Exceptions and interrupts in Pipelined processors

E. Sanchez

Politecnico di Torino
Dipartimento di Automatica e Informatica

EXCEPTIONS

Exceptions are events that modify the normal execution order of the program.

Exceptional events (exceptions, interrupts, or faults) are more complex to handle in pipelined architectures because several instructions are being executed at a time.

EXCEPTIONS (2)

The terms *exception* and *interrupt* are often confused

Exception usually refers to an internal CPU event such as

- floating point overflow
- MMU fault (e.g., page fault)
- trap (SWI)

Interrupt usually refers to an external I/O event such as

- I/O device request
- reset

Exception sources

Possible causes of exceptions are:

- I/O device request
- Operating system call by a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow or underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Hardware malfunction
- Power failure.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume vs. terminate.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. non-maskable
- Within vs. between instructions
- Resume vs. terminate

An exception is synchronous if it can be triggered always at the same position in the code. Asynchronous exceptions are normally generated by external devices.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. b
- Resume v

User requested exceptions are similar to procedures.
Coerced exceptions are out of the control of the user program.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instru
- Resume vs. ter

The user can normally force the hardware not to answer to maskable exception requests.

Exceptio

- Synchronous vs
- User requested
- User maskable vs
- Within vs. between instructions
- Resume vs. terminate.

Exceptions can occur either within or between instructions. In the former case they are generated by the instruction itself.

Exception Cla

- Synchronous vs. asynchro
- User requested vs. co
- User maskable vs
- Within vs. between instructions
- Resume vs. terminate.

After activating an exception, the program can either terminate or execute something and then resume.

Restartable machines

Restartable machines are able to handle an exception, save the state, and restart without affecting the execution of the program.

Nearly all processors nowadays are restartable machines.

Stopping the execution

When an exception occurs, the pipeline must execute the following steps:

- force a trap instruction into the pipeline on the next IF stage
- until the trap is taken, turn off all writes for the instruction that raised the exception (i.e., the faulty instruction) and for all the following instructions in the pipeline
- when the exception-handling procedure receives control, it immediately saves the PC of the faulty instruction.

Restarting the execution

After the exception has been handled, special instructions return the machine from the exception by reloading the PC and restarting the instruction stream.

Interrupt protocol in 80x86

1. An external device sends an interrupt request
2. The CPU detects the interrupt
3. The CPU reads the Interrupt Type *N* from the bus
4. The CPU saves the value of the processor status word (PSW) and the return address (Code Segment and Instruction Pointer registers) in the stack
5. The CPU resets the interrupt flags disabling external interrupts and trap mode
6. Using N as an index, the processor reads from the Interrupt Vector Table the address where to jump
7. The CPU jumps to the Interrupt Service Routine.

Interrupt protocol in ARM

1. An external device sends an interrupt request
2. The CPU detects the interrupt
3. The CPU pushes into the current stack the information of the current stack frame composed by 8 registers including the Program Counter and the Processor Status Register.
4. The CPU updates the processor flags and jumps to the address given by the interrupt type N, on that position a new jump to the actual Interrupt Service Routine may be placed.

Interrupt protocol in RISC-V

When an exception occurs, the processor:

1. **saves the current PC and the exception cause to a couple of special registers.**
2. **jumps to the exception handler address.**
3. **updates the processor status to reflect the new privilege level and disables interrupts.**

Precise exceptions

A processor has *precise exceptions* if the pipeline can be stopped so that

- the instructions just before the instruction that triggered the exception are completed, and
- the instructions following the instruction that triggered the exception can be restarted from scratch.

Restarting after an exception may be really hard if exceptions are not handled precisely.

Precise exception handling is a must for most architectures, at least for integer instructions.

Cost of precise exceptions

Some processors (Alpha 21064, MIPS R800) implement precise exceptions in debug mode, only.

This mode is about 10 times slower than usual normal mode.

Imprecise exceptions

Guaranteeing precise exceptions is more complex with multiple cycle instructions.

Example

```
fdiv.d  f0, f2, f4
fadd.d  f10, f10, f8
fsub.d  f12, f12, f14
```

Instructions `fadd.d` and `fsub.d` complete before `fdiv.d` (*out-of-order completion*). If an exception arises during `fsub.d`, an imprecise exception occurs.

Solutions

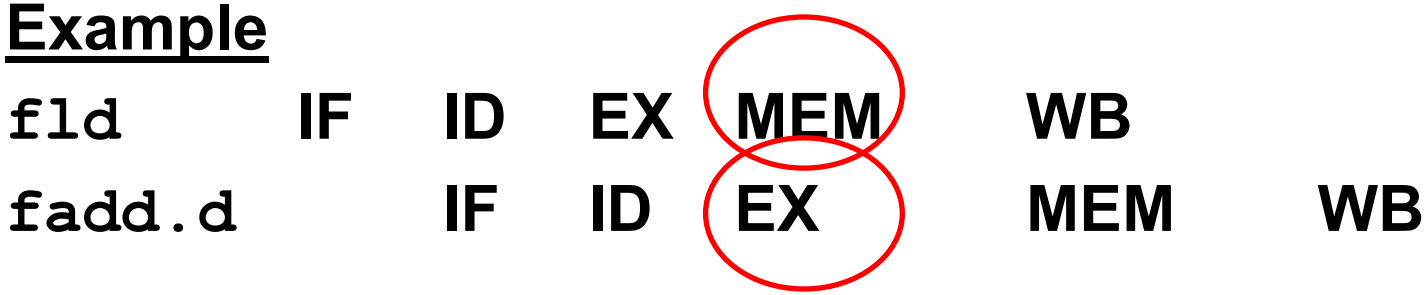
- Accepting imprecise exceptions
- Providing a fast, but imprecise operating mode, and a slow, but precise one (only one FP instruction is executed at a time)
- Forcing the FP units to early determine whether an instruction can cause an exception, and issuing further instructions only when the previous ones are guaranteed not to cause an exception
- Buffering the results of each instruction until all the previously issued instructions have been completed.

RISC-V: Possible sources of Exceptions

Pipeline stage	*Cause of exception*
IF	Page fault on instruction fetch
	Misaligned memory access
	Memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch
	Misaligned memory access
	Memory-protection violation
WB	None

Contemporary exceptions

Example



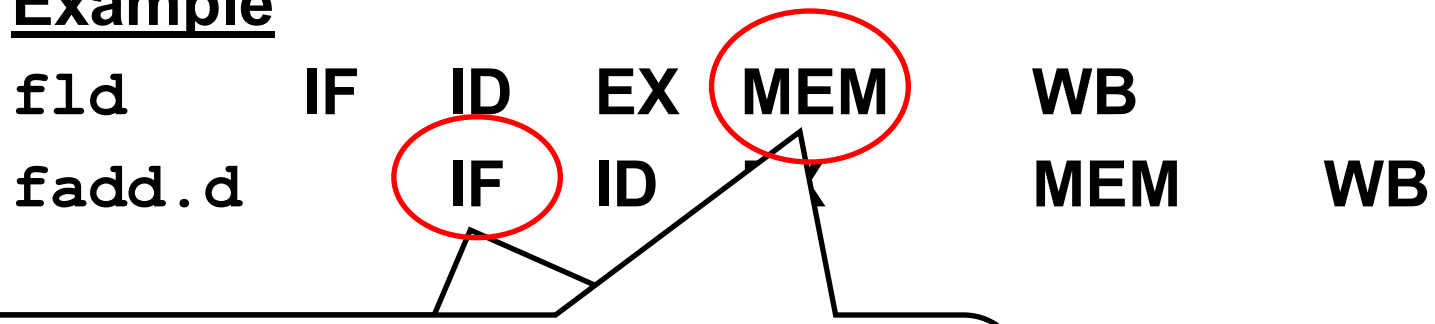
A data page fault exception can occur in the MEM stage of the `fld` instruction, and an arithmetic exception in the EX stage of the `fadd.d` instruction.

The first exception is processed and, if its cause is removed, the second exception is handled.

Exception order

There are cases in which two exceptions can occur in the opposite order of the instructions they relate to.

Example



An exception caused by an instruction page fault in the second instruction occurs before an exception caused by a data page fault in the first instruction.

Exception order (cont'd)

A possible solution is the following:

- a status flag is associated to each instruction in the pipeline
- if an instruction causes an exception, the status flag is set
- if the status flag is set, the instruction cannot perform any write operation
- when an instruction reaches the last stage, and its status flag is set, an exception is triggered.

Instruction set complications

When an instruction is guaranteed to complete it is called *committed*.

Some machines have instructions that change the state before they are committed (e.g., those using autoincrement addressing modes).

If one of these instructions is aborted because of an exception, it leaves the machine state altered. Implementing precise exceptions is thus very difficult.

A possible solution is based on allowing to roll-back all the state changes made by an instruction before it is committed.

Instruction set complications (cont'd)

Instructions implicitly updating condition codes create complications:

- they can cause data hazards
- they require to be saved and restored in the case of an exception
- they make more difficult the task of the compiler for filling possible delay slot between the instruction writing the condition codes and the branch one.

Instruction set complications (cont'd)

Complex instructions are difficult to implement in a pipeline. Forcing them to have the same length can hardly be achieved.

Sometimes these problems are solved by pipelining the microinstructions implementing each instruction.