Interrupts and Exceptions

Notes

Material in this set from Section 3.6.

The book uses "exception" as a general term for all interrupts ...
... in these notes interrupt is used as the general term ...
... and a narrower definition is used for exception.

The definitions of trap, interrupt, and exception given here ...
... are not explicitly provided in the text ...
... but are widely used.

Interrupts

Interrupt:
Event that interrupts normal program flow.

Operating system "takes over" computer ...
... attends to whatever caused the interrupt ...
... and (most of the time) resumes interrupted program.

Interrupt Terminology

Handler:
The OS program that "takes over" in response to interrupt.

Privileged Mode:
A state in which the CPU controller and memory system ...
... do not restrict instructions that can be executed ...
... or memory that can be accessed.

Processor switches into privileged mode in response to interrupt ...
... and out of privileged mode when resuming the program.

Three Types of Interrupts.

- *Trap:*
  Sort of a subroutine call to OS.
- *Exception:*
  Something went wrong, triggered by an executing instruction.

  Exception has both a general and this specific meaning.
- *Hardware Interrupt:*
  Something outside the CPU is trying to get the computer's attention.

  Interrupt has both a general and this specific meaning.

Traps

Trap:
(1) An instruction intended for user programs that transfers control to the operating system (privileged code).
(2) The execution of such an instruction.

Sort of a subroutine call to OS.

Trap causes branch to OS code and a switch to *privileged mode*.

Privileged Mode:
A processor mode in which there are fewer restrictions on instruction execution.

Some instructions can only be executed in privileged mode.

When in privileged mode a *trap handler* is executed to service request.

Trap Handler:
A program, running in privileged mode that responds to a trap.

Traps typically used for I/O, memory allocation, etc.

Example, SPARC V8 trap instruction:

`ta` ⟨**rs1**⟩,⟨**imm**⟩.

ISA has a *trap base register (TBR)* that is used to construct the trap address.

Trap address is in *trap table*, each entry holds first four instructions of trap handler.

Trap Address Construction:

OS initializes TBR with upper 20 bits of trap table base.

When, say, `ta r1,3` executed, bits 4-10 set to low seven bits of `r1+3`.

Low four bits of TBR always zero.

Example: Using trap for read on Solaris (Sun OS)

```
System Calls                                                read(2)

NAME
     read, readv, pread - read from file

SYNOPSIS
     #include <unistd.h>
     ssize_t read(int fildes, void *buf, size_t nbyte);

DESCRIPTION
     The read() function attempts to read nbyte  bytes  from  the
     file  associated with the open file descriptor, fildes, into
     the buffer pointed to by buf.
```

```
! Read System Call.
! Parameters placed in %o0, %o1, %o2.
!  %o0:  File descriptor (fildes)
!  %o1:  Buffer pointer (buf, address where read data copied to).
!  %o2:  Number of bytes to read. (nbyte)

mov  SYS_read, %g1   ! Argument for trap.
ta   %g0, ST_SYSCALL ! Call trap.
```


Some SPARC Trap Codes

```
/* Copyright (c) 1989 by Sun Microsystems, Inc. */
/* Software traps (ticc instructions). */
/* In sys/trap.h */
#define ST_OSYSCALL 0x00
#define ST_BREAKPOINT 0x01
#define ST_SYSCALL 0x08
#define ST_GETCC 0x20  // Move condition code to reg.
#define ST_SETCC 0x21  // Move condition code from reg.

/* In sys/syscall.h */
#define SYS_syscall 0
#define SYS_exit 1
#define SYS_fork 2
#define SYS_read 3
#define SYS_write 4
#define SYS_open 5
#define SYS_close 6

mov  SYS_read, %g1   ! Argument for trap.
ta   %g0, ST_SYSCALL ! Call trap.
```


Exceptions

Exception:
An interruption in normal execution triggered by an instruction that could not complete execution.

An exception occurs when an instruction cannot fully execute.

Faulting Instruction:
Instruction that caused an exception.

Some Exception Causes

- Access to unallocated memory, a segmentation fault.
- Access to memory that's paged out (on disk), a page fault.
- Division by zero.

In response to an exception . . .
. . . OS either fixes problem and re-tries instruction . . .
. . . or terminates program.

Exceptions frequently occur in the *middle* of an instruction . . .
. . . which has to be re-started when the program resumes.

Exception Example

Exception in MEM stage of `lw`.

Example:

```
add  r1, r2, r3         !  Compute address.
lw   r6, 0(r1)          !  Load data. (May encounter page fault.)
sub  r5, r6, r7         !  Use loaded data.
```

Here, `lw` may generate a page-fault exception.

If so, page fault handler starts after `add` finishes.

When handler returns, execution resumes with `lw` (its second try).

Interrupt

Caused by an external event . . .
. . . which typically needs routine attention.

For example,

- Disk drive has data that was requested 20 ms ago.
- User pressed a key on the keyboard.
- User sneezed, causing mouse to move.
- Timer (used by the OS as an alarm clock) expired.

