Exception handling

Donatella Sciuto: donatella.sciuto@polimi.it
Interrupts: altering the normal flow of control

An external or internal event that needs to be processed by another (system) program. The event is usually unexpected or rare from program’s point of view.
Causes of Interrupts

Interrupt: an event that requests the attention of the processor

- Asynchronous: an external event
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure

- Synchronous: an internal event (a.k.a. exceptions)
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - virtual memory exceptions: page faults, TLB misses, protection violations
  - traps: system calls, e.g., jumps into kernel
History of Exception Handling

- First system with exceptions was Univac-I, 1951
  - Arithmetic overflow would either
    1. trigger the execution a two-instruction fix-up routine at address 0, or
    2. at the programmer’s option, cause the computer to stop
  - Later Univac 1103, 1955, modified to add external interrupts
  - Used to gather real-time wind tunnel data

- First system with I/O interrupts was DYSEAC, 1954
  - Had two program counters, and I/O signal caused switch between two PCs
  - Also, first system with DMA (direct memory access by I/O device)

[Courtesy Mark Smotherman]
DYSEAC, first mobile computer!

- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

[Courtesy Mark Smotherman]
Classes of exceptions

- Synchronous vs asynchronous
  - Asynchronous events are caused by devices external to the CPU and memory and can be handled after the completion of the current instruction (easier to handle)

- User requested vs coerced
  - User requested are predictable: treated as exceptions because they use the same mechanisms that are used to save and restore the state; handled after the instruction has completed. Coerced are caused by some hw event not under control of the program

- User maskable vs user nonmaskable
  - The mask simply controls whether the hardware responds to the exception or not
Classes of exceptions

- Within vs between instructions
  - Exceptions that occur within instructions are usually synchronous since the instruction triggers the exception. The instruction must be stopped and restarted.
  - Asynchronous that occur between instructions arise from catastrophic situations and always cause program termination.

- Resume vs terminate
  - Terminating event: program’s execution always stops after the interrupt.
  - Resuming event: program’s execution continues after the interrupt.
Asynchronous Interrupts: invoking the interrupt handler

- An I/O device requests attention by asserting one of the prioritized interrupt request lines.

- When the processor decides to process the interrupt:
  - It stops the current program at instruction $I_i$, completing all the instructions up to $I_{i-1}$ \textit{(precise interrupt)}.
  - It saves the PC of instruction $I_i$ in a special register (EPC).
  - It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode.
Interrupt Handler

- Saves PC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move PC into GPRs
  - need a way to mask further interrupts at least until PC can be saved
- Needs to read a status register that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (return-from-exception) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state
Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a particular instruction.

- In general, the instruction cannot be completed and needs to be restarted after the exception has been handled:
  - requires undoing the effect of one or more partially executed instructions.

- In the case of a system call trap, the instruction is considered to have been completed:
  - a special jump instruction involving a change to privileged kernel mode.
An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which all instructions before that instruction have committed their state and no following instructions including the interrupting instruction have modified any state.

- This means, effectively, that you can restart execution at the interrupt point and “get the right answer”
- Implicit in our previous example of a device interrupt:
  - Interrupt point is at first *lw* instruction

```
add    r1, r2, r3
subi   r4, r1, #4
slli   r4, r4, #2
lw     r2, 0(r4)
lw     r3, 4(r4)
add    r2, r2, r3
sw     8(r4), r2
...    
```
Why are precise interrupts desirable?

Many types of interrupts/exceptions need to be restartable. Easier to figure out what actually happened:

I.e. TLB faults.
Need to fix translation, then restart load/store
IEEE gradual underflow, illegal operation, etc:

e.g. Suppose you are computing: 
Then, for 

\[ f(x) = \frac{\sin(x)}{x} \]

\[ f(0) = \frac{0}{0} \Rightarrow NaN + \text{illegal operation} \]

Want to take exception, replace NaN with 1, then restart.
Why are precise interrupts desirable?

- Restartability doesn’t require preciseness. However, preciseness makes it a lot easier to restart.
- Simplify the task of the operating system a lot
  - Less state needs to be saved away if unloading process.
  - Quick to restart (making for fast interrupts)
How to handle multiple simultaneous exceptions in different pipeline stages?
How and where to handle external asynchronous interrupts?
Precise Exceptions in simple 5-stage pipeline:

- Exceptions may occur at different stages in pipeline (i.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts? The doctor’s mandate of “do no harm” applies here: try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as “cause exception or not” and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc
Another look at the exception problem

- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don’t act on exception until it reaches WB stage

- Handle interrupts through “faulting noop” in IF stage

- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!
Approximations to precise interrupts

- Hardware has imprecise state at time of interrupt
- Exception handler must figure out how to find a precise PC at which to restart program.
  - Done by emulating instructions that may remain in pipeline
  - Example: SPARC allows limited parallelism between FP and integer core:
    - possible that integer instructions #1 - #4 have already executed at time that the first floating instruction gets a recoverable exception
    - Interrupt handler code must fixup <float 1>, then emulate both <float 1> and <float 2>
    - At that point, precise interrupt point is integer instruction #5
    - Vax had string move instructions that could be in middle at time that page-fault occurred.
    - Could be arbitrary processor state that needs to be restored to restart execution.
Exception Handling 5-Stage Pipeline

PC → Inst. Mem → Decode → E → M → Data Mem → W

- PC address Exception
- Illegal Opcode
- Overflow
- Data address Exceptions
- Asynchronous Interrupts

Select Handler PC → Kill F Stage → Kill D Stage → Kill E Stage → Asynchronous Interrupts

PC → Kill Writeback
Exception Pipeline Diagram

\[ \text{time} \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots . \]

\[ (I_1) \ 096: \text{ADD} \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{nop} \quad \text{nop} \quad \text{overflow!} \]

\[ (I_2) \ 100: \text{XOR} \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{nop} \quad \text{nop} \]

\[ (I_3) \ 104: \text{SUB} \quad \text{IF}_3 \quad \text{ID}_3 \quad \text{nop} \quad \text{nop} \quad \text{nop} \]

\[ (I_4) \ 108: \text{ADD} \quad \text{IF}_4 \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \quad \text{nop} \]

\[ (I_5) \text{ Exc. Handler code} \quad \text{IF}_5 \quad \text{ID}_5 \quad \text{EX}_5 \quad \text{MA}_5 \quad \text{WB}_5 \]

Resource Usage

- IF
- ID
- EX
- MA
- WB
Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Inject external interrupts at commit point (override others)
- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage
How to achieve precise interrupts when instructions executing in arbitrary order?

- Jim Smith’s classic paper discusses several methods for getting precise interrupts:
  - In-order instruction completion
  - Reorder buffer
  - History buffer
- We will discuss these after we see the advantages of out-of-order execution.
Speculating on Exceptions

- **Prediction mechanism**
  - Exceptions are rare, so simply predicting no exceptions is very accurate!

- **Check prediction mechanism**
  - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types

- **Recovery mechanism**
  - Only write architectural state at commit point, so can throw away partially executed instructions after exception
  - Launch exception handler after flushing pipeline

- **Bypassing** allows use of uncommitted instruction results by following instructions