proc. 12th Ann. Int’l Symp. Computer Architecture,* CS Press, Los Alamitos, Calif., 1985, pp. 36-44.
6. 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.
7. W.M.W. Hwu and Y.N. Patt, “Checkpoint Repair for Out-of-Order Execution Machines,” *IEEE Trans. Computers,* Vol. C-36, No. 12, Dec. 1987, pp. 1,515-1,522.
8. R.M. Keller, “Look-Ahead Processors,” *Computing Surveys,* Vol. 7, No. 4, Dec. 1975, pp. 177-195.
9. G.S. Sohi and S. Vajapeyam, “Instruction Issue Logic for High-Performance, Interruptible Pipelined Processors,” *Proc. 14th Ann. Int’l Symp. Computer Architecture,* CS Press, Los Alamitos, Calif., 1987, pp. 27-34.
10. R.M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM Journal of Research and Development,* Vol. 11, Jan. 1967, pp. 25-33.
11. N. Ullah and M. Holle, “The MC88110 Implementation of Precise Exceptions in a Superscalar Architecture,” *Computer Architecture News,* Vol. 21, No. 1, Mar. 1993, pp. 15-25.
12. W.A. Walker, “Interrupt-Processing Strategies in Pipelined Microprocessors” master’s thesis, University of Texas at Austin, Dec. 1992.

Wade Walker, a PhD candidate at the University of Texas at Austin, is an engineer at Advanced Micro Devices in the Argon/K7 design group. His research interests are in computer architecture with special emphasis on precise interrupts for concurrent processors.

Walker received BS and MS degrees in electrical engineering from the University of Texas at Austin.

Harvey G. Cragon is a professor at the University of Texas at Austin, where he has held the Ernest Cockrell Jr. Centennial Chair in Engineering since 1984. His research interests are in computer architecture design and high-speed computers.

Cragon received a BSEE degree from Louisiana Polytechnic Institute in 1950. He was presented the IEEE Emanuel R. Piore Award in 1984 and the ACM/IEEE Eckert-Mauchly Award in 1986. Cragon is an IEEE Fellow, an ACM Fellow, and a member of the National Academy of Engineering, the Computer Society, and the ACM.

Readers can contact the authors at wade.walker@amd.com and cragon@uts.cc.utexas.edu.