Exceptions, Traps, and Interrupts

Exceptions (as the word indicates) are rare events that are triggered by the hardware and force the processor to execute an exception handler. The fact that this event is triggered by the hardware and is not explicitly scheduled in the code is the major difference between exceptions and branches/jumps, although the difference can be sometimes very subtle, such as for exceptions caused by traps to the kernel. Because of their similarities, in most machines, branches and exceptions share the same hardware mechanisms. Exceptions are part of the ISA specification and must be supported by any hardware implementation.

Exceptions may be caused by an instruction, by external interrupts, or by hardware malfunctions. Exceptions caused by program instructions are synchronized with that instruction. If the program must resume after the exception it must first be stopped at the faulting instruction and then resumed at the faulting instruction. By contrast exceptions due to I/O interrupts and hardware failure/malfunction are not synchronized with a program instruction. They may stop program execution at any instruction. However, interrupts should be taken promptly, lest they may be lost. Because interrupts are not synchronized with program instructions, they are generally easier to handle.

Let's describe some examples of exceptions.

• **I/O Device interrupt.** Most I/O operations are interrupt-driven because of their large latency. Typically the CPU starts an I/O operation by programming an I/O device such as a DMA (Direct Memory Access) controller or an I/O processor. Then the I/O device executes the I/O operation while the CPU continues its processing. When the I/O operation is completed, the I/O device signals the CPU by raising an interrupt signal. When the CPU sees the signals it executes an interrupt handler. Because the I/O operation is external to the CPU and is unrelated to the current program executing on the CPU, the CPU can select the cycle in which it wants to take the interrupt.
• **Operating system call.** When the user wants to invoke a service from the operating system it executes a TRAP instruction. Usually the trap instruction has a parameter indicating an entry in a trap table that gives the entry point of the handler for the requested service. This type of exception is very similar to a jump to subroutine.
• **Instruction Tracing and Breakpoints.** Most machines have hardware support for tracing the execution of programs. In tracing mode, the CPU traps on every single instruction so that the exception handler can record the state of the CPU before the execution of the instruction in a trace, which is then saved on disk. With the trace, all kinds of valuable information can be obtained for the design of compilers and architectures. The trace can also be used to simulate other architectures. Breakpoints used by program debuggers are also similarly supported in hardware through exceptions.
• **Integer or floating-point exceptions.** Arithmetic instructions (especially floating point) may cause a whole range of exceptions. The most familiar ones are underflow and over-

flow exceptions. On an overflow or underflow the handler may resume the process or may not. Arithmetic exceptions may also signal illegal operations such as division by 0.

• **Page Faults.** Modern systems support virtual memory. The system physical memory only contains a few of the pages (or blocks) of data/instructions needed by the processor. If the page is not found in memory on an instruction fetch or data access, the CPU is trapped to the kernel to perform memory management functions and bring the faulting page into memory.

• **Misaligned memory access.** A data is not aligned in memory according to its size.

• **Memory protection violation.** Memory access may be out of bounds or may violate access rights (for example, write access to a read-only page). In these cases the program must be stopped before it goes awry.

• **Undefined instruction.** If the instruction decoder detects an unknown/illegal code (opcode or addressing mode), it does not know what to do, thus it must trap the process. This exception can be used to extend the ISA. For example, an instruction set without floating-point instructions may be extended by assigning illegal opcodes to floating point instructions and then emulating them in software in the exception handler.

• **Hardware failure/alarm.** Various hardware components such as buses and memories are protected by error detection/correction code. When a hardware component cannot recover from an error it must trap the process running on the CPU. In modern systems, sensors are distributed throughout the machine and may trap the CPU when conditions (such as local temperature) enter a danger zone.

• **Power failure.** When power fails the voltage drops slowly because of capacitive effects. Thus there may be time to salvage as much as possible of the current computation and environment before the system fails, so that it can be recovered later.

When an exception is synchronized with an instruction i and the process must be resumed after the exception has been processed, the processor's state at the end of instruction i-1 and before instruction i must be saved before executing the handler so that the process' execution can be resumed. Therefore all instructions preceding instruction i in process order must be completed and instruction i plus all following instructions in process order must ~~but~~ be aborted. Process order is the dynamic order of instruction execution when instructions are executed one at a time. Such exceptions are called *precise exceptions*. Exceptions such as page fault or arithmetic overflow must be precise. Other exceptions such as hardware failure or memory access violation do not have to be precise because the process will be terminated at the end of the handler.

Precise exceptions constrain~~t~~ what can be done by the hardware or the compiler. Only exceptions intended by the programmer may be triggered. Although exceptions are rare, they can be caused by virtually any instruction and are unpredictable until the instruction has reached the stage of its execution where the exception can be detected. Because exceptions are rare, there is no need to try to speedup their handling when they occur. However the compiler and the architecture must have built-in mechanisms to detect and recover from precise exceptions.

Dealing with precise exceptions in the 5-stage pipeline

In the 5-stage pipeline precise exceptions may be triggered in the IF stage (page fault), the ID stage (undefined instruction), the EX stage (arithmetic overflow), or the ME stage (page fault). A precise exception cannot happen in the WB stage because at that point the only remaining action is to write to the register file and only a hardware error could trigger an exception.

When a precise exception occurs in the 5-stage pipeline, the hardware must do the following:

• all preceding instructions in process order must complete,
• all instructions following the faulting instruction plus the faulting instruction itself must be squashed
• the execution of the handler must be started.

One tempting approach would be to take the exception in the cycle when it occurs. However this will be very complex, because 1) the stages to flush vary with the stage in which the exception occurs, 2) the exception condition and excepting PC must be accessible to treat the exception and may reside in different stages, 3) multiple exceptions may occur in the same clock cycle, and 4) exceptions must be taken in process order, not in temporal order.

To understand the significance of this last requirement, assume for example that a page-fault occurs in IF. If we take the exception at the end of the clock cycle, then the page-fault handler will start in the next cycle. However, one or several preceding instructions in the ID or EX stages may cause an exception later as they complete their execution. Since these instructions precede the faulting instruction in process order their exception should have been taken first.

A radical, simple solution to these problems is to flag an exception when it happens and keep the exception “silent” until the instruction reaches the write-back stage. Each instruction “carries” its PC and an Exception_Status_Register (ESR) with it. On the first exception, the exception is recorded in the ESR of the instruction and the instruction is NOOPed. When the instruction reaches the WB stage, the exception is taken. There are several technical issues such as a store in ME must be disabled if the preceding instruction in WB takes an exception.