Hardware Interrupt Example

Example:

```
add  r1, r2, r3   IF  ID  EX  MEM WB
sub  r4, r5, r6       IF  ID  EX  MEM WB
xor  r7, r8, r9           IF  ID  EX  MEM WB
```

As execution reaches code above, *achoooo* (user sneezes) . . .
. . . moving mouse, triggering an interrupt.

Based on time of sneeze, hardware completes `add` and `sub` . . .
. . . but squashes `xor` (for now).

The handler starts . . .
. . . the screen pointer (the little arrow) is moved . . .
. . . the handler finishes . . .
. . . and execution resumes with `xor`.

Interrupt Mechanisms: Goals and Difficulties

Goals

Exceptions: Handle in program order. (Not trivial.)

All Interrupts: Ability to resume execution as though nothing happened.

Precise Exception: Last instruction must immediately precede faulting instruction.

Difficulties

Interrupt mechanisms are hard to design . . .
. . . because it's hard to stop a pipeline in the middle of something . . .
. . . and have it resume again later . . .
. . . as if nothing happened.

Actions Initiated by HW Interrupt & Exceptions — Simple Case

Hardware executes up to and including last instruction . . .
. . . and squashes all following instructions.

- Type and timing of interrupt determine a *last* instruction.
- The last and all preceding instructions allowed to complete . . .
  . . . instructions following last are squashed (nullified).
- A `trap` instruction or its address is inserted in pipe by hardware . . .
  . . . which jumps to handler (OS code) and switches to privileged mode.
- Handler attends to interrupt.
- If appropriate, state is restored and program resumes . . .
  . . . with the instruction following last.

The Last Instruction

The last and all preceding instructions must execute completely . . .
. . . while instructions following the last must have no effect at all . . .
. . . even though they may have already started when the interrupt occurred.

These *squashed* instructions will be executed after the handler completes.

Choice of Last Instruction

Choice depends on type of interrupt.

- For traps, the trap instruction itself is last.

  No problem.

- For hardware interrupts, a convenient last instruction can be chosen.

  No problem again.

- Precise exception, the instruction preceding faulting instruction.

  Problem: exception can occur in any of several stages.

  Problem: more than one instruction can raise exception.

  Problem: an instruction can raise exception before its *predecessor*.

  Problem: despite problems, some exceptions must be precise.

Squashing Instructions

When an interrupt occurs instructions may be squashed.

When the handler finishes and the program resumes . . .
it must be as though the squashed instructions never even started.

So . . .
. . . they cannot write registers . . .
. . . they cannot write memory . . .
. . . or set any kind of condition codes . . .
. . . *unless* . . .
. . . the state change can be un-done.

Squashing Instructions in DLX

In the DLX implementation it's easy to squash integer instructions . . .
. . . because they only change state in the last two stages (MEM & WB).

To squash an instruction in the IF, ID, or EX stages . . .
. . . the opcode is replaced with a NOP . . .
. . . or any control bits that initiate a memory or register write . . .
. . . are set to perform no action.

An instruction in WB cannot easily be squashed . . .
. . . because the following instruction, in MEM, . . .
. . . would already be changing state.

Fortunately, there's never a need to squash an instruction in WB . . .
. . . although an already-squashed instruction can enter WB.

An instruction in MEM cannot be squashed . . .
. . . *unless* the memory operation fails . . .
. . . which, luckily, is the only reason to squash the instruction.

Implementing Exceptions in DLX

Each stage has its own hardware to detect exceptions.

IF: Page fault etc on instruction fetch, detected by mem port.

ID: Illegal opcode, detected by decode logic.

EX: Arithmetic exception, detected by ALU.

MEM: Page fault on load/store, detected by memory port.

Pipeline latches have exception registers.

Normally set to null.

If exception occurs, written with exception info.

When exception occurs:

- Exception register (of faulting instruction) written.
- NOPs written to IRs for following (to the left) instructions.
- Instruction fetch stops.
- Following instructions proceed normally.

In writeback stage:

Exception register checked (every cycle).

If exception register non-null . . .
. . . exception info, PC, and other information copied somewhere . . .
. . . and `trap` instruction placed in IF stage of pipeline.

Note

- Exceptions handled in program order because exception register tested in WB.

Actions Initiated by Interrupts & Exceptions — Complex Case

Hardware executes *some* instructions up to and including last ...
... and squashes all following instructions ...
... handler must arrange things so execution can resume where it left off.

- Type and timing of interrupt determine a *last* instruction.
- Last and preceding instructions may or may not complete.
- A `trap` instruction is inserted in pipe by hardware ...
  ... which jumps to handler (OS code) and switches to privileged mode.
- Handler attends to interrupt.
- If program is to resume ...
  ... handler may have to ...
  ... determine which instructions finished ...
  ... and which were in progress.
- The handler would have to restore state so that ...
  ... an interrupted instruction can resume in the middle.

Precise Exceptions

Precise exceptions are necessary for some instructions ...
... and expensive for others.

They are necessary for instructions ...
... such as memory loads and stores.

For other instructions they are a convenience ...
... for example FP instructions ...
... that can write error values instead of numbers ...
... if they don't complete.

In many systems precise exceptions are optional for floating point ...
... but always provided for other instructions